Superhuman Performance in Reinforcement Learning

Edan Toledo
Department of Computer Science
University of Cape Town
TLDEDA001@myuct.ac.za

Abstract

This paper discusses core reinforcement learning (RL) concepts and methods in addition to elaborating on the field's significant successes in game-AI. The focus is on explaining the systems used in achieving superhuman performance in a variety of game domains. This includes perfect-information games: Go, Chess, and Shogi; Imperfect-information games: Poker; and video games: the Atari suite, Dota 2 and StarCraft. This report is split into three main sections. First, preliminary terminology and formalisms are introduced. Second, foundational single-agent reinforcement learning methods are explained. Third, methods that achieved superhuman performance in various game domains are elaborated on. Additionally, there is a final section illustrating the performance of implemented value-based methods.

Contents

1 Introduction

1.1 Introduction to Reinforcement Learning

Reinforcement learning is a sub-field of machine learning that concerns itself with teaching an agent how to map environmental situations to actions so as to maximise the cumulative reward given by the environment. The goal for the agent is simple: to learn on its own and discover which actions ultimately lead to better outcomes. Through simple trial-and-error, reinforcement learning agents can learn to achieve remarkable performances in a variety of tasks.

Within artificial intelligence, games are an incredible test-bed for new methods and techniques. The development of AI to play games is not anything new within the broader AI field, as some of the earliest AI research set their sights on *solving* games. An example of this is the mastery of a digital Tic-Tac-Toe game by A. S. Douglas in 1952. Early research on games such as chess eventually culminated in incredible achievements such as Gary Kasparov's defeat at the *hands* of IBM's Deep Blue [17]. Over the last decade, game-related AI research has expanded to include games of all types and complexities, including modern video games. This research has pioneered many techniques that are becoming mainstream artificial intelligence, such as Monte Carlo Tree Search, automated game design, procedural content generation and using high-sensory inputs, such as screen capture, to optimise control.

In recent years, we have seen reinforcement learning dominate the field of game-AI. Not only has superhuman performance been achieved in both perfect and imperfect information games, but reinforcement learning has done so exclusively by trial-and-error and self-play. These achievements have sent shockwaves through the world by showing that machines can surpass human intuition and skill in complex domains without any human intervention. The success of these methods seemingly pave the way towards general artificial intelligence systems.

1.2 Preliminary Concepts

The following are preliminary concepts and methods that are required to understand how reinforcement learning can achieve superhuman performance.

1.2.1 The Reinforcement Learning Problem

The reinforcement learning problem asks the question of **how** to *learn* the actions one should take to, ultimately, maximise some given numerical *reward* signal. The entity responsible for learning which actions to take is termed the *agent*, and the world the *agent* finds itself in is termed the *environment*. In RL, learning generally starts *tabula rasa* meaning the agent has no prior information to aid in its discovery of optimal actions. The agent must discover on its own, through means of environment interaction, which actions to take to achieve its goal. This learning by trial-and-error is a distinguishing characteristic of reinforcement learning. Formally, this problem is often modelled using the Markov Decision Process (MDP) framework - see section 1.2.4.

1.2.2 Agent-Environment Interface

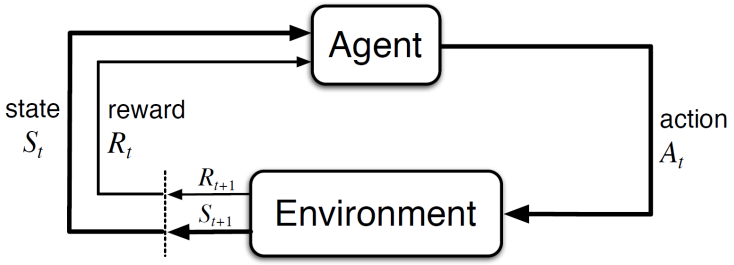

Figure 1: Agent's interaction with environment [59]

The agent-environment interface (see figure 1) describes how interaction/communication takes place between the agent and environment. This interaction cyclically occurs over time. Each cycle represents a new time-step t where the environment presents a new situation (state s_t) to the agent. The agent then needs to decide what to do, i.e. choose an action (action a_t). Each time-step, the environment also gives a numerical reward (r_t) after the execution of some action, which the agent is trying to maximise over time. An agent continues in this cycle of state, action, reward until the terminal time step T is reached. The full sequence of states, actions and rewards is called a trajectory or episode. $(s_1, a_1, r_1, ..., s_T, a_T, r_T)$

1.2.3 States, Actions & Rewards

States represent the underlying environmental situations the agent can find itself in. An example of this, in the game of chess, would be a certain board configuration. The agent makes use of the state s_t to decide which action a_t to take that will maximise the cumulative reward going forwards $\sum_{t=0}^{T} r_t$. An agent might not have direct access to the environments underlying states, thereby receiving some observation o_t, representing s_t. In simple environments, the observation is identical to the state $o_t = s_t$, but this is not always the case. An example of this is agents learning how to play games directly from pixels. The pixels themselves are not the state of the environment, but they form a representation of the state. The set of all possible states the environment has to offer is termed as the state space $\mathcal{S}$.

Actions are the way the agents interact with the environment. Each state inherently has a set of possible actions for the agent to take. These actions can be discrete, such as a list of cardinal directions to move in, or continuous, such as the number of degrees to rotate a steering wheel. The set of all possible actions an environment has to offer is termed the action space $\mathcal{A}$.

Rewards are the numerical signal an agent is trying to maximise. Depending on the environment and goal of the agent, the reward function is created. The reward function is the fundamental driving force to teach an agent what is good or bad to do in a situation. The reward function depends on the current state s_t and current action a_t. Often the reward function is deemed the most important modelling factor of the reinforcement learning problem because if the reward signals do not truly capture the goal of the task, the agent will not learn correctly or at all. David Silver, one of the leading pioneers of modern reinforcement learning research, posits that the objective of maximising reward is enough to create behaviour that exhibits most attributes of intelligence found in nature [56]. This idea further enforces the importance of a correctly structured reward function.

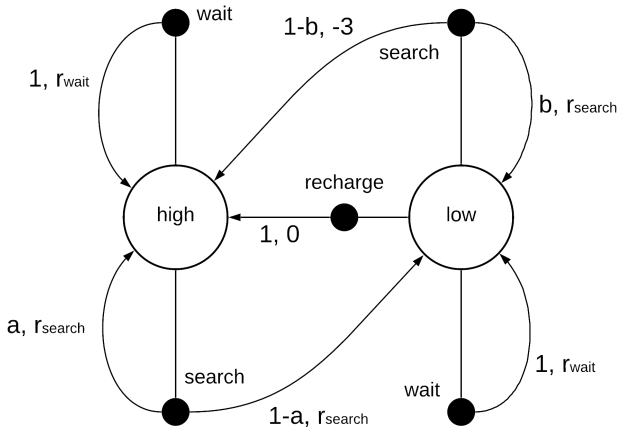

Figure 2: Example of a MDP [59]

1.2.4 Markov Decision Processes

Markov decision processes (MDP) are used to model an environment in a reinforcement learning problem. Formally an MDP is represented by the tuple ($\mathcal{S}$, $\mathcal{A}$, $\mathcal{P}$, $\mathcal{R}$, γ) where:

- $\mathcal{S}$ represents the set of all possible states.
- $\mathcal{A}$ represents the set of all possible actions.
- $\mathcal{P}$ represents the state transition probability function which gives the probability of outcome states $s_{t+1} \in \mathcal{S}$ given a current state $s_t \in \mathcal{S}$ and current action $a_t \in \mathcal{A}$.
- $\mathcal{R}$ represents the reward function which gives the numerical reward depending on the current state $s_t \in \mathcal{S}$ and current action $a_t \in \mathcal{A}$.
- γ represents the discount factor that decides the weighting of importance given to immediate/future rewards.

MDPs (or POMDPs) are used to describe the RL agents environment formally. A POMDP (Partially Observable Markov Decision Process) is an MDP whereby the agent cannot observe the MDP's true underlying state. This results in an MDP where instead of states, there are observations and an associated observation transition probability function which allows for a degree of inference about the underlying state. Formally, POMDPs are represented by the tuple ($\mathcal{S}$, $\mathcal{A}$, $\mathcal{P}$, $\mathcal{R}$, Ω, $\mathcal{O}$, γ) which is the same as an MDP with the addition of Ω which represents the set of observations and $\mathcal{O}$ which represents the observation probability function.

Since the reinforcement learning problem is mathematically idealised by Markov Decision Processes, one can construct precise theoretical statements about these problems using them [59]. This is what allows for many of the theoretical guarantees of tabular reinforcement learning methods.

Figure 2 shows an example of an MDP. The circles represent the states, and the arrows represent transitions with specific probabilities. As can be seen, states can offer more than one potential transition. An agent will transition from state to state with a certain probability depending on the action/transition it decides to take. Figure 1 shows the agent's interaction with the environment in an MDP. The agent and environment (MDP) interact at each step in a sequence of discrete time steps. At every step the agent receives an observation o_t of the state $s_t \in \mathcal{S}$, and selects an action $a_t \in \mathcal{A}$ based on this observation according to it's policy π. Once the agent has executed its action, one time-step later, the agent will receive a reward $r_t \in \mathcal{R} \subset \mathbb{R}$ due to its action and the state it ends up in. This reward can be positive or negative, depending on the outcome. The agent now finds itself in a new state $s_{t+1} \in \mathcal{S}$ and the cycle repeats until a terminal state is reached.

It is important to note that MDPs have the Markov property meaning that the future states are independent of the past states given the present state, i.e $P[s_{t+1}|s_1, s_2, ..., s_t, a_1, a_2, ..., a_t] = P[s_{t+1}|s_t, a_t]$. Intuitively this means that every state must include the necessary information about all aspects of the

past agent–environment interaction so that actions can be accurately decided solely based upon the current state the agent is in. Many modern problems do not obey the Markov Property and require the agent to have some form of memory, but this is elaborated in future sections.

1.2.5 Policies and Value Functions

Agent's use a function, termed a **policy**, to decide their actions at every time-step. Intuitively, the policy can be seen as the agent's brain. A policy π is defined as a mapping of environment states $\mathcal{S}$ to an action [59]. The policy can be constructed as a stochastic or deterministic function. In the deterministic case, the function $\pi(s_t)$ outputs a single action a_t that the agent will take. If stochastic, $\pi(s_t)$ will output a probability distribution over all possible actions $a_t \in \mathcal{A}$ whereby an agent will sample an action a_t from this distribution.

In the process of finding the optimal policy π^*, a large proportion of reinforcement learning algorithms estimate the value function v. The value function is a function that receives a state (s_t) or state-action pair (s_t, a_t) as input to estimate the expected cumulative reward to be received going forwards (*value*). Since the cumulative reward is dependant on future actions in future states, the value function is defined with respect to the policy the agent is following. The expected cumulative reward G_t can be defined as follows:

$$G_t = \sum_{k=t}^{T} \gamma^{k-t} r_k$$

where T is the terminal time step and γ discount factor (the weight of importance given to immediate and future rewards). Formally the value function for any given policy is defined as:

$$v_\pi(s) = \mathbb{E}[G_t | s_t = s], \text{ for all } s \in \mathcal{S}$$

The bellman equation for the value function decomposes the function into two parts, the immediate reward and the discounted value of successor states. This can be formulated as :

$$v_\pi(s) = \mathbb{E}[r_t + \gamma * v(s_{t+1}) | s_t = s]$$

The bellman equation illustrates an important recursive property used for most methods that involve estimating a value function.

1.2.6 Prediction and Control

In reinforcement learning, there are two different goals: Prediction - where the agent's task is to evaluate how well a given policy performs. An example of this is to estimate the value function of an MDP given some policy.

The other goal, which is usually the focus of many, is control - where the task of the agent is to find the optimal policy to some MDP [59].

Often, in the pursuit of optimal control, prediction is used to improve the current policy. An agent uses its experience to learn and estimate the value function, after which it then improves its policy by acting greedily. With its new policy, the process of estimating the new value function starts again, thereby creating a cycle of policy improvement and evaluation, ideally converging on the optimal policy and value function over time. This cycle of policy evaluation (prediction) and policy improvement (control) is termed *Generalised Policy Improvement* (see figure 3). Almost all reinforcement learning methods can be described as a form of *Generalised Policy Improvement*.

1.2.7 Model Free

Model-free methods are those in which the agent has no internal representation of the MDP/Environment. The agent has no knowledge of the MDP transition or reward dynamics and cannot use this information in learning. Model-free agents do not have the ability to think and plan about how their environments might change in response to their actions. Since models have to be accurate to give utility, model-free methods often have the advantage in complex environments, which is why there has been much success with these methods in the past. Model-free methods are also an essential building block for model-based methods, which are becoming more popular in recent years.

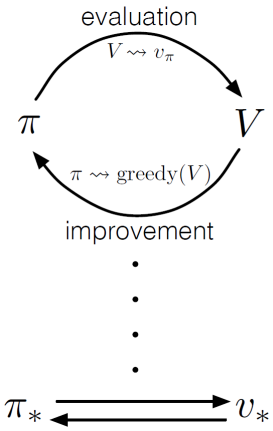

Figure 3: *Generalised Policy Improvement* [59]

1.2.8 Model-Based

Model-based methods are methods in which the environment dynamics such as reward and state transition functions are learned by (or given to) the agent and modelled into some representation that the agent can use to choose the optimal action. Models can help agents learn faster and find better policies in fewer interactions with the environments (i.e. it is sample efficient) [33]. Models can also aid agents with the ability to predict the future, and this can be very beneficial in certain situations [58]. Model-based methods are becoming more popular in recent years with the creation of techniques capable of learning complex models that can be reasoned about e.g. MuZero [49].

1.2.9 Monte Carlo Methods

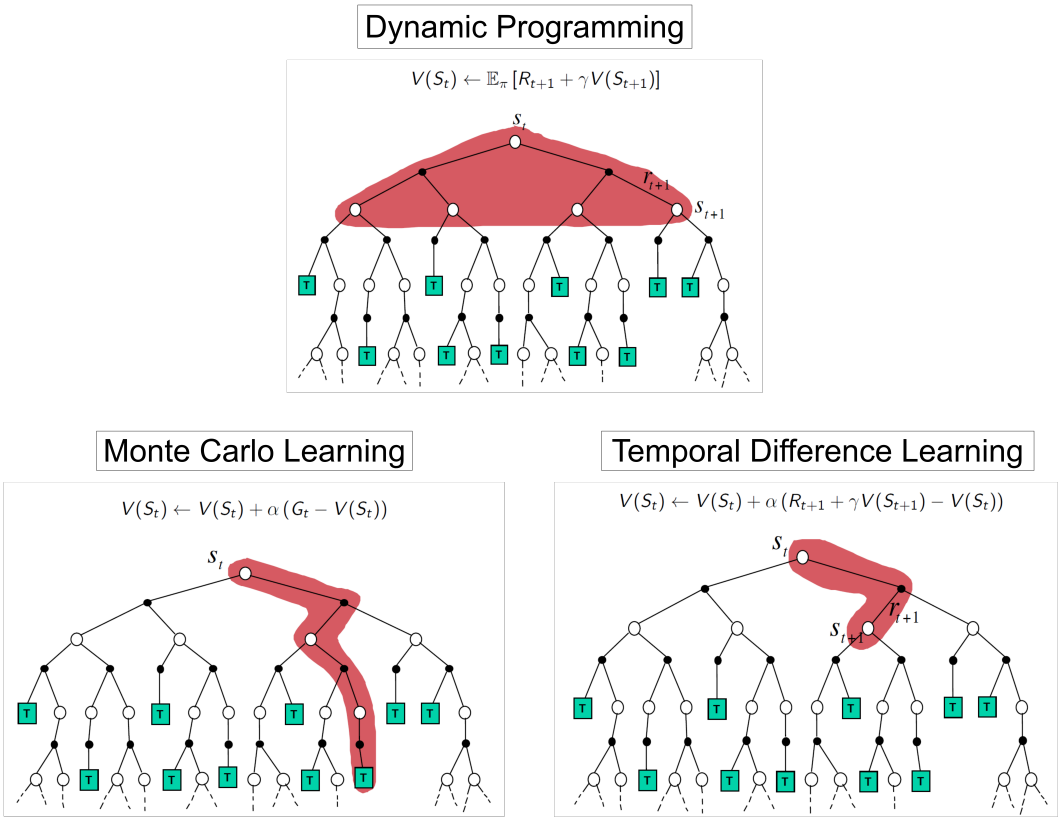

Figure 4: Dynamic Programming, Monte Carlo, and Temporal Difference Backup illustrated

Monte Carlo methods aim to solve the reinforcement learning problem by directly learning from experience and averaging sample returns. Monte Carlo methods have the assumption that all episodes will terminate and that experience is divided into episodes (sequence of time steps starting in a start state and ending in a terminal state). The agent simulates episodes of experience and learns directly from this experience, thereby not requiring any knowledge of how the environment functions. In the learning process, an agent generates episodes of experience $(S_0, A_0, R_0, S_1, A_1, R_1, ...S_T, A_T, R_T)$ and uses this to learn a value function or policy function. Since Monte Carlo methods have to simulate entire episodes of experience before learning can occur, these methods cannot be applied to infinite-horizon MDPs. What distinguishes Monte Carlo methods apart from classic dynamic programming and temporal difference learning is that Monte Carlo learning does not employ bootstrapping. The estimates for one state do not build upon the estimates of any other state. Thus, Monte Carlo methods have a lower bias compared to other methods, but the trade-off is a higher variance as episode trajectories can be highly different [59]. An example of a Monte Carlo update for a value function is as follows:

$$V(s_t) \leftarrow V(s_t) + \alpha[G_t - V(s_t)]$$

where G_t represents the cumulative value seen from time t onward and α represents the step size. As shown in the update, a Monte Carlo method directly uses the sampled value G_t it has observed from experience.

1.2.10 Temporal Difference Learning

Temporal difference methods are the same as Monte Carlo methods in that they directly learn from experience without the need of a model, but unlike Monte Carlo, they do not need to wait for episode termination to learn. Instead, they bootstrap from the sample trajectory [59]. TD methods make use of every individual time-step to learn. At time $t + 1$, the experience of 1 or n time steps is used to update either the agent's policy or value function. An example of a one-step TD update for the value function is as follows:

$$V(s_t) \leftarrow V(s_t) + \alpha[r_t + \gamma V(s_{t+1}) - V(s_t)]$$

This is updating the agent's current state value estimate in the direction of the successive state's value estimation plus the immediate reward it has seen. Intuitively, this is a combination of dynamic programming and Monte Carlo ideas, where the agent updates its estimates based on other learned estimates and does this from raw experience. Since TD methods bootstrap, the variance between state updates is much lower than its Monte Carlo counterparts. However, this causes an increase in bias as values are updated based on other estimates instead of experienced trajectories.

2 Foundational Methods

2.1 Value-based

Value-based methods are those in which the agent tries to learn the value function of the MDP it is situated in. The value function is highly beneficial as it gives the agent information on what states it should transition to. The agent can use the value function for action selection simply by acting greedily and choosing the action that takes it to the state with the highest value. Most value-based methods use the greedy policy (specifically epsilon-greedy)[59] as it is simple to implement and is effective. The epsilon-greedy policy is precisely like the greedy policy, except every time-step, the agent has a slight chance to take a completely random action. The epsilon value decides how large this chance is. The reason for not always acting greedily is to allow the agent to explore new states and actions it has not seen before.

Practically, most value-based methods try to learn the value of action-state pairs, otherwise known as the Q-value. The idea behind this is that if the agent does not know the dynamics of the environment, it does not know which action will take it to the desired state. Hence, the solution is to create a function that estimates the total expected cumulative reward if an agent takes a specific action in a specific state:

$$Q(s, a) = \mathbb{E}[G_t | s_t = s, a_t = a], \text{ for all } s \in \mathcal{S} \text{ and } a \in \mathcal{A}$$

2.1.1 Q-Learning

Q-learning [72] is regarded as one of the early breakthroughs in reinforcement learning [59]. The Q-learning control algorithm is a way for an agent to learn the optimal action-state value function

Q^* directly instead of repeatedly performing policy evaluation and iteration. Q-learning achieves this by keeping a table of each action-state pair and its associated Q-value estimate. As the agent interacts with the environment, it iteratively updates its estimates whilst acting greedily or ϵ-greedily. Q-learning makes use of TD-learning to update the Q-function. An example of this Q-learning update using one-step TD-learning is:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha\big[r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)\big]$$

Where α is the learning rate, and γ is the discount factor. In Q-learning, the optimal state-action value function Q^* being learned is independent of the behavioural policy that the agent is following. Due to this, an agent can use a more exploration-focused policy whilst still learning how to perform optimal control. Although Watkins et al. [72] has shown the convergence proofs illustrating that Q-learning can effectively solve MDP's, this is only in the tabular setting.

Algorithm 1: Tabular Q-learning

Initialise Q(S,A) to arbitrary values.
for *each episode:* **do**
  Initialise S
  for *Each step in episode* **do**
    Choose A from S using policy derived from Q
    Take action A and observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_{A'} Q(S', A') - Q(S, A)]$
    $S \leftarrow S'$
    until S is terminal
  end
end

2.1.2 DQN

When modelled into MDPs, most modern problems have vast state spaces (potentially continuous state space). Solving these MDPs can prove challenging for tabular setting algorithms, such as Q-learning, as computers have limited storage. The difficulty of dealing with large state spaces becomes worse when learning the action-state pair value estimates as the number of states is multiplied by the number of actions. This high dimensionality makes traditional Q-learning for larger problems computationally infeasible as action-state pairs' visitation count must tend to infinity for the Q-function to converge. The solution to this problem is function approximation. With modern advancements in deep learning, one type of function approximation used heavily in RL is neural networks. Although any differentiable function approximation can be used, neural networks are incredibly versatile and high performing. Deep Q-Networks (DQNs) [39] have been shown, with a few additions such as experience replay buffer [38] and target networks, to be able to solve large dimensional MDPs, such as Atari games, effectively. Initially, Q-learning, using neural networks, could not solve even elementary problems due to three main issues. Firstly, neural networks require non-correlated data, which is not the case in reinforcement learning as data collected is sequential and temporally related. Secondly, the neural Q-network is highly sensitive to changes which makes training unstable. Lastly, the Q-targets when bootstrapping are highly correlated to the Q-values being predicted. These issues effectively prevented neural networks and the advancements of deep learning from being adopted in RL. However, with the additions added by Mnih et al., the DQN has become one of the most widely used and expanded upon algorithms.

