Overview of deep reinforcement learning in partially observable multi-agent environment of competitive online video games

Jaana Louhio

Helsinki November 3, 2020

UNIVERSITY OF HELSINKI
Department of Computer Science

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Studieprogram — Study Programme
Faculty of Science	Computer Science

Tekijä — Författare — Author
Jaana Louhio

Työn nimi — Arbetets titel — Title
Overview of deep reinforcement learning in partially observable multi-agent environment of competitive online video games

Ohjaajat — Handledare — Supervisors
Dorota Glowacka

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Master's thesis	November 3, 2020	69 pages + 0 appendices

Tiivistelmä — Referat — Abstract

In the late 2010's classical games of Go, Chess and Shogi have been considered 'solved' by deep reinforcement learning AI agents. Competitive online video games may offer a new, more challenging environment for deep reinforcement learning and serve as a stepping stone in a path to real world applications. This thesis aims to give a short introduction to the concepts of reinforcement learning, deep networks and deep reinforcement learning. Then the thesis proceeds to look into few popular competitive online video games and to the general problems of AI development in these types of games. Deep reinforcement learning algorithms, techniques and architectures used in the development of highly competitive AI agents in Starcraft 2, Dota 2 and Quake 3 are overviewed. Finally, the results are looked into and discussed.

ACM Computing Classification System (CCS):
Computing methodologies → Machine learning → Learning paradigms → Reinforcement learning → Multi-agent reinforcement learning
Computing methodologies → Machine learning → Machine learning approaches → Partially-observable Markov decision processes
Computing methodologies → Machine learning → Machine learning approaches → Neural networks

Avainsanat — Nyckelord — Keywords
deep reinforcement learning, competitive online video games, POMDP, multi-agent systems

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Thesis for the Algorithms, Data Analytics and Machine Learning subprogramme

Contents

1 Introduction

Online video games of various flavor have been played competitively for almost three decades. Real-time strategy, first-person shooter, multiplayer online battle arena and driving games have captivated their audiences for decades. Popular competitive online video games, such as Counter-Strike, Fortnite, Dota 2 and Starcraft 2 are being played competitively by professional players. These games are starting to move to the forefront of research in deep reinforcement learning, a form of machine learning in which an agent must learn by trial and error an optimal behaviour by interacting with its environment combined with deep neural networks.

DeepBlue [CHJH02] defeated the grandmaster of chess in late the 1990's and AlphaGo [SHM+16] defeated best Go player in the world in 2010's. Further, AlphaZero [SHS+18] improved upon AlphaGo to such an extent that it managed to defeat the best players in Chess, Go and Shogi, and did so with just self-play – without requiring any prior knowledge like DeepBlue and AlphaGo did. These results proved that classical two player adversarial games with fully visible game state and limited amount of moves to be made are 'solved' with deep reinforcement learning techniques.

This brings us to the new frontier: competitive online video games. They offer a new challenge with games taking tens of thousands of moves with dozens of different game pieces and games being played in real time with partially observable varying environments. These games may even include more than two players competing and cooperating simultaneously, creating a challenging multi-agent environment. These games are an incredible challenge to play even for the best professional human players. Competitive online video games as an reinforcement learning environment have also one particularly tempting upside: they offer a ready environment to test the performance of the AI solutions by a steady stream of human competitors.

There have been attempts to 'solve' or at least offer a semi-competitive AI with reinforcement learning techniques throughout the 2000's and 2010's, but these have failed to provide any sort of competition to good players, let alone to best the professional players. This has been until the rise of deep learning techniques which use artificial neural networks with several connected layers. Combining deep networks with reinforcement learning has led to encouraging results in several different categories of competitive online video games. AlphaStar [VBC+19], a deep reinforcement learning AI agent for real-time strategy game of Starcraft2, managed to reach

Grandmaster rank, the highest rank in the online ranked ladder of the game. OpenAIFive [OB+19] managed to defeat the best professional human team in a restricted version of Dota 2, a multiplayer battle-arena game. FTW agent [JCD+19a] was able to reach level of strong human player in a capture the flag gameplay in first person shooter game Quake3. These results in older first person shooter game could be encouraging in an attempt to solve much more rich environments like Counter-Strike and Fortnite. During the writing of this thesis, there were also results published in a driving game of Gran Turismo [FSK+20]. Results in these rich and complex online video game environments could also serve as a stepping stone to even harder real-life environments like robotics and computer vision citevoulodimos2018deep.

The aim of this thesis is to provide an overview of the concepts of reinforcement learning, deep networks and deep reinforcement learning. Then, to overview the challenges of competitive online video games as a partially observable reinforcement learning environment and then look into how new techniques have recently performed recently in these games. Second and third chapter aim to introduce the concepts of reinforcement learning and deep networks with some theoretical examples. The fourth chapter overviews how the combination of these techniques, the deep reinforcement learning, was applied and how the AI agents performed in the challenging environments of competitive online video games of Starcraft 2, Dota 2 and Quake 3. Finally, the fifth last chapter contains a discussion on these results with their pros and cons and what these results mean today and what they could mean in the future. Chapter 6 concludes this thesis.

2 Reinforcement learning

Reinforcement learning [KLM96, SB18] is a form of machine learning in which an *agent* must learn by trial and error an optimal behaviour by interacting with its environment. An agent starts interacting with the *environment* with an arbitrary *policy* for choosing *actions*. On each step of interaction, the agent receives an input of some indication of the *state* of the environment. Then the agent chooses an action to perform to generate output and change the state of the environment. The agent then receives *rewards* (positive reinforcement) when the actions lead to successful performance and *penalty* (negative reinforcement) when its actions lead to unwanted results. As the agent explores its environment, its performance changes from nearly random to nearly deterministic as it finds routes to high rewards and learns to avoid

actions that result in punishment. In the long run, agent should choose its actions to maximize its received reinforcement. This is called *optimal policy* and it can be found by trial and error, guided by a wide variety of different algorithms.

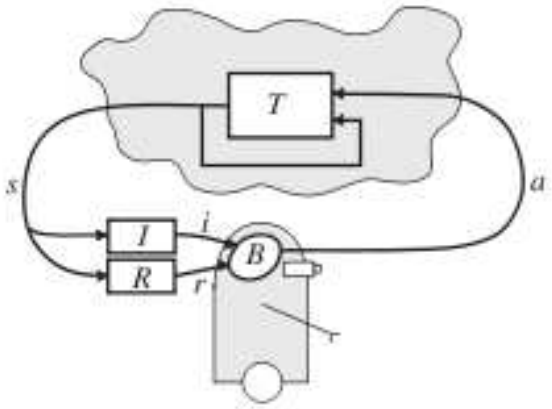


Figure 1: The standard reinforcement learning model.I is the input function of the agent that it uses to perceive the state s of the environment T. R is its reinforcement model from which agent receives scalar reinforcement r after each action a according to the new, perceived, state s of the environment T. B is its "brain" that contains its policy π, which maps agents actions a to states s of environment T. Aim of the agent is to maximize received reinforcement over time, this is done by so called 'optimal policy', which is learned by trial-and-error by acting in the environment.

One of the biggest problems in reinforcement learning is that the reinforcement-learning agent must explicitly explore its environment. The problem lies in the question: To which extend should the agent exploit its knowledge and to which extend should it explore? Should the agent choose the best known action always or should it sometimes choose an action, which is less explored but seems worse at this time? This problem is often referred to as *exploration/exploitation trade-off* and is discussed in more detail in chapter 2.2.

Reward assignment if another important concept in reinforcement learning. Agent learns by receiving a scalar reinforcement signal after each time step when it performs an action in its environment. A positive reinforcement signal, a reward, is received upon performing an action that moves the state of the environment closer to a desired outcome and negative reinforcement signal, a punishment, is received upon undesired behaviour. Agent evaluates its behaviour at each state for each action by these reinforcement signals. Over time, agent learns to optimize its actions in a trial-and-error fashion, learning to maximize its reward. In the reward assignment lies

another big problem of reinforcement learning: How to reward or punish the agent for its performance when the results of it's actions can be seen only after a long time period? Should agent be given single reward at the end of the learning epoch and propagate this reward back in time? This rewarding scheme is called *delayed reinforcement*. Or should agent be rewarded after each action at each time step if it moved towards its goal? This is called *immediate reinforcement*. Depending on the reinforcement learning problem one of these might work better or a mixture of both. [SB18, KLM96].

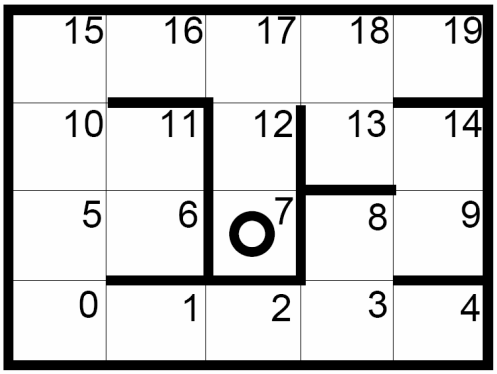


Figure 2: A simple reinforcement learning problem example known as the Maze20. This is a simple discrete world where an agent can move from one square to a square next to it to any direction as its actions. Agent can't move through walls denoted by thick lines and if it attempts to do so, it fails and stays put. Environment consists of the numbered square. The agent starts at a random position and it must find shortest path to the square marked with a circle. To ensure agent tries to find the shortest path we give agent reinforcement of -1 for each move and reinforcement of 100 for reaching the goal. This ensures agent to try to minimize steps taken, because each move gives it penalty. At each time step agent tries to move from one square in the maze to another and receives this new square (or the same if it tried to move against a wall) and reinforcement of -1 (or 100 if it reached the goal) as input. Agents policy is a mapping of a movement to a certain direction from each square in the maze. An optimal policy is the policy that maximizes the received reinforcement signal, which is the shortest path to the goal. At the beginning of the learning period agent starts exploring the maze at random, it knows not where the goal is or if it can move through walls. In time, it will learn to act in the environment and the shortest path by trial and error through many learning periods.

Over the last three decades there have been many applications of reinforcement

learning in classical board games. An example of applying reinforcement learning for games, an algorithm called temporal difference learning[Sut88] has been successfully applied to solving the classic game of Backgammon with TD-Gammon[Tes95] in the mid 90's . Later in the late 90's IBM's deep blue chess playing computer defeated reigning chess world champion Gary Gasparov for the first time ever [CHJH02]. For decades, game of Go still remained elusive for computers to play at human, let alone super human level, due to its computational complexity and depth. After the advent of deep learning techniques and increased computational capacity, new strides for classic games have been made. In 2016 even the GO was mastered by AlphaGO[SHM$^+$16] and the best human GO player was defeatead. Two years later, a general deep learning algorithm called AlphaZero, plays Go, Chess and Shogi at superhuman level [SHS$^+$18]. Now, researchers look to apply these now techniques for solving competitive video games where new challenges can be found.

2.1 Markov decision process

In a single agent reinforcement learning the process can be formulated as an Markov decision process (MDP) [KLM96, SB18]. MDP is formally defined as a tuple $< \mathbf{S}, \mathbf{A}, T, \mathbf{R}, \gamma >$, where $\mathbf{S}$ is finite and discrete state space and $\mathbf{A}$ is the finite discrete action set for the agent. $T : \mathbf{S} \times \mathbf{A} \rightarrow \prod(\mathbf{S})$ is the state transition function which maps each state $s \in \mathbf{S}$ and the action $a \in \mathbf{A}$ of the agent to the next state. State transition function may be stochastic and define a probability distribution over the state space $\mathbf{S}$. $R_i : \mathbf{S} \times \mathbf{A} \rightarrow \mathbb{R}$ is the reward function for the agent and $0 \leq \gamma < 1$ is the discount factor for future rewards.

The aim of the agent is to maximize the sum of its expected gained reinforcement over time. The maximized reinforcement is given by the *optimal policy*:

$$\pi^*(s) = \arg\max_a (r(s,a) + \gamma \sum_{s' \in S} p(s,a,s') V^*(s')), \tag{1}$$

where $p(s,a,s')$ is state transition probability . Policy can be described as a mapping of states to actions that give the maximal long term reward.

2.2 Value function and solving optimal policy

Agents optimal policy π^* can be solved from *optimal value function* [KLM96, SB18]:

$$V^*(s) = \max_a \Big(R(s,a) + \gamma \sum_{s' \in S} T(s,a,s') V^*(s') \Big), \forall s \in S. \tag{2}$$

From the value function we can see that the value of the state s is immediate reward plus the next state's value reduced by a factor of γ. From this we can solve for the optimal policy π by choosing to perform action a in state s, which maximises the value of the optimal value function V^* in state s. Then, the optimal policy π^* is in fact a greedy policy where actions are chosen by the optimal value function. [KLC98].

Optimal value function can be solved algorithmically with *value iteration* [KLM96]. It performs iterative improvement to value function setup with arbitrary values. It tries to get as close to optimal value function as possible and stops [KLM96]. It has been shown that value iteration algorithm converges to correct optimal value function values [Bel66]. There is no guarantee as to how many iterations are needed for the convergence to optimal value function, which can be costly on the processing time. However, greedy policy performed from the value iteration algorithm is optimal long before value iteration algorithm has converged [KLM96].

Another way of solving optimal policy is policy iteration algorithm. It works by directly interacting with the policy. It starts with a arbitrary policy and improves it iteratively, until optimal policy is achieved [KLM96].

Both algorithms are exact algorithms that solve exactly for the optimal policy. The problem with these methods, along with other methods, is computational intensity. These algorithms need vast memory space to store all the state, action pairs along with the massive state space. Usually, some problem specific methods are needed to manage the computational and memory requirements to achieve successful results. This limits the universal applicability of these methods [KLM96].