The DQN algorithm functions like traditional tabular Q-learning, except a neural network, instead of a table, represents the Q-function that estimates the Q-value. To train the neural network, as the agent interacts with the environment, the *online* Q-network $Q_\theta(s_t, a_t)$ predicts the action-state values and updates its weights θ according to how close its prediction is to the target Q-value. To calculate the target Y_t, we bootstrap as follows:

$$Y_t = r_t + \gamma \max_a Q_\theta(s_{t+1}, a)$$

By using this as the target value, we can then use MSE loss:

$$\mathcal{L} = \frac{1}{T}\sum_{t=1}^{T}(Q_\theta(s_t, a_t) - Y_t)^2$$

To calculate the Q-network parameter gradients and perform gradient descent.

These changes alone would not allow the DQN agent to solve complex high-dimensional MDPs. Two critical additions, introduced by Mnih et. al, are needed:

Experience Replay Buffer:

The agent uses the experience replay buffer to store information about the interactions it has experienced over multiple episodes. Usually these interactions are stored as tuples in the form (s_t, a_t, r_t, s_{t+1}). The reason for collecting and storing experience over multiple episodes is twofold. Firstly, it increases sample efficiency by allowing the agent to use more experience than a single episode when updating the neural network weights. Secondly, it allows for the experience to be shuffled when used, thereby decorrelating the data the neural network uses to train. Due to the replay buffer having a limited fixed size, when new experience is stored, the oldest experience is deleted. This first-in-first-out replacement scheme helps the agent have more relevant experience to use in training. When the agent samples experience from the replay buffer, the data is randomly shuffled into a batch. These batches are ultimately what the DQN is trained on, similarly to supervised learning.

Target Network:

An additional neural network, the target network $\hat{Q}_{\theta^-}$, is created and used to produce the Q-targets Y_t instead of the *online* network Q_θ. The target network decouples the Q-targets from the Q-values produced by the *online* network. The weights θ^- of the target network are fixed as this network is not trained. Instead, the *online* Q-networks weights θ are copied over to the target network after a fixed number of agent time-steps or training updates so that the Q-targets are more stationary as they are not changing every update. This stationarity stabilises training significantly and assists in the general performance of the DQN algorithm. The introduction of the target network changes the Q-target Y_t to be:

$$Y_t = r_t + \gamma \max_a \hat{Q}_{\theta^-}(s_{t+1}, a)$$

As successful as the DQN algorithm is, there are still limitations to the approach. It was shown that non-linear function approximation, such as neural networks, can cause the Q-networks to diverge [64]. Q-learning in general also tends to overestimate the actual Q-value, which can eventually lead to sub-optimal policies [62]. Even though convergence is not theoretically guaranteed when using neural network function approximation and overestimation still occurs, practically, we see successful results in applying the DQN algorithm to specific problems [39, 22, 75].

2.1.3 DDQN

A well-known issue with Q-learning is the overestimation of Q-values. Generally, if overestimation does occur, it is unknown whether or not this overestimation will negatively impact an agent's performance. Since in Q-learning, the greedy policy (or ϵ-greedy) is used, if all values are uniformly higher, the agents' action preferences would remain the same, thereby having no impact on the quality of policy learnt. The problem lies in a non-uniform overestimation whereby Q-values can initially favour non-optimal states, negatively impacting exploration. With a lack of exploration caused by non-uniform overestimation, sub-optimal policies can be learnt [63].

The cause of this overestimation in Q-learning is due to the max operator. When calculating the target Q-values, the maximisation operation favours initial overoptimistic estimates thereby potentially introducing a source of bias which tends to keep action selection favouring these initially optimistic states and potentially trap the agent in choosing sub-optimal actions. Hasselt et. al. [65] show that Q-learning's overestimation extends to the deep neural network setting and how this can impact the learning of policies. Additionally, Hasselt et. al. generalise and extend the tabular Double Q-learning algorithm [26], a proposed solution to Q-learning overestimation, to the deep neural network setting and show how this improves the DQN algorithm to achieve new state of the art results on the Atari suite at the time of publication.

Algorithm 2: DQN

Initialise Replay Memory $\mathcal{D}$ to capacity $\mathcal{N}$
Initialise action-value function Q with random weights θ
Initialise target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
for *Episode 1...M* **do**
  Initialise s_1
  for *time step* $t = 1...T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = argmax_a Q_\theta(s_t, a)$
    Execute action a_t in environment and observe s_{t+1} and r_t
    Store transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{D}$
    Sample random minibatch of transitions (s_j, a_j, r_j, s_{j+1}) from $\mathcal{D}$
    Set $Y_j = \begin{cases} \mathrm{r}_j & \text{if episode terminates at step } j+1 \\ \mathrm{r}_j + \gamma max_{a'} \hat{Q}_{\theta^-}(s_{j+1}, a') & \text{otherwise} \end{cases}$
    Perform gradient descent step on $(Y_j - Q_\theta(s_j, a_j))^2$ with respect to the network parameters θ
    Every C steps, set $\theta^- = \theta$
  end
end

The Tabular Double Q-learning algorithm reduced overestimation by decoupling the action selection and evaluation performed in the max operation. This algorithm used two Q-functions, whereby action evaluation was performed by the one and action selection by the other. Double DQN (DDQN), the extension of Double Q-learning to DQN, aims to reduce overestimations in the same manner. In the DQN algorithm, the target network is already created and fulfils a similar role as a second Q-function in the original tabular algorithm. Although using the target network does not provide a complete decoupling of action selection and evaluation, it prevents the need to create a third neural network and works well in practice. The DDQN algorithm proposes a simple change. The target network $\hat{Q}_{\theta^-}$ is still used to estimate the Q-value of the target but the action used in the target Q-value estimation is decided by the online network Q_θ. Hence, the target values are calculated as follows:

$$Y_t = r_t + \gamma \hat{Q}_{\theta^-}(s_{t+1}, argmax_a Q_\theta(s_{t+1}, a))$$

With this simple change in the update target, the overestimation is reduced, and higher-quality policies are seemingly learnt.

2.1.4 Dueling DQN

Deep learning greatly impacted reinforcement learning by allowing algorithms to scale up to solve larger and more complex problems. The success of algorithms such as DQN and DDQN have illustrated the potential of deep learning coupled with reinforcement learning. Despite these successes, prior to Dueling DQN [71], most research iterations and improvements focused on algorithmic changes and exclusively used standard neural network architectures. Dueling Deep Q-Networks is a proposed architectural improvement to the DQN algorithm that takes a new approach to estimating Q-values and can be used with any existing algorithm that uses a Q-network.

In understanding the change in architecture, it is important to know that the Q-value can be decomposed into two separate parts:

1. The advantage function - The advantage/value of the taking an action in a state compared to the other possible actions :
$$A(s, a) = Q(s, a) - V(s)$$

2. The state value function - The value of the state i.e the expected cumulative reward to be received following a given policy :
$$V(s) = \sum_{a \in \mathcal{A}} Q(s, a)$$

The dueling network architecture separates the Q-value estimation into two streams - one to represent the advantage function $A(s, a)$ and the other to represent the value function $V(s)$. These two streams are ultimately combined using an aggregation layer to produce the final Q-values. The intuitive reason for this decoupling of estimators is that the Dueling DQN can learn, separately, which states are or are not valuable without the need to learn the effect of each action at each state. The authors propose that there is no point in learning/calculating action values if the entire state value is terrible and can be avoided, e.g. why calculate all actions at one state when all these actions lead to death? This decoupling is also helpful in states where actions have no impact on the environment in any critical or relevant way. In this situation, there is no need to calculate and learn the value for each action as the choice has no importance.

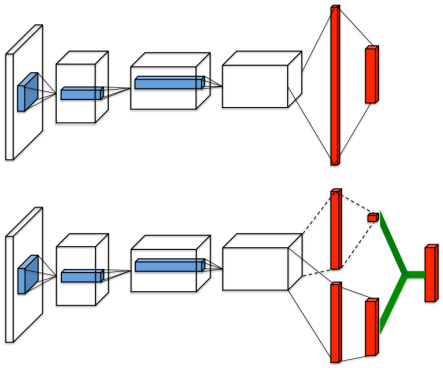

Figure 5: The standard single stream Q-network (top) and the dueling Q-network (bottom). The dueling network has two streams to separately estimate (scalar) state-value and the advantages for each action; Both networks output Q-values for each action.

After the separation of the advantage and value functions, one might think the aggregation layer is as simple as:

$$Q(s, a) = A(s, a) + V(s)$$

However, the issue with this is one of identifiability. Given a Q-value calculated in this way, the advantage $A(s, a)$ and value $V(s)$ cannot be recovered, which is a problem for backpropagation. To solve this, the authors propose forcing the advantage function to have zero advantage at the chosen action by using the following formula:

$$Q(s, a) = V(s) + (A(s, a) - \max_{a' \in |\mathcal{A}|} A(s, a'))$$

An alternative method is also given that replaces the max operator with an average:

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

This replacement loses the original semantics of V and A since it puts them off target by a constant - but this increases the stability of the optimisation since advantages only need to change as fast as the mean instead of having to compensate any change to the advantage of the optimal action. This is ultimately what the authors used in their paper.

Dueling DQN was shown to decrease the time taken to converge to optimal policies and, at the time, set a new state of the art result on the Atari suite.

2.1.5 Prioritised Experience Replay Memory

In the original implementation of DQN, the experience buffer is used to store the agents' experience for training. The buffer's primary purpose is to break the temporal correlations by mixing recent and past experience for the agent to use in its updates. It also provides the benefit of allowing experiences to be used more than once, thereby enhancing learning and sample efficiency. An issue with the traditional experience replay buffer is that all experience is treated as equally important. Due to this, an agent can potentially only make use of trivial or redundant transitions/experience in updates, which can significantly slow down or inhibit learning in certain environments. In 2016, Deepmind

investigated the effect of allowing this buffer of experience to prioritise the data that the agent is trained on and created the Prioritised Experience Replay Memory [45].

Temporal Difference Prioritisation:

To prioritise experience, some criterion with which to judge the value needs to be used. Ideally, one would measure the amount that could be learnt directly from a specific transition; however, this metric is not accessible. As a proxy for this ideal criterion, one could use the magnitude of the temporal difference error δ as an approximation. The temporal difference error essentially measures an agents' disbelief or surprise in a transition. This error can be used as a measure of the *amount* of new information to be learned. Deepmind proposed the use of the calculated TD-error to prioritise the experience in the replay buffer. This algorithm, referred to as Greedy TD-error prioritisation, stores the most recently calculated TD-error inside the transition tuple $(s_t, a_t, r_t, s_{t+1}, \delta_t)$ where:

$$\delta_t = r_t + \gamma max_a Q(s_{t+1}, a) - Q(s_t, a_t)$$

When an agent samples from the replay buffer, transitions that contain larger TD-errors are prioritised. Additionally, new transitions are given a maximum priority to ensure every transition is sampled at least once. When experience is sampled from the replay buffer, the TD-errors of the sampled transitions are updated before being reinserted into the buffer after an agent update. This way, as the agent learns, more relevant experience will be prioritised. The Greedy TD-error approach showed significant results; however, several issues presented themselves. To avoid computationally expensive sweeps over the replay buffer, temporal difference errors are only updated when experience is sampled. This means that any transitions with a small TD error on their first visit may never be sampled again and ultimately never seen again due to old data being removed as new data enters. This method is also susceptible to noise spikes that can be made worse due to bootstrapping as potentially noisy data can be reused consistently. Lastly, this prioritisation tends to focus on a small subset of the data. Since errors decrease slowly, experiences that initially had high TD-errors might get sampled frequently. Due to this, the diversity of data sampled from the replay buffer is small, proving to be detrimental to generalisation and cause overfitting.

Stochastic Prioritisation:

To solve the greedy algorithm's issues, Deepmind also introduced a stochastic sampling method to interpolate between a greedy and uniform sampling approach. This method ensures a transition tuple's probability of being sampled is proportional to its priority but is always non-zero. This means even the lowest priority transitions have a chance of being sampled from the replay buffer. A transition tuple's sampling probability is assigned as follows:

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$$

where $p_i > 0$ is the priority of transition i. The parameter α decides upon how much prioritisation is used with $\alpha = 0$ eliminating prioritisation and giving uniform sampling.

Even with stochastic prioritisation, not all issues were solved. It was seen that the use of any prioritisation scheme introduced a bias as the estimation of an expectation requires updates corresponding to the same distribution of that expectation. The distribution of updates changes if specific experiences are valued over others. To correct this bias, Deepmind show that one can use importance-sampling weights:

$$w_i = \left(\frac{1}{N} \cdot \frac{1}{P(i)}\right)^\beta$$

where the parameter β controls the strength of bias correction. This weight can be used in the Q-learning updates by using the target $w_i \delta_i$ instead of just δ_i.

Deepmind show that the use of a prioritised experience replay buffer helps agents converge to optimal policies faster. Additionally, in the original paper, state of the art results on the Atari suite (at the time) were achieved through prioritised memory.

2.2 Policy-based

Policy-based methods are methods in which the agent tries to learn the policy function directly. These methods offer several advantages over their value-based counterparts. Firstly, policy-based methods are able to learn stochastic and deterministic policies, whereas value-based methods are essentially

deterministic but always have a slight ϵ probability of selecting an action at random. Secondly, policy-based methods can also effectively learn continuous or high dimensional action spaces and have better convergence properties. Lastly, in certain environments, the policy function might be a lot simpler to approximate thereby making it easier for the agent to learn optimal control. Policy gradient methods search for a local maximum in the objective function $J(\theta)$ by gradient ascent:

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$$

where α is the step-size and:

$$J(\theta) = \mathbb{E}\big[\sum_{t=0}^{T} r_t | \pi_\theta\big]$$

2.2.1 REINFORCE

REINFORCE [73] is a Monte Carlo method that updates the policy function's parameters directly using the policy gradient with respect to the objective function $J(\theta) = \mathbb{E}\big[\sum_{t=0}^{T} r_t | \pi_\theta\big]$. Sutton et al. [60] show that the gradient of the objective function to maximise expected total cumulative reward is:

$$\nabla_\theta J(\theta) \propto \mathbb{E}\Big[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) Q_\pi(s_t, a_t)\Big]$$

By using the Monte Carlo estimate $G_t = \sum_{k=t}^{T} \gamma^{k-t} r_k$ in place of the Q value, it is relatively easily calculate the gradient and update the policy's parameters directly using the REINFORCE update:

$$\theta_{t+1} = \theta_t + \alpha * G_t * \nabla_\theta \log \pi_\theta(a_t|s_t)$$

As per the shortcomings of Monte Carlo methods, REINFORCE suffers from high variance with a noisy gradient estimate and no clear credit assignment to positive or negative actions throughout the episode [59]. An easy way to improve REINFORCE is to reduce the variance of the empirical returns G_t by subtracting a baseline function $b(s)$ in the policy gradient. The baseline function can be any function that does not rely on the agents' actions as a parameter. A popular option for the baseline function is the state value function $V(s_t)$. The new value after subtracting the baseline from G_t is the advantage $A(s_t) = G_t - V(s_t)$. This changes the REINFORCE update to:

$$\theta_{t+1} = \theta_t + \alpha * A(s_t) * \nabla_\theta \log \pi_\theta(a_t|s_t)$$

This requires the REINFORCE agent to learn the value function alongside the policy and can introduce a bias as the cost of lowering variance. The value function is jointly learned by minimising the MSE loss:

$$MSE = \sum_{t=0}^{T-1} (G_t - V(s_t))^2$$

Algorithm 3: REINFORCE with Baseline

Input : a differentiable policy parameterisation $\pi(a|s, \theta)$
Input : a differentiable state-value function parameterisation $v(s|\phi)$
Algorithm parameters: step sizes $\alpha^\theta > 0, \alpha^\phi > 0$
Initialise policy parameter $\theta \in \mathbb{R}^{d'}$ and state-value weights $\phi \in \mathbb{R}^d$
for *each episode:* **do**
  Generate an episode $s_0, a_0, r_0, ...s_T, a_T, r_T$
  for *Each step t in episode* **do**
    $G \leftarrow \sum_{k=t}^{T} r_k$
    $\delta \leftarrow G - v(s_t|\phi)$
    $\phi \leftarrow \phi + \alpha^\phi [\gamma^t \nabla v(s_t|\phi)]$
    $\theta \leftarrow \theta + \alpha^\theta [\gamma^t \nabla \ln \pi(a_t|s_t, \theta)]$
  end
end

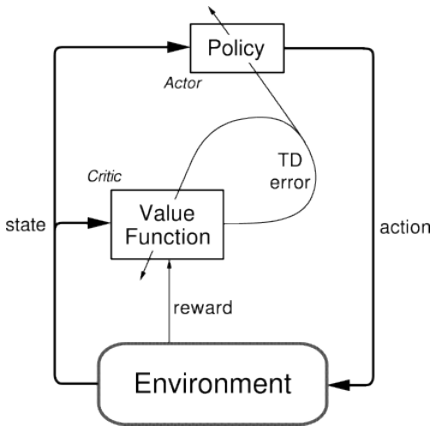

Figure 6: Actor-Critic Architecture [59]

2.3 Actor-Critic

Actor-Critic methods [36] are a combination of policy-based and value-based methods to combine each method's strong points. Actor-Critic methods consist of two models: The critic which estimates some value function e.g($Q(S_t, A_t)$) (intuitively, it tells the actor how good the action was) and the actor which decides which action to take and updates the policy parameters in the direction suggested by the critic. REINFORCE with baseline is similar to an advantage actor-critic. The difference between the two is that the actor-critic makes use of $r_t + V(s_{t+1})$ instead of the Monte Carlo return G_t. This change allows it to bootstrap experience and learn in non-terminating environments but increases the learning bias.

2.3.1 PPO

Due to the use of reinforcement learning with neural network function approximators, a variety of problems have become significantly apparent. Popular methods such as DQN, REINFORCE, and TRPO [50] each have unique issues that can cause the inability to feasibly solve certain real-life problems. DQN can often fail on simple problems [19] and is generally not well understood. REINFORCE and other vanilla policy-gradient methods have low sample efficiency and lack robustness to hyperparameters. Lastly, Trust Region Policy Optimization (TRPO) which is often seen as the predecessor to PPO, tries to tackle similar issues as PPO but is a much more complicated method that does not support use with a variety of architectures, i.e. those that include noise or parameter sharing. Another issue that is common in reinforcement learning with function approximation is the case of sudden sharp decreases in policy performance known as *catastrophic forgetting*. This is due to large gradient steps that occur in gradient-steep portions of the parameter space. Since the loss surface can be very unique and non-convex, too large a step might increase loss and worsen the learnt parameters, ultimately seeming as if the agent has *forgotten* how to act in the environment. In the pursuit of creating an algorithm to combat all these issues, PPO was proposed. Beyond solving the problems above, PPO also aims to take the largest gradient step possible without causing *catastrophic forgetting*.

Proximal Policy Optimisation (PPO) [52] is an actor-critic method that has two primary variants, PPO-Penalty and PPO-Clip. Both variants aim to constrain the magnitude of gradient steps to increase stability and prevent performance collapse.

PPO-Penalty:

PPO-Penalty is similar to TRPO in that it aims to approximately solve a KL-constrained update. The difference is that instead of making the KL-divergence a hard constraint in the policy objective function, it penalises it and automatically adjusts this penalty coefficient over time to ensure appropriate scaling. PPO-Penalty, experimentally shown in the original paper [52], performs worse than PPO-Clip but is seen as an important baseline upon which to compare the Clip variant.

PPO-Clip: PPO-Clip does not make use of the KL-divergence and does not use a constraint term at all. Not using KL-divergence or a constraint term simplifies the objective function and implementation in general. The PPO-Clip algorithm relies on a specific clipping mechanism in the objective function

to inhibit extremely large and potentially performance harming gradient steps. PPO-Clip makes use of the following update:

$$\theta_{k+1} = \arg\max_{\theta} \mathbb{E}_{s,a\sim\pi_{\theta_k}}\left[\mathcal{L}^{CLIP}(s,a,\theta^{(k)},\theta)\right]$$

This essentially means the PPO-Clip objective is maximised with respect to the policy parameters θ. An important observation that needs to be noted is that in each optimisation update the fixed parameter vector $\theta^{(k)}$ from the previous (k^{th}) update is used. This is to allow the Clip objective to make use of the previous policy and prevent too large gradient steps. The Clip objective is defined as:

$$\mathcal{L}^{CLIP}(s,a,\theta^{(k)},\theta) = \min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}A^{\pi_{\theta_k}}(s,a),\ g(\epsilon, A^{\pi_{\theta_k}}(s,a))\right)$$

where

$$g(\epsilon, A) = \begin{cases}(1+\epsilon)A & A \geq 0\\ (1-\epsilon)A & A < 0\end{cases}$$

$A(s,a)$ represents the advantage function $Q(s,a) - V(s)$ or any other advantage estimation. The objective function aims to create a ceiling and floor to the updates that can occur. The intuition behind this objective function is quite simple. When the advantage estimate is positive, the objective function reduces to:

$$\mathcal{L}^{CLIP}(s,a,\theta^{(k)},\theta) = \min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, (1+\epsilon)\right)A^{\pi_{\theta_k}}(s,a)$$

This means that the objective function value will increase if the action selected becomes more likely, i.e. $\pi_\theta(a|s)$ increases, but the min term limits the increase where as soon as $\pi_\theta(a|s) > (1+\epsilon)\pi_{\theta_k}(a|s)$ the objective function cannot increase further than the ceiling of $(1+\epsilon)A^{\pi_{\theta_k}}(a|s)$. This means that the new policy experiences no benefit to differ more from the old policy.

When the advantage estimate is negative, the objective function reduces to:

$$\mathcal{L}^{CLIP}(s,a,\theta^{(k)},\theta) = \max\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, (1-\epsilon)\right)A^{\pi_{\theta_k}}(s,a)$$

This means the objective function value will increase if the action selected becomes less likely, i.e. $\pi_\theta(a|s)$ decreases, but the max term limits the increase where as soon as $\pi_\theta(a|s) < (1-\epsilon)\pi_{\theta_k}(a|s)$ the objective function cannot increase further than the ceiling of $(1-\epsilon)A^{\pi_{\theta_k}}(a|s)$. This creates the same effect as last time where the new policy experiences no benefit to differ more from the old policy.