An example of value function is Q-function. In Q-learning [WD92] the agent executes action a in the current state s, then receives an immediate reward r and the next state s'. The current estimated Q-value can then be updated with the following equation:

$$Q_{k+1}(s,a) = (1-\alpha_k)Q_k(s,a) + \alpha_k[r + \gamma \max_{a \in A} Q_k(s',a)], \tag{3}$$

where k is the current time step, $0 \leq \alpha_k \leq 1$ is the learning rate and $0 \leq \gamma \leq 1$ is the decaying factor. Learning rate controls how much the new experiences affect the agent. Decaying factor decreases the impact of future rewards gained from this state. Only the value for the current state and action are ever updated, other states remain unchanged. If the state-action pairs are updated an infinite number of times, that is $k \to \infty$, and decaying factor is $\alpha_k \to 0$. For Q-values the following is true: $V^*(s) = \max a Q^*(s;a)$ [KLM96]. Then, Q_k converges to the optimal policy with

probability of 1 [WD92].

As mentioned in chapter 2.1, exploration/exploitation trade-off dilemma is one of the biggest problems in reinforcement learning. How long should agent try to explore its environment more to learn a better policy for choosing actions and when to exploit its current known best policy to try to maximize its reward? To answer the exploration/exploitation trade-off dilemma, the agent may choose its action according to different action selection strategies. One such strategy is to choose action a in state s according to the greedy policy, where the action that leads to greatest reward is chosen. Or $\epsilon - greedy$ policy, where the action that leads to the greatest reward is chosen with probability of ϵ and a random action with probability of $1 - \epsilon$.

One popular biased action selection strategy is Boltzmann exploration [KLM96]. In state s an action is selected according to the following probability:

$$p(a) = \frac{e^{\frac{Q(s,a)}{T}}}{\sum_a e^{\frac{Q(s,a)}{T}}}, \tag{4}$$

where T is the *temperature* parameter. T is selected in such way that the exploration is at first almost completely random and T decays so that in the end actions are chosen greedily. This can be achieved by choosing $T = \frac{T_0}{b+1}$, where T_0 is a constant and b is the number of actions finished by the agent.

2.3 Partially observable environments

In many real-world environments and in video games, it will not be possible for the agent to have perfect and complete perception of the state of the environment. This can happen, for example, in a real-time strategy game (rts) when a fog of war covers the map, obscuring visibility of enemy units. Unfortunately, for many of the reinforcement learning algorithms, complete observability is necessary. Further, in video games state space of the vast three dimensional environments might not be finite. In such cases, state space must be discretized and made finite or there can be no guarantee of finding optimal policy. Still, reinforcement learning has been used successfully in many real-world applications [SB18].

Partially observable Markov decision process (POMDP) is an extension MDP. Key difference for reinforcement learning in POMDP is learning additional model parameters. T is the probability of moving from state s to s', O is the probability distribution for observations, and $R = (s, a, s')$ is the reward function, where now,

due to uncertainty, must also be considered the state s' agent ended up in by performing action a in state s. In POMDP the agent does not know exactly in which state it is so it must maintain belief state, which is a probability distribution over the states in $\mathbf{S}$. It is updated after each action $a \in \mathbf{A}$ according to the received observation $o \in \mathbf{O}$ about the next state $s' \in \mathbf{S}$. Optimal policy π can then be solved by value iteration algorithms. Another simpler way is to return POMDP back to MDP and using reinforcement learning methods for MDP. This is done by using most likely belief state b as the true state s. This solution works well, it the state s can be separated form other states in S with high probability. The greater the uncertainty about the state s is, the closer distribution of belief state b is to a uniform distribution and the worse this solution works [KLM96, SB18].

Formally, we can define partially observable Markov decision process (POMDP) as a tuple $< S, A, T, R, \Omega, O, \gamma >$, where $\mathbf{S}$ is finite and discrete state space and $\mathbf{A}$ is the finite discrete action set for the agent. $T : \mathbf{S} \times \mathbf{A} \to \prod(\mathbf{S})$ is the state transition function which maps each state $s \in \mathbf{S}$ and the action $a \in \mathbf{A}$ of the agent to the next state. State transition function may be stochastic and defines a probability distribution over the state space $\mathbf{S}$. $R_i : \mathbf{S} \times \mathbf{A} \times \mathbf{S} \to \mathbb{R}$ is the reward function for the agent and $\Omega = \{o_1, o_2, ..., o_j, ...\}$ is the space of observations received from the environment. $O(o|s', a)$ is probability distribution for the observations, where $s' \in S$ is the state where agent moved after performing action $a \in A$ and $o \in \Omega$ is the observation agent received of its new state s'. And finally, $0 \leq \gamma < 1$ is the discount factor for futere rewards.

Here we can see that the addition of POMDP to MDP is that the observation agent receives after change of state is uncertain. And because the agent has uncertainty about the state it arrives in, this model need some other way to represent agents state instead of a single state s. A good candidate is *belief state* b, which is a probability distribution over all the possible states $s \in S$. $b(s)$ is probability that the environment is in state s. It is usually not accurate enough to represent the most probable state as the state agent is at, but also represent the amount of uncertainty there is about the state. The closer the distribution of states is to uniform distribution, the greater the uncertainty of the state is. On the other hand, if the probability mass is distributed mostly to just one state the certainty of the state is high. POMDP can be returned to MDP, when we assume that observation o represents arrival at state s' and probability of observation is $O(o|s', a) = 1$. Then we assume that the belief state always represents the true state of the environment and the observation as having perfect information.

Belief state is agents internal belief about the state of the environment. The environment is always at certain state, but the agents belief of the state can be different. Agent acts in the environment by performing actions and changing the state of the environment. After this agent always receives observation of the environment and updates its belief state. The longer the agent acts in the environment and by using history from previous observation (eg. RNN or LSTM discussed in chapter 3) the agents belief state becomes more accurate. Belief state can be updated easily, for example, by using Bayes' Theorem $P(A|B) = \frac{P(B|A)*P(A)}{P(B)}$.

Learning solutions in POMDP are divided in two categories: solutions with memory and solutions without memory. Learning in POMDP without memory is usually very inefficient and are shown that learning the optimal policy without memory in POMDP is an NP-difficult problem [Hau00]. In solutions with memory, agents can learn a better understanding of their environment instead of trying to 'guess' according to uncertain observations. If same kind of observations are made in different states, it can prove a difficult learning environment. Best solution is to keep a complete history for the agent, since it gives the best possibility to figure out the agents true state in the environment surrounded by uncertain information. However, keeping track of agents complete history can prove impossible due to memory restrictions and solutions to compact it are usually needed. Simplest solution is to limit memory to a number of most recent states, but such heuristic method has no guarantee of convergence to an optimal policy. A better way to utilize memory is to use artificial neural networks. They can be used to efficiently propagate in time, even to the start of the learning process. They are introduced in chapter 3 Deep Learning.

Because solving optimal policy in POMDP without memory has proven to be NP-difficult problem, so the only solution appears to be to approximation algorithms. There have been proposed multiple heuristic approaches. But for a long time, these approximate algorithms are too inefficient, they often fail to find the optimal policy or take too long time to converge. Biggest issue to widespread application reinforcement learning techniques to POMDP environments, like real-world applications and complex video games, is the computational complexity. Deep recurrent neural networks, overviewed in chapter 3 of this thesis, provide the memory needed for these algorithms to function properly in POMDP. This combined with reinforcement learning and added computational power have proven to provide much needed assistance to these issues.

2.4 Solving optimal policy in partially observable environments

Usually it is easiest to solve reinforcement learning problems in POMDP model free fashion, which means that we try not to learn the individual parameters of the environment but rather try to just learn the optimal policy in the environment. Unlike MDP, in POMDP the number of belief states is continuous, so the exact algorithms cannot converge to optimal policy. Limiting timescale for belief states, on the other hand, has proven to be a PSPACE-difficult problem. However, there are approximate algorithms that can produce good results with reasonable time and space requirements [SB18].

Optimal policy can be solved with optimal value function like in MDP. Optimal value function is defined as follows [Hau00]:

$$V^*(b) = \max_{a \in A} \Big\{ \sum_{s \in S} R(s,a)b(s) + \gamma \sum_{o \in O} \sum_{s' \in S} O(o|s',a)V^*(\tau(b,o,a)) \Big\}, \tag{5}$$

where next belief state is b' on

$$b'(s) = \tau(b,o,a)(s) = \beta \sum_{s' \in S} O(o|s',a)b(s'). \tag{6}$$

β is the normalizing constant so that $0 \leq b'(s) \leq 1$.

The difference in optimal value function between MPD and POMDP is that now we must iterate over all the possible observations $o \in O$ we can have when moving from state $s \in S$ to $s' \in S$ with action $a \in A$. State has been replaced by the belief state b. The number of belief states is infinite, which makes optimal value function computationally impossible to exactly solve, but it can be approximated. One way to approximate it is to present value function by parts of convex linear function $\alpha_t(s)$ [Hau00]. Convex linear value function for time t is:

$$V_t(b) = \max_{\alpha_t \in \Gamma_t} \sum_{s \in S} b(s)\alpha_t(s), \tag{7}$$

where b and α_t are $|S|$ dimensional vectors and Γ_t is finite set of linear vectors α_t. The number of these linear vectors grow exponentially in time. This makes exact solutions for problems with large state and observation spaces computationally very expensive.

Another way to approximate optimal value function is to use value iteration algorithm and statistical sampling to discretize belief state [Thr00, SB18]. Then, a

balance between computational intensity and sample size must be struck, so that neither speed nor accuracy suffers too much. Statistical sampling methods need a large enough sample size to be accurate. Approximate value iteration algorithms are less exact, but less computationally intensive as convex linear functions, so they are capable of solving more complex problems. But even value iteration algorithms have lacked the ability to tackle the most complicated learning tasks.

Monte Carlo method is an effective way of statistically sampling belief states and solving reinforcement learning problems in POMDP, when the problem is episodic in nature [Thr00, JSJ95]. When using model-free methods we do not have the knowledge of properties of the environment or specifically the transition probabilities between states $p(s, a, s')$. We would need to learn all the information by exploring the environment. Monte Carlo method works instead by updating values directly to an arbitrary policy π for selecting actions. It runs a full learning episodes (or simulations) from start to finish. After each episode, rewards are propagated back in time to each state visited. Value for an action at a state is updated with received discounted reward that is weighed with the times this state has been visited. From this we can construct a greedy policy by selecting actions to perform with the highest value of value function at each state. For Monte Carlo method, there is no guarantee of convergence to optimal policy, but it has good convergence properties, when a large enough number of episodes can be run for each time step. Monte Carlo methods are simple to use and have been shown to work well in many episodic problems, like classic games and video games. They however, suffer from high variance and each episode must be fully completed.

2.5 Multi-agent reinforcement learning

Many competitive online video games are an example of a problem where multi-agent reinforcement learning can be used. Multi-agent reinforcement learning is split into competitive, cooperative and semi-cooperative tasks. Video games are usually either semi-cooperative, where some agents are teammates and other agents opponents or fully competitive, where one agent competes against another agent. The key objective in semi-cooperative games is for the agents to learn to work together to defeat their opponent. Semi-cooperative games require extensive cooperation between friendly agents. Some tasks become much easier with cooperation, for example defeating an opposing agent with superior numbers or coordinating base defence and offensive between multiple agents. A real-world example and widely

researched topic on semi-cooperative multi-agent reinforcement learning is robot soccer. [SSK05, KRS+19]

The theory of MDP is central in most reinforcement learning techniques. In MDP it is assumed that the agents environment is stationary and therefore it cannot contain other learning agents[Lit94]. The actions of other agents in multi-agent environments make the environment no longer stationary. In the multi-agent case, the policies of all the agents for selecting actions play a crucial role in convergence to a joint policy.

Cooperative learning tasks can be split into two categories: centralized learning and distributed (or concurrent) learning. In team learning there exists only one learning process, a master agent, and the other agents are slaves of the master. Centralized learning can be reduced to single-agent learning case, but it also suffers from an exponential state and actions space in the number of agents. Since the learner needs to keep track of each agents states and actions and learn a joint policy of optimal actions for each agent. In distributed learning there are as many learning processes as there are agents. Each agent is learning individually and there may even not be a centralized joint optimal policy. Or the optimal joint policy can be constructed from the individual agents optimal policies.

Concurrent, or distributed, learning is often more feasible for video games or real-world applications since it suffers less from the curse of dimensionality than centralized learning. The problem is that the guarantee of convergence to optimal policy is lost. This kind of learning task with multiple agents can be investigated from a game theoretic viewpoint as a Markov game (or stochastic game). This extends Markov decision process (MDP) to the multi-agent case. An important concept here is a Nash equilibrium, which is useful tool for reasoning with multiple agents optimal policies in an environment. These concepts are explained in chapter 2.6.

Same challenges present in single agent case apply: the exploitation-exploration trade-off and the curse of dimensionality. In the multi-agent case new challenges arise in specifying the learning goal, the need for coordination between agents and the non-stationary nature of the learning environment. With multi-agent systems, the problem complexity easily arises, even with simple problems. Multiple agents and stochastic environment add complexity but also widen the spectre of emergent behaviours within the agents. This makes it harder to predict the outcome of the learning process and makes the learned solution less smooth. Even small changes in an agent's behaviour could result in radical changes in the emergent behaviour.

Assigning appropriate delayed reinforcement is problematic in a single agent environment. It is even more problematic in a multi-agent environment due to the interactions between other agents. How to reward agents if they do well as a team? There are multiple schemes for rewarding and punishing in multi-agent environment. Simplest reward scheme is *global rewards*, where whenever an agent receives reward or punishment, each other agent receives equal reward or punishment. Global rewards might not scale to complex tasks which require more tailored rewards for agent's specific actions. In some tasks with concurrent agents it may be difficult to calculate global rewards. Another extreme is to reward and punish each agent individually based on its own behaviour without any shared reinforcement. *Local rewards* discourages laziness, which may occur with global rewards, but it also discourages cooperation and greedy individual behaviours may emerge.

Which credit assignment scheme to use depends on the task. In tasks where specialization is not necessary, such as foraging, local rewards work better. While in other tasks, where specialization is mandatory, such as robot soccer, global schemes might work better. Global and local rewards can be combined to find a balance between them. Agent receives full local reward, but each other agent receives a smaller global reward. Another way of rewarding agents is to punish non-cooperation and reward cooperation instead of progress in the task. In this way, agents eventually learn to coordinate their actions [GM10]. One way of giving direct rewards in a multi-agent environment is *progress estimator*. It could result in good learning behaviour [Mat97]. Instead of rewarding agents when they successfully complete a task (drop what they have collected home), progress estimator is used to measure if the current behaviour is moving the agent towards doing something useful. This allows stopping the current behaviour if that behaviour does not progress the agent towards the goal well enough. Various other rewarding schemes, like vicarious reinforcement, social reinforcement, and observational reinforcement, have also been suggested for improving multi-agent learning behaviour [PL05, Mat94].

2.6 Game theory

Game theory is study of mathematical models of strategic interaction with rational decision-makers [VNM47, Mye13]. Game theoretic problems are numerous, but classical ones include two-player zero-zum game and prisoner dilemma. Key concept for each game theoretic game is finding mixed-strategy equilibrium, where each player is using a strategy which is best for them against each other player. It

has many applications in social science, logic and computer science. Game theory is exceptionally well suited for reasoning about multi-agent systems (or multiple players in the game theoretic setting). Markov games (or stochastic games) are and extension of game theory to MDPs [Lit94]. In concurrent multi-agent learning we can use this extended MDP. Markov games gives us a framework which can be used to provide theoretical results to the multi-agent problem. Markov game (MG) can be formally defined as a tuple: $< \mathbf{S}, \mathbf{A_1}, ..., \mathbf{A_n}, T, \mathbf{R_1}...\mathbf{R_n}, \gamma >$, where $\mathbf{S}$ is finite and discrete state space and $\mathbf{A_1}, ..., \mathbf{A_n}$ are the finite discrete action sets of individual agents. $T : \mathbf{S} \times \mathbf{A_1} \times ... \times \mathbf{A_n} \rightarrow \prod(\mathbf{S})$ is the state transition function which maps each state and one action from each agent to the next state. State transition function may be stochastic and define a probability distribution over the state space $\mathbf{S}$. $R_i : \mathbf{S} \times \mathbf{A_1} \times ...\mathbf{A_n} \rightarrow \mathbb{R}$ is the reward function for agent and $0 \leq \gamma < 1$ is the discount factor for futere rewards

In MG's learning agents try to maximize the sum of their joint expected rewards. The main difference between MDP and MG is that unlike in MDP, in MG the joint actions of the agents determine the next state. After choosing their actions the environment transitions to the next state. Each agent receives rewards according to their reward functions and they observe the next state in the partially observable case.

2.6.1 Nash Equilibrium

An important concept in Markov games is *Nash equilibrium* [N$^+$50, Mye13]. It is a game theoretic concept involving two or more players, in which each player is assumed to:

- Do their best to maximize their expected payoff in the game.
- Play flawlessly.
- Have sufficient intelligence to deduce the solution to the game.
- Know the equilibrium of each player.
- Believe that deviation from their own strategy does not cause deviation by other players.
- There is common knowledge that each player know these conditions, they know each other player knows these conditions and that they all must meet these

conditions.

If these are true, no player can benefit from changing its strategy in an attempt to gain benefit. If each player keeps their chosen strategies, this collective set of strategies and the corresponding rewards of those strategies form the Nash equilibrium. Nash equilibrium forms a joint policy of all single agents policies, where no agent in the environment has any incentive to change away from the equilibrium assuming that each of the agents are using their optimal policies. As agents have no control over other agents' policies, changing their own policy could lead to some other agent exploiting their policy for personal advantage. Many concurrent multi-agents try to converge to Nash equilibrium, since it always exists, even if it may not correspond to the optimal team behaviour. In cooperative tasks with global reward function, where each agent gains the same reward, Nash equilibrium is the globally optimal equilibrium.

2.6.2 Cooperative Markov Game Example

A simple example of fully cooperative two player Markov game is the climbing game [CB98]. The climbing game (Table 1) represents a problem where miscoordination is highly penalized, there is single optimal joint action and safe actions that may steer agents away from optimal joint action. This makes the climbing game a simple yet challenging problem for learning coordination. In the climbing game it is difficult for both agents to converge to this single optimal joint action, because of the high penalty involved in miscoordination. This may lead both agents to avoid that action, even if by coordinating their actions they could reach high reward. If one agent chooses action a and another agent action b they receive penalty of -30. This could lead to both agents avoid those actions, which would be detrimental to the learning process. By choosing action c is not punished for either agent, which could lead both agents to play it safe by choosing action c. However, if both agents would choose action a they would both gain higher reward of 11.

	a	b	c
a	11	-30	0
b	-30	7	6
c	0	0	5

Table 1: Two player climbing game.

A modification of the climbing game is a stochastic climbing game (Table 2), where all joint action selections are linked with two rewards, each gained with probability of 0.5. The average of these rewards is the same as in the original climbing game (Table 1). In the stochastic version of the climbing game, the maximum reward, which is gained with actions (b,b), is not associated with the optimal action, which is actions (a,a).

	a	b	c
a	10/12	5/-65	8/-8
b	5/-65	14/0	12/0
c	5/-5	5/-5	10/0

Table 2: Two player stochastic climbing game with equal 50% probabilities.

3 Deep learning

Deep learning is a family of machine learning techniques based on *artificial neural networks*.[Nie15, GBC16] Teaching these artificial neural networks may be done with supervised, semi-supervised or unsupervised learning techniques.[GBC16] Artificial neural networks have an inspiration from biology: neurons and their connection in a brain.[GBC16] Deep part comes from usage of multiple layers of neurons in these artificial neural networks.

Deep learning and deep networks have a long history. Back in 1967, first deep multilayer feedforward perceptrons[IL67] were introduced. Deep networks were popular in the 70's but after that waned in popularity due to their inability to solve more complex and less well defined problems. In the 90's, deep networks started to regain popularity due to their ability to solve more complex problems. This is due to the increase in computational power and the availability of larger and more complex data sets. In the 2010s, the advent of graphical processing unit (GPU) powered deep learning further increased the computational power to start a new deep learning revolution. Now, it is a critical component in many computational tasks. For example, it has found success in areas of computer vision[VDDP18], speech recognition[DHK13] and natural language processing[YHPC18].

In an example of image recognition, the raw input to the network may be the pixels of the image. The first layer could abstract pixels and output edges to the next

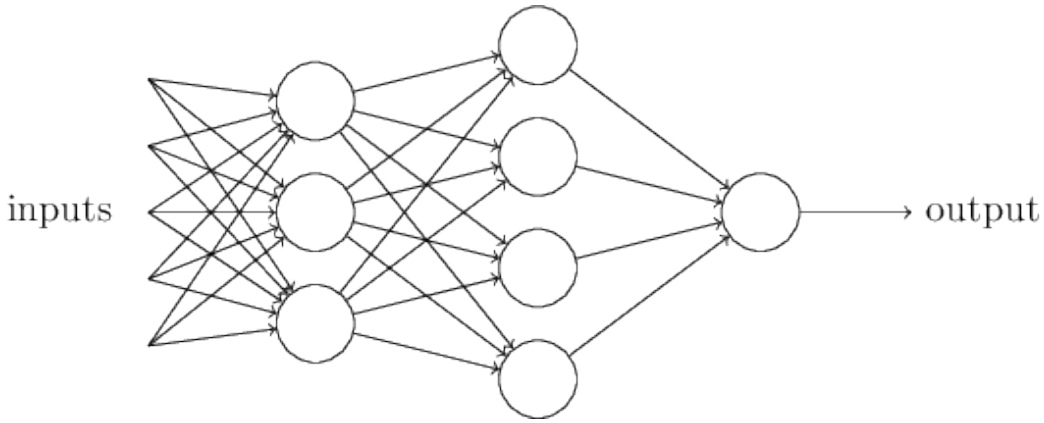


Figure 3: An example of a neural network (also called multilayer perceptron). Input flows through the network from left to right. The first layer of neurons take the input to the network. They process and output a weighed input to the next level of neurons, which, in turn, output their weighed input to a single neuron which gives the output of this neural network. This way, each layer can make more complex and abstract decisions than the previous one. (Image taken from "Neural Networks and Deep Learning", Chapter 1: Using neural nets to recognize handwritten digits, taken in 24.08.2020 [Nie15].

layer. Second layer could in turn encode the arrangement of these edges, third layer could already recognize features from the image, like eye or nose. The fourth layer could then recognize that the image is of a face.

3.1 Artificial neural networks

Artificial neural network (ANN) is a collection of nodes, called *neurons* and connections, called *edges*, between neurons. They get their inspiration from biology of brain: neurons and synapses connecting them. Edges between neurons transmit signals, which are typically real numbers. Neurons and edges have typically *weights* and *biases* associated with them. Weights increase or decrease the strength of the connections. Each neuron takes as an input a weighed sum of its signal and adds bias to that result. This is called *activation* of the neuron. Neuron then calculates (a non-linear) function to produce an output signal to the next layer of neurons. Neurons usually have a threshold over which to pass its output signal. There are typically multiple *layers* of neurons with connections between layers. Each layer may perform a different non-linear transformation function to its signal. First layer is called *input layer* and the last layer which produces the output of the network

is called *output layer*. Layers in the middle of the network are called *hidden layers* and there may be multiple hidden layers in the network. Signals travel through the entire network from input layer to output layer, each layer applying its non-linear transformation function on the way.

In feedforward neural networks, input flows from left to right, input to output. There are no backward connections. In recurrent neural networks feedback loops are possible, but they have not been as popular due to learning algorithms for them being less powerful. They, however better mimic how the brain works. ANN's can have many hidden layers, each layer having different type of neurons. They can be fully (each neuron connecting to every neuron on next layer), or sparsely connected (some connections are missing).

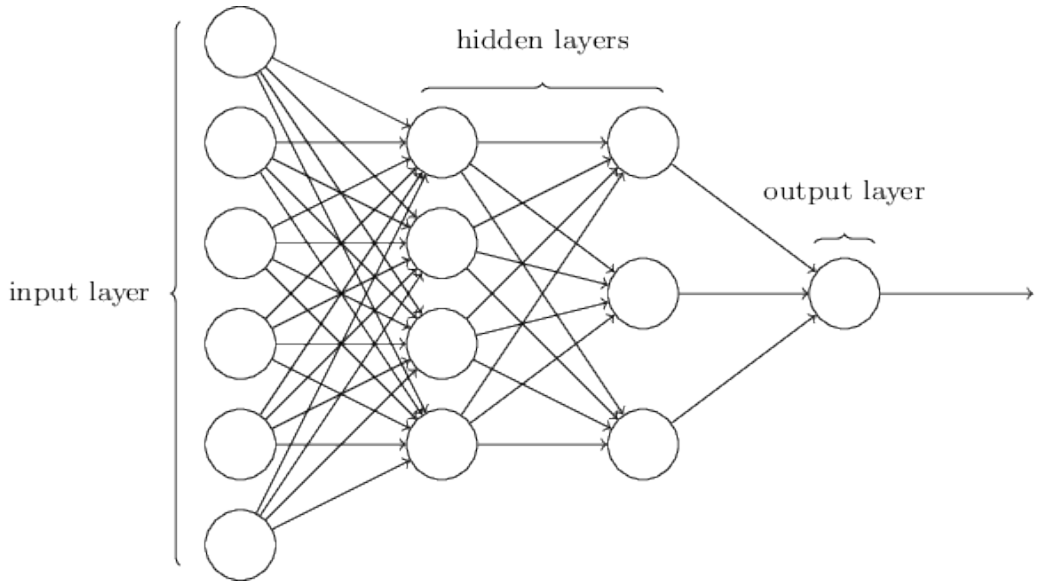


Figure 4: An example of a fully connected feedforward neural network with six neurons in input layer and two hidden layers and an output layer of single neuron. Arrows denote the direction for the signal. Each neuron in the hidden layer is connected to each neuron in the next hidden layer. Image taken from "Neural Networks and Deep Learning", Chapter 1: Using neural nets to recognize handwritten digits, taken in 24.08.2020 [Nie15].

Learning is done by adjusting weights and biases of the network, which can be automated by learning algorithms as we will see later. Essentially, ANN's can be used to approximate any function, which make them ideally suitable for machine learning tasks [GBC16].

3.1.1 Neurons

Artificial neural network learns by adjusting the weights and biases of its neurons. Neuron takes a vector of its inputs $x = (x_1, x_2, ..., x_n)$ and adjusts it by its weight $w = (w_1, w_2, ..., w_n)$ and finally subtracts or adds a bias b. Bias can be thought as a term that dictates how easy or difficult it is to get the neuron to activate, that is to have a value over certain treshold (usually zero) depending on the activation function. Suppose that $z = w \cdot x + b$ is the input to the neuron and f is the activation function of the neuron. Activation of the neuron is then $f(z) = a$. Weights and input can be real valued or vector valued.