The ϵ hyperparameter allows one to control how far policies can change whilst improving the objective function. The clipping mechanism is thus seen as a regulariser to prevent the policy to change dramatically. Although PPO-Clip aims at preventing dramatic gradient steps, it is still possible for catastrophic forgetting to occur, albeit much less likely. Due to this, implementation-specific changes can be made to prevent this. An example of a change is simply early stopping, whereby if the mean KL-divergence of the new policy increases beyond a set threshold, gradient steps are stopped.

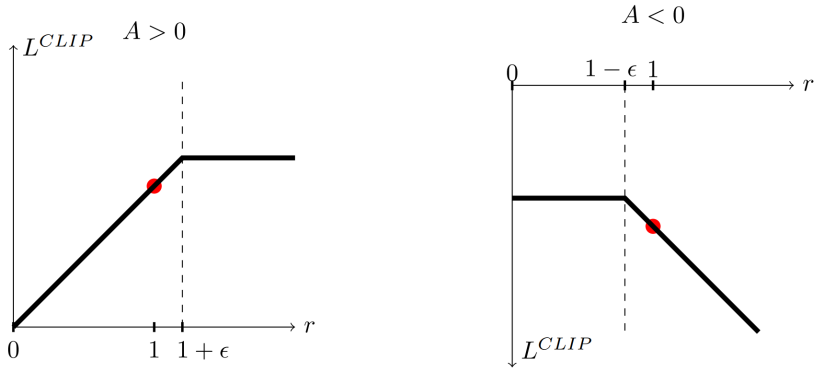

Figure 7: Plots showing the reduced function when the advantage estimate is positive or negative. The red circle shows the starting points for the optimization i.e when $r = \frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} = 1$

Algorithm 4: PPO-Clip

Input: initial policy parameters θ_0, initial value function parameters ϕ_0

for $k = 0, 1, 2, ...$ **do**

Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.

Compute rewards-to-go $\hat{R}_t$.

Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.

Update the policy by maximizing the PPO-Clip objective:

$$\theta_{k+1} = \arg\max_\theta \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t), \quad g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right),$$

typically via stochastic gradient ascent with Adam.

Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_\phi \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2,$$

typically via some gradient descent algorithm.

end

2.4 Tree Search

A less specific concept to reinforcement learning but one that has been crucial to the creation of game-AI is tree search. *Tree search* algorithms explore state or action spaces by constructing a tree structure consisting of nodes and edges. The root node represents the agent's current state. Edges represent an agent's actions, and nodes represent the successive states conditioned on the action taken. Since every state can have multiple actions that lead to multiple states, each node can have many children. Different tree search algorithms mainly differ in the way the search tree is constructed and explored.

Classical tree search algorithms such as depth-first search and breadth-first search are regarded as uninformed search algorithms as the tree is exhaustively looked at, and every node is seen. It is very rare to see these algorithms used in a game-AI context as the size of search trees can become so large as to make them computationally infeasible to exhaustively look at. Additionally, although these searches can theoretically be used in single-player games to find the optimal path to victory, they cannot be used for two-player adversarial games where another player is trying to win. This is because the path to victory is completely dependent on the opponents' action choice, which is unknown. For games such as this, e.g. chess, one needs an adversarial search.

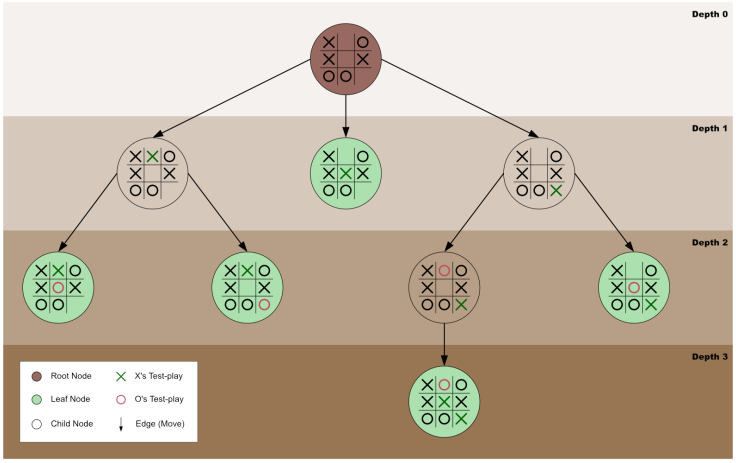

Figure 8: Example of a game tree in tic-tac-toe [57]

2.4.1 Minimax

Minimax is a basic adversarial search algorithm that has been used very successfully in perfect-information two-player games such as chess and checkers. The algorithm core loop alternates between both players where one is termed the *min* player and the other the *max* player. Each player explores all possible moves. Within the resulting states, all possible moves of the opponent are explored. This continues until every possible combination of moves is explored and constructed in the search tree until the game ends with some result, i.e. win or lose. The game's outcome gives each leaf node a value that is ultimately backed up the tree to each parent node indicating the value of the current game configuration. The backup process assumes that each player is playing optimally. From the standpoint of the *max* player, the actions selected try to maximise the score, whereas, from the standpoint of the *min* player, the actions selected try to minimise the *max* player's score. If either player does not take the optimal action, the other player will know the winning path. This algorithm will always work and find the optimal policy in a two-player adversarial game, but the problem is that in games with large state and action spaces, the process of building the search tree is infeasible. Even a simple game such as tic-tac-toe has a game tree size of 9! which is 362,880 states. This is a feasible size, but games such as chess have a game tree of approximately 10^{154} nodes. This means practically, if minimax is to be used, the search tree needs to be cut off at a given depth and use some state evaluation function to predict the outcome of the game from a leaf node.

Minimax is a fundamental tree search algorithm that has seen great success even with its computational requirements. Improvements to the algorithm such as $\alpha - \beta$ pruning [35], which removes unnecessary nodes that make the algorithm slow, and the use of tailor-made non-deterministic state evaluations functions have allowed for incredibly high performing programs to be made, such as IBM's Deep Blue [17].

2.4.2 Monte Carlo Tree Search

As stated previously, Minimax is already a solution to games without large state and action spaces, but this is seldom the case in most games played today. All games with a high branching factor, indicated by the possible actions at each state, cannot use Minimax effectively even with modern improvements. This is because the higher the branching factor, the lower the depth that is possible to compute. This means searches will be less accurate and potentially non-useful. Another problem with Minimax is that even if the tree has a relatively large depth, without an accurate and high performing state evaluation function, the search will be useless. The ancient board game of Go is a useful example of both of these problems. There is an extremely high branching factor of approximately 300, and the positional nature of Go makes it almost impossible to judge the value of a board configuration accurately. Originally, the best Go-playing programs used Minimax, but these programs barely exceeded the beginner level of play. Additionally, Minimax cannot solve imperfect information or non-deterministic games due to the inherent nature of the algorithm.

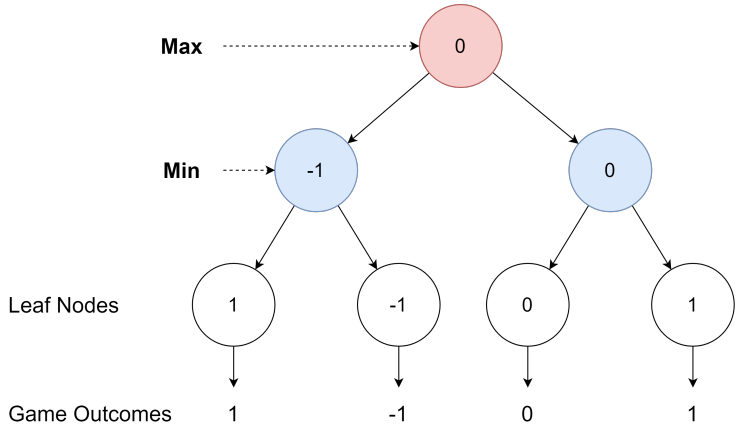

Figure 9: MiniMax Search Tree - As can be seen, the *min* player chooses the moves that minimize the *max* players scores. Of these minimum choices, the max player chooses the maximum score leading the game to a draw instead of losing.

Monte Carlo Tree Search was invented to combat Minimax's limitations and has since seemingly dominated all game-AI related searches. MCTS not only handles high branching factors and lack of sufficient state evaluation functions but can also be used for imperfect information and non-deterministic games. MCTS accomplishes this by not searching all branches of the search tree to an even depth but instead focusing on sufficiently searching the most promising branches. This makes it possible to perform deep searches even in high branching factor games. Additionally, to solve for Minimax's non-deterministic and imperfect information limitations, MCTS makes use of *rollouts* to estimate the value of a game state. A *rollout* is a random playthrough of the rest of the game from a specific game state to see the outcome, i.e. win or lose. At the start of a MCTS run, the search tree only consists of the root node representing the game's current state. The algorithm then proceeds to iteratively build the search tree with the addition and evaluation of new nodes. An incredible aspect of MCTS is that it is an **anytime** algorithm meaning it can be interrupted at any time, and inferences can be drawn. MCTS only requires the game rules and terminal state evaluation, i.e. the ability to ascertain the outcome of a game.

MCTS consists of **four** main phases:

Selection:

In the selection phase, the algorithm decides which node should be expanded and searched further. This process starts at the root and continues until a node is selected which has unexpanded children. Every time an action (node) has to be selected, a child node i is chosen so as to maximise the UCB1 formula:

$$UCB1 = \bar{X}_i + 2C_p\sqrt{\frac{2\ln N}{N_i}}$$

where $\bar{X}_i$ is the average reward of all nodes beneath the node being evaluated, C_p is an exploration constant, N is the number of times the parent node has been visited, and N_i is the number of times the child node i has been visited. The UCB1 formula is a popular method for action selection, although it is not the only one. Epsilon-greedy and Thompson sampling are examples of other options.

Expansion:

Once a node has been selected that has unexpanded children, one of the child nodes is chosen for expansion. This means that a **simulation** is performed using the child node as the starting state. The selection of which child node to expand is often random.

Simulation:

After a node has been selected for expansion, a simulation, i.e. rollout, is performed using the child node as the starting state. This effectively performs a playthrough of the entire rest of the game, from the selected starting state to see the game outcome of victory or defeat. This can be a completely

random playthrough or make use of some specific action selection method. The outcome is used as the reward received, i.e. +1 for winning, 0 for drawing and -1 for losing. This reward is then propagated back up the tree to the expanded node.

Backpropagation: Once an entire rollout has been performed and a reward is received, the reward is added to the total reward of the new node. This reward is further backed up to its parent and ancestor nodes, all the way to the root, to add to their total rewards.

With these four steps, the MCTS algorithm has eliminated almost all limitations of the Minimax algorithm and practically yields a relatively unbiased expectation of reward for action selection. MCTS executes as many rounds (execution of all four steps) as time will allow. The more rounds performed, the more accurate the results, which improves move selection. As soon as MCTS was invented, the strength of Go-playing systems dramatically increased, albeit still not to the level of a professional. An issue with MCTS is that even though random playthroughs can potentially be very fast, for games that have extremely long sequences of actions until termination i.e tens or thousands of steps, most simulations will either take too long or not finish at all thereby yielding no information about the nodes being expanded. This means MCTS works best in games guaranteed to terminate in a relatively small number of moves.

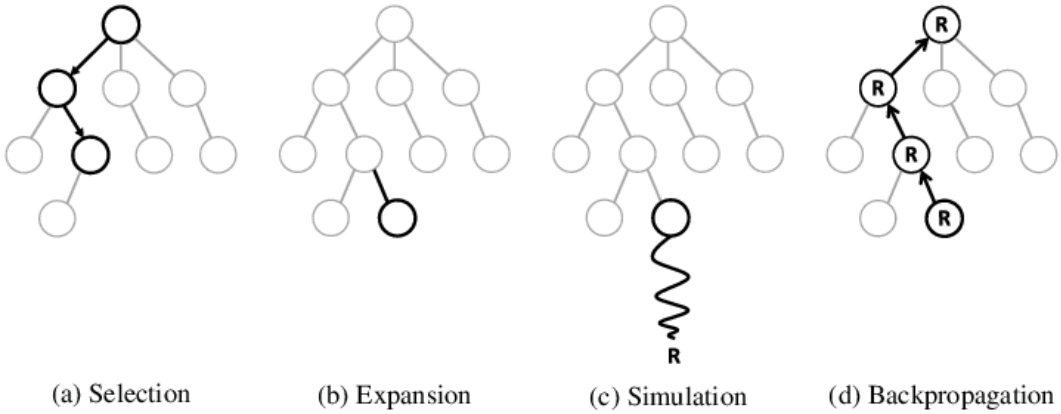

Figure 10: MCTS Phases

3 Achieving Superhuman Performance

With games being seen as a form of *Turing* test for AI, the achievements within this domain always create much excitement (or fear) around the world. As research progresses, one game after another falls prey to artificial intelligence's mastery. Although current methods are seemingly capable of playing any game, certain games prove to be very difficult for machines to surpass humans. Incredibly vast search spaces, high sensory inputs, and imperfect information are some of the challenges current AI methods struggle with. The following section describes current methods and techniques used to achieve superhuman performance in games that traditionally proved extremely difficult for AI to master.

3.1 Go

Go is a classical board game originating in China over 3000 years ago. The game is simple: there are two players using either white or black stones. Players place their stones on the 19 x 19 board every turn to capture board space or their opponents' stones. The game only terminates once all possible moves have been played upon which the number of stones and empty spaces are counted. The player that scores the highest is the winner. With these simple rules, a profoundly complex game arose, deemed an impossible task for artificial intelligence.

Go was, historically, seen as the most difficult of classical games for a computer to play. The reason for this is because Go has an enormous search space due to the number of possible board configurations being around 10^{170}, which exceeds the number of atoms in the known universe. Additionally, evaluating board positions and moves is exceedingly challenging due to the complexities of the game. These two factors make the use of existing search algorithms very challenging and arguably infeasible.

Despite decades of work, the performance of Go-playing AI has never exceeded a professionals' level of play. The creation of a Go-playing agent capable of defeating a human professional player was theorised to be years still away. However, to the world's surprise, in 2016, DeepMind released AlphaGo [53].

3.1.1 AlphaGo

AlphaGo was the first computer program to defeat a professional Go player and the first to defeat a Go world champion. At the time of creation, it was also arguably the best Go player in the world. AlphaGo, the first iteration of the Alpha algorithms, tried to capture the intuitive nature of Go with a new approach. Through a combination of neural networks, supervised learning, reinforcement learning and an advanced MCTS tree search method, computers finally managed to achieve victory in the game of Go. Due to the highly complex nature of Go, Deepmind deemed that human expertise was needed. All the neural networks that AlphaGo uses make use of expertly designed heuristic features of the Go board as input which helps instil a form of *human knowledge*.

AlphaGo consists of the following main components:

Supervised Learning policy network:

The first stage of the AlphaGo training pipeline is the creation of the Supervised Learning policy network (SL policy network) and Rollout Policy Network. The SL policy network $p_\sigma(a|s)$ is a neural network trained in a traditional supervised manner to predict expert moves/actions where s represents the input features, i.e. state. This network uses convolutional neural network layers with parameters σ and rectifier nonlinear activation functions. The output of this network uses a softmax layer to create a probability distribution over all the legal moves a on the 19 x 19 board which predicts the probability that an expert would make a particular move. The SL network is trained on an extensive database of expert game data consisting of state-action pairs (s, a). The network optimises the parameters σ to maximise the likelihood of the human move a when given s. The network is trained independent of historical moves and aims to predict based on the single state input.

$$\Delta\sigma \propto \frac{\partial \log p_\sigma(a|s)}{\partial\sigma}$$

AlphaGo used a 13 layer network trained on 30 million positions. This network only achieved an accuracy of 57.0% on a held-out test set. Even with the accuracy not seeming impressive at first glance, the previous state of the art model only achieved a 44.4% accuracy. The relatively minor improvements in accuracy led to significant improvements in playing strength.

Rollout policy network:

Due to the size of the SL policy network, the evaluation time is slow during a search which inhibits the number of simulations that can be performed. The solution is to create a smaller network called the rollout policy, which is much faster. The rollout policy is trained in the same fashion as the SL policy, although instead of using convolutional layers, it uses simple linear layers. The trade-off is that as size and complexity decrease, so does the accuracy of predicting expert moves. The rollout policy $p_\pi(a|s)$ achieved an accuracy of only 24.2% but required 2 μs to select an action instead of the 3 ms that the SL policy network uses.

RL policy network:

The second stage of the AlphaGo training pipeline is improving the policy network by using policy-gradient reinforcement learning. This involves the creation of the RL policy network p_ρ. Using the same architecture as the SL network, the RL policy network is initialised with the SL network's learnt parameters, i.e. $\rho = \sigma$. This gives the RL policy starting knowledge of moves that are deemed *good* to play. Once the RL policy network is created, self-play occurs. Games are played between the current best RL policy p_ρ and randomly selected policy networks from a pool of previous iterations $p_{\rho^{(k)}}$. This randomisation of opponents is seen to stabilise the training and prevent overfitting of the current RL policy. The reward function used in training is very sparse. All intermediate rewards are zero, with terminal rewards being 1 for winning and -1 for losing. The outcome of the game z_t is set to the terminal reward. The parameters ρ are then updated at each time step t by stochastic gradient descent in the direction that maximises the expected outcome z_t:

$$\Delta\rho \propto \frac{\partial \log p_\rho(a_t|s_t)}{\partial\rho} z_t$$

This expected outcome is the same for all time steps, thereby giving equal credit assignment for all moves in the game. The RL policy's performance was evaluated against the SL policy network where moves are sampled from the networks outputted probability distributions, i.e. $a_t \sim p_\rho(\cdot|s_t)$. The RL policy achieved victory in more than 80% of the games played against the SL policy network. Surprisingly, a similar result was seen when the RL policy, without the use of any search mechanism, played against the strongest open-source Go program at the time, Pachi [5], where the RL policy network won in 85% of the games played. This incredible feat shows the power of reinforcement learning and self-play as a simple policy network with no search whatsoever could beat a MCTS program that executes 100,000 simulations per move. Ultimately, the RL policy network is not even used in the final AlphaGo program but is pivotal in creating the value network.

Value Network:

The last stage in the AlphaGo training pipeline is the creation of the value network. The value network $v_\theta(s)$ is created and trained to estimate the value function $v^p(s)$ which knows the outcome z_t of any game, from board positions s, that is played using the RL policy p_ρ.

$$v^p(s) = \mathbb{E}\Big[z_t|s_t = s, a_{t...T} \sim p\Big]$$

All perfect-information games have an optimal value function $v^*(s)$ under perfect play. Practically, if one does not know what *perfect play* is, then it is impossible to estimate this function. A solution is to estimate the value function when played under an extremely high performing specific policy that one has access to. Thus since the RL policy is the strongest policy available, i.e. an approximation of perfect play, a neural network $v_\theta(s)$ is created, which tries to estimate $v^p(s)$. Essentially, this means the value network is trying to approximate an approximation of the optimal value function i.e $v_\theta(s) \approx v^{p_\rho}(s) \approx v^*(s)$. The neural network $v_\theta(s)$ uses the same architecture as the policy network except outputs a single value. This network is then trained on state-outcome pairs (s, z) to minimise the mean squared error between the predicted value and outcome z.

$$\Delta\theta \propto \frac{\partial v_\theta(s)}{\partial\theta}(z - v_\theta(s))$$

When trained on state-outcome pairs of complete games, the value network simply memorised game outcomes as a lot of the data is highly temporally correlated. This is because successive positions in one game all share the same outcome. It is shown that when the value network is trained on the expert data, which consists of this highly correlated data, the minimum MSE on the test set was 0.37, which is high compared to the MSE of the training set 0.19. This indicates overfitting occurred where the value network did not learn to generalise. The solution implemented was to create a new data set, using self-play between the RL policy and itself, consisting of 30 million distinct positions and outcomes where each was sampled from a different game. When trained on this new self-play data set, the value network achieved similar test and training MSE scores of 0.234 and 0.226, respectively. It was seen that the value function $v_\theta(s)$ was consistently more accurate than Monte Carlo rollouts using the rollout policy p_π. Additionally, when using Monte Carlo rollouts with the RL policy p_ρ, the value network approached the same level of accuracy but with 15,000 times less computation.

Lookahead Search Even though the creation of the RL policy network already achieved an incredible level of play, it most likely still would not have the ability to beat a professional. The addition of a search algorithm gives AlphaGo the performance enhancement it needs to compete at the professional level. AlphaGo makes use of both the policy and value networks in an adapted MCTS algorithm for action selection. Each edge which represents a state-action pair stores the action value $Q(s, a)$, visit count $N(s, a)$ and a prior probability $P(s, a)$. These prior probabilities are given by the SL policy network p_σ. In the MCTS simulation phase, at each time step t, action selection of a_t, i.e. the edge to traverse, is selected by maximising the action value and a bonus:

$$a_t = \arg\max_a(Q(s_t, a) + u(s_t, a))$$

where

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

This bonus makes action-selection use the prior probability but decay its impact with repeated visits to ensure and encourage exploration. When the traversal of the tree reaches a leaf node s_L, this

leaf node may be expanded and is processed just once by the SL policy network to calculate the prior probabilities for each possible action in state s_L. Since the SL policy network is very slow in evaluation, the search algorithm tries to limit its use as much as possible by only using it once for unexpanded leaf nodes. The leaf node is then evaluated in two ways, first by the value network $v_\theta(s)$ and second by the outcome z_{s_L} of a randomly played game played until termination using the rollout policy p_π. Both evaluations are combined, which results in a final evaluation score $V(s_L)$.

$$V(s_L) = (1-\lambda)v_\theta(s_L) + \lambda z_{s_L}$$

where λ is a mixing parameter which ended up being 0.5 in the final AlphaGo program. At the end of a simulation, all the traversed edges have their action values and visit counts updated:

$$N(s,a) = \sum_{i=1}^{n} 1(s,a,i)$$

$$Q(s,a) = \frac{1}{N(s,a)} \sum_{i=1}^{n} 1(s,a,i) V(s_L^i)$$

Where s_L^i is the leaf node from the i_{th} simulation, and $1(s,a,i)$ indicates whether an edge (s,a) was traversed during the i_{th} simulation. Once the MCTS algorithm has finished with respect to the given time constraints, AlphaGo chooses the most visited move from the root position of the search tree.