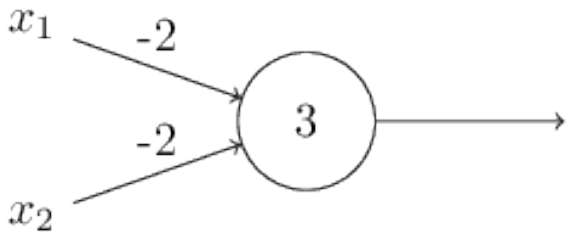
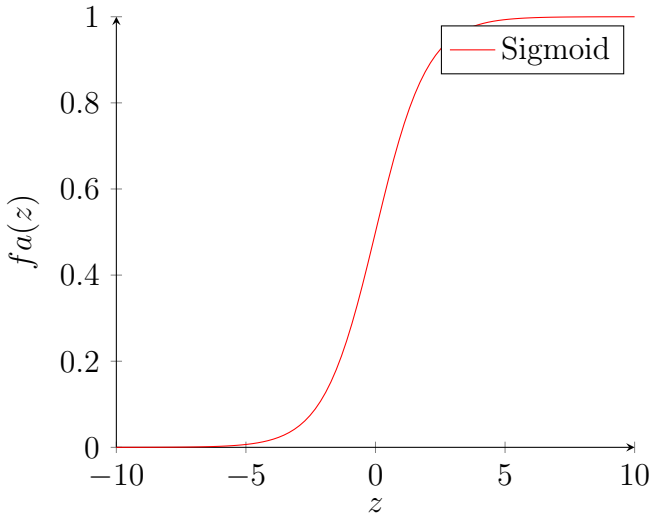


Figure 5: An example of a neuron in artificial neural network. This neuron takes two inputs x_1 and x_2. It has weight equal to -2 and bias of 3. The input to its activation function is therefore $-2x_1 - 2x_2 + 3$. (Image taken from "Neural Networks and Deep Learning", Chapter 1: Using neural nets to recognize handwritten digits, taken in 24.08.2020 [Nie15].)

Neurons activation then happens in its activation function which outputs its result to connected neurons. There are many different types of activation functions that can be used in neurons [GBC16]. Sigmoid and rectified linear unit are some of the more common types of activation functions for neurons. Sigmoid has been historically the most used neuron activation function. It is defined as

$$f(z) = \frac{1}{1 + e^{-z}}, \tag{8}$$

where $z = w \cdot x + b$. When z is very large, Sigmoid function is close to 1 and when z is very small sigmoid function is close to 0. As can be seen in the plot, it resembles a smoothed out version of a step function. Sigmoid does suffer from the problem of vanishing gradient for very large or very negative input values. In these cases, output of the sigmoid saturates, making gradient in these regions very close to zero. This causes major slowdown or even a full halt in learning. RELU, introduced later, handles this much better. Also, sigmoids may suffer from zig-zag behaviour when updating weights with gradient updates due to its non-centered nature.

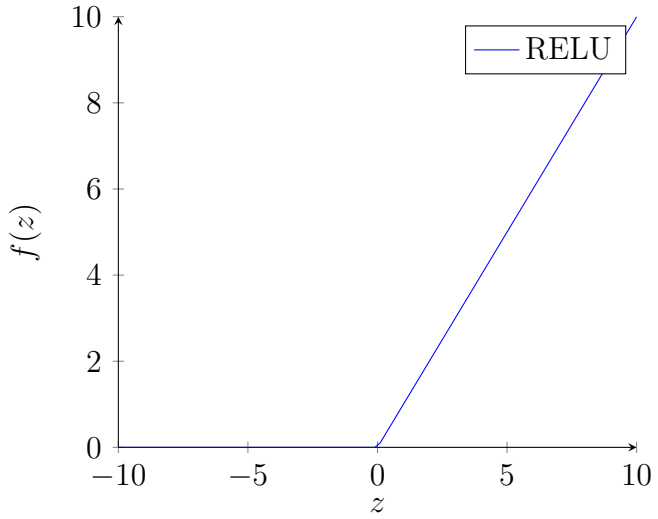


Rectified linear unit (RELU) is defined as:

$$f(z) = \max(0, z), \tag{9}$$

RELUs have found more use in recent times. Previously, there have been reluctance to use them due to the point of non-continuity at zero. Practice has however shown, that this is not an issue, since RELU is continuous almost everywhere [GBC16] (a property of measure theory [Hal13], out of scope of this thesis).

RELU solves many of the issues of sigmoids with large positive numbers. Gradient is always positive, large and consistent when neuron is active and gradient is always zero when output is zero or negative, meaning when the neuron is not active. This has some useful and unfortunate properties for gradient based learning algorithms. RELU cannot learn with gradient based methods on samples when their activation is zero. If learning rate is not set carefully, many of RELU neurons in network die and stop learning altogether. There are more advanced versions of RELU like leaky RELU [MHN13] and PRELU[HZRS15], which have their own advantages and disadvantages.



3.1.2 Convolutional neural networks

Convolutional neural networks [LBD+89] or CNN use specialized architecture, which is well suited for image recognition tasks. This architecture allow fast training of deep, many layer networks. Today, they are the most used type of neural network in image recognition tasks.

CNN uses three basic ideas: *local receptive fields*, *shared weights* and *pooling*. In local receptive fields, an area of the input image, for example 5x5 pixels, will be connected to one neuron in the first layer. Next local receptive field will be shifted by a stride length of one or more pixels up or right. This way, each local receptive field will be overlapping. Then, each neuron of these neurons connected to their respective overlapping local receptive fields will share their weights and biases. The shared weights and biases are said to define *a kernel* or *a filter*. The idea behind this is that each neuron in one layer would detect the same feature from the image, like an edge. Because image recognition requires multiple features, the first hidden layer would then contain multiple of these so called *feature maps*. Convolutional layer gets its name from convolution operation which is involved in the input activation of the feature maps.

In addition to convolutional layers constructed from these hidden maps, CNNs use pooling layers. They are located immediately after the convolutional layers and their task is to simplify the information from them. It takes a certain region of neurons from previous layer and condenses them to a single neuron. One of the simplest pooling methods is *max-pooling* which simply takes the maximum activation from

these neurons. This pooling is done for each feature map in the convolutional layer separately. The intuition for this layer is to check if certain feature was found in the region of the image.

More detailed description of CNNs can be found in "Deep Learning" by Goodfellow et al [GBC16] or "Neural Networks and Deep Learning" by Nielsen [Nie15].

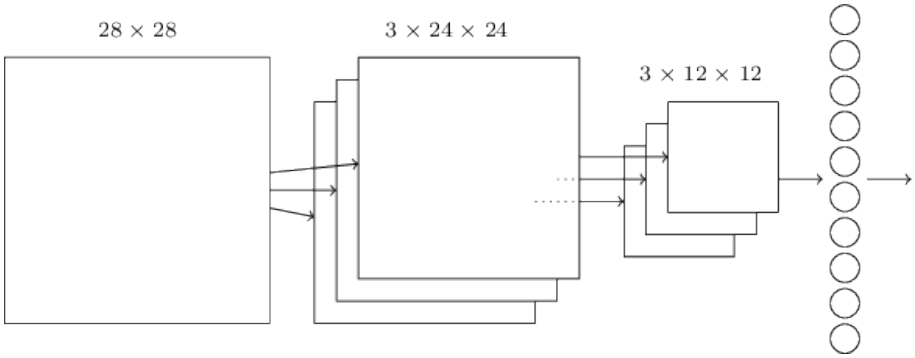


Figure 6: An example of convolutional neural network. Here, 28x28 pixel image is extracted to three 24x24 feature maps using 5x5 pixel local receptive fields with stride length of one. These are then pooled with 2x2 regions resulting in 3x12x12 neurons in the hidden layer. Every layer is then connected to 10 output neurons. (Image taken from "Neural Networks and Deep Learning", chapter 6: Deep Learning, taken in 24.08.2020 [Nie15].)

3.1.3 Recurrent neural networks

Recurrent neural networks [RHW86] or RNNs are a specialized type of artificial neural network suited for processing sequential data, such as handwriting or speech. RNNs have an internal state-space to process variable length sequences as input. This state-space can be a state vector, graph or another neural network. There are many ways to construct an RNN and as with other neural networks, they too, benefit from depth and width of the network as well as depth and width of its memory state. One way to construct an RNN is a network that produces output a each time step and has recurrent connections between the neurons of each successive layer. Different types of activation functions in the recurrent connections between hidden layers result can be used. Internal memory of RNNs to allow it to propagate effectively backwards in time, even to the start of the learning process. However, they have difficulty learning long-term memory, because backward propagating signals decay exponentially in time. This results in vanishing gradient problem discussed in earlier

chapters and to some extend exploding gradient problem.

More detailed description of RNNs can be found in "Deep Learning" by Goodfellow et al [GBC16].

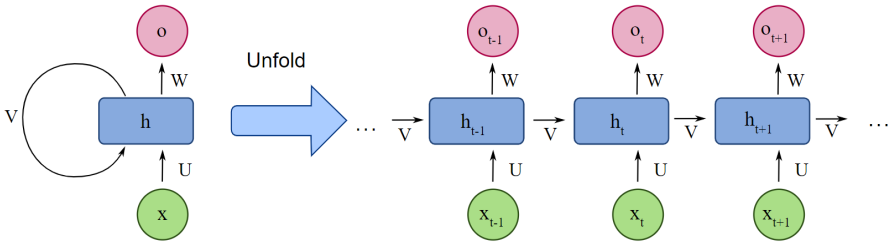


Figure 7: An example for a one-unit RNN: x is the input state, h is the hidden state and o is the output state. U,V,W are the network weights. Left is a compressed form and right is unfolded form. (Taken from https://en.wikipedia.org/wiki/Recurrent_neural_network on 19.10.2020)

3.1.4 Long short-term memory recurrent neural network

Vanishing gradient problems of RNNs can be alleviated using Long Short-Term Memory (LSTM) [HS97]. It does however suffer from the exploding gradient as RNNs do. LSTM is a specialized RNN that can learn long term dependencies from arbitrarily long time periods. As with RNNs, they can be constructed with a variety of different architectures. A basic idea is to have a *cell*, which is the memory unit of LSTM. Then, there are three *gates* (input, output and a forget gate) that control the flow of information in the LSTM. Some architectures may not have some of these gates or have even more of them. Cell is tasked in keeping track of the dependencies between the elements of the sequential input sequence. Input gate controls the extent to which new values flow into the cell. Forget gates controls the extent to which a value remains in the cell. Output gate controls to which extent each of those values stored in the cell is used in the computation of the output activation function of a neuron in the LSTM network.

More information on LSTMs can be found in "Deep Learning" by Goodfellow et al [GBC16].

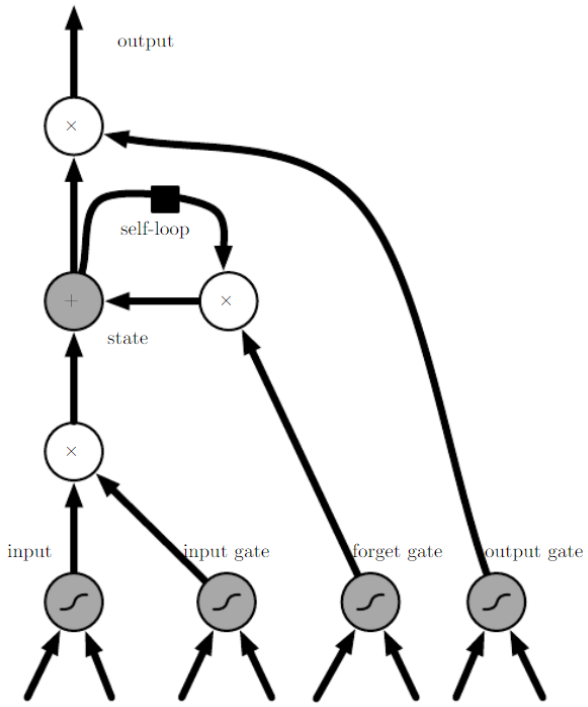


Figure 8: Architecture of a memory cell in LSTM network. Cells are connected recurrently to each other, replacing the regular hidden units in RNNs. Self recurrent connection represents feedback with a delay of one time step and weight controlled by forget gate. Input and output gate units close and open the access to and off the cell. Image taken from book "Deep Learning" page 405 by Goodfellow et al.

3.2 Training deep networks

The goal of the neural network is to accept an input x and produce an output $\hat{y}$. Information flows through the network. This is called *forward propagation*. During training, network is fed *training samples*, and forward propagation continues until it produces a desired cost (or objective) of $C(\theta)$ (sometimes $J(\theta)$, where θ is the parameter vector of the network containing weights w and biases b. *Backpropagation* or *backprop* for short, allows information from cost function to then flow back through the network in order to compute a gradient. Backpropagation does compute gradient in simple and computationally inexpensive way. Then, another algorithm called *stochastic gradient descent* can be used to perform learning from this gradient. In learning algorithms such as stochastic gradient descent, we often require gradient of the cost function $\nabla_\theta C(\theta)$ in respect to the parameters of the network. If we want to learn the weights and biases of the network, we require partial derivatives $\nabla_w C(w)$

and $\nabla_b C(b)$. The aim of the gradient descent is to minimize the *cost function* $C(\theta)$.

3.2.1 Cost function

Aim of the cost function $C(x, y)$ is to compute a single scalar cost or loss. Using this scalar cost we can determine how close our networks output $\hat{(}y)$ is to the actual desired output of y for an input of x. When our network is doing a good job at estimating the output we expect our cost function $C(x, y) \approx 0$.

It is important to choose the right type of cost function for the problem at hand. There are several types of cost functions. One of the simplest is the quadratic cost function [Nie15]:

$$C(x, y) = \frac{1}{2n} \sum_x ||y(x) - f^L(x)||^2, \tag{10}$$

where $f^L(x)$ is the activation function of the last layer of the network for an input of x, L is the number of layers in the network.