With these components, AlphaGo achieved remarkable performance by achieving a 99.8% win rate against all other Go-playing programs at the time. Additionally, as stated before, AlphaGo achieved the first computer program victory against a professional human player. Ultimately, DeepMind created a distributed version of AlphaGo that used 40 search threads, 1,202 CPUs, and 176 GPUs which achieved higher performance. See figure 11 for additional information on performance.

The initial implementation of AlphaGo was not at the level of play to defeat the world champion Lee Sedol. However, over time AlphaGo became increasingly stronger upon which world champion level of play was achieved. AlphaGo was beyond a simple emulation of human performance. AlphaGo possessed creativity and played several novel game-winning moves that showcased its incredible level of play.

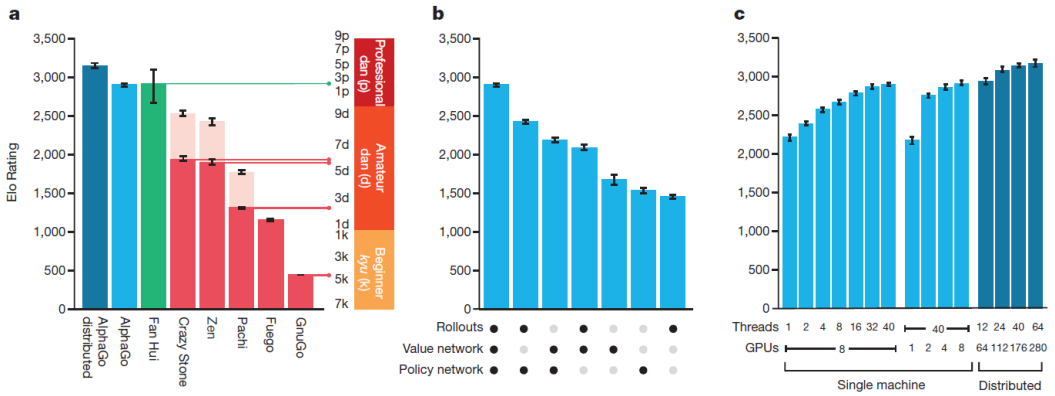

Figure 11: **a:** The Elo scores of AlphaGo, Fan Hui, the European Go champion as well as the other Go-playing computer programs available at the time. **b:** The Elo differences of AlphaGo when making use of a different combination of components. **c:** The Elo increases as computational resources increase.

3.1.2 AlphaGo Zero

AlphaGo was an incredible achievement, but some would say the use of expertly designed heuristics and large amounts of expert data diminishes its impact on general game-playing AI. Expert datasets are often unavailable, and expertly designed heuristics are hard to create and are variable in performance. AlphaGo Zero [55], the next iteration of DeepMind's Go-playing agents, set out to remove these limitations and constraints. The goal of AlphaGo Zero was to achieve superhuman proficiency in

the game of Go without the use of any human data, i.e. *tabula rasa*. AlphaGo Zero has four critical differences from AlphaGo. The first difference is that AlphaGo Zero uses only the black and white stone positions on the board as input, thereby removing expertly designed heuristics or features. The second and most important difference is that training is done exclusively through self-play without any human supervision. This completely removes the need for human-generated data. The third difference involves changing AlphaGo's original neural architecture to a singular neural network to perform both roles of the policy and value networks. The final difference is that AlphaGo Zero uses a less complex tree search. This more straightforward tree search does not use any Monte Carlo rollouts to simulate the rest of the game; leaf nodes are evaluated using only the value network. Additionally, the tree search does not require multiple neural networks for different aspects of the search. AlphaGo Zero implements these changes along with a newly designed reinforcement learning algorithm that incorporates an agents' lookahead search in the training loop. The following sections elaborate on AlphaGo Zero and its changes:

Architecture:

AlphaGo Zero uses a single neural network $f_\theta(s)$ to output both the action probabilities p, which give the probability of selecting each move $Pr(a|s)$, and the state value v which gives the probability of winning from board position s. This singular neural network serves the roles of both the policy and value networks used in the original AlphaGo system. The network architecture consists of residual blocks [28] of convolutional layers using batch normalisation [31], instead of the standard convolutional layers that AlphaGo used, and rectifier nonlinearities [25]. Combining the policy and value networks into a singular network allows the dual objective to regularise the network to learn a more common representation that supports various use cases. Additionally, the singular network is more computationally efficient, allowing increased MCTS simulations within a limited time frame.

Training:

AlphaGo Zero learns entirely from scratch with only being told the rules of the game. This is done through self-play and a novel reinforcement learning algorithm. This novel algorithm incorporates MCTS in the training process as a policy improvement operator whereby the discovered search probabilities π almost always select stronger moves than the raw probabilities p outputted by the neural network $f_\theta(s)$. Self-play with search, which selects each move using the enhanced MCTS-based policy and then uses the game outcome z as a sample of the *value* v, may be considered a strong policy evaluation operator. The idea behind this novel algorithm is to use both the policy improvement operator and policy evaluation operator repeatedly in a policy iteration procedure [8, 46]. There are two stages in this training process that occur every iteration: Self-play and Neural Network Training.

Self-play: AlphaGo Zero plays games of Go against itself to generate training data. In each state s, the agent performs a lookahead tree search using the MCTS algorithm (see section 3.1.1) which uses the neural network $f_\theta(s)$ to output the prior probability $P(s, a)$ for actions and *value* $V(s)$ of a state. Each execution of the MCTS algorithm produces search probabilities π, for action selection, proportional to the visit count for each move.

$$\pi \propto N(s, a)^{\frac{1}{\tau}}$$

where τ is a temperature parameter. Each game is played until completion where at each time step t, tuples (s_t, π_t, z) are stored, where z is the final outcome of the game i.e $\{+1, -1\}$ for winning or losing respectively.

Neural Network Training: The neural network $f_\theta(s)$ is trained on the stored data tuples to, firstly, make the raw probabilities p more closely match the improved search probabilities π and, secondly, make the state evaluation value v more closely match the game outcome z. This optimisation is done using gradient descent on the following loss function $\mathcal{L}$.

$$(p, v) = f_\theta(s) \text{ and } \mathcal{L} = (z - v)^2 - \pi^T \log p + c||\theta||^2$$

where c controls the level of L2 regularisation to prevent overfitting. This training happens every iteration where data tuples (s, π, z) are sampled uniformly from all time steps of the previous game.

Outcomes:

AlphaGo Zero started training with completely random behaviour and continued to train without any human intervention for three days. In only 36 hours out of the 72 hour training time, AlphaGo Zero outperformed AlphaGo Lee - the AlphaGo iteration that defeated the world champion Lee Sedol.

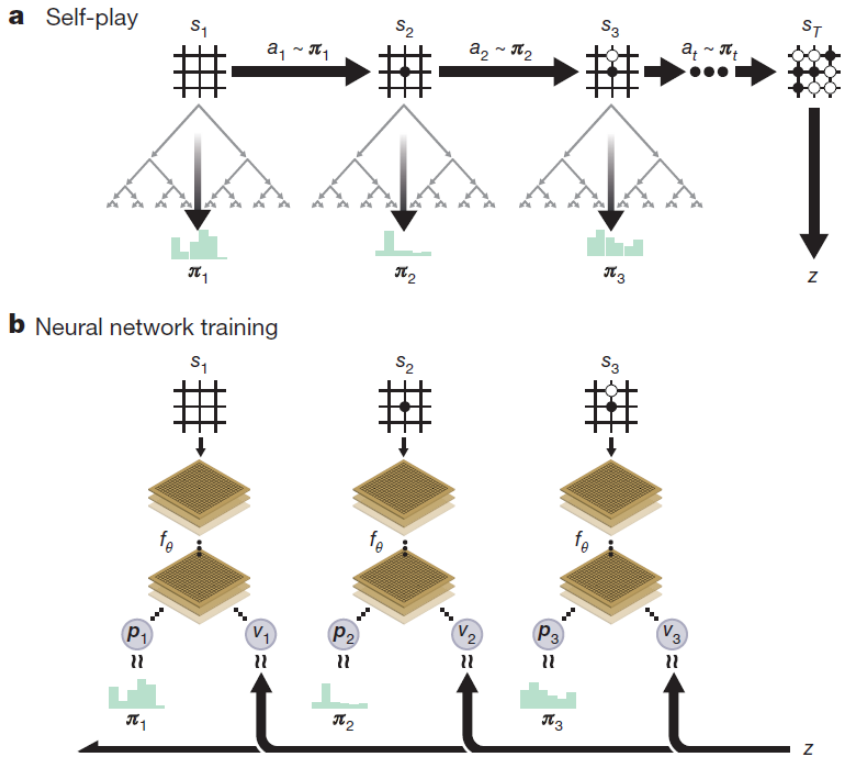

Figure 12: **a)** illustration of the self-play process. **b)** illustration of the training of the neural network.

This remarkable achievement is made even more apparent when considering that AlphaGo Lee was trained over several months. Additionally, AlphaGo Lee was distributed over many machines and used 48 TPUs, whereas AlphaGo Zero only used a single machine with 4 TPUs. After 72 hours of training, AlphaGo Zero defeated AlphaGo Lee by 100 games to 0.

Deepmind subsequently decided to retrain AlphaGo Zero from scratch using a larger neural network and longer training time. This AlphaGo Zero agent was trained for approximately 40 days. Upon the completion of training, an internal tournament was held, which pitted AlphaGo Zero against previous versions of AlphaGo, including AlphaGo Fan, AlphaGo Lee, and AlphaGo Master - the strongest existing Go program. AlphaGo Master uses the same architecture and algorithm as AlphaGo Zero but uses human data and features. In this tournament, AlphaGo Zero achieved an Elo rating of 5,185 - the highest in the tournament. This suggests that agents learning solely through self-play learn fundamentally different strategies that are higher performing, on average, than their human counterparts.

AlphaGo Zero comprehensively demonstrates the potential of a pure reinforcement learning approach even in highly challenging domains such as Go. We see that it is possible to train to a superhuman level without any human data or knowledge of the domain beyond the rules. Additionally, AlphaGo Zero demonstrates that a pure reinforcement learning approach can achieve higher performance than human-assisted methods.

3.2 Chess & Shogi

In the history of artificial intelligence, the game of chess has been a central subject of research. For decades, humans were never able to create a program that could defeat a professional chess player. However, in 1997, Deep Blue [17] defeated the human world chess champion, Gary Kasparov, marking a watershed moment for artificial intelligence. In the following decades, computer chess systems progressed to the superhuman level. However, these programs use a large number of clever heuristics and domain-specific adaptations with expertly designed handcrafted features. This makes all the advances in chess-playing performance relatively inapplicable to other domains. Shogi, a Japanese variant of chess, is another classic board game that only recently has seen a computer

program, Elmo, defeat a human champion [2]. Like most chess programs, Elmo used many domain-specific adaptations, which again prevented the system from being generally applicable. With the creation of AlphaGo Zero, which eliminated the need for human data and expertly designed heuristics, the question remained if it can easily be applied to other classical board games, such as chess and shogi, which have less complex game trees in comparison to Go. Although AlphaGo Zero removed most game-specific knowledge, certain assumptions remained, such as Go being invariant to rotation and reflection. AlphaGo Zero exploited this property of symmetry to enhance training data and the MCTS search. Since properties such as this cannot be assumed for other classical games, Deepmind again iterated on their previous work to create the next algorithm in their Alpha series: AlphaZero.

3.2.1 AlphaZero

AlphaZero [54] is the generalisation of AlphaGo Zero to allow the original algorithm to achieve superhuman performance in a variety of challenging games beyond just Go. As with AlphaGo Zero, AlphaZero starts entirely from scratch, only knowing the rules of the game. With minor changes to the original AlphaGo Zero algorithm, AlphaZero significantly defeated world champions in chess, shogi, and Go.

AlphaZero differs from AlphaGo Zero in the following aspects:

1. AlphaZero does not assume that games have a binary win or loss outcome. Games such as chess and shogi have the potential for players to draw. Instead, unlike AlphaGo Zero which estimates and optimises for the probability of winning, AlphaZero estimates and optimises the actual expected outcome.

2. AlphaZero has no game property assumptions and treats games as asymmetric, thereby not exploiting any game-specific information to enhance training data or tree searches.

3. AlphaGo Zero generated data through self-play of the highest performing player of all previous iterations. After every iteration, the newly updated player would be evaluated against the best player; if the new player won by more than a 55% margin, it was deemed the new best player. AlphaZero throws away this approach and simply maintains a single neural network that is continually updated instead of waiting for training iterations to complete. This means that all self-play data is generated using the latest parameters of the neural network.

AlphaZero, without the use of expertly crafted heuristics and features, managed to defeat the strongest computer programs in chess, shogi and Go. Additionally, AlphaZero searches around 80 thousand positions per second in chess and 40 thousand in shogi, compared to 70 million for Stockfish (strongest MCTS chess program) and 35 million for Elmo. This shows that AlphaZero uses an arguably more human-like approach where its deep neural network focuses on the most promising variations instead of the more brute force-like search performed by traditional systems.

AlphaZero shows the potential for a game-agnostic algorithm to achieve superhuman performance in various perfect information games, thereby bringing us closer to the longstanding goal in artificial intelligence of general superhuman game-playing systems.

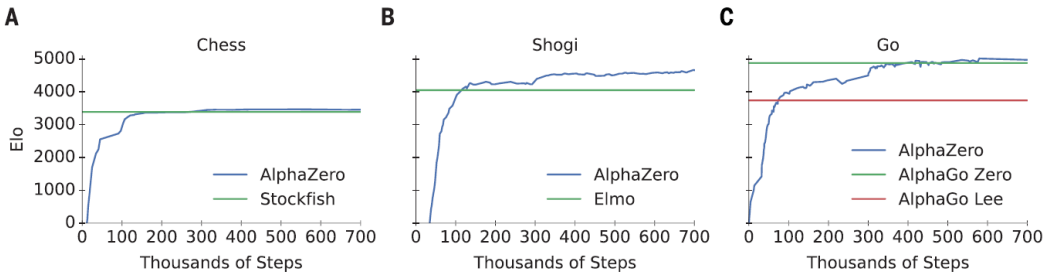

Figure 13: **A:** Performance of AlphaZero in chess compared with the 2016 TCEC world champion program Stockfish. **B:** Performance of AlphaZero in shogi compared with the 2017 CSA world champion program Elmo. **C:** Performance of AlphaZero in Go compared with AlphaGo Lee and AlphaGo Zero

3.3 Poker

AlphaZero has marked an incredible milestone in artificial intelligence whereby a single algorithm has surpassed human performance in various traditional board games. Even though AlphaZero is a relatively game-agnostic algorithm, there is one explicit limitation. AlphaZero requires perfect information where both players know the exact underlying state of the game at any point in time. This is a severe constraint on the application of AlphaZero in more complex real-life domains. In imperfect information games, information about the true underlying state of the game is hidden from players. Hidden information adds a significant degree of complexity to a game as beyond simply searching for optimal action sequences; a player must ensure an opponent does not find out too much about their private information. Additionally, imperfect information games cannot be subdivided into different tasks as the optimal strategy for a current situation can depend on the strategy played in future or alternative situations. This means that the strategy played must consider the game as a whole.

Poker has long since been a benchmark challenge for AI that can take into account hidden information [9, 1, 42]. Due to the game's strategic complexity and large size, No-limit heads-up Texas hold 'em has been the primary variant of poker used in AI research. No-limit Texas hold 'em is regarded as the most popular form of poker in the world. The heads-up variant (only two players) prevents the possibility of collusion between opponents and scenarios where bad players can cause mediocre players to do well. This variant, therefore, allows for a clear winner to be determined. Prior attempts at creating poker-playing agents mainly focused on game theory concepts such as minimising regret [12]. Although these agents achieved minor success, none had come close to defeating top-ranking human players. This changed in 2017 when an agent named Libratus [13] was developed. Libratus is an agent that managed to defeat top human professional poker players in no-limit heads-up Texas hold 'em without large amounts of expert domain knowledge or any human data.

3.3.1 Libratus

The Libratus agent contains three main modules in its game-solving approach:

1. The first module computes an abstraction of the game being played. This abstraction is a smaller and easier to solve *game* represented by a game tree. With the abstraction, the module then proceeds to compute game-theory based strategies to solve the abstracted game tree. These strategies provide detailed policies for the early game rounds, i.e. the first two rounds of betting; however, they only approximate how to play in the later rounds when the game becomes more complex. This solution to the abstracted game is pre-computed before any games have taken place and is termed the blueprint strategy.

2. The second module comes into play as the game progresses to later rounds. The second module is used to construct a more accurate and finer-grained abstraction for the subgame, i.e. remainder of the game tree, and solves it in real-time. The second module does not solve the finer-grained abstraction in isolation, and it ensures that the subgame fits within the created blueprint strategy of the whole game.

3. The third module is the self-improver. This module aims to enhance the blueprint strategy by filling in missing branches in the blueprint's abstraction. Additionally, the third module ensures that game-theory based strategies are created for the missing branches.

In theory, all computations can be done in advance to create an optimal strategy for all possible situations. However, much like Go or chess, the game tree is too large for this approach to be feasible.

First Module:

To address the problem of imperfect information, an agent has to reason about the entire game as a whole instead of certain pieces of it. This differs from perfect information games like chess. For example: in chess, the optimal play does not rely on any other subgames (remaining game trees from a specific board configuration) that could have been reached and only relies on the current board configuration. In general, when reasoning about the game in its entirety, a solution can be pre-computed ahead of play. An example of an algorithm capable of doing this is counterfactual regret minimisation plus (CFR+) which was used to near-optimally solve heads-up **limit** Texas hold'em [10]. This was done entirely offline, and when playing, the agent would simply look up the strategy to play when certain states were reached. This was feasible since **limit** poker is much simpler with around 10^{13} unique decision points. In comparison, **no-limit** heads-up Texas hold'em has around

10^{161} decision points. This makes traversing the entire game tree even a single time practically infeasible. Due to this, pre-computing a strategy for every point is also impossible. To reduce this complexity, one can perform action abstraction since, fortunately, in poker, many decision points are very similar, e.g. there is practically no difference between betting 100 or 101. Abstraction in this regard refers to a smaller, more simple game that retains as many strategic aspects of the original game as possible. This can be employed to vastly reduce the size of the game tree. Libratus makes use of action abstraction by rounding bets to a predetermined set of bet sizes. Another abstraction performed by Libratus is chance-dependent action abstraction or, in the case of poker, card abstraction. This treats similar hands as the same state, e.g. a King-high flush only differs slightly from a Queen-high flush. By treating these hands as identical, the game tree drastically reduces in size. This may be beneficial for reducing computational complexity; however, there are differences between these hands that play a role at the top level of play. Libratus does not use card abstraction in the first two betting rounds but uses very dense action abstraction. Additionally, in the last two betting rounds, which contain a significantly larger number of possible states, Libratus only performs the abstraction in the blueprint strategy that is pre-computed. Once the abstracted game tree is constructed, Libratus creates the blueprint strategy through self-play using an algorithm called Monte Carlo Counterfactual Regret Minimisation (MCCFR).

MCCFR estimates and maintains a regret value, i.e. how much would the agent regret not taking a certain move, for each action in the game tree. When actions are encountered during self-play, Libratus chooses the action with higher regret more often, i.e. higher probability. As self-play occurs, MCCFR ensures that a players average regret for any action approaches zero. This means Libratus's average strategy during self-play gradually improves. In each simulated game of self-play, MCCFR chooses one player to explore every possible action and update its regrets (called the traverser) whilst the other player plays according to the strategy determined by the current regrets. After every game, the algorithm switches the roles of the players. The regrets for each action form a probability distribution which is used to sample actions when playing. At the traverser's decision points, MCCFR explores every action in a depth-wise manner. At the other player's decision points, actions are sampled. This process repeats until the end of the game, whereby a reward is given, which backpropagates up the tree much like MCTS. Libratus uses a slightly improved version of MCCFR whereby a smaller portion of the tree is traversed every iteration. In certain games, there are obvious suboptimal actions and routes that waste computational resources by repeated exploration. Instead of exploring every hypothetical action, Libratus's MCCFR skips over the unpromising actions that have very negative regret as the tree gets deeper. MCCFR is run on the abstracted game tree to derive the blueprint strategy, which is very detailed for the first two rounds but less detailed for the final two rounds. Libratus makes use of the blueprint strategy for the first two rounds but not in the last two. In the last two rounds, the blueprint strategy is used to estimate the reward a player should expect with a particular hand in a subgame. This estimate is then used to determine a more optimal strategy during actual play.

Second Module:

Abstraction-based approaches have previously managed to create high performing poker AI [9, 24] but have not been sufficient to reach superhuman performance. Beyond the abstraction performed by Libratus in the first module, the second module is aimed at solving subgames whereby a more fine-grained, detailed strategy is created for a particular part of the game that is reached. As stated previously, in the first two rounds of betting, Libratus plays according to its pre-computed blueprint strategy. This is because the early parts of the game allow for more detailed abstraction to take place. When the third round is reached, the remaining game tree is small so Libratus can construct a new abstraction for the remaining subgame. This subgame tree is then solved in real-time. The problem is that in imperfect information games such as poker, subgames cannot be solved in isolation since optimal strategies can depend on other subgames. Previous solutions have assumed that opponents play according to the blueprint strategy, but this poses a considerable risk. The opponent can exploit this assumption by simply switching to a different strategy. This can cause the newly learnt subgame strategies to be significantly worse than the blueprint strategy. The technique used by Libratus is termed safe subgame solving. The idea behind this is that Libratus makes no strict assumption about the opponents' strategy but instead makes an assumption about the values they could receive in an equilibrium. An equilibrium (Nash Equilibrium) is a profile of strategies that no player can improve upon by deviating; therefore, if both players are playing the equilibrium strategy, it ensures you will not lose in expectation. These equilibrium expectations are approximated using the blueprint strategy.

Libratus then creates the strategy according to these values that will make the opponents worse off. There is no guarantee that the strategy created will make the opponents worse off, but at the very least, the strategy should make the opponent no better off than the action taken in an equilibrium. Since the equilibrium values are not known and only approximated, there is a margin of error, but this method has been empirically shown to have high performance. The strategy created for the subgame is what is used for the last two betting rounds of a poker game. This subgame strategy was created by using the CFR+ [10] algorithm, combined with specialised optimisations [32] for equilibrium finding, on the subgame game tree. The technique used by Libratus, safe subgame solving, follows the theorem that if the subgame estimated values are close to the true equilibrium values, then the strategy created plays close to a Nash Equilibrium strategy.

Third Module:

Libratus's third module is dedicated to self-improvement. The module's responsible for enhancing the blueprint strategy in the background. Due to action abstraction, the number of branches in the blueprint strategy does not represent all the actions possible. For example, if an opponent made a bet of 80, the blueprint strategy may have to round this to 100 as it does not have a branch for that specific action. This can cause the blueprint strategy to experience rounding errors and misplay certain moves. The solution to this is the third module. After a day of playing, the third module records the most popular actions the opponents are playing that are the furthest from actions the blueprint strategy has already accounted for. Overnight, Libratus then creates and fills in new branches in the blueprint abstraction and computes game-theory based strategies for these branches. This way, Libratus can progressively narrow the gaps that are in the action abstraction. By doing this, if the opponents can find a weakness in the action abstraction, this weakness will not persist for the whole tournament.

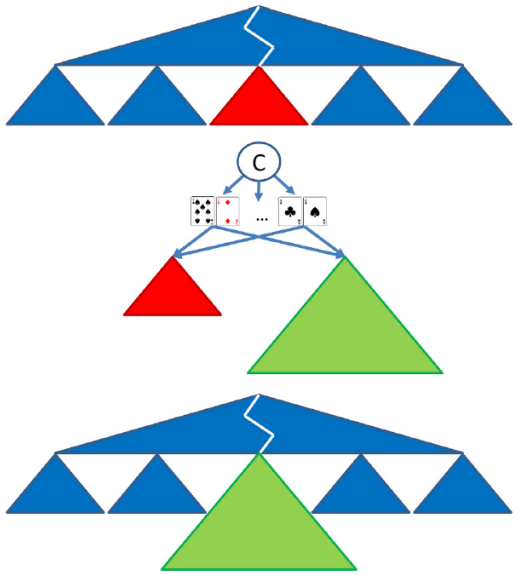

Figure 14: The following is taken directly from the original paper [13]. **Subgame solving** Top: A subgame is reached during play. Middle: A more detailed strategy for that subgame is determined by solving an augmented subgame, in which on each iteration the opponent is dealt a random poker hand and given the choice of taking the expected value of the old abstraction (red), or of playing in the new, finer-grained abstraction (green) where the strategy for both players can change. This forces Libratus to make the finer-grained strategy at least as good as in the original abstraction against every opponent poker hand. Bottom: The new strategy is substituted in place of the old one.

Libratus is the first poker AI ever to defeat top-ranking professionals in poker, albeit only against single opponents. Libratus managed to capture uniquely human strategies of the game such as "bluffing" as well as uniquely inhuman strategies such as using a large variety of different bet sizes. The underlying techniques used by Libratus are relatively domain-independent, allowing it to be used for a variety of imperfect information domains. As of 2019, an agent named Pluribus [14]

which builds on Libratus has defeated top professionals in multiplayer poker, meaning the two-player constraint is no longer required.

3.3.2 Pluribus

Checkers, Chess, Go, and heads-up no-limit texas hold 'em all have a common factor. They are two-player zero-sum games. Superhuman performance in these games have all been reached by approximating a Nash equilibrium strategy instead of some method to exploit and adapt to their opponents. Two-player zero-sum games have a special property whereby the Nash equilibrium is guaranteed to not lose in expectation no matter the opponent's strategy. This means the Nash equilibrium gives the optimal strategy in a two-player zero-sum game. Deviating off the Nash equilibrium strategy to exploit opponents weaknesses is a viable shift in play; however, this shift would open one up to being exploited. Nash equilibriums are proven to exist in any finite game, although discovering them is a non-trivial task. No polynomial-time algorithm is known that can find Nash equilibriums in two-player zero-sum games. Finding Nash equilibriums in zero-sum games that contain more than two players is at least as hard, if not harder, than two-player games. Another challenge in multiplayer games is that it is not clear that simply playing a Nash equilibrium would be the best choice. If every player independently computes and plays an equilibrium strategy, then the group of strategies may not be an equilibrium. The Lemonade Stand Game [76] is an example of this (see figure 15 for an explanation). These issues with multiplayer Nash equilibriums and lack of game-theoretic solutions make multiplayer games a tough challenge to strategise for. Due to this, Pluribus's goal is not a specific game-theoretic solution but rather the construction of an artificial intelligence system that empirically can defeat human opponents consistently. Pluribus makes use of algorithms that are not guaranteed to converge to a Nash equilibrium in non-two-player zero-sum games but are observed to create strong strategies capable of defeating human professionals. A key observation of Pluribus is that even with a lack of theoretical guarantees, superhuman performance can be achieved.

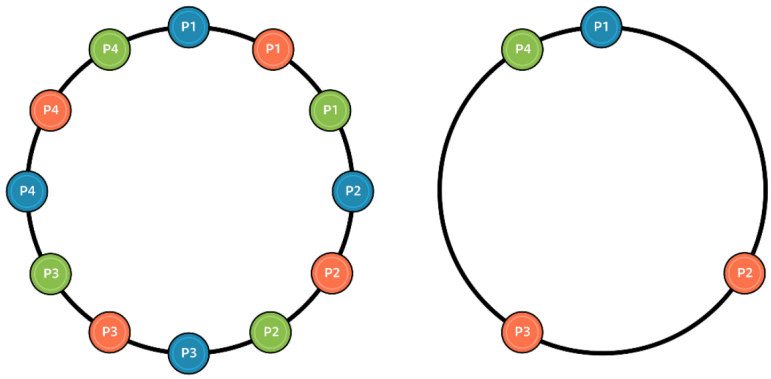

Figure 15: The following is taken directly from the original paper [14]. **An example of the equilibrium selection problem**: In the Lemonade Stand Game, players simultaneously choose a point on a ring and want to be as far away as possible from any other player. In every Nash equilibrium, players are spaced uniformly around the ring. There are infinitely many such Nash equilibria. However, if each player independently chooses one Nash equilibrium to play, their joint strategy is unlikely to be a Nash equilibrium. Left: An illustration of three different Nash equilibria in this game, distinguished by three different colors. Right: Each player independently chooses one Nash equilibrium. Their joint strategy is not a Nash equilibrium.

Pluribus, like Libratus, produces its main strategy via self-play, whereby copies of the agent play against each other. Starting from random play, Pluribus gradually improves as it discovers which actions and strategies lead to better outcomes. Pluribus uses the same core ideas as Libratus: First, the computation of a blueprint strategy for the entire game using the MCCFR algorithm, which happens offline before any games are played. Second, improving the blueprint strategy in real-time depending

on the situations the agent finds itself in. Additionally, Pluribus also uses the same action and card abstraction as Libratus to reduce the complexity of the game tree.

Pluribus differs from Libratus in the way it improves the blueprint strategy. Unlike Libratus, which uses the blueprint strategy for the first two rounds, Pluribus only plays according to it in the first round. In the second round and above, Pluribus uses a real-time search to discover a stronger, more fine-grained strategy for its current situation. This real-time search is also used in the first round if an opponent chooses a bet size that is significantly different from the blueprint abstraction. In imperfect-information games, real-time search is difficult as the value of non-terminal leaf nodes is not fixed and depends on the searcher's and opponents' strategy. A potential solution, used by ReBeL [11], is to make the value of leaf nodes conditioned on the belief distribution of both players at that point in the game [41]. A problem with this, is the computational resources needed for calculation. This method requires one to solve a large number of subgames that are conditional on beliefs . This complexity increases with the number of players and amount of hidden information, making it infeasible for multiplayer setups and games like reconnaissance chess. Libratus avoided this complexity by exclusively performing a real-time search (nested subgame solving) when the remaining subgame was short enough that the depth limit would reach the end of the game, thereby not requiring an evaluation function. Unfortunately, when dealing with more than two players, always solving to the end of the game also becomes computationally expensive.

Pluribus's solution to real-time search uses a modified form of Depth-Limited Solving [15] where the searcher considers that any or all players may shift to different strategies beyond the leaf nodes of a subgame (see figure 16). This means that, unlike other methods that assume all players are using a fixed strategy beyond the leaf node, Pluribus assumes that each player may choose between k different strategies, specialised to each player, to play for the rest of the game. Pluribus specifically used a k value of four and assumed players could choose between the following four strategies.

1. The precomputed blueprint strategy.
2. A modified version of the blueprint strategy that is biased toward folding.
3. A modified version of the blueprint strategy biased toward calling.
4. A modified version of the blueprint strategy biased toward raising.

Because picking an imbalanced strategy (e.g. always playing Rock in Rock-Paper-Scissors) would be exploited by an opponent moving to one of the other continuing tactics, this approach leads to the searcher discovering a more balanced strategy that accounts for multiple opponents and their playstyles.

Another challenge in imperfect information games is that optimal play relies on what a player would do in every situation from their opponents' perspective. For example, if a player is holding the best possible hand, betting in this situation is a good action. However, if the player exclusively bets in this situation, the opponents would always fold as a response. Pluribus calculates how it would act with every potential hand in its current state, balancing its approach over all hands to stay unpredictable to the opponent, regardless of whatever hand it is actually holding. Using a newly calculated balanced strategy, Pluribus then takes action for its current hand.

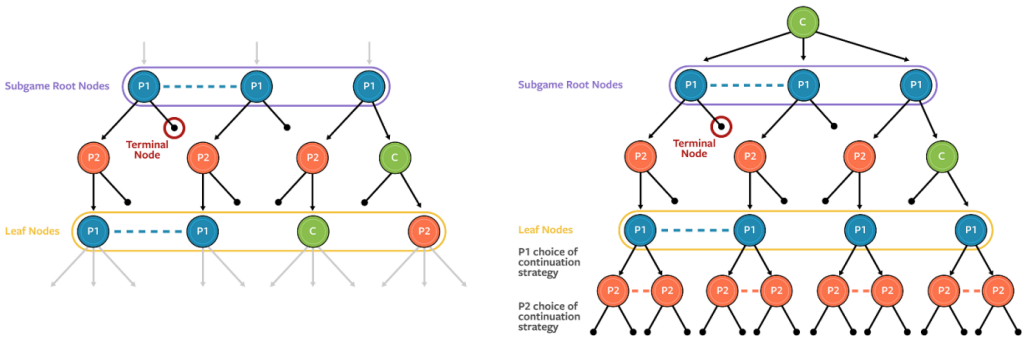

Figure 16: The following is taken directly from the original paper [14]. **Real-time search in Pluribus**: The subgame shows just two players for simplicity. A dashed line between nodes indicates that the player to act does not know which of the two nodes she is in. Left: The original imperfect information subgame. Right: The transformed subgame that is searched in real time to determine a player's strategy. An initial chance node reaches each root node according to the normalized probability that the node is reached in the previously-computed strategy profile (or according to the blueprint strategy profile if this is the first time in the hand that real-time search is conducted). The leaf nodes are replaced by a sequence of new nodes in which each player still in the hand chooses among k actions, with no player first observing what another player chooses. For simplicity, k = 2 in the figure. In Pluribus, k = 4 . Each action in that sequence corresponds to a selection of a continuation strategy for that player for the remainder of the game. This effectively leads to a terminal node (whose value is estimated by rolling out the remainder of the game according to the list of continuation strategies the players chose).

3.3.3 ReBeL

In an attempt at merging the success of AlphaZero and Libratus, Recursive Belief-based Learning (ReBeL) [11] was created. The ReBeL agent aims to create a single algorithm that can play perfect and imperfect information games. ReBeL is a general RL and real-time search framework that converges to a Nash equilibrium (optimal policy) in two-player zero-sum games. Much like AlphaZero, ReBeL trains a value network and policy network exclusively through self-play reinforcement learning and makes use of these functions for real-time search. ReBeL managed to defeat top poker players in heads-up no-limit Texas hold 'em, like Libratus, using far less domain knowledge whilst also being applicable to perfect-information games.

Imperfect-information games can be converted into continuous state and action space perfect-information games where the state description contains a probabilistic belief distribution of all agents. By doing this, perfect-information game techniques can be applied with minor modifications to imperfect-information games. To elaborate the idea of probabilistic belief distributions, consider a toy card game consisting of two players, where each player is dealt 52 cards privately. Every turn, a player has three actions. They can call, raise or fold. Eventually, the game will terminate, and some player will receive a reward. Since each player has hidden information, this is an imperfect information game. Now consider a modification to this game by not allowing either player to see their private cards and introducing a third-party *referee* to view both their cards for them. On each turn, a player announces the probability of taking a specific action with each possible private card. The *referee* then samples an action on the players' behalf from the players announced probability distribution given their actual card. At the start of the game, each player has a uniform random belief distribution about their private card, but after each action performed by the *referee*, players can update their belief distribution about the card they are holding via Bayes's Rule. Additionally, players can update their belief distribution about the card their opponent has by looking at the *referee's* actions. Ultimately, throughout the entire game, the probability that each player is holding each private card is common knowledge for all players in the game. Interestingly, both versions of the game are strategically identical, but the modified version contains no hidden information, thereby becoming a continuous state and action space perfect-information game. This requires the assumption that, although players do not directly announce their policies in the first game, all players' policies are common knowledge; therefore, the probabilities associated with each action are also common

knowledge. This is a common assumption in game theory solutions. An argument for it is that an opponent would eventually determine an agent's policy in repeated play.

The first version of the game illustrated is referred to as the discrete representation, whereas the modified version is referred to as the belief representation. Using the example above, a history in the belief representations, called Public Belief State (PBS), comprises the sequence of public observations and 104 probabilities (the probability that each player holds each of the 52 cards). An action is comprised of 156 probabilities (one per discrete action per private card.). Since a PBS is a history of the perfect-information belief-representation game, a subgame can be rooted at a PBS.

The interpretation of imperfect-information games as continuous state perfect-information games is not ReBeL's novel contribution. ReBeL's novelty is the combination of this interpretation with self-play reinforcement learning for adversarial settings. The conversion of imperfect information to perfect information can, in principle, allow one to run algorithms such as AlphaZero directly. However, this is practically inefficient. The state and action dimensions can increase exponentially. Traditional reinforcement learning and search methods prove unwieldy in such situations. ReBeL uses the convex property of these high-dimensional state and action spaces in two-player zero-sum games to use a gradient-descent-like algorithm (CFR) to search efficiently. Simply put, these high-dimensional belief representations are convex optimization problems. This allows the value of leaf nodes to be calculated and depth-limited search to be used.

Although ReBeL is the first algorithm to offer sufficient Reinforcement learning and Search in imperfect-information games, the current implementation has certain drawbacks. Most notably, the amount of calculation ReBeL needs becomes unpractical in games with strategic complexity but little common knowledge, such as Recon Chess.

3.4 Atari

Traditional board games have long been within the artificial intelligence field, but recently, research has shifted towards more modern games. A natural progression from traditional board games, video games have become an extensively researched topic in reinforcement learning. In this domain, a variety of new and exciting techniques have been created with the *routine* goal of superhuman performance.

Video games, differing from traditional board games, have high-dimensional sensory inputs. Additionally, game dynamics can be very complex and hard to model. This inhibits the use of tree search as game simulation is not possible in a model-free context. Originally, learning to control agents directly from high-dimensional sensory inputs like vision was a longstanding challenge of reinforcement learning. The reason for this is that the extraction of high-level features from raw sensory input was extremely difficult and often required handcrafted features. This changed as the field of deep learning advanced, and as seen in 2013 with the creation of DQN [39], reinforcement learning was successfully applied to learn to play Atari games. Although complex control policies were learnt in these high dimensional environments, superhuman performance was far from reached with these original methods.

The Arcade Learning Environment (ALE), consisting of 57 Atari 2600 games, was offered as a benchmark collection of tasks. These iconic Atari games present a wide range of problems for an agent to master. This benchmark is extensively used in the research community to assess progress in developing increasingly intelligent agents that can operate with high-dimensional sensory inputs. As research progressed, the performance on the Atari suite of games improved with specific algorithms achieving superhuman performance in a variety of the games [34, 30]. Although this performance was reached, the problem of generalisation remained. Programs that could achieve superhuman performance in one game could fail to perform in another.

3.4.1 MuZero

MuZero [49], the next step in Deepmind's pursuit of a game-agnostic algorithm, managed to achieve the same performance as AlphaZero in Go, chess and shogi as well as achieve the state-of-the-art results (at the time) on the Atari suite. This was all done with yet another game-specific piece of information removed. AlphaZero managed to achieve superhuman performance in Go, chess and shogi without using human data or crafted heuristics, but it still required the rules of each game. MuZero, on the other hand, does not require knowledge of any rules or game dynamics. Due to the use of planning and lookahead searches, algorithms like AlphaZero cannot be used in environments with unknown or highly complex game dynamics as knowledge of action consequences is required

for simulations. In a video game context, this would mean an agent would need to know how its actions will influence the raw pixel observations it uses as input. Model-based reinforcement learning attempts to learn a model that can be used for planning but for visually rich domains, learning a model to predict the high dimensional sensory inputs is exceptionally challenging. Due to this, the most successful methods in environments such as Atari used to be model-free methods. MuZero builds upon the AlphaZero algorithm to be compatible with environments of all kinds and achieve success in visually complex domains whilst maintaining superhuman performance in precise planning tasks such as Go, chess and shogi.

MuZero is a model-based reinforcement learning technique that learns a model of the environment. What separates MuZero from other model-based reinforcement learning methods is that instead of trying to learn and model the entire environment, it learns a model that exclusively focuses on the aspects deemed necessary for planning and the decision-making process. Once this model is learnt, MuZero can use AlphaZero's methodology in any environment, including visually rich domains such as Atari.

To learn this model, three important elements that are crucial for planning are estimated. The three elements are: the value function, i.e. how *good* is the current position? The policy function, i.e. which action is the *best* to take? The reward function, i.e. how *good* was the last action? These three elements are all that are needed for MuZero to plan in environments with complex or unknown dynamics.

The MuZero model μ_θ consists of the following three connected components:

1. The Representation Function $s^0 = h_\theta(o_1...o_t)$
2. The Prediction Function $(p^k, v^k) = f_\theta(s^k)$
3. The Dynamics Function $(s^k, r^k) = g_\theta(s^{k-1}, a^k)$

At every time step t, predictions are made for every hypothetical planning step $k = 1..K$. The model μ_θ is conditioned on a sequence of the past observations $o_1, ..., o_t$ and future hypothetical actions $a_{t+1}, ..., a_{t+k}$ to predict the policy p_t^k, the value v_t^k and the immediate reward r_t^k.

$$p_t^k \approx \pi(a_{t+k+1}|o_1, ..., o_t, a_{t+1}, ..., a_{t+k})$$
$$v_t^k \approx \mathbb{E}\big[u_{t+k+1} + \gamma u_{t+k+2} + ...|o_1, ..., o_t, a_{t+1}, ..., a_{t+k}\big]$$
$$r_t^k \approx u_{t+k}$$

where u is the true reward, π is the policy that is used to select real actions, and γ is the discount factor.

The dynamics function g_θ is a recurrent process that mirrors the structure of a traditional MDP model as it computes the expected reward and state transition of a given state and action. The crucial difference here is that the dynamics function's state representation s^k has no semantics of the true environment attached. s^k is simply an internal representation used to accurately predict the relevant future estimates of policies, values and rewards. The value function and policy function are jointly computed from the internal state representation s^k using the prediction function $(p^k, v^k) = f_\theta(s^k)$. The starting state of any search, i.e. the root state s^0, is created using the representation function $s^0 = h_\theta(o_1...o_t)$ that encodes past observations into the internal hidden state representation. With these three components, this model can be used to search over any future trajectories of actions $a^1, ..., a^k$ conditioned on the past observations $o^1, ...o^t$.

MuZero uses the MCTS algorithm, as used by AlphaZero, to simulate game trajectories and produce the MCTS improved policies π_t and a improved value function $\mathbf{v}_t$. Like with AlphaZero, MuZero selects actions from the MCTS improved policy $a_{t+1} \sim \pi_t$. For every hypothetical step k in the model's simulation, the parameters of the model θ are jointly trained to accurately match the policy, value and reward to their respective true values observed after k actual time-steps. Firstly, MuZero aims to more closely match the model's produced policies p_t^k to the MCTS improved policies π_{t+k} making use of MCTS as a powerful policy improvement operator, as done in AlphaZero. Secondly, MuZero aims to minimize the error between the model's value estimates v_t^k and the outcome targets of the search z_{t+k} where z_t can be the actual outcome value or be a bootstrapped estimate $z_t = u_{t+1} + \gamma u_{t+2} + ... + \gamma^{n-1} u_{t+n} + \gamma^n \mathbf{v}_{t+n}$. Lastly, MuZero aims to minimise the

error between the model's predicted reward r_t^k and the true observed reward u_{t+k}. This creates the loss function $\mathcal{L}_t$:

$$\mathcal{L}_t(\theta) = \sum_{k=0}^{K} l^r(u_{t+k}, r_t^k) + l^v(z_{t+k}, v_t^k) + l^p(\pi_{t+k}, p_t^k) + c||\theta||^2$$

where l^r, l^v, and l^p are the loss function for reward, value and policy respectively. An additional L2 regularisation term is added at the end to prevent overfitting.