Since in most cases our parametric model defines a probability distribution $p(y|x; \theta)$, we can use the principle of maximum likelihood. A more useful cost function is then defined by the cross-entropy between the training data x and model θ's prediction $\hat{y}$. [GBC16]

$$C(x, y) = -\mathbb{E}_{x,y \sim \hat{p}_{data}} \log p_{model}(y|x) \tag{11}$$

3.2.2 Backpropagation algorithm

The aim of the backpropagation algorithm is to compute the partial derivatives of the cost function C in respect to its weights and biases: $\frac{\partial C}{\partial w}$ and $\frac{\partial C}{\partial b}$. When written in vector form the gradients $\nabla_b C(x, y)$ and $\nabla_b C(x, y)$. [Nie15, GBC16]

Essentially, backprop uses the chain rule of calculus to compute the expression for the derivative of the cost function as a product of partial derivatives between layers – from left to right. Given cost function C, L the number of, layers in the network, biases b^l for layer l, weight matrix between layers $l - 1$ and l is $W^l = (w^l_{ik})$, input x and output y. Then, the weighed input of a layer l is z^l, the output activation function of that layer is f^l and activation output of layer l is $f^l = a^l$. Then going backwards from output layer to input layer, the cost functions is:

$$C(x, f^l(W^L f^{L-1}(W - 1) f^{L-2}(...f^1(W^1 x)))). \tag{12}$$

The derivative of the loss function is given by the chain rule evaluated at each node in the network for the input x:

$$\frac{dC}{da^L}\frac{da^L}{dz^L}\frac{dz^L}{da^{L-1}}...\frac{da^1}{dz^1}\frac{\partial z^1}{\partial x}. \tag{13}$$

This can be written with weight matrices W of the network as:

$$\frac{dC}{da^L}(f^L)'W^L(f^{L-1})'W^{L-1}...(f^1)'W^1. \tag{14}$$

Gradient is then the transpose of the derivative of the output of the network in terms of the input x :

$$\nabla C_x = (W^1)^T(f^1)'(W^2)^T(f^2)...(W^L)^T(f^L)'\nabla_{a^L}C. \tag{15}$$

Backpropagation essentially consists of evaluating this expression from left to right, computing gradient at each layer on the way. We can now define δ^l as a gradient of the input values at layer l of the network. It can be thought of a s a measure of 'error' at level l. δ^l is a vector with elements equal to nodes at level l of the network.

$$\delta^l = (W^l)^T(f^l)'(W^{l+1})^T(f^{l+1})...(W^L)^T(f^L)'\nabla_{a^L}C. \tag{16}$$

δ^{l-1} can be easily calculated recursively as $\delta^{l-1} = (f^{l-1})'(W^l)^T\delta^l$. Now, we can calculate the gradient for the biases in layer l as $\nabla_{b^l}C = \delta^l$ and weights as $\nabla_{W^l}C = a^{l-1}\delta^l$.

3.2.3 Stochastic gradient descent

We would like our network to be as efficient as possible, that is to minimize the cost function C. To do that we need to learn the weights and biases that minimize it. We can do that using *stochastic gradient descent* or sgd for short [Nie15, GBC16]. Sgd works by selecting m random *mini-batches* $X_1, X_2, ..., X_M \in X$ from the training set instead of using the full training set as in *gradient descent*. Provided that there are enough mini batches, we can estimate the gradient using mini-batches:

$$\frac{\sum_{j=1}^{m}\nabla C_{X_j}}{m} \approx \nabla C. \tag{17}$$

If we let η be our learning rate, we can now update the weights W and biases B of our network by applying stochastic gradient descent update rule:

$$W^l \to W^l - \frac{\eta}{m} \sum_j \nabla_{W^l} C_{X_j} \tag{18}$$

$$b^l \to b^l - \frac{\eta}{m} \sum_j \nabla_{b^l} C_{X_j} \tag{19}$$

Essentially, in gradient descent, we try to minimize our cost function by adjusting the weights and biases of the network. Gradient for the weights and biases at each step point towards a minimum of the cost function. We then take tiny steps towards this point at each epoch. In sgd we go through one mini batch of the training sample in one epoch, then move to the next one. Training sample is scaled with the number of training samples in total, but sometimes it may be omitted, since it is already scaled with the learning rate η. The sgd gradient is not exact, but it points to the general direction of the gradient, which is what we care about. Sgd is usually much more efficient and faster than regular gradient descent, since we are only using partial samples of the training, but using enough training samples the estimate is 'good enough' in practice even if it would not converge to the exact gradient.

Gradient descent, and also stochastic gradient descent, may suffer from *exploding gradient* or *vanishing gradient*. This is when, during numerical computation. numbers are close to zero or too large. Then gradient explodes to infinity or vanishes to zero during multiplication in backprop when moving from end of the network towards the beginning of the network. This may cause neurons earlier in the network to stop learning completely, adversely affecting the performance of the network as whole. RELUs suffer less from vanishing gradient as do long short-term memory (LSTM) RNNs. [GBC16]

3.3 Challenges of deep networks

Deep networks are a powerful tool in modern machine learning, but they do suffer from certain issues. Learning algorithms for deep networks are, even with every bit of optimization, computationally complex. They grow exponentially in time when the width and depth of network increases. Deep networks usually need to be hand crafted for the specific task. Usually this means carefully selecting types of neurons for each layer (and their activation functions), selecting best possible cost function and fine tuning learning parameters.

Training sets need to be selected and crafted carefully for the task at hand. When training sets, network and parameters are not well tuned, *overfitting* may occur. This is when the network learns its training data and noise contained in that data so well, that it can no longer generalize on new data. Regularization techniques such as weight decay or L2-regularization may also be used to combat vanishing and exploding gradient. Increasing or performing transformations to training data to increase its size may also help against overfitting [Nie15, GBC16].

Underfitting may also occur, when the network fails to model the task at hand properly and will therefore have a poor performance. Underfitting usually occurs when the network is too simple for the problem at hand (it has too few layers or too few neurons per layer) or the layer types in the network are not suitable for the task.

3.4 Deep reinforcement learning in POMDP

Deep reinforcement learning aims to introduce the concepts of deep networks to reinforcement learning. Now, the aim is to train a deep artificial neural network, typically, a recurrent neural network or convolutional neural network, instead of a policy. This allows reinforcement learning to scale to a level previously unattainable [ADBB17].

Policy gradient methods are fundamental in success of recent deep reinforcement learning applications, but they do come with their issues. With supervised learning we can easily implement cost function for our network and run gradient descent on expecting results with fine tuning of parameters. Reinforcement learning case is not as simple. Moving parts in the algorithms are hard to debug and require substantially more time to fine tune. In reinforcement learning, the algorithms are usually also even more computationally complex making it harder to scale them for larger problems.

A simple solution for solving an optimal policy in partially observable environments would be to apply tried and tested Q-learning[WD92] algorithm to deep networks. Deep Q-networks (DQN)[MKS$^+$15] have shown to be very capable in 2-dimensional Atari games. As most modern online video games are partially observable, it would make sense to apply this in POMDP. Regular DQN method would fail, since partial observability and long time dependencies require memory. Future game states and rewards depend on more than the current game state. In POMDP we can

use deep recurrent Q-network (DRQN) [HS17]. This approach leverages the advantage of deep networks by combining DQN with LSTM[HS97] neural network. It has been demonstrated that LSTM networks can solve POMDPs when trained with policy gradient [WFPS07] methods and Q-learning[WD92] can be expanded to solving POMDPs with LSTM networks with DQRN. Both methods along with actor-critic[KT00] model and a novel proximal policy optimization (PPO)[SWD$^+$17] approach offer new tools for RL in POMDP.

These new techniques allow deep reinforcement learning algorithms, for example, to directly learn from the pixels on the screen. This has lead to promising results for deep reinforcement learning in 2-dimensional Atari games [MKS$^+$13] and as we will see in chapter 4, in much more complicated 3-dimensional multi-player online video games. These results, while already promising, are only a stepping stone in applications to real-world deep reinforcement learning tasks [ADBB17]. In addition to video games, there are applications to robotics, computer vision, health care, finance, transportation etc. [Li18]

3.4.1 Deep recurrent Q-network

Even in 2-dimensional Atari games, let alone modern online video games, there are far too many unique states and actions for vanilla Q-learning[WD92] to handle. Instead, in DQN, a model is used to approximate Q-values. This is done with a deep neural network parametrized by its weights W and biases B denoted as θ. Q-values are estimated online by querying output nodes of the network given a input state s. This is denoted by $Q(s, a|\theta)$, where parameters are same as in regular Q-learning. Instead of updating Q-value directly, the parameters θ of the network are updated to minimize a loss function:

$$L(s, a|\theta_i) = (r + \gamma \max_{a'} Q(s', a'|\theta_i) - Q(s, a|\theta_i))^2 \tag{20}$$

$$\theta_{i+1} = \theta_i + \alpha \nabla_\theta L(\theta_i) \tag{21}$$

To maintain learning stability three different techniques are used. Experiences $e_t = (s_t, a_t, r_t, s_{t+1})$ are recorded into replay memory D and then sampled uniformly at training time. Secondly, a separate target network $\hat{Q}$ provides update targets to main network Q, decoupling feedback resulting from the network generating its own targets. $\hat{Q}$ is identical to main network Q, except its parameters θ^- are updated to match θ every 10000 iterations. Finally, to compensate for the lack of fixed training

set, a adaptive learning rate method is used to maintain per-parameter learning rate α according to the history of gradient updates to that parameter. At each training iteration i, an experience $e_t = (s_t, a_t, r_t, s_t + 1)$ is sampled uniformly from the replay memory D. [HS17]

The loss of the network is determined as follow:

$$L_i(\theta_i) = \mathbb{E}_{(s_t, a_t, r_t, s_{t+1}) \sim D}[(y_i - Q(s_t, a_t; \theta_i))^2] \tag{22}$$

where $y_i = r_t + \gamma \max_{a'} \hat{Q}(s_{t+1}, a'; \theta^-)$ is the update target given by the target network Q. [HS17]

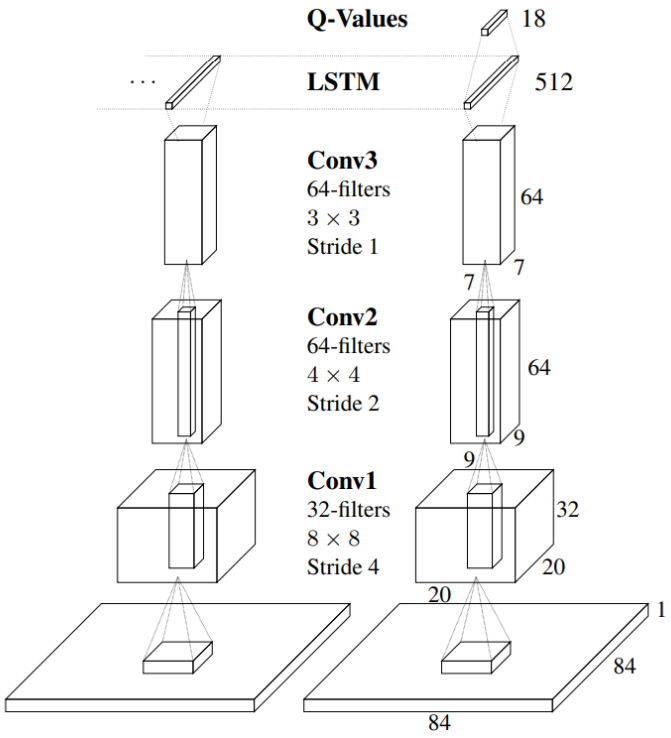


Figure 9: Example of DQRN network convolves three times over a single-channel image of the game screen of an Atari game. The resulting activations are processed through time by an LSTM layer. LSTM outputs become Q-Values after passing through a fully-connected layer. Convolutional filters are depicted by rectangular sub-boxes with pointed tops. (Image and example taken from "Deep Recurrent Q-Learning for Partially Observable MDPs" by Hausknecht and Stone, page 3) [HS17].)

A vanilla Q-learning has no way of deciphering the current state in POMDP. In practice, estimating Q-value from observation can be arbitrarily bad, since $Q(o, a|\theta) \neq$

$Q(s, a|\theta)$. Adding recurrency to Q-learning allows it to better estimate actual Q-values from observations so that $Q(o, a|\theta) \approx Q(s, a|\theta)$. This can be done by replacing the first fully connected layer of the network by a LSTM layer (Figure 9).

3.4.2 Actor-critic model

> Actor-critic methods are TD-methods that have a separate memory structure to explicitly represent the policy independent of the value function. The policy structure is known as the actor, because it is used to select actions, and the estimated value function is known as the critic, because it criticizes the actions made by the actor. Learning is always on-policy: the critic must learn about and critique whatever policy is currently being followed by the actor. The critique takes the form of a TD error. This scalar signal is the sole output of the critic and drives all learning in both actor and critic. Actor-critic methods are the natural extension of the idea of reinforcement comparison methods to TD learning and to the full reinforcement learning problem. Typically, the critic is a state-value function. After each action selection, the critic evaluates the new state to determine whether things have gone better or worse than expected. [SB18]

The objective of reinforcement learning is to find a policy π parametrized by θ that maximizes the cumulative future discounted reward. We can define the objective function for policy gradient as:

$$J(\theta) = \mathbb{E}[\sum_{t=0}^{T} r_{t+1}|\pi_\theta], \tag{23}$$

where t is the current time step, T is the terminal time step and r_{t+1} is the reward gained when performing action a_t at state s_t at timestep t. This is a maximization problem so we optimize the policy by taking gradient descent with the partial derivative of the objective function with respect to policy parameters θ. This results in:

$$\nabla_\theta J(\theta) = \mathbb{E}[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) G_t], \tag{24}$$

where $G_t = \sum_{t'=t+1}^{T} r_{t'}$. But, as discussed earlier, vanilla policy gradient method would run into serious issues in POMDP. To solve policy in POMDP we need some kind of memory. One way to improve policy gradient is by introducing a baseline, making cumulative reward smaller and to make gradient smaller and more stable:

$$\nabla_\theta J(\theta) = \mathbb{E}[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)(G_t - b(s_t))], \tag{25}$$

This baseline can take various values, which leads us to actor-critic methods[KT00]. In actor-critic the 'critic' $G_t - b(s_t)$ estimates the value function and the 'actor' $\log \pi_\theta(a_t|s_t)$ updates policy distribution in the direction suggested by the 'critic', for example with policy gradients. We get Q actor-critic when the 'critic' is $G_t - b(s_t) = Q_w(s_t, a_t)$, where Q is Q-value parametrized by neural network w. Or advantage actor-critic when $G_t - b(s_t) = A(s_t, a_t) = Q_w(s_t, a_t) - V_v(s_t)$, where A is called advantage value and Q as previously. Using value function V as the baseline function we subtract Q-value with the V-value. This means how much better it is to take a specific action compared to to the average action at that state. From the Bellman optimality equation[Bel66] we get $Q_t(s_t, a_t) = \mathbb{E}[r_{t+1} + \gamma V(s_{t+1})]$ and therefore $A(s_t, a_t) = r_{t+1} + \gamma V_v(s_{t+1}) - V_v(s_t)$, where value function V is parametrised by a neural network v.