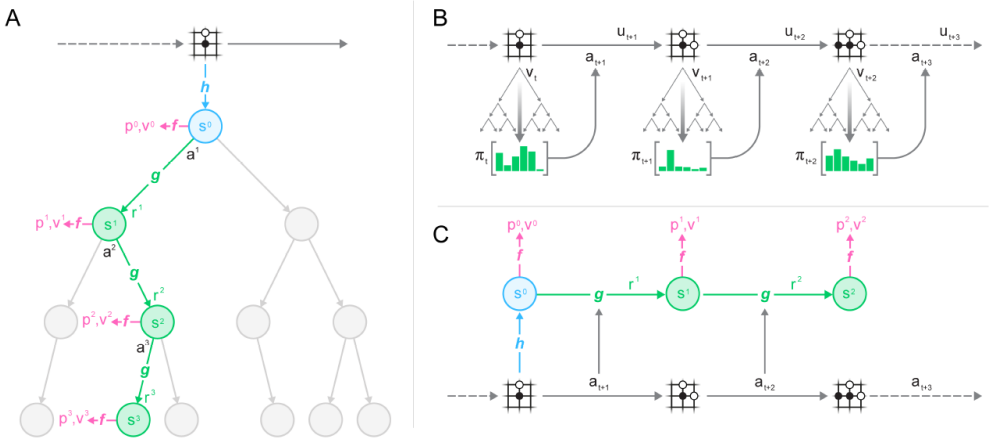

Figure 17: The following is taken directly from the original paper [49]. **Planning, acting, and training with a learned model. (A)** How *MuZero* uses its model to plan. The model consists of three connected components for representation, dynamics and prediction. Given a previous hidden state s^{k-1} and a candidate action a^k, the dynamics function g produces an immediate reward r^k and a new hidden state s^k. The policy p^k and value function v^k are computed from the hidden state s^k by a prediction function f. The initial hidden state s^0 is obtained by passing the past observations (e.g. the Go board or Atari screen) into a representation function h. **(B)** How *MuZero* acts in the environment. A Monte-Carlo Tree Search is performed at each timestep t, as described in A. An action a_{t+1} is sampled from the search policy π_t, which is proportional to the visit count for each action from the root node. The environment receives the action and generates a new observation o_{t+1} and reward u_{t+1}. At the end of the episode the trajectory data is stored into a replay buffer. **(C)** How *MuZero* trains its model. A trajectory is sampled from the replay buffer. For the initial step, the representation function h receives as input the past observations $o_1, ..., o_t$ from the selected trajectory. The model is subsequently unrolled recurrently for K steps. At each step k, the dynamics function g receives as input the hidden state s^{k-1} from the previous step and the real action a_{t+k}. The parameters of the representation, dynamics and prediction functions are jointly trained, end-to-end by backpropagation-through-time, to predict three quantities: the policy $p^k \approx \pi_{t+k}$, value function $v^k \approx z_{t+k}$, and reward $r_t^k \approx u_{t+k}$, where z_{t+k} is a sample return: either the final reward (board games) or n-step return (Atari).

A second version of MuZero, termed *Reanalyze*, was created that specifically aimed to optimise sample efficiency. MuZero Reanalyze executes MCTS simulations on its past time steps but using the models latest parameters. The idea is that, the models latest parameters will potentially allow the MCTS to provide better action probabilities than the original search. These new search policies are used as the policy network's target for 80% of the training updates. Additionally, a target value network $v^- = f_\theta^-(s^0)$ is created and used to provide a more recent and stable n-step bootstrapped target for the value function, $z_t = u_{t+1} + \gamma u_{t+2} + ... + \gamma^{n-1} u_{t+n} + \gamma^n v_{t+n}^-$. MuZero Reanalyze also changed several hyperparameters to increase sample reuse and avoid overfitting. The Reanalyze variant of MuZero managed to achieve remarkable scores in Atari, achieving a mean score of 2,168% (100% is human performance), whilst using magnitudes less data than its primary variant (only 200 million frames compared to 20 billion).

MuZero has equalled the superhuman performance of high-performance planning algorithms in their preferred domains, e.g. chess and Go, whilst also outperforming state-of-the-art model-free

reinforcement learning algorithms in their preferred domains, e.g. visually complex Atari games. Incredibly, MuZero exceeded the performance of AlphaZero whilst also achieving a new state of the art (at the time) for both the mean and median normalised scores across all 57 games in the Atari suite. MuZero outperformed the previous best method, R2D2 [34] (a model-free method) in 42 out of the 57 games. Even though MuZero achieved incredible results in most of the games, certain games such as Montezuma's Revenge proved too challenging. This meant the race to create the first general agent to achieve superhuman performance in all of the Atari games was still on.

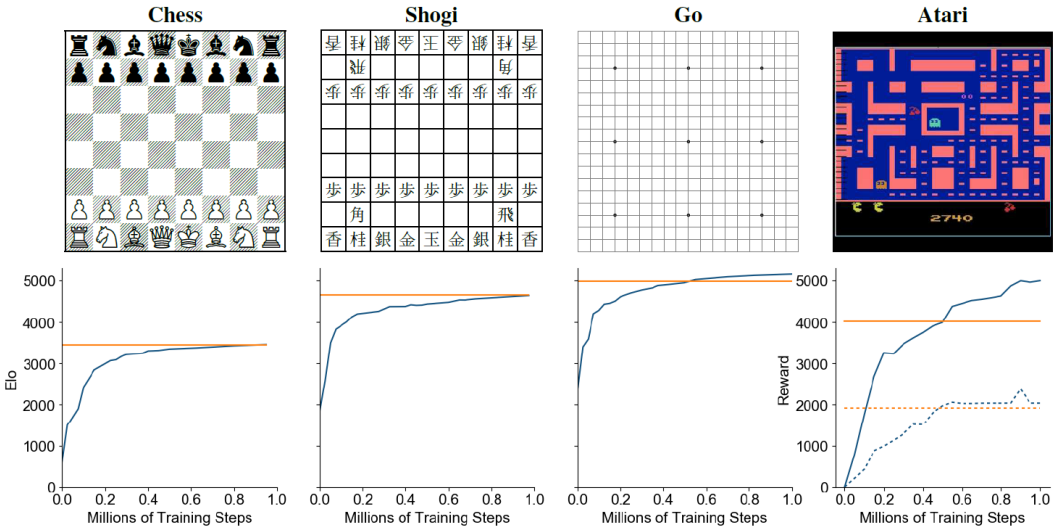

Figure 18: The following is taken directly from the original paper [49]. **Evaluation of MuZero throughout training in chess, shogi, Go and Atari**. The x-axis shows millions of training steps. For chess, shogi and Go, the y-axis shows Elo rating, established by playing games against AlphaZero using 800 simulations per move for both players. MuZero's Elo is indicated by the blue line, AlphaZero's Elo by the horizontal orange line. For Atari, mean (full line) and median (dashed line) human normalised scores across all 57 games are shown on the y-axis. The scores for R2D2 [34], (the previous state of the art in this domain, based on model-free RL) are indicated by the horizontal orange lines. Performance in Atari was evaluated using 50 simulations every fourth time-step, and then repeating the chosen action four times, as in prior work [40].

3.4.2 EfficientZero

Sample efficiency is still a critical challenge within reinforcement learning. Methods such as AlphaZero need to play approximately 21 million games to train to superhuman performance in Go. A professional player playing five games per day would need around 11,500 years to achieve the same level of experience. MuZero, even with its Reanalyze variant, requires large amounts of data to succeed. In game-like settings, this may not be an issue as experience can continuously be generated, but this limits the applicability of the method to simulations and games. EfficientZero [74] is a proposed expansion to MuZero designed to achieve high performance with limited data. EfficientZero makes use of the following three components to improve sample efficiency whilst maintaining superhuman performance: a self-supervised environment model, a mechanism to alleviate the model compounding error, and a method to correct off-policy issues. The use of these components are solutions to three critical problems that MuZero faces:

1. MuZero's learned model is trained exclusively through the reward, policy and value functions. This proves to be challenging since the reward is a scalar and most often sparse signal; the value functions are trained through bootstrapping, thereby introducing noise, and policy functions are trained with the search process. Therefore, these three values often cannot provide enough training signals to learn the environment model accurately with little data.
2. The learned model's reward prediction has significant prediction errors even with large amounts of data, and the reason for this is the aleatoric uncertainty of the underlying environment. These reward prediction errors accumulate in the MCTS process, which then results in sub-optimal exploration and evaluation.

3. When MuZero computes the value functions target value, the multi-step reward observed in the environment is used. This causes severe off-policy issues and hampers convergence in a limited data setting.

Self-Supervised Consistency Loss:

Since the use of MCTS as a policy improvement operator is so dependent on the environment model, it is imperative to have an accurate one. Since the output $\hat{s}_{t+1}$ of the learned models dynamics function g_θ should be the same as the output s_{t+1} of the representation function h_θ with the input of the next observation, we can use this property to help supervise the predicted next state $\hat{s}_{t+1}$.

$$h_\theta(o_1...o_{t+1}) = s_{t+1} = g_\theta(s_t, a_t) = \hat{s}_{t+1}$$

This provides an additional training signal to aid in the learning of the model. Using the SimSiam [1] self-supervised framework, the difference between the state representations is used in the training process.

End-To-End Prediction of the Value Prefix:

Model-based agents need to predict the future states conditioned on the current state and hypothetical actions. The further these predictions extend into the future, the more difficult it is to maintain accuracy due to the accumulation of prediction errors. This is referred to as the state aliasing problem. This problem dramatically harms the performance of MCTS and results in suboptimal use of the search. The estimation of the Q-value is:

$$Q(s_t, a_t) = \sum_{i=0}^{k-1} \gamma^i r_{t+i} + \gamma^k v_{t+k}$$

where the sum of the reward $\sum_{i=0}^{k-1} \gamma^i r_{t+i}$ is termed the value prefix. MuZero uses the predicted reward of each unrolled state $\hat{s}_{t+i}$ which can contain large prediction errors. EfficientZero proposes an end-to-end method to predict the *value prefix* instead of each reward independently. The value-prefix is predicted from all the unrolled states $(s_t, \hat{s}_{t+1}, ..., \hat{s}_{t+k-1})$. EfficientZero uses an LSTM to take in the variable number of inputs and output a scalar value. At every timestep during training, the LSTM is supervised as there is always a new value prefix when a new state comes in. This allows the LSTM to be trained well, even with limited data. The end-to-end prediction is also more accurate which reduces the error accumulation and aids in MCTS performance.

Model-Based Off-Policy Correction:

As seen in MuZero Reanalyze, the value target is given by computing $z_t = \sum_{i=0}^{k-1} \gamma^i u_{t+i} + \gamma^k v_{t+k}$ from a sampled trajectory of past data. This causes off-policy issues as the trajectory is rolled out using an older policy, thereby causing the value target to no longer be accurate. Since the agent has a model of the environment, it can use the model to simulate an *online* experience. EfficientZero proposes using the rewards of a dynamic horizon l from the old trajectory, where $l < k$ and l should be smaller if the trajectory is older. Intuitively, this can be seen as cutting off the old trajectory and using it as a seed to produce "imagined" data to train on. This process reduces policy divergence. Additionally, a new MCTS search is performed with the current policy on the last state of sampled data s_{t+l} whereby the empirical mean value at the root node is calculated. The value target used is:

$$z_t = \sum_{i=0}^{l-1} \gamma^i u_{t+i} + \gamma^l v_{t+l}^{MCTS}$$

where $l <= k$ and l is inversely proportional with the age of the sampled trajectory. $v^{MCTS}(s_{t+l})$ is the root value of the MCTS tree using the current policy, like MuZero Reanalyze.

EfficientZero is benchmarked on the Atari 100k environment, which limits training data of Atari games to 100 thousand environment steps. This is roughly equivalent to 2 hours of real-time gameplay. EfficientZero is seen to vastly improve upon prior methods, achieving state-of-the-art results by a large margin.

3.4.3 Agent57

Agent57 [3], yet another agent produced by Deepmind, is the first method to achieve, at the minimum, human performance on all 57 games in the Atari suite. It does this by making use of prior reinforcement learning algorithmic/architectural improvements along with an efficient exploration algorithm

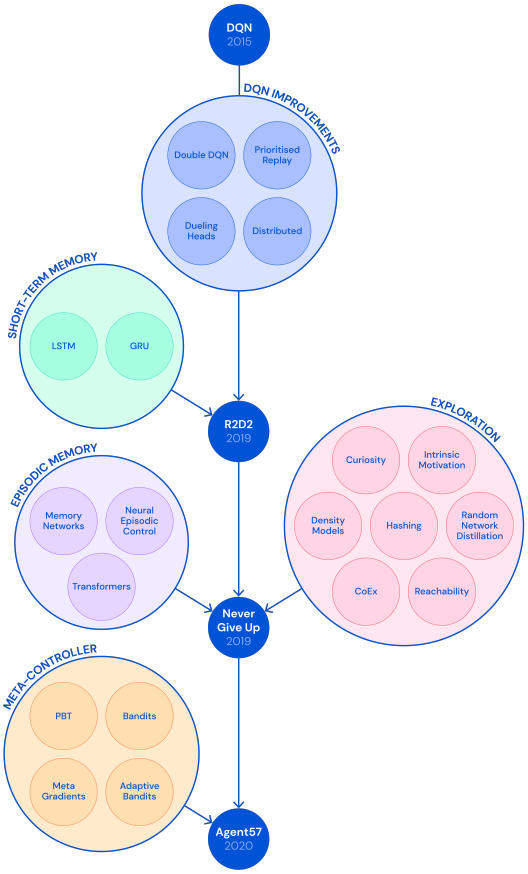

Figure 19: Agent57's Lineage

and meta-controller. Since the creation of the DQN, numerous extensions have been developed, e.g. DDQN [65], prioritised experience replay [45], and Dueling Networks [70], but these methods consistently fail in 4 specific games. Montezuma's Revenge, Skiing, Pitfall, and Solaris. MuZero [49] and R2D2 [34], two state-of-the-art algorithms, surpass 100% of the human normalised score on 51 and 52 games, respectively. Although these algorithms achieve superhuman performance in a large percentage of games (e.g. more than 1000% of the human normalised score), even these algorithms consistently fail in the four games specified. This failure is primarily due to two important reinforcement learning issues:

1. ***Long-term credit assignment*:** the problem of knowing which actions are deserving of credit for the positive or negative outcome is difficult when an environment is sparsely rewarded. This means long sequences of actions can occur before a reward is given, making it difficult for the agent to know which action influenced the outcome. In certain games such as *Skiing*, rewards are only given at the end, but certain actions performed during the playthrough are responsible for the outcome. The agent does not know which actions produced certain penalties and rewards.

2. ***Exploration*:** In reinforcement learning, efficient exploration can be crucial to learning viable policies. Certain games, such as Montezuma's Revenge or Pitfall, potentially require hundreds of actions before the first positive reward is given. Despite the difficulty of obtaining positive rewards, the agents must continue investigating the environment in order to succeed. This issue becomes even more prominent in vast, high-dimensional state spaces, especially when function approximation is required.

To solve these issues and achieve superhuman performance on these games without sacrificing a strong performance on others, Agent57 uses years of research in the reinforcement learning field in a

combination of methods and components. The following sections elaborate on the components used in Agent57:

DQN Improvements:

DDQN, prioritised experience replay and dueling networks are three improvements to the original DQN algorithm that greatly enhanced its performance. Specifically, these improvements increased its learning efficiency and stability - two common issues with reinforcement learning algorithms. These improvements are discussed in section 2. Agent57 makes use of and iterates on these ideas along with the remainder of its components.

Distributed Learning:

Distributed learning algorithms allow for simultaneous execution on many computers. This enables increased learning speed and experience generation since multiple simulations can happen in parallel. Agent57 is a distributed agent whereby the data collection and generation is decoupled from the actual learning process. In this decoupling process, multiple *actors* are responsible for environment interaction, and a single *learner* is responsible for updating network weights according to the collected data. For data collection, the *actors* simultaneously generate experience through interaction with their independent copies of the environment. This experience is then sent to a centralised, prioritised experience replay memory buffer. Once there is sufficient data, the learner can sample the trajectories to construct the necessary loss functions that optimise the parameters of its neural network. Each actor is periodically sent the learner's updated network weights to whereby each actor uses these network weights in a specialised manner determined by individual priorities. Figure 20 illustrates the distributed learning process.

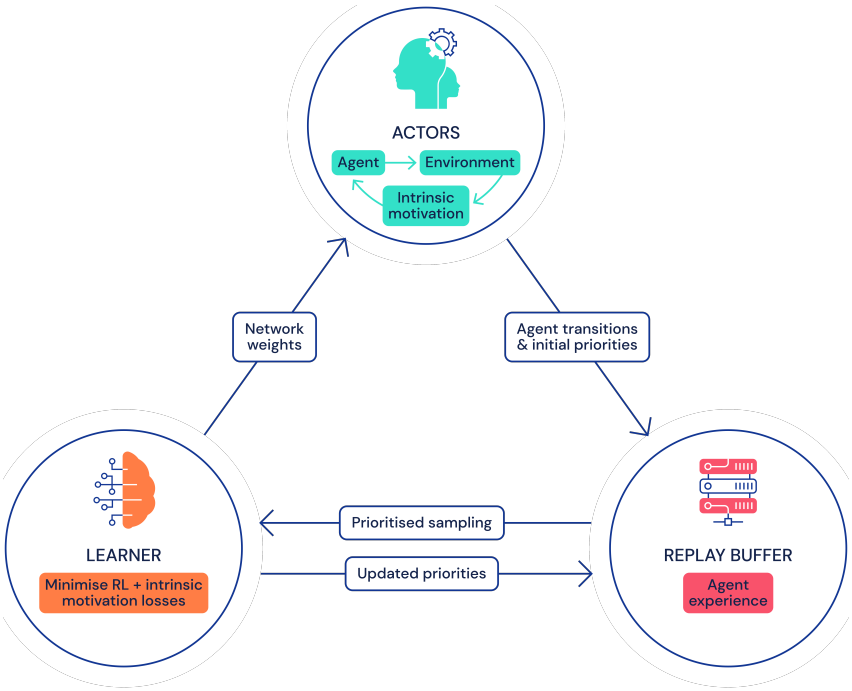

Figure 20: Agent57's distributed process of learning

Short-term memory:

Certain environments prove to be challenging to act in with only a partial current observation. The use of past observations can reveal more about the environment and allow an agent to make improved decisions. Incorporating a sense of memory in an on-policy learner is a relatively trivial task as the value of direct actions are being learned; therefore, *remembering* direct experience is simple. Incorporating memory into an off-policy learner, on the other hand, which can learn about optimal actions even when not actively performing those actions, is a lot more challenging as the agent needs to know what is *possibly remembered* when executing different behaviours. Off-policy learning

is desired since the agent can use more exploratory behaviour whilst still learning optimal control. Additionally, the ability to use past data allows for an agent to be more sample efficient. Deep Recurrent Q-Network [27] (DRQN) was the first agent to combine a sense of memory with off-policy learning. DRQN made use of a single LSTM [29] layer (a recurrent neural network) in addition to the original DQN architecture. This LSTM layer gave the agent a form of short-term memory with which to aid its decision making. The DRQN agent is seen to have increased performance in partially observable environments and allows for the ability to infer high-level information using past observations.

With the advancements in distributed learning showing significant performance increases and research showing the benefit of short-term memory inclusion, a new method for training distributed, off-policy, recurrent neural networks was required. R2D2 [34] was this method. The R2D2 algorithm incorporated recurrent neural networks, a dueling DQN architecture, prioritised experience replay and distributed learning into a single algorithm. R2D2, like DRQN, used an LSTM network layer atop a convolutional module. How R2D2 trained its recurrent neural networks required a change to the way data is stored. Instead of simply storing regular environment transitions (s_t, a_t, r_t, s_{t+1}), R2D2 stores large sequences of trajectories in the form $((s_1, a_1, r_1), (s_2, a_2, r_2), ..., (s_t, a_t, r_t))$. These trajectories are stored and given a priority level in the replay buffer. Experience is sampled from the replay buffer as normal, except instead of receiving batches of transitions, the agent receives batches of transition sequences to train on.

Episodic Memory:

Never Give Up (NGU) [4] was designed to augment R2D2 with episodic memory in addition to its short-term memory. This gives the agent the ability to detect when it encounters novel states. The reason for this is to allow the agent to focus on exploring the novel parts of the game it encounters. Due to this, the agents' behaviour dramatically shifts to favour exploration, making it difficult for the agent to learn the optimal policy for its original goal of achieving a high score. This makes off-policy learning required so that the exploration experience generated can still be effectively used. NGU uses an episodic memory buffer that stores the agents' observations within a single episode to calculate intrinsic motivation for efficient exploration.

Intrinsic Motivation:

For many sparsely rewarded environments, shaping the reward function is not possible. Using random exploration methods rely upon the agent stumbling onto the goal state by chance. This can be practically impossible in large environments and will result in failure to learn. Intrinsic motivation can be seen as a new way of learning which requires no extrinsic rewards r^e from the environment. It is a controlled form of exploration. The two most popular formulations of intrinsic reward can be grouped as follows: The first class of methods encourages the agent to explore states it has not seen before. Bellemare et al. [6] have shown improved results in very sparsely rewarded environments, such as Montezuma's Revenge, using such exploration with DQN. The second class of methods focus on encouraging the agent to take actions that lower the error in the agent's ability to predict the consequences of its actions. Intuitively this aims to increase the agent's knowledge about the environment [47, 48, 18]. Measuring the novelty, i.e. how different a state is from a previous state, or building an internal environmental model to predict the next state can be complex in high dimensional state spaces. This is compounded with environment stochasticity and noise, which ultimately makes intrinsic reward calculation difficult.

Random network distillation [16] (RND) is an exploration method that introduces a new approach for the second class of methods. Two neural networks are created: one that is fixed where weights are randomised called the target network $f(o_t)$ and one that is trained on data collected by the agent called the predictor $f_\theta(o)$. The target network simply receives the agent's observation o_t as input and outputs a fixed random embedding of the state. The predictor network receives an observation and also outputs a random embedding of the state. The predictor network is then trained to minimise the difference between its output and the fixed target output.

$$\mathcal{L}_\theta(o) = |f_\theta(o) - f(o)|^2$$

This process essentially "distils" the predictor network into a trained one. The prediction error between the two networks is expected to be larger for novel states that are more dissimilar to the states the predictor network has seen and been trained on. This prediction error $\mathcal{L}_\theta$ forms part of the intrinsic reward r^i as a way to encourage moving to novel states.

NGU introduced a new way to calculate intrinsic reward. This intrinsic reward has three main properties:

1. It discourages returning to the same state inside the same episode.
2. It gradually discourages trips to states that have been visited many times across episodes.
3. The concept of state overlooks parts of the environment that are unaffected by an agent's actions.