3.4.3 Proximal policy optimization

A novel approach for policy gradient methods in POMDP's is Proximal Policy Optimization (PPO) [SWD+17]. It is a variant of advantage actor-critic[KT00]. It uses a slightly different approach to vanilla policy gradient methods and tries to address issues with fine tuning and complexity. PPO tries to compute an update at each time step to the cost function while ensuring the deviation from the previous update is small. PPO adds a constraint that can be optimized with first-order optimizer like gradient descent. This constraint can be broken at some intervals without interfering with the results in a long run. It can be shown that both the constrained policy and the actual policy can be optimized to the same optimal policy. A way to apply PPO is to use various clipping strategies, like clipped objective function [SWD+17]. Importance sampling is a general technique from statistics for estimating properties of a particular distribution by only using samples generated from another distribution. With this idea, we can evaluate a new policy from samples collected from old policies. But, over time, difference between older policies and our actual policy get

larger, so we need to update our sampled policy to our current policy from time to time. *Advantage function* is the difference between expected rewards for the new policy at the state minus a baseline value of the current state. Baseline can be for example, the average reward between actions at that state. We can then construct a objective function that clips the estimated advantage function if it steers away too much from the old policy. If the probability between the new and the old policy fall outside the range of $(1-\epsilon, 1+\epsilon)$ the advantage function will be clipped.

For proximal policy optimization with clipping or PPO-clip policy update rule at time step k is:

$$\theta_{k+1} = \arg\max_{\theta} \mathbb{E}_{s,a\sim\pi_{\theta_k}}[L(s,a,\theta_k,0)], \tag{26}$$

by taking multiple steps of sgd with mini-batches, where L is given by:

$$L(s,a,\theta_k,\theta) = \min(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}A^{\pi_{\theta_k}}(s,a), \text{clip}(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, 1-\epsilon, 1+\epsilon))A^{\pi_{\theta_k}}(s,a), \tag{27}$$

where $\pi_\theta(a|s)$ is the policy we want to optimize, $\pi_{\theta_k}(a|s)$ is the policy used to collect samples and $A^{\pi_{\theta_k}}(s,a)$ is the advantage function.

This can be simplified to:

$$L(s,a,\theta_k,\theta) = \min(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}A^{\pi_{\theta_k}}(s,a), g(\theta, A^{\pi_{\theta_k}}(s,a)), \tag{28}$$

where

$$g(\theta, A) = \begin{cases} (1+\epsilon)A & A \geq 0 \\ (1-\epsilon)A & A \leq 0 \end{cases} \tag{29}$$

When advantage A is positive:

$$L(s,a,\theta_k,\theta) = \min(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, (1+\epsilon))A^{\pi_{\theta_k}}(s,a), \tag{30}$$

then the objective increases if the action becomes more likely, that is $\pi_\theta(a|s)$ increases. min puts a limit to how much the objective can increase. When $\pi_\theta(a|s) \geq (1+\epsilon)\pi_{\theta_k}(a|s)$, the minimum kicks in and the equation hits a ceiling of $(1+\epsilon)A^{\pi_{\theta_k}}(s,a)$. Then, the new policy does not benefit from going further from the old policy.

When advantage A is negative:

$$L(s, a, \theta_k, \theta) = \max(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, (1-\epsilon))A^{\pi_{\theta_k}}(s, a), \tag{31}$$

then the objective increases if the action becomes more likely, that is $\pi_\theta(a|s)$ decreases. max puts a limit to how much the objective can increase. When $\pi_\theta(a|s) \leq (1-\epsilon)\pi_{\theta_k}(a|s)$, the maximum kicks in and the equation hits a ceiling of $(1-\epsilon)A^{\pi_{\theta_k}}(s, a)$. Then, the new policy does not benefit from going further from the old policy. Clipping serves as a regularizer by removing incentives to dramatically change the policy.

4 Deep reinforcement learning in online video games

Recent progress in areas of computer vision[VDDP18], speech recognition[DHK13] and natural language processing [YHPC18] has allowed the resurgence of deep learning, which provides a powerful toolkit for non-linear function approximation of neural networks. These techniques have proven successful in three dimensional simulations, robotics [SBS+18] and even in video games as we see in later chapters.

One AI milestone is to exceed human capabilities in complex online video games like Starcraft 2 or Defence of the Ancients (Dota2). As previously the goal was exceeding human performance in classical games like Go or Chess. Complex nature of the video games could model the messiness, uncertainty and continuous nature of the real world with multiple simultaneous agents learning to cooperate and compete against each other. Advancements in video game AI could lead to general applications in real world.

Recently, AlphaGo [SHM+16] and AlphaZero [SHS+18] have demonstrated how deep reinforcement learning techniques can be used for the system to achieve super-human performance in classic fully observable adversary two player games of Go, Chess and Shogi. AlphaZero learns entirely from self-play, but its predecessor AlphaGo used replays from human players to kick-start the learning. The style of play and strategies developed by Alphazero agent are uncanny and could alter how humans approach these games in future. AlphaStar[VBC+19] demonstrates how seeding the learning progress with supervised learning techniques using replays of human play, like in AlphaGo, can help achieve playing performance of top-tier human players also in an complex online video game. OpenAI five [OB+19] and DeepMind FTW

[JCD+19a] show the power of self-play in modern games, although game's in these scenarios were restricted in complexity.

Deep reinforcement learning techniques are ideally suited for complex three dimensional online video games due to the following reasons:

1. They offer rich stream of input, ideal for deep neural networks.
2. They have a clear measures for success, ideal for reinforcement learning.
3. They can be simulated and run on a large scale, ideal for deep reinforcement learning.
4. They are very difficult and complex even for human play, providing an interesting field of research.
5. They are run everywhere with the same parameters, making comparisons between solutions easy.
6. They have a vast pool of players all over the world, even professional players. This makes it easy to benchmark AI agents against skilled human players.

4.1 General challenges of video game AI

In comparison to classical games of GO and Chess, online video games are much more complex. Games of GO generally end before 150 moves and games of Chess typically end in 40 moves with every move being strategically significant. The average number of available actions at each state in chess is 35; in Go, 250. Chess board is represented by 70 enumeration values, 8x8 game board and 6 piece types. Game of Go is represented by 400 enumeration values of 19x19 board and two piece types. In classic games, game pieces mostly have only one action they may perform in several different directions. Action pool in video games on the other hand consists of dozens of different actions and with each agent having their own set of actions.

As the state-space is continuous in a video games, some form of abstraction is needed. Even if locations of the agents would be abstracted to discrete locations, the state-space would still be huge in addition to other things agent needs to keep track of. To deal with the size and complexity of the state-pace, it can be abstracted to *conditions*. Action-space can be abstracted to hard-coded *behaviours*, which then contain the set low-level actions for that behaviour. Such an abstraction scheme

is highly dependent on the expert knowledge of the task and needs to be applied beforehand for each task separately. This abstraction scheme simplifies the learning task to finding an optimal policy of a mapping of conditions to behaviours. Other agents can be abstracted as a number of agents assigned to each task (or behaviour) or action, instead of keeping track of each agent individually. Example of a low-level action would be a game piece or unit moving to a another location. A behaviour could be for example to gather resources. This would consist of several low-level actions, such as multiple move actions to a location of a mine, a mining action, then again multiple move actions back to home base to deposit mined resources.

Partial observability also poses a new and difficult issue. Some parts of the game map may be shrouded by fog-of-war and be not visible to all agents. Recurrent neural networks, or other variants, can be used to address partial observability with their internal memory structure. Multi-agent environment also poses its own difficulties. Real-time strategy games are often played with two players like Go or Chess, but multiplayer battle arena games and first person shooter games may have ten or more players in the same game cooperating with other players and competing against others.

In video games it may often be easy to assign global rewards in semi-cooperative multi-agent environments due to the rules of the game: If the game was won or lost. Internal game scoreboard may also provide a clear way to assign rewards. This particularly is useful for games in which agents need to specialize, for example in multiplayer online battle arena games. On the other hand reward assignment solely for loss or win over long time periods may prove difficult resulting in too sparse reward structure. Games may take tens of thousands of actions with very long time dependencies. Rewards need to be accurately assigned even over long periods of time then and propagated properly in time.

Typically AI in games has been done with scripted behaviours, path-finding algorithms and hard-coded rules [Mil19]. This approach works well enough in most single-player games but even then, often AI needs to be given unfair advantages. This does not work at all in competitive online video games where the objective is to match and to win against the best professional players. Here, deep reinforcement learning techniques come into play. Before the rise of popularity of deep learning in the last two decades, reinforcement learning techniques were not even close to be successful in video games. Next sections take a closer look at modern approaches in competitive game AI.

4.2 Defence of the Ancients 2

Defence of the Ancients 2 (Dota2)[Val] is a complex and popular online video game which has been played alongside it's first instalment for more than 15 years. It is a multiplayer online battle arena (moba) game which is played by two teams of five players. It's being played competitively in many professional leagues all over the world, being one of the most lucrative eports games alongside with its rival League of Legends (LoL). LoL being more simple of these two trying to capture more casual audience without requiring in-depth knowledge of the game. Objective of Dota2 is to destroy a structure defended by opposing team known as the ancient. Game is played in real time on a three dimensional map viewed from an isometric perspective. Each player plays a character chosen in a pregame draft process from a pool of over 100 different characters with different abilities. These characters are also known as heroes are split into two different classes 'core' and 'support'. Core heroes typically start weak and develop into powerful heroes by gaining experience or new equipment and are able to 'carry' team to victory. Support heroes typically lack ability to directly deal with opposing heroes but provide crucial support to core heroes. Each hero have four distinct abilities and more powerful ones are unlocked during the game as heroes gain experience to level up and gold by defeating opposing heroes, buildings and creeps. Each ability has a cool-down period so timing of abilities is crucial. Heroes also have powerful 'ultimate' ability with very long cool down. When a hero is defeated it is removed from the game until re-spawn timer counts to zero. In addition to heroes becoming more powerful via leveling up through gained experience they may purchase items with gold.

The two opposing teams known as 'Radiant' and 'Dire' occupy heavily fortified bases on the opposing corners of the map divided by a river. These bases are connected by three paths referred as 'lanes'. Lanes are defended by turrets that shoot opposing heroes and creeps on sight. Lanes also spawn creeps who move in predetermined paths along the lanes attacking opposing heroes, creeps and buildings on sight. There are also camps of neutral creeps in 'jungles' between lanes on both sides of the map. Act of defeating creeps for gold to purchase equipment is referred as 'farming'. There spawns a powerful neutral creep named 'Roshan' at the center of the map from time to time. If heroes manage to defeat it, they gain powerful items which the may use to swing the tide of the game. Each time 'Roshan' re-spawns it becomes more powerful. Temporary power-ups called 'runes' provide short term bonuses and spawn at the map every two minutes.

Figure 10: Two heroes from the Radiant (green) and Dire (red) are battling it out on a lane. Few creeps are also present on the battlefield. Bar on the lower end of the screen shows how many actions one hero can use. On the lower left is the minimap which shows the scale of the map where the gameplay takes place. On the upper side of the screen, all 10 heroes from both teams, present in the game are shown.The UI conveys a vast amount of information to the player. Taken from Dota 2 official site at https://www.dota2.com/play/ on 28.10.2020.

Dota2 is known for its steep learning curve and complexity. Games of Dota2 can last up to 45 minutes with each player making over 20,000 moves during the game. Moves at any given time can have minimal impact in the game or be game breaking moves and these moves can have long term consequences. Units, buildings and heroes can see the map around them. Rest is hidden by a fog of war constraining information available to partially observable. Players must plan their strategy based on incomplete information while trying to gather information on what opposing team is up to. Each hero can take dozens of possible actions against other heroes, creeps, buildings and ground. Not every action is valid at any given time. Roughly 1000 different actions are available for each hero at any give time. Dota 2 is played on a large continuous map containing 10 heroes on two teams, dozens of buildings, dozens of other units and various other game features such as trees runes etc. The state of Dota 2 can be expressed by roughly 20,000 floating point numbers. In summary, a game of Dota 2 is a game of long term planning on high-dimensional continuous partially observable environment with actions drawn from high-dimensional continuous action space.

In 2019 a team of bots using deep learning AI called OpenAI Five [OB+19] has been shown the capability to defeat even the best professional team twice in a row in a live match. Supplementary material for the OpenAI five containing gameplay videos can be found on the blog post for the OpenAI five[Ope18]. Open AI was able to achieve this in a year after losing under similar circumstances in 2018. The only major change from 2018 to 2019 was addition of 8 times more training computation, leading to about 250 years of simulated experience per day per. This was meant utilizing thousands of GPUs for months. In the end, OpenAI achieved a 99.4% win rate against human opponents on the internet over 7000 games. This was done in a restricted version of the game. Hero pool was limited to 17 heroes of 117 in total, two items were banned alongside summons, creatures which heroes may call for their aid and no scanning. OpenAI Five's heroes were also given invulnerable, scripted couriers to carry items for them into the front lines. Addition of five couriers led to criticism that the games played unlike real Dota2 games and also led to the 'signature' aggressive playing style of OpenAI Five. It was able to keep pushing while receiving healing items from couriers in states of game where heroes normally would have to retreat. Reaction time of OpenAI Five was restricted to 200ms to match that of professional players and not to give it unfair advantage in reflexes when micromanaging battles. [OB+19]

Policy of OpenAI Five agents was constructed as a function from a history of obser-

vations to a probability distribution over actions. This was then parametrized to a 159 million parameter single-layer 4096 unit LSTM recurrent neural network[HS97]. This neural network emits actions as action heads, which have a semantic meaning as a game actions. This helps to address partial observability of the game. Games are played repeatedly with current observation being passed at each time step to sample an action from the distribution. Same observations are shared with each hero, since they share the same visibility in game. Observations are not received from the pixels of the screen but from approximate data arrays. Even though some of the game states are partially observable, OpenAI Five was able to learn to deduct some of them based on other states. For example, it could learn to avoid area attacks it could not see based on health being lost. [OB+19]

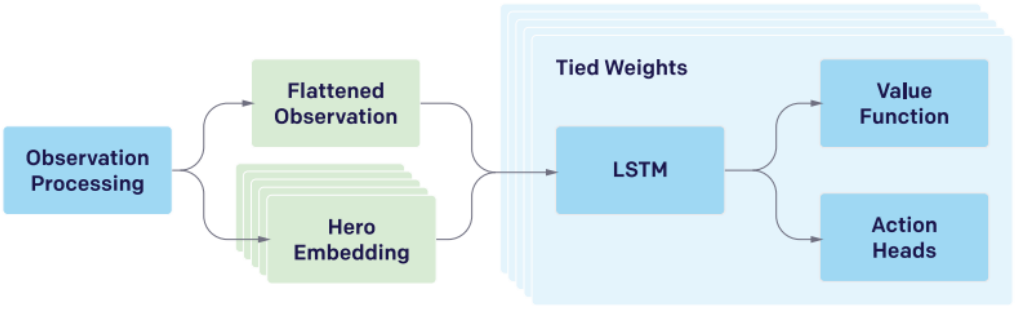


Figure 11: Simplified version of Open AI five architecture. The complex multi-array observation space of more than 15,000 total action values has a tree structure. A full game state has various attributes such as global continuous data and a set of allied heroes and each allied hero in turn has a set of abilities, a set of modifiers, etc. This is processed into a single vector and passed through 4096-unit LTSM. Each node in the tree is processed according to its data type. For example for spatial data, the data within each cell is concatenated and then applied to a 2 layer convolutional network. Then the outputs of that LSTM are projected to produce outputs using linear projections. Each of the five heroes are controlled by a replica of this architecture with nearly identical inputs (each hero shares their visibility) but each with their own hidden states. Number of actions available to a hero during a game varied between 8,000 to 80,000 from a total action space of 1, 837,080 dimensions. Taken from "Dota 2 with Large Scale Deep Reinforcement Learning" page 4. [OB+19]

Task was to find a policy, which could defeat professional human players, with self-

play. The policy was trained with Proximal Policy Optimization (PPO) algorithm [SWD+17] with clipping on a massive scale. Generalized Advantage Estimation (GAE) [SML+18], an advantage based variance reduction technique, was used to stabilize and accelerate learning. OpenAI Five learned entirely from self-play starting from random parameters and not using replays of humans playing. Experience from the self-play was collected to experience buffers located at a central GPU pool asynchronously. From these experience buffers mini-batches were sampled randomly to compute gradients at each of these optimizer GPUs. Gradients are then averaged across the pool and then applied to the parameters of the LSTM network. To prevent collapse of strategies and to avoid development of easily exploitable strategies, self play is 80% of the time against its current iteration and 20% of time against its former self. Pre game-drafting portion of the game could have used different reinforcement learning framework altogether, since it differs from the game itself. But, instead MinMax-algorithm was used from the win percentage estimates with certain teams against certain teams. When an opponent chooses a hero, the AI then chooses its hero to maximize its win percentage to opponents expected worst case selection. [OB+19]

Reward function was constructed with team members prior knowledge of the game, what players use to track how well they are playing: net worth, kill/death ratio, etc. This reward is then post-processed by subtracting the average reward of opposing team from it to prevent agents from finding positive-sum reward situations in games. it contained signals like characters dying or resources being collected. To address the extremely long time dependencies of Dota2 in which games can take tens of thousands of actions, the learning agent is trained to maximize the exponentially decayed sum of future rewards $\frac{T}{1-\gamma}$, where γ exponential decay factor and T is game time corresponding to each time step. In the later runs, it was weighted such that rewards were valued at half-life of about 5 minutes. This allowed rewards to be accurately assigned even over long time periods and maximize actions up to 6-12 minutes into the future. To address the rising power level of heroes during game, a simple renormalization is used, multiplying all rewards other than the win/loss reward by a factor which decays exponentially over the course of the game. Each reward ρ_i earned a time T since the game began, is scaled with $\rho_i \leftarrow 0.6^{T/10mins}\rho_i$. OpenAI Five did not contain explicit channel for communication between heroes neural networks, but instead during learning period had a hyperparameter called 'team spirit' τ, ranging from 0 to 1. It measures how much hero shares of its reward with other heroes. This results in final reward function for hero $r_i = (1-\tau)\rho_i + \tau\overline{\rho}$,

where $\overline{\rho}$ is the mean of ρ. This controls how much hero should put weight on teams average reward functions instead of just its own reward function, 0 meaning every hero for themselves and 1 meaning equal split of rewards, $r_i = \overline{\rho}$. In between, reward is interpolated. Ideally, training should optimize for $\tau = 1$ to maximize the teams reward. After training period, value of team spirit is set to maximize team effort. Each hero is also assigned randomly a subset of lanes to play in at the beginning of each game. It is penalized for straying away from them too long. [OB+19]

As the training period of 10 months progressed the environment changed for several reasons: as the team experimented, they learned more and adjusted the architecture, over time more of the game mechanics were added, and from time to time, games publisher released new versions of the game. These changes would have each time typically resulted in making adjustments and restarting the learning progress form scratch. In long training period this would have meant loss of months of data. Instead they performed 'a surgery', a collection of tools, to perform offline transformation to the old learned policy to make it compatible in the new environment. In the simplest case, when the environment, action or observation space did not change, the new policy simply implements the same function as the old. These surgeries allowed the learning to continue throughout the process without loss in training performance [OB+19].

Even given the restrictions to the base game of Dota2, OpenAI Five has hundreds of items, dozens of buildings, abilities, unit types and a lot of game mechanics to learn. Exploring such a vast combinatorial space efficiently poses a problem. Learning from self play, gives OpenAI Five a natural approach to exploration. In the first games, it just wanders aimlessly around the map, like a toddler walking for the first time. After several hours of training basic game concepts such as 'laning', 'farming' and fighting over the middle ground start to emerge. Several days into training, basic strategies, like rotating around the map to gain advantage on a single lane, start to emerge. As the training goes on, more and more advanced strategies, such as pushing advantage as a team of five heroes, are being learned. Good reward management also helped in the exploration.

The agents were trained solely on self-play with all agents controlled by the AI and playing against other AI agents. They were able to transfer their learning to controlling a subset of heroes in cooperation with human players which presents a compelling vision of human-AI interaction. It was evaluated by the professional players that OpenAI Five was weak at 'last hitting', a technique where a fatal blow

is given, against opponents and creeps, being at a level with average Dota2 player. But, its objective prioritization matched that of common strategies of professional teams. OpenAI Five was able to sacrifice short term goals such as farming in favor of long term goals such as strategic map control, which leads to the belief that it is indeed able to optimize over long time horizons. [OB+19]

4.3 Starcraft 2

Starcraft 2[Bli] is the most popular and complex online real-time-strategy (rts) game designed to challenge human players across the world. It has been played along with its original title for more than 20 years. It is one of the most popular and difficult competitive online video games and a relevant artificial intelligence challenge target due to its raw complexity and multi-agent challenges. Previously, strongest AI agents have simplified aspects of the game, used superhuman capabilities or employed hand-crafted systems to challenge human players. Still, they have not come close to matching the skill level of top human players in the game even in the first edition of the game [OSU+13]. Learning environment of SC2LE has been created to expose Starcraft 2 as a research environment [VEB+17]. This has been done in cooperation with the game's developer Blizzard interactive.

A competitive game of Starcraft 2 is played through point and click interface from birds eye view on three dimensional world between two players battling on varying maps shrouded by fog of war hiding crucial information. These maps contain various elements such as ramps, choke points and resources to gather. Players play as one of three factions available in the game. These three factions, 'Terrans', 'Protoss' and 'Zerg', each posses different units, buildings and technologies to mix up in a strategy to defeat your opponent. Each player starts the game with one worker unit and a base structure. Worker unit is used to gather resources (crystals and gas) and to build more advanced buildings, which in turn allow building of more units and technologies. Players must carefully craft their strategy on micro level, controlling each individual unit, and on macro level, the economy on whole. Careful balance must be found between investing in buildings, technologies, on the amount of workers harvesting resources and between units of war. This must be adjusted on the fly so that the chosen strategy is not easily countered by the opponent. Generally, as most rts games, games are split between early game expansion and economy phase and mid- to lategame battle phase. Although, some players may favor early game rush, where the idea is to attack other player before they are ready to defend itself.

Starcraft 2 poses an immense challenge on AI research. Players must take actions continually as the game progresses in real-time, unlike in classic games, like chess, where players take alternative moves in subsequent turns. Action space in Starcraft 2 is very large. Hundreds of different units and buildings in the players faction must be controlled at once in real-time, resulting in a huge combinatorial space of actions. In addition to this technologies that can be researched in the game can augment these possible actions. Best human players perform hundreds of actions per minute. Games take tens of thousands of time steps and players need to perform thousands of actions in them.

Figure 12: Early game expansion of Zerg in game of Starcarft. Early game consist of gathering resources (crystals and gas) with builder/gatherer units as fast as possible to have as strong economy as possible. Then, the priority is to start building structures and to expand military to defend against early rush attacks or to start them against opponent. Picture taken from https://en.wikipedia.org/wiki/StarCraft_II:_Wings_of_Liberty on 27.10.2020.

In game theory aspect Starcraft, 2 is a game of rock-paper-scissors where there is no single strategy that is the best. AI training process needs to continually explore and adapt its strategic knowledge. Game of Starcraft 2 is a game of imperfect information unlike a game of chess where each player can see the complete game state. Crucial information to form a winning strategy is hidden by the fog of war shrouding the map

and active scouting to lift the fog of war and to discover this hidden information is of paramount importance. Cause and effect in game of Starcraft 2 is not instantaneous and actions taken in the early game can take a long time to pay off. Games can take up to an hour to complete, nut are usually over in less when one side has gained an substantial advantage that the other side cannot anymore overcome.

Figure 13: Mid and late game battle of Protoss against Terrans sees powerful units brought to battle against each other to destroy enemy bases and to defend own bases. This phase sees a lot of micro level maneuvering from professional players to gain upper hand in the battle. These battles often may mean the difference between victory and defeat. Picture taken from https://starcraft2.com/en-us/media on 27.10.2020.

Alphastar [VEB+17, VBC+19] is the first artificial intelligence program to defeat some of the top professional players. Supplementary material for the Alphastar containing gameplay videos can be found on the blog post for the Alphastar [VBC+]. Alphastar's behaviour is generated by a deep LSTM[HS97] neural network capable of handling partial observability is used that takes input from the isometric view of the game camera and inputs series of actions within the game. General purpose machine-learning techniques were used in Alphastar. To address the vast action space Alphastar use a recurrent pointer network [VFJ15] and an auto-regressive policy[MIJD17]. Auto-regressive policy is a history dependent policy that produces action samples according to stationary autoregressive stochastic process. Since Alphastar could in theory, input actions at a super-human rate, its actions per minute or APM for short, were limited to lower then what is recorded for best human play-

ers. This constraint was approved by professional players. Surprisingly, this limit allowed Alphastar to spend more time in refining macro strategies instead of micro level managing and led to better results overall. [VBC+19]

The policy of Alphastar is a mapping $\pi_\theta(a_t|s_t, z)$ of previous observations $s_t = o_{1:t}$, actions $a_{1:t-1}$ and strategy statistics z to a probability distribution over actions a_t for the current time step t. Policy π_θ is implemented with LSTM deep neural network, which maintains memory between time steps. The observations o_t are encoded into vector representations, combined and processed by the LSTM network. The actions a_t are sampled with auto-regression[MIJD17], conditioned to the output of the LSTM and observation encoders. This is a architecture using the state of the art advances in deep learning. Alphastar's network has 139 million weights in total, but 55 million are used at once during inference. [VBC+19]

Alphastar's artificial neural network was first trained by a multi-agent supervised learning algorithm from a dataset of 971,000 anonymous recorded games played by the top 22% of human players. One policy is trained for each of the three races. From each replay, strategy statistic z is extracted, which encodes build order of buildings, units, upgrades etc. To train the policy, at each step the the current observations and output a probability distribution over each action argument is input. For these arguments, Kullack-Leiber (KL) divergence is computed between human actions and the policy's output. Then Adam-optimizer[KB17] and L^2 regularization is applied. The policy is then further fine tuned by only using winning replays which improved win-rate against built-in bot from 87% to 96%. [VBC+19]