The intrinsic reward is made up of two modules: the *episodic novelty module* and *life-long novelty module*. The episodic novelty module uses the episodic memory buffer to calculate an *episodic intrinsic reward* $r_t^{episodic}$ for each observation based on the difference between the observation in question and all other observations seen within the episode. The intrinsic reward given is linearly proportional to the size of the difference. The episodic intrinsic reward rapidly diminishes if the agent starts to visit similar states within the episode repeatedly. The reason for this is to encourage an agent to explore as much as possible within a single episode time limit. The second module, the *life-long novelty module* is used to control how much exploration is performed across episodes. This is achieved with the use of a curiosity factor α_t. Random Network Distillation is used to control this factor as follows:

$$\alpha_t = 1 + \frac{\mathcal{L}_\theta - \mu}{\sigma}$$

where $\mathcal{L}_\theta$ is the RND error as discussed above, μ and σ are the running mean and standard deviation of $\mathcal{L}_\theta$ respectively. The curiosity factor then uses the episodic intrinsic reward $r_t^{episodic}$ to create the NGU intrinsic reward r_t^i:

$$r_t^i = r_t^{episodic} \cdot \{\max\{\alpha_t, 1\}, L\}$$

Where L is the chosen maximum reward scaling. Through the mixing of rewards in this fashion, long-term novelty detection is leveraged whilst still encouraging exploration. This means that if the curiosity factor and episodic intrinsic rewards are high, the agent receives a large intrinsic reward. Suppose the curiosity factor is low and the episodic intrinsic rewards are high. In that case, the resulting reward will be scaled down but never vanish entirely, thereby always incentivising the agent to continue exploring (the agent will ***Never Give Up***).

Every time step t, N different augmented reward signals are created by NGU. This is done by adding N differently scaled versions of the intrinsic reward $\beta_j r_t^i$ to the extrinsic reward r_t^e thereby producing N reward signals calculated by:

$$r_{j,t} = r_t^e + \beta_j r_t^i$$

Each reward $r_{j,t}$ therefore offers a greater or lesser degree of exploration incentivisation that depends on the scaling value β_j. NGU then proceeds to try and learn N different optimal Q functions Q_{rj}^* each associated with the different rewards r_j. NGU, therefore, learns N different policies associated with different degrees of exploratory behaviour controlled by the exploration rate β_j. Additional control is also performed by giving each of the N policies a different discount factor γ_j to use in training. Since NGU is a distributed learning algorithm, *actors*, which gather experience, are each given different β_j and γ_j parameters, thereby contributing to a very diverse pool of experience for the *learner* to use. NGU achieved high scores on the four hard-exploration Atari games but sacrificed performance in the others.

Meta-controller:

A problem with NGU is that all experience generated was treated as equally valuable, not taking into consideration the contribution to learning. The Atari suite provides a variety of games that all have very different exploratory needs. NGU focused too much on exploration in games that did not need it, resulting in lower performance than desired. This begs the question of whether or not one can teach an agent to decide for themselves when it is better to explore or exploit. This is precisely what the meta-controller is used for. Agent57 uses a meta-controller to directly decide the level of exploration going to be performed at the start of an episode. Since Agent57 is an expansion of NGU, most aspects are the same. Like NGU, multiple policies, parameterised by an exploration rate β_j and discount factor γ_j, are maintained and used by different *actors* in generating experience. Agent57 differs in that in every episode, each *actor* runs a non-stationary multi-armed bandit algorithm to select one of the multiple policies such that the policy maximises the expected return of that episode. Essentially, this sets out to teach the agent to pick the best performing policy for a specific episode. This gives

each actor the ability to pick the level of exploration it deems necessary to achieve. Through this, Agent57 can prioritise policies during training time to allow for more efficient use of resources. An additional change is also made to the Q-network to increase training stability. This sees to decompose the contributions of the intrinsic and extrinsic rewards by creating two separate neural networks such that:

$$Q_\theta(s,a) = Q^e_\theta(s,a) + Q^i_\theta(s,a)$$

whereby $Q^e_\theta(s,a)$ is the extrinsic component and $Q^i_\theta(s,a)$ is the intrinsic component.

Agent57, the most general of Atari-playing agents, is built from years of research and incremental improvements in the reinforcement learning field. It is the only algorithm to achieve at least human and generally superhuman performance in all 57 Atari games and marks another milestone in reinforcement learning produced by Deepmind.

3.5 Dota 2

From Backgammon [61] to Chess [17] to Atari [3], step-by-step complex games have been mastered. In 2016, AlphaGo showed the world the ability for artificial intelligence methods to surpass human intuition even in extremely complex perfect information games. In 2018, AlphaZero managed to accomplish this in various games without needing human supervision or intuition capturing engineering. Upon this success, it was deemed that the next major AI milestone was collaboration. With this in mind, OpenAI decided to embark on creating a team of agents to play the highly complex game of Dota 2.

Dota 2 is a popular online multiplayer real-time strategy video game. The game is conceptually simple: There are two teams consisting of 5 players each. Each team attempts to destroy a specific structure on their opponents' territory whilst defending their own. The complexity arises through the range of possible moves, the number of pre-game choices, the number of in-game choices and possible strategies that are needed to secure victory. The game also does not provide players with perfect information as the sight of enemy players is not always available. Due to this, algorithms such as AlphaZero, which use tree search, cannot accurately predict what the game will look like even one step ahead. Additionally, the complex rules and game logic make it challenging to develop game models even with the use of algorithms such as MuZero [49].

OpenAI's key solution to the additional problems presented by a domain such as Dota 2, was a seemingly common approach in the world of artificial intelligence. The solution was simply to scale existing reinforcement learning systems to unprecedented levels of compute, making use of thousands of GPUs over several months. OpenAI Five [7] is the name of the Dota 2-playing agent that ultimately went on to beat the only two-time world champions. For any Dota 2 agent to be created, the three main problems needed to be addressed are:

1. **Long Time Horizons:** Dota 2 games last on average 45 minutes with a 30 frames per second output. OpenAI Five limits its action selection to every fourth frame, which yields approximately 20,000 steps every episode. This is a stark difference to games such as Go or chess, which on average last 150 and 80 moves, respectively.

2. **Partial observability:** The only portion of the game state that is visible to a team is near their units and buildings, and the rest of the map is hidden and unobservable. Due to this, a crucial part of playing Dota 2 is the ability to infer situations based on incomplete visible data.

3. **High dimensionality of observation and action spaces:** In Dota 2, games are played on a large map containing ten unique characters, several buildings, infinitely spawning non-player units and a variety of complex game features that influence the game, such as trees, wards, etc. As a consequence of this, OpenAI Five approximately observes 16,000 values every time step. The action space is also extremely large, where depending on the character being played, the discretised action space is between 8000 to 80,000 actions.

These reasons make the game of Dota 2 an extremely challenging domain for humans and computers alike.

3.5.1 OpenAI Five

The creation of the OpenAI Five agent can be distinguished into two main stages; interaction and optimisation.

Interaction:

Every time-step (4th frame), OpenAI Five receives an observation from the game, which has been encoded to represent all the visible information available that a human player has access to. OpenAI Five uses this observation to return a discrete action that encodes the desired movement, attacks, etc. Due to the game's complexity, OpenAI Five relies on several human crafted scripts for specific game mechanics. This entails the order in which characters purchase items and choose abilities, the control of the unique courier unit and the selection of which items are kept in reserve. In hindsight, OpenAI believes that the system would be capable of higher performance without relying on these scripts, but this was unfortunately never investigated due to the computational and financial cost of retraining. The agents' policy functions π_θ use a parameterised recurrent neural network that receives a history of the agents' observations as input and outputs a probability distribution over all the possible actions. The recurrent neural network used is a single layer 4096-unit LSTM [23] with weights θ. For each of the five characters, separate policy functions are created that all use the same parameters θ. Since the available information for each character is almost identical, each character receives highly similar observations differing only in small features such as distance to the enemy. The OpenAI Five agent does not learn directly from pixels as usual in video games. Instead, the available information is summarised in a set of arrays. Due to this, the observations received by the agents are imperfect, with small pieces of information that are available to humans not being encoded. This might be seen as giving the agent an unfair disadvantage. However, the agent has access to all the given information simultaneously at every single time step, which is not feasible for a human. For the policy function to consider the unique differences of characters being played, the LSTM receives an extra input that indicates which of the five heroes is being controlled (see figure 21).

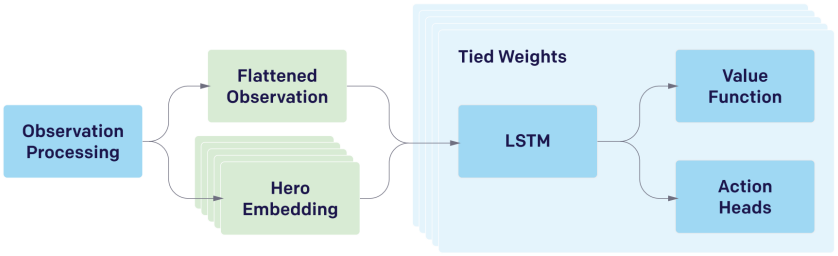

Figure 21: The following is taken directly from the original paper [7]. The complex multi-array observation space is processed into a single vector, which is then passed through a 4096-unit LSTM. The LSTM state is projected to obtain the policy outputs (actions and value function). Each of the five heroes on the team is controlled by a replica of this network with nearly identical inputs, each with its own hidden state. The networks take different actions due to a part of the observation processing's output indicating which of the five heroes is being controlled.

Optimisation:

In order to optimise the policy function π_θ, a reward function is needed. Due to the long time horizon of a Dota 2 game, credit assignment is a significant learning challenge. To combat this, a shaped reward function is created that does not only give rewards for winning or losing. Agents are given rewards for a set of actions that are deemed good by human players. This reward function instils human knowledge in the learning process, thereby giving the agents a more clear path of how to win. The algorithm used to train agents is PPO (see section 2.3.1). Additionally, the optimisation algorithm Generalised Advantage Estimation [51] is used to stabilise and accelerate learning. Much like AlphaZero, the neural network used by the agent outputs both the policy function and value function. Through self-play, experience is collected and stored in a centralised buffer similar to the buffer used in R2D2 [34]. A pool of optimiser GPUs receive data and store it locally within their own experience buffers. Each optimiser then samples mini-batches of experience and computes the network gradients. These separately calculated gradients are then averaged before being applied to the parameters. Due to this manner of calculating gradients, the effective batch size used is the batch size of each GPU, 120 samples, with 16 time-steps, multiplied by the number of GPUs used, which

was 1536 at the peak. Every 32 gradient steps, the new parameters are pushed to a central storage called the controller. Each Rollout worker runs self-play games in parallel to generate experience. An important note is that each rollout worker does not run the policy, only the game engine. The rollout workers communicate with a separate pool of GPU machines that run forward passes of the policy and value function network. These machines also frequently get the latest parameters from the controller.

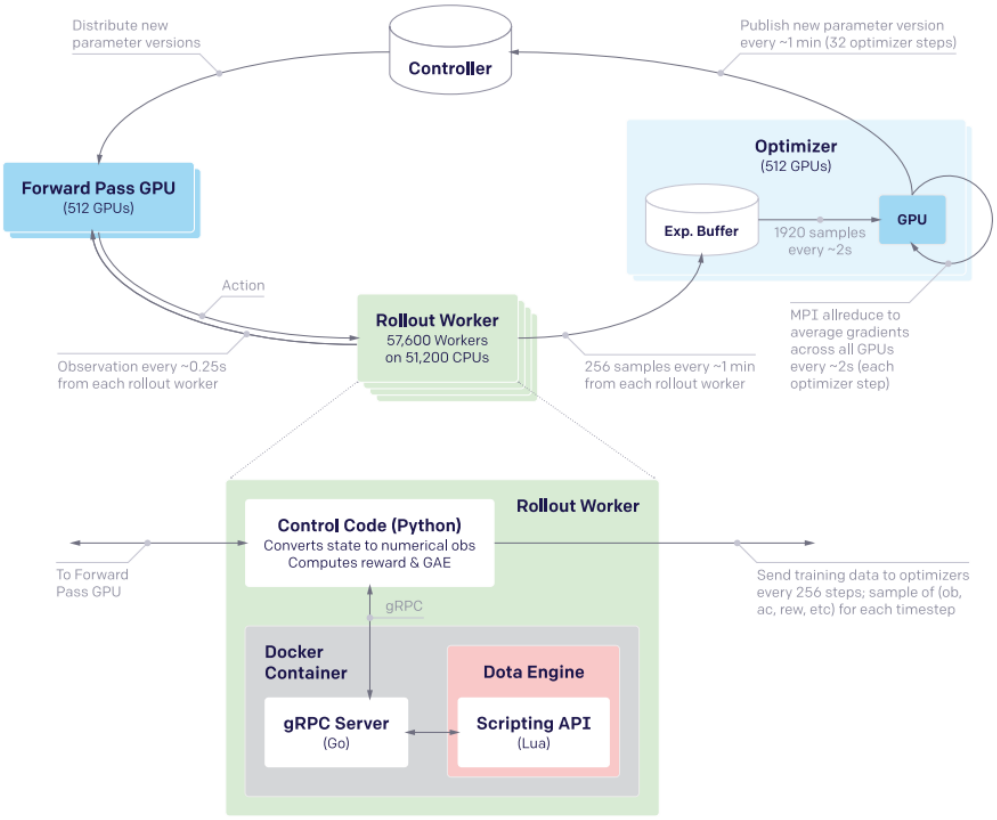

Figure 22: OpenAI Five System Overview

OpenAI launched the OpenAI Five Arena, in which OpenAI Five was exposed to the public for competitive online games to see if it could be regularly abused by creative or out-of-distribution play. OpenAI Five played 3,193 teams in 7,257 games and won 99.4% of the time. Additionally, OpenAI Five managed to defeat the first two-time world champions. This achievement has demonstrated that with significant compute, deep reinforcement learning can not only achieve superhuman performance in an incredibly complicated domain but can do so in collaboration dependent domains.

3.6 StarCraft II

When it comes to video games, the Atari suite can be seen as incredibly simple. Still, artificial intelligence methods struggled for years to achieve human or above performance in all 57 games. Video games offer a diverse range of problems that pose no real challenge to humans yet greatly prevent machine mastery. StarCraft, widely regarded as one of the most difficult Real-Time Strategy (RTS) games and one of the most popular esports of all time, has emerged as a "grand challenge" for AI research in recent years. Despite notable accomplishments in video games like Atari, Mario, and Dota 2, AI methods have struggled to cope with the complexity of StarCraft. Handcrafting significant aspects of the system, placing severe limits on the game rules, granting systems superhuman abilities,

or playing on simpler maps is what has allowed prior success. Despite these modifications and constraints, no algorithm has yet to come close to matching the ability of professional players. AlphaStar [67] is Deepmind's StarCraft-playing agent that is trained using supervised and reinforcement learning to play the entire game without any simplifying modifications.

StarCraft II pits two players against each other in a highly complex strategy game whereby each player controls up to 200 units at a time. At the start of the game, each player selects one of three *races*, Terran, Zerg, or Protoss. Each *race* has very distinct abilities and characteristics that define their play style and strategy. Both players start the game with basic units that gather resources to build structures, units and technologies. These creations allow a player to further bolster their military, economy and territory in the hopes of developing the capabilities to defeat their opponent. High-level play involves complex macro-management of their economy and territory and micro-management to control their individual units' movement and activities. This proves to be very difficult for AI systems as long-term, and short-term goals need to be balanced along with adapting to unexpected situations. The following are the challenges that have been deemed crucial to solve in order to create a StarCraft agent:

1. **Game Theory:** StarCraft does not present a single optimal strategy. Due to this, an AI system needs to explore and gain new strategic information continually as existing game-theoretic solutions cannot be applied.
2. **Imperfect Information:** StarCraft is an imperfect information game where pivotal information is hidden from the player. To discover this information in the game, the player needs to move units to *scout* areas that are invisible.
3. **Long Term Planning:** StarCraft actions do not have immediate consequences, and games can last up to one hour to finish. This means actions taken may only pay off far into the future.
4. **Real Time:** StarCraft games are played in real-time without a turn-based system. This means that players need to act within the game until it is finished continually.
5. **Large Action Space:** Since hundreds of units need to be controlled simultaneously, the combinatorial space of possibilities is vast. Additionally, actions, which are hierarchical in nature, can be augmented and modified, thereby increasing the action space further.

These challenges severely prevent the use of existing methods and pose incredible difficulty for computer mastery.

3.6.1 Alpha Star

AlphaStar, as with most modern reinforcement learning solutions, makes use of a deep neural network as a policy function $\pi_\theta(a_t|s_t, z)$ that receives its input directly from the raw game interface. The game interface essentially provides a list of units and properties that are visible to the agent. AlphaStar's neural network uses all previous observations $s_t = (o_{1:t}, a_{1:t})$ to output a sequence of instructions that represent an action within the game. AlphaStar's policy is also conditioned on a summarised strategy z that is sampled from human data. The specific architecture used is relatively complicated, using a variety of recent advances in deep learning research. The architecture consists of a transformer [66] (for observation processing) combined with an LSTM [29] (to aid with partial observability), an auto-regressive policy head [68] with a pointer network [69] (to manage the combinatorial action space), and a centralised value baseline [21]. AlphaStar additionally uses a novel multi-agent learning algorithm. Similarly to the original AlphaGo implementation, AlphaStar trains its deep neural network on human games to predict each action a_t. In this training process, the policy was either solely conditioned on s_t or also on z. This resulted in a diverse set of strategies similar to human play. The use of supervised learning intended to teach AlphaStar to imitate the basics of micro and macro-strategies. With supervised learning alone, AlphaStar managed to defeat a "gold" level player, the 5th highest rank, around 95% of the time. Subsequently, agents were trained using a new form of self-play and a reinforcement learning algorithm to maximise the win rate against various opponents. The reinforcement learning algorithm is an off-policy actor-critic algorithm [20] that makes use of experience replay, self-imitation [43] and policy distillation [44].

The discovery of novel strategies is one of the critical challenges in StarCraft. Due to the large action space, it is improbable that naive exploration will discover long-term strategies to aid the agents' victory. To encourage diverse and robust behaviour, as shown by human players, agents utilise human

data. Every agent is initialised with the parameters learnt by the initial supervised learning. During the reinforcement learning training process, each agent is either trained unconditionally or is conditioned on z, a human strategy. Agents conditioned on z are rewarded for following the human strategy, and non-conditioned agents are free to choose their own strategy. Human exploration ensures that various strategies and ideas are seen and are continued to be explored during the reinforcement learning training. The novel form of self-play used by AlphaStar is termed *league* training. Traditional self-play can result in cycle-chasing, e.g. agent A defeats agent B, who defeats agent C, but agent A loses to agent C. This results in the creation of agents who only can beat specific strategies. A form of self-play called Fictitious self-play (FSP) [37] manages to avoid these cycles through the computation of a *best-response* against a uniform mixture of all previous policies. This mixture is seen to converge to a Nash equilibrium in two-player zero-sum games. *League* training extends FSP to compute the best response from a non-uniform mixture of opponents. The *league*, which emulates a tournament-like structure, includes a wide variety of agents and their policies from current and previous iterations of training. Every iteration, agents play against opponents sampled from the league. Agent parameters are then updated from the outcomes of the games using the reinforcement learning algorithm. In normal self-play, each agent tries to increase its chances of winning against its opponents; however, this is only half a solution. In reality, a player who wants to get better at StarCraft could team up with partners to practice specific strategies. As a result, their training partners are not playing to win against every conceivable opponent but rather to expose their friend's shortcomings to help them grow as a player. A primary discovery in league training is that the idea of playing to win is not enough to create superhuman performance: we need main agents who want to win against everyone and exploiter agents who want to assist the main agent in getting stronger by revealing its vulnerabilities, rather than maximising their own win rate against all players. League training takes this idea and implements it into the self-play process. The league is comprised of three types of agents using differing opponent selection schemes:

1. **Main Agents:** The main agents use a prioritised FSP mechanism that adapts the mixture probabilities proportionally to the win rate of each opponent against the agent. The reason for this is to provide agents with an increased number of opportunities to defeat the most problematic opponents.

2. **Main Exploiter Agents:** The main exploiter agents competes solely against the current iteration of main agents. Their goal is to discover possible exploits in the main agents and encourage them to resolve such vulnerabilities.

3. **League Exploiter Agents:** League exploiter agents employ a similar Prioritised FSP technique as main agents but are not targeted by main exploiter agents. Their mission is to identify the entire league's systemic flaws.

Both main exploiters and league exploiters are re-initialised regularly to promote greater variety, and they may quickly find specialised techniques that are not always resistant to exploitation. *League* training takes population-based and multi-agent reinforcement learning further by creating a process that consistently explores the large strategic space of StarCraft. This process ensures all the agents know how to perform against the strongest strategies whilst not forgetting the earlier ones experienced.

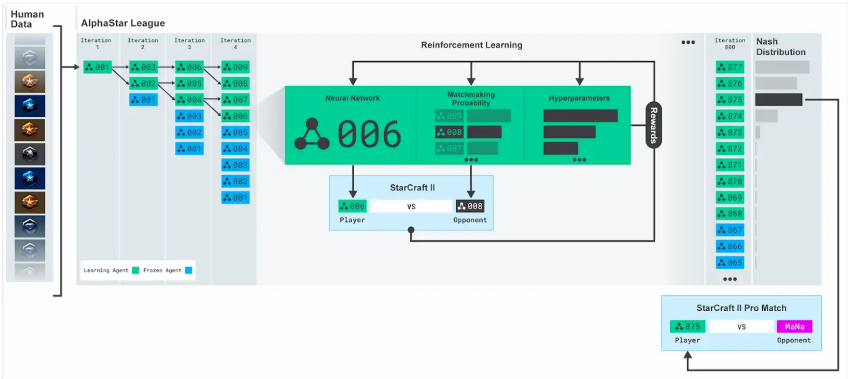

Figure 23: League Training Overview

The league was trained using a single main agent and main exploiter agent for each StarCraft *race* and six league exploiter agents. Each agent was trained using 32 TPUs over 44 days. During league training, almost 900 distinct players were created. The final AlphaStar agent achieved a Grandmaster ranking, the highest rank, in StarCraft II competitive play, placing it above 99.8% of ranked human players. This is the first computer program ever to reach this level of play.

4 Implementations

The following sections show the results of DQN, DDQN and Dueling Networks when applied to learn how to play the Atari game of Pong. All models are trained from pixel observations. In the game of Pong, the total score represents the difference between points scored by the agent and opponent with a max score of 21. All the code to my implementations can be found here: `https://github.com/EdanToledo/RL-Algorithms` and `https://github.com/EdanToledo/DuelingDDQN-and-AlphaZero`. Additionally, a minimal version of AlphaZero was implemented to learn the game of Connect Four.

4.1 DQN

Figure 24 shows the results of the DQN agent. This can be seen as a baseline to compare the improvements of algorithms such as Double DQN and Dueling Networks.

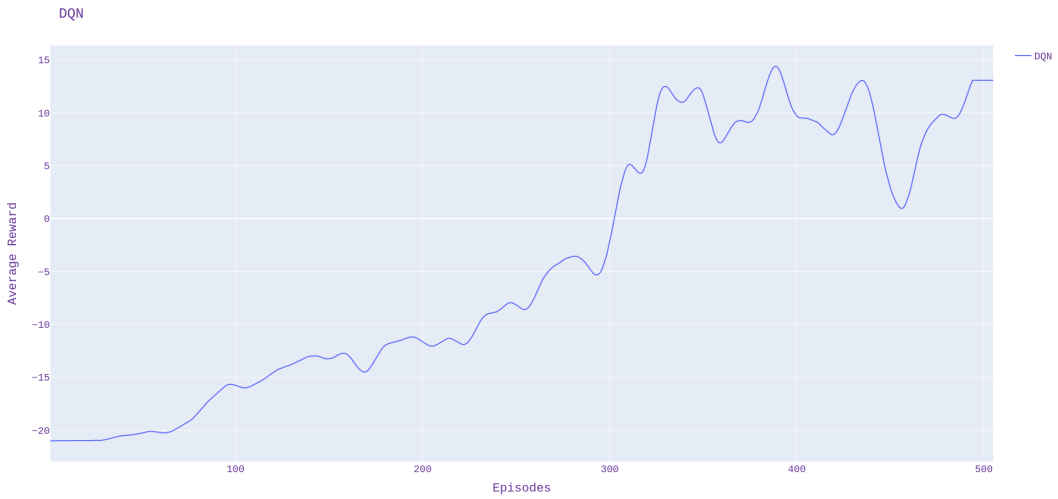

Figure 24: Running average of total scores achieved by the DQN

4.2 DDQN

Figure 25 shows the results of the DDQN agent. We can see that the DDQN agent has increased performance and requires less data to perform as well.

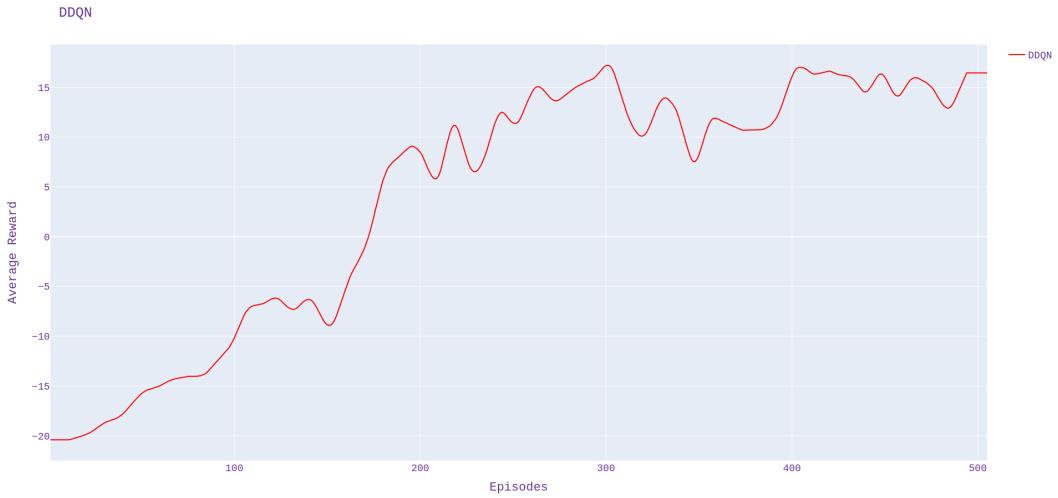

Figure 25: Running average of total scores achieved by the DDQN

4.3 Dueling DQN

Figure 26 shows the results of the Dueling DQN agent. We can see that the Dueling DQN agent has the highest performance thus far with the lowest requirement of data but experienced unstable training with high variation in performance as training progressed.

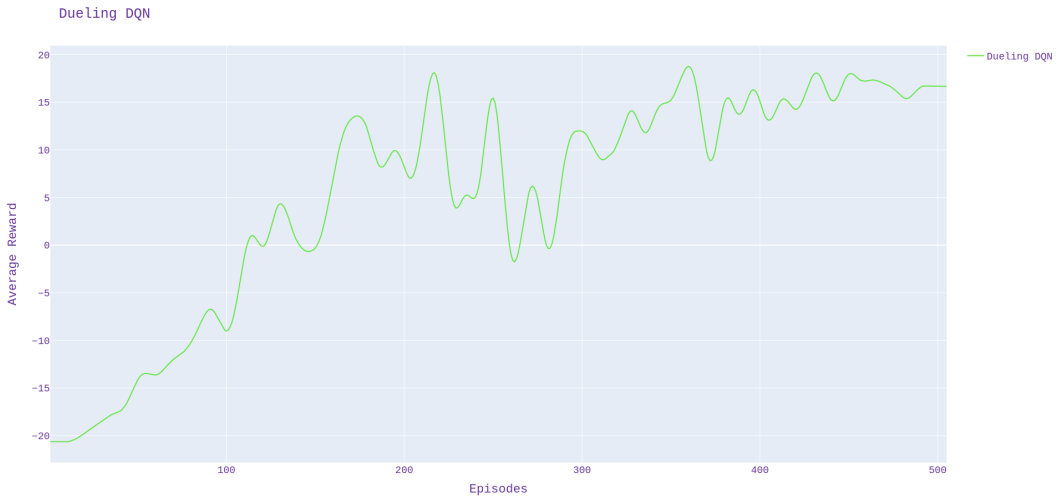

Figure 26: Running average of total scores achieved by the Dueling DQN

4.4 Dueling DDQN

Figure 27 shows the results of the Dueling DDQN agent. We can see that the Dueling DDQN agent performed slightly worse than the Dueling DQN agent (approximately the same as the DDQN agent) but had the most stable learning process out of all the agents.

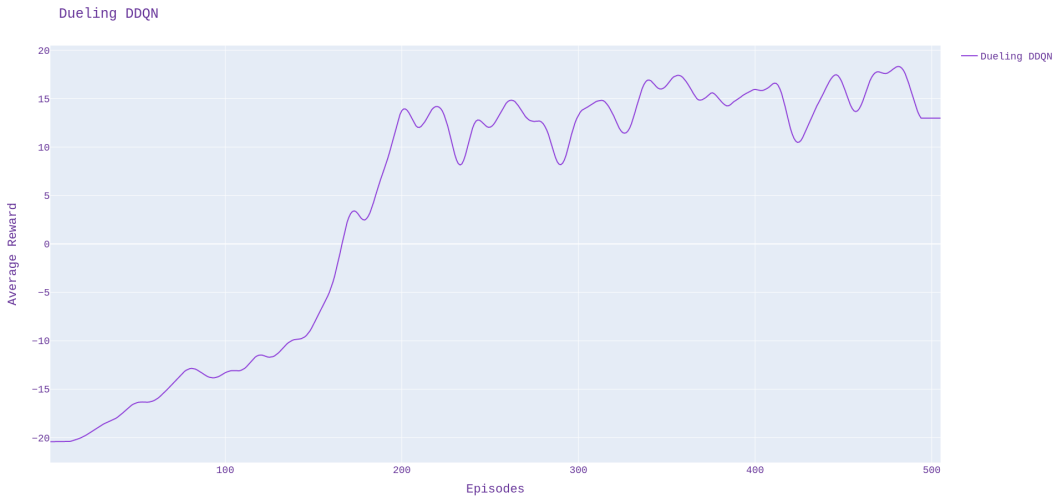

Figure 27: Running average of total scores achieved by the Dueling DDQN

Figure 28 shows the results of all the agents on a single plot. From this we can easily see that Dueling DDQN is the best performer as learning was stable and high performing.

4.5 AlphaZero

AlphaZero, after several hours of training, managed to capture key strategies in the game of Connect Four. Using a standard convolutional neural network architecture, the AlphaZero algorithm achieved

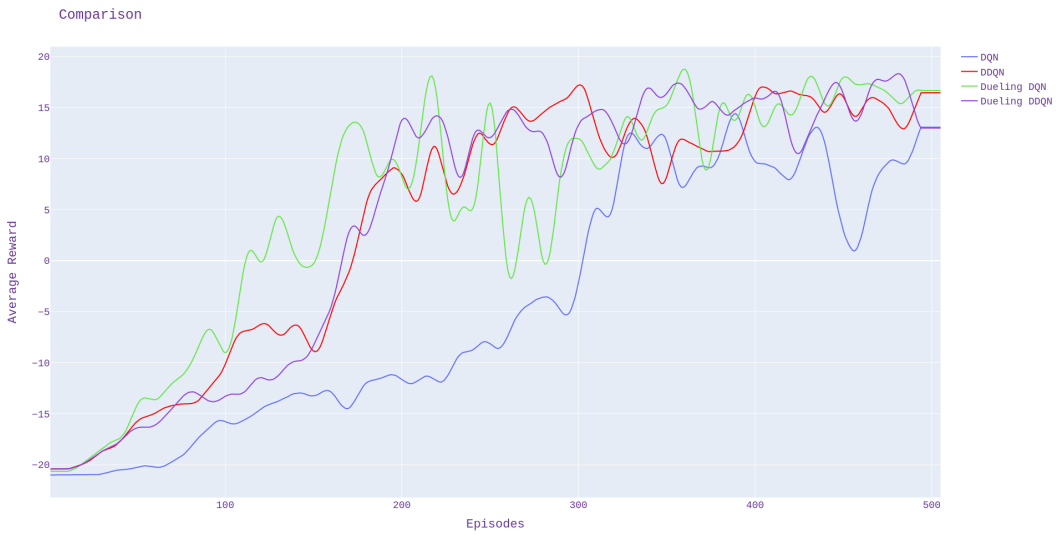

Figure 28: Comparison of all agents

strong human performance. This AlphaZero model has been made available to play against at `https://connect-four-vs-alpha-zero.vercel.app/`.

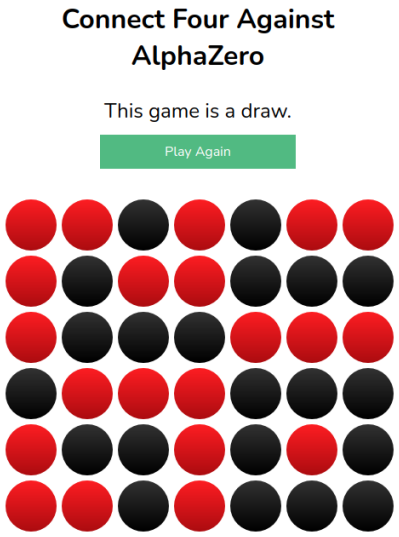

Figure 29: AlphaZero match ending in draw

References

[1] *AAAI'05: Proceedings of the 20th National Conference on Artificial Intelligence - Volume 4*. AAAI Press, 2005.

[2] Computer Shogi Association. Results of the 27th world computer shogi championship, 2017.

[3] Adrià Puigdomènech Badia, Bilal Piot, Steven Kapturowski, Pablo Sprechmann, Alex Vitvitskyi, Zhaohan Daniel Guo, and Charles Blundell. Agent57: Outperforming the atari human benchmark. *CoRR*, abs/2003.13350, 2020.

[4] Adrià Puigdomènech Badia, Pablo Sprechmann, Alex Vitvitskyi, Zhaohan Daniel Guo, Bilal Piot, Steven Kapturowski, Olivier Tieleman, Martín Arjovsky, Alexander Pritzel, Andrew Bolt, and Charles Blundell. Never give up: Learning directed exploration strategies. *CoRR*, abs/2002.06038, 2020.

[5] Petr Baudiš and Jean-loup Gailly. Pachi: State of the art open source go program. In H. Jaap van den Herik and Aske Plaat, editors, *Advances in Computer Games*, pages 24–38, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[6] Marc Bellemare, Sriram Srinivasan, Georg Ostrovski, Tom Schaul, David Saxton, and Remi Munos. Unifying count-based exploration and intrinsic motivation. In D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc., 2016.

[7] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemyslaw Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Christopher 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. Dota 2 with large scale deep reinforcement learning. *CoRR*, abs/1912.06680, 2019.

[8] Dimitri P. Bertsekas. Approximate policy iteration: a survey and some new methods. *Journal of Control Theory and Applications*, 9(3):310–335, July 2011.

[9] Michael Bowling, Neil Burch, Michael Johanson, and Oskari Tammelin. Heads-up limit hold'em poker is solved. 347(6218):145–149, January 2015.

[10] Michael Bowling, Neil Burch, Michael Johanson, and Oskari Tammelin. Heads-up limit hold’em poker is solved. *Science*, 347(6218):145–149, 2015.

[11] Noam Brown, Anton Bakhtin, Adam Lerer, and Qucheng Gong. Combining deep reinforcement learning and search for imperfect-information games. *CoRR*, abs/2007.13544, 2020.

[12] Noam Brown and Tuomas Sandholm. Baby tartanian8: Winning agent from the 2016 annual computer poker competition. In *IJCAI*, 2016.

[13] Noam Brown and Tuomas Sandholm. Superhuman AI for heads-up no-limit poker: Libratus beats top professionals. 359(6374):418–424, January 2018.

[14] Noam Brown and Tuomas Sandholm. Superhuman ai for multiplayer poker. *Science*, 365(6456):885–890, 2019.

[15] Noam Brown, Tuomas Sandholm, and Brandon Amos. Depth-limited solving for imperfect-information games. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[16] Yuri Burda, Harrison Edwards, Amos J. Storkey, and Oleg Klimov. Exploration by random network distillation. *CoRR*, abs/1810.12894, 2018.

[17] Murray Campbell, A.Joseph Hoane, and Feng hsiung Hsu. Deep blue. *Artificial Intelligence*, 134(1):57–83, 2002.

[18] Nuttapong Chentanez, Andrew Barto, and Satinder Singh. Intrinsically motivated reinforcement learning. In L. Saul, Y. Weiss, and L. Bottou, editors, *Advances in Neural Information Processing Systems*, volume 17. MIT Press, 2005.

[19] Yan Duan, Xi Chen, Rein Houthooft, John Schulman, and Pieter Abbeel. Benchmarking deep reinforcement learning for continuous control. *CoRR*, abs/1604.06778, 2016.

[20] Lasse Espeholt, Hubert Soyer, Rémi Munos, Karen Simonyan, Volodymyr Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, Shane Legg, and Koray Kavukcuoglu. IMPALA: scalable distributed deep-rl with importance weighted actor-learner architectures. *CoRR*, abs/1802.01561, 2018.

[21] Jakob N. Foerster, Gregory Farquhar, Triantafyllos Afouras, Nantas Nardelli, and Shimon Whiteson. Counterfactual multi-agent policy gradients. In *AAAI*, 2018.

[22] Ziming Gao, Yuan Gao, Yi Hu, Zhengyong Jiang, and Jionglong Su. Application of deep q-network in portfolio management. In *2020 5th IEEE International Conference on Big Data Analytics (ICBDA)*, pages 268–275, 2020.

[23] Felix A. Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to Forget: Continual Prediction with LSTM. *Neural Computation*, 12(10):2451–2471, 10 2000.

[24] Andrew Gilpin, Tuomas Sandholm, and Troels Bjerre Sørensen. A heads-up no-limit texas hold'em poker player: Discretized betting models and automatically generated equilibrium-finding programs. In *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 2*, AAMAS '08, page 911–918, Richland, SC, 2008. International Foundation for Autonomous Agents and Multiagent Systems.

[25] Richard H. R. Hahnloser, Rahul Sarpeshkar, Misha A. Mahowald, Rodney J. Douglas, and H. Sebastian Seung. Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit. *Nature*, 405(6789):947–951, June 2000.

[26] Hado Hasselt. Double q-learning. In J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems*, volume 23. Curran Associates, Inc., 2010.

[27] Matthew J. Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps. *CoRR*, abs/1507.06527, 2015.

[28] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[29] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, November 1997.

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

[31] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 448–456, Lille, France, 07–09 Jul 2015. PMLR.

[32] Michael Johanson, Kevin Waugh, Michael Bowling, and Martin Zinkevich. Accelerating best response calculation in large extensive games. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume One*, IJCAI'11, page 258–265. AAAI Press, 2011.

[33] Lukasz Kaiser, Mohammad Babaeizadeh, Piotr Milos, Blazej Osinski, Roy H. Campbell, Konrad Czechowski, Dumitru Erhan, Chelsea Finn, Piotr Kozakowski, Sergey Levine, Ryan Sepassi, George Tucker, and Henryk Michalewski. Model-based reinforcement learning for atari. *CoRR*, abs/1903.00374, 2019.

[34] Steven Kapturowski, Georg Ostrovski, Will Dabney, John Quan, and Remi Munos. Recurrent experience replay in distributed reinforcement learning. In *International Conference on Learning Representations*, 2019.

[35] Donald E. Knuth and Ronald W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975.

[36] Vijay R. Konda and John N. Tsitsiklis. OnActor-critic algorithms. *SIAM Journal on Control and Optimization*, 42(4):1143–1166, January 2003.

[37] David S. Leslie and E.J. Collins. Generalised weakened fictitious play. *Games and Economic Behavior*, 56(2):285–298, 2006.

[38] Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 8(3-4):293–321, May 1992.

[39] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[40] 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):529–533, 02 2015.

[41] Matej Moravčík, Martin Schmid, Neil Burch, Viliam Lisý, Dustin Morrill, Nolan Bard, Trevor Davis, Kevin Waugh, Michael Johanson, and Michael Bowling. Deepstack: Expert-level artificial intelligence in heads-up no-limit poker. *Science*, 356(6337):508–513, 2017.

[42] John Nash. Non-cooperative games. *Annals of Mathematics*, 54(2):286–295, 1951.

[43] Junhyuk Oh, Yijie Guo, Satinder Singh, and Honglak Lee. Self-imitation learning. *CoRR*, abs/1806.05635, 2018.

[44] Andrei A. Rusu, Sergio Gomez Colmenarejo, Çaglar Gülçehre, Guillaume Desjardins, James Kirkpatrick, Razvan Pascanu, Volodymyr Mnih, Koray Kavukcuoglu, and Raia Hadsell. Policy distillation. *CoRR*, abs/1511.06295, 2016.

[45] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[46] Bruno Scherrer. Approximate policy iteration schemes: A comparison. In Eric P. Xing and Tony Jebara, editors, *Proceedings of the 31st International Conference on Machine Learning*, volume 32 of *Proceedings of Machine Learning Research*, pages 1314–1322, Bejing, China, 22–24 Jun 2014. PMLR.

[47] Jürgen Schmidhuber. A possibility for implementing curiosity and boredom in model-building neural controllers, 1991.

[48] Jürgen Schmidhuber. Formal theory of creativity, fun, and intrinsic motivation (1990–2010). *IEEE Transactions on Autonomous Mental Development*, 2(3):230–247, 2010.

[49] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy Lillicrap, and David Silver. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, December 2020.

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

[51] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation, 2018.

[52] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[53] 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. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, January 2016.

[54] 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. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[55] 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. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, October 2017.

[56] David Silver, Satinder Singh, Doina Precup, and Richard S. Sutton. Reward is enough. *Artificial Intelligence*, 299:103535, 2021.

[57] Oluwatobi Sofela. Minimax algorithm guide: How to create an unbeatable ai, Apr 2021.

[58] Richard S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *In Proceedings of the Seventh International Conference on Machine Learning*, pages 216–224. Morgan Kaufmann, 1990.

[59] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[60] Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In S. Solla, T. Leen, and K. Müller, editors, *Advances in Neural Information Processing Systems*, volume 12. MIT Press, 2000.

[61] Gerald Tesauro. Temporal difference learning and td-gammon. *Commun. ACM*, 38(3):58–68, March 1995.

[62] Sebastian Thrun and Anton Schwartz. Issues in using function approximation for reinforcement learning. In *In Proceedings of the Fourth Connectionist Models Summer School*. Erlbaum, 1993.

[63] Sebastian Thrun and Anton Schwartz. Issues in using function approximation for reinforcement learning. In Michael Mozer, Paul Smolensky, David Touretzky, Jeffrey Elman, and Andreas Weigend, editors, *Proceedings of the 1993 Connectionist Models Summer School*, pages 255–263. Lawrence Erlbaum, 1993.

[64] John N. Tsitsiklis and Benjamin Van Roy. An analysis of temporal-difference learning with function approximation. Technical report, IEEE Transactions on Automatic Control, 1997.

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

[66] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[67] Oriol Vinyals, Igor Babuschkin, Wojciech M. Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H. Choi, Richard Powell, Timo Ewalds, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja Huang, Laurent Sifre, Trevor Cai, John P. Agapiou, Max Jaderberg, Alexander S. Vezhnevets, Rémi Leblond, Tobias Pohlen, Valentin Dalibard, David Budden, Yury Sulsky, James Molloy, Tom L. Paine, Caglar Gulcehre, Ziyu Wang, Tobias Pfaff, Yuhuai Wu, Roman Ring, Dani Yogatama, Dario Wünsch, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy Lillicrap, Koray Kavukcuoglu, Demis Hassabis, Chris Apps, and David Silver. Grandmaster level in StarCraft II using multi-agent reinforcement learning. 575(7782):350–354, October 2019.

[68] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John P. Agapiou, Julian Schrittwieser, John Quan, Stephen Gaffney, Stig Petersen, Karen Simonyan, Tom Schaul, Hado van Hasselt, David

Silver, Timothy P. Lillicrap, Kevin Calderone, Paul Keet, Anthony Brunasso, David Lawrence, Anders Ekermo, Jacob Repp, and Rodney Tsing. Starcraft II: A new challenge for reinforcement learning. *CoRR*, abs/1708.04782, 2017.

[69] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc., 2015.

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

[71] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*, pages 1995–2003. PMLR, 2016.

[72] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3-4):279–292, May 1992.

[73] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3-4):229–256, May 1992.

[74] Weirui Ye, Shaohuai Liu, Thanard Kurutach, Pieter Abbeel, and Yang Gao. Mastering atari games with limited data, 2021.

[75] Xingdi Yuan, Marc-Alexandre Côté, Jie Fu, Zhouhan Lin, Chris Pal, Yoshua Bengio, and Adam Trischler. Interactive language learning by question answering. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 2796–2813, Hong Kong, China, November 2019. Association for Computational Linguistics.

[76] Martin A. Zinkevich, Michael Bowling, and Michael Wunder. The lemonade stand game competition: Solving unsolvable games. *SIGecom Exch.*, 10(1):35–38, March 2011.