This training strategy allowed Alphastar to learn basic micro- and macro- strategies used by human players. Then, to further make Alphastar stronger, these agents taught from human replays were then used to seed a multi-agent reinforcement learning algorithm. Multi-agent environment required asynchronous off-policy learning [SB18] for policy updates. During reinforcement learning, the policy is updated by V-trace[ESM+18] with clipped importance sampling, an off-policy actor-critic[KT00] RL algorithm similiar to advantage actor-critic[MBM+16], and the value estimates are updated using TD(λ)[Sut88] (Temporal difference) learning. [VBC+19]

In the actor-critic model of Alphastar, the actor generates a trajectory $(x_t, a_t, r_t)_{t=s}^{t=s+n}$ following a policy π. Then, a n-step V-trace for $(V(s_h))$ can be defined and the value approximation at state s_h is:

$$v_s = V(s_h) + \sum_{t_h}^{h+n-1} \gamma^{t-h} (\prod_{i=h}^{t-1} c_i) \delta_t V, \tag{32}$$

where $\delta_t V = \rho_t(r_t + \gamma V(x_{t+1}) - V(s_t))$ is temporal difference algorithm (or similiar value function update rule) for V, and $\rho_t = \min(\overline{\rho}, \frac{\pi(a_t|s_t)}{\hat{\pi}(a_t|s_t)})$ and $c_i = \min(\overline{c}, \frac{\pi(a_t|s_t)}{\hat{\pi}(a_t|s_t)})$ are truncated importance sampling weights such that $\overline{\rho} \geq \overline{c}$. [ESM+18]

In addition to the V-trace policy update, a novel upgoing policy update (UPGO) rule based on self-imitation learning[OGSL18] algorithm is used, which updates the policy parameters in the direction of:

$$\rho_t(G_t^U - V_\theta(s_t, z)) \nabla_\theta \log \pi_\theta(a_t|s_t, z) \tag{33}$$

where

$$G_t^U = \begin{cases} r_t + G_{t+1}^U & if\ Q(s_{t+1}, a_{t+1}, z) \geq V_\theta(s_{t+1}, z) \\ r_t + V_\theta(s_{t+1}, z) & otherwise \end{cases} \tag{34}$$

is an upgoing return, $Q(s_t, a_t, z)$ is an action-value estimate, $\rho_t = \min(\frac{\pi_\theta(a_t|s_t,z)}{\pi_{\theta'}(a_t|s_t,z)}, 1)$ is the clipped importance ratio and $\pi_{\theta'}$ is the policy that generated the trajectory in the actor. Similarly to imitation learning, the idea is to update the policy from partial trajectories with better-than-expected returns by bootstrapping when the behaviour policy takes a worse-than-expected action. With the difficulty of approximating $Q(s_t, a_t, z)$ in vast action space of Starcraft 2, action values are estimated with one-step target $Q(s_t, a_t, z) = r_t + V_\theta(s_{t+1}, z)$. The overall loss is a weighed sum of policy and value function losses, corresponding to the win-loss reward r_t, the pseudo-rewards from human data, KL divergence loss with respect to the supervised policy and the standard entropy regularization loss. Overall loss is optimized with Adam[KB17]. Compared to typical value function, in which agent values how good it is to be at a certain state, advantage function in actor-critic captures how much better certain action is compared to other actions at a given state. Self-imitation learning further improves on actor-critic and its variant PPO[SWD+17] in this instance. Prior reinforcement learning approaches to updating weights were found ineffective due to massive action space of Starcraft2. [VBC+19]

In the reinforcement learning framework, terminal reward r_t is given from $(-1, 0, 1)$ according to lose, tie, or win, at the end of the game without any discount. Following the actor-critic model, a value function $V_\theta(s_t, z)$ is trained to predict r_t and to train the policy $\pi_\theta(s_t, a_t, z)$. Starcraft 2 poses a complex domain with sparse reward structure as a RL-problem to exploration. Human data is therefore used to guide

in the exploration and to preserve strategic diversity throughout learning period. First, policy is initialized to the supervised learning policy and continually minimize the KL divergence between supervised policy and current policy. Second, the main agents are trained with pseudo-rewards to follow a strategy z randomly sampled from human replay data. These pseudo-rewards measure the edit distance between sampled and executed build orders, and the Hamming distance between sampled and executed cumulative statistics. Human data was found critical in domain of Starcraft 2 to achieve good performance with reinforcement learning. [VBC+19]

A league was then created to match agents against each other, just like human players would play ranked ladder games against each other. Each agent learned from games played against each other. Matchmaking used three different strategies. First group, the main agents used prioritized fictitious self-play [Bro51], a mechanism that utilizes mixture of probabilities proportionally to the win rate against other agents. This mixture converges to Nash equilibrium in two-player zero sum games. This encourages playing against opponents of about equal level and opponents that are problematic. Second group, the exploiters only were matched against main agents to encourage them to address their weaknesses. Third and last group were the league exploiters tasked to find the weaknesses of the entire league. They were match-maked similarly to main against but not against main exploiters. Main and league exploiters were reinitialized from time to time, but main agents remained in the league the whole time. Leagues consisted of one main agent of each race, one main exploiter of each race and two league exploiters of each race. Over time the performance of main agents increased and the performance of exploiters decreased [VBC+19].

This league process allowed agents to constantly learn against the best learned strategies while constantly exploring the vast strategic space of Starcraft 2. When new strategies emerged from adding new agents to the league, some were refinement of old strategies and some were counter-strategies to popular strategies in the league. Early in the league cheesy, but easy to counter, strategies were favored, but these were later discarded as training progressed. To encourage diversity in training, each agent was given a learning objective. This objective could be to defeat a single strategy, or to prefer a specific unit, etc. This helped the main agents grow stronger when they were exposed to cheesy strategies or strategies to exploit their weaknesses. Typical weakness of self-play is forgetting: agents forget strategies needed to perform well against strategies encountered further back in timeline in favor of performing well against strategies encountered more recently. Or they may

run into a cycle of moving between few known strategies and newer really improving. This was mitigated by letting agents in the league also play against former versions of themselves. The idea of the league is to produce a single strong agent, not to maximize the reward of every agent [VBC+19].

During training of 44 days on 32 third generation tensor processing units, each AlphaStar main agent playing the league received equal of 200 years of real-time Starcraft 2 playing experience. The final AlphaStar agent consisted of Nash distribution of the mixture of strategies discovered in the league. This single agent could be run real-time on a single modern desktop GPU. Alphastar was then finally tested at the Battle.net official matchmaking system. In the final results, Alphastar ranked at grandmaster level and in the top 0.02 percentile of human players in the ranking. Grandmaster is the highest tier of ranked ladder. Professional players defeated by the Alphastar commented that playing against it felt like not playing against superhuman opponent. It was strong at some aspects of the game and weak in some and they felt it was not undefeatable. Alphastar played with skill and reflexes of professional player, but with a unusual strategy and style entirely of it's own. Like previously seen with AlphaGo's gameplay, AlphaStar was able to develop playing style and strategies previously unexplored by human players. This could lead to human players increasing in skill and adapting new strategies to their gameplay. [VBC+19]

4.4 Quake 3 Arena

Historically, the most popular type of online video games is the first person shooter games. They provide a complex multi-agent environment where agents need to learn to interact with their opponents and cooperate with their teammates. Challenges lie in learning team play, strategy, tactics and hand-eye coordination. Learning actions needs to be done from raw-pixel input stream and actions produced through virtual controller to constrain AI agents reaction times to similar level of professional human players.

Counter Strike is one of the most popular first person online shooter games, played competitively for more than 20 years. Counter Strike is classical two team game where one team plays terrorist trying to plant a bomb at one of two sites at a three dimensional varying map. Other team plays counter-terrorist tasked with preventing planting of the bomb and defeating terrorist. Newer and very popular category of these competitive first person shooter video games are so called 'battle

royale' games where last man standing wins in a vast game map that gets smaller over time. Most popular games in this category are Fortnite and Player Unknown's Battleground. Counter Strike is the most watcher friendly competitive online video game of first person shooter games providing exciting, gripping, edge-of-your-seat type of watching experience.

However, there are as of yet no available deep learning solutions for these complex modern online first person shooter games. Advancements in deep learning during last decade, especially in the area of computer vision and the ability to develop learning algorithms from the pixel input on the screen has led to success in 2-dimensional Atari games [MKS+13]. This has lead to research in pseudo 3-dimensional precursor of all first person shooter games from the early 90's: the Doom. [KWR+16, LC17] Results in Doom are promising, deep recurrent Q-learning framework for POMDP's[LC17] presented was able to train agents to outperform ingame bots and human players in a deathmatch scenario and to be able to keep up the same performance in new maps not present during training [LC17]. In deathmatch gameplay, each player plays against each other player trying to score as many kills as possible.

Recently, advancements have been made in a slightly more modern 3-dimensional game, classic capture the flag gameplay in modified version of Quake 3 Arena [ID], which has laid foundation for many modern competitive online first person shooter video games. Quake 3 Arena, along its successors, still has an alive professional scene. In Quake 3 Arena capture the flag (CTF) game mode players have a flag at their bases. Both teams have a symmetrical objective of capturing enemy flag and safely carrying it to their own base while defending their own flag. Team with most captured flags after five minutes wins the game. Conflicts when encountering other team's members are sorted out with various types of firearms. Rules of CTF are simple but the gameplay is complex.

AI agent dubbed 'for the win' (FTW) [JCD+19a] demonstrated ability to play at high competitive level at a multi-player first person online video game of modified Quake 3. Supplementary material for the FTW containing gameplay videos can be found on the blog post for the FTW[JCD+19b]. It become much stronger than baseline AI bots and strong human players. Gameplay against professional human players was not conducted. To level the playing field with human players, FTW uses the same visual information that is available to human players. Learning was done from raw RGB pixel input stream and actions issued through a virtual controller.

Humans have very slow sensory input to biomotoric response compared to AI agent and this needed to be leveled to not give AI agents a superhuman advantage that humans could not match. Agents actions were delayed by 267ms witch is comparable to average human players. Maps were randomized for each match to avoid agents memorizing map layouts and forcing them to learn more general and robust methods of play in variety of team compositions. Professional human players will memorize maps and practice the nuances of each map and adopt different strategies for each map. However, main strategic concepts remain the same. Each agent has its own reward signal which allows it to generate its own goal. This can be for example to capture enemy flag or defend own flag.

Figure 14: FTW agent from a red team chasing blue teams flag carrier, trying to tag them, in a game of capture the flag in a simplified version of Quake 3. Taken from "FTW blog post" at https://deepmind.com/blog/article/capture-the-flag-science on 28.10.2020. [Ope18]

Agent's policy π is parametrized by the multi-timescale recurrent neural network with external memory [GWR+16]. This is constructed by two LSTM[HS97] recurrent neural networks with different timescales: for fast and slow timescales with a shared memory module. This allows agent to react to immediate game events while executing a long term strategy or game plan. Input x_t is drawn from the same pixel input on the screen as human players at each time step t. Actions $a_t \sim \pi$ are

generated conditionally on a stochastic latent variable, whose distribution is modulated by slowly evolving prior process. These actions are then simulated with virtual gamepad. The resulting model constructs a temporally hierarchical representation space in a novel way to promote the use of memory and temporally coherent action sequences.

For ad hoc teams, a agent should try to optimize their policy π_0 for maximizing the probability of winning for their team $\pi_0, \pi_1, ..., \pi_{\frac{N}{2}-1}$:

$$\mathbb{P}(\pi_0\text{'s team wins}|\omega, (\pi_n)_{n=0}^{n-1}) = \mathbb{E}_{a\sim(\pi_n)_{n=0}^{n-1}}[\pi_0, \pi_1, ..., \pi_{\frac{N}{2}-1} * win * \pi_{\frac{N}{2}}, ..., \pi_{N-1}], \quad (35)$$

where N is the total number of players in the game and $*win*$ is winning operator which returns 1 if the team on left wins and 0 if the team on the right wins. Ties are resolved randomly. $\omega \sim \Omega$ represents specific map instance generated randomly. Giving rewards only at the end would result in too sparse reward function. Rewards are instead given accordingly to $r_t = w(p_t)$, according to points agent receives on the game's scoreboard, giving a dense internal reward function [SLBS10]. This allows agents to learn a transformation w such that the policy optimization on the internal rewards r_t for the win. Hence the name 'for the win' or FTW agent. [JCD+19a]

Agents are trained from scratch to cooperate and compete in a completely unknown environment. Rather than training a single agent, in FTW a population of 30 different agents are trained by playing with each other to provide diversity. Their levels are estimated by calculating ELO scores used in Chess and various other games and then matched against and as teammates agents of similiar level with stochastic match-making scheme[JCD+19a]. Thousands of CTF games are played simultaneously in parallel. This population of agents is also used to meta-optimize the internal rewards and hyperparameters of the RL process itself. This results in two tier RL problem[JCD+19a]:

$$J_{inner}(\pi_p|\mathbf{w}_p) = \mathbb{E}_{t\sim m_p(\boldsymbol{\pi}),\omega\sim\Omega}\mathbb{E}_{\mathbf{a}\sim\boldsymbol{\pi}}[\sum_{t=0}^{T}\gamma^t \mathbf{w}_p(\rho_p, t)]\ \forall \pi_p \in \boldsymbol{\pi} \quad (36)$$

$$J_{outer}(\mathbf{w}_p, \boldsymbol{\phi}_p|\boldsymbol{\pi}) = \mathbb{E}_{t\sim m_p(\boldsymbol{\pi}),\omega\sim\Omega}\mathbb{P}(\pi_p^{\mathbf{w},\boldsymbol{\phi}}\text{'s team wins}|\omega, \pi_t^{w,\boldsymbol{\phi}}) \quad (37)$$

$$\pi_p^{\mathbf{w},\boldsymbol{\phi}} = \text{optimize}_{\pi p}(J_{inner}, \mathbf{w}.\boldsymbol{\phi}) \quad (38)$$

The inner problem is to maximize each agents future discounted internal rewards, which is solved by reinforcement learning in POMDP, since agents do not have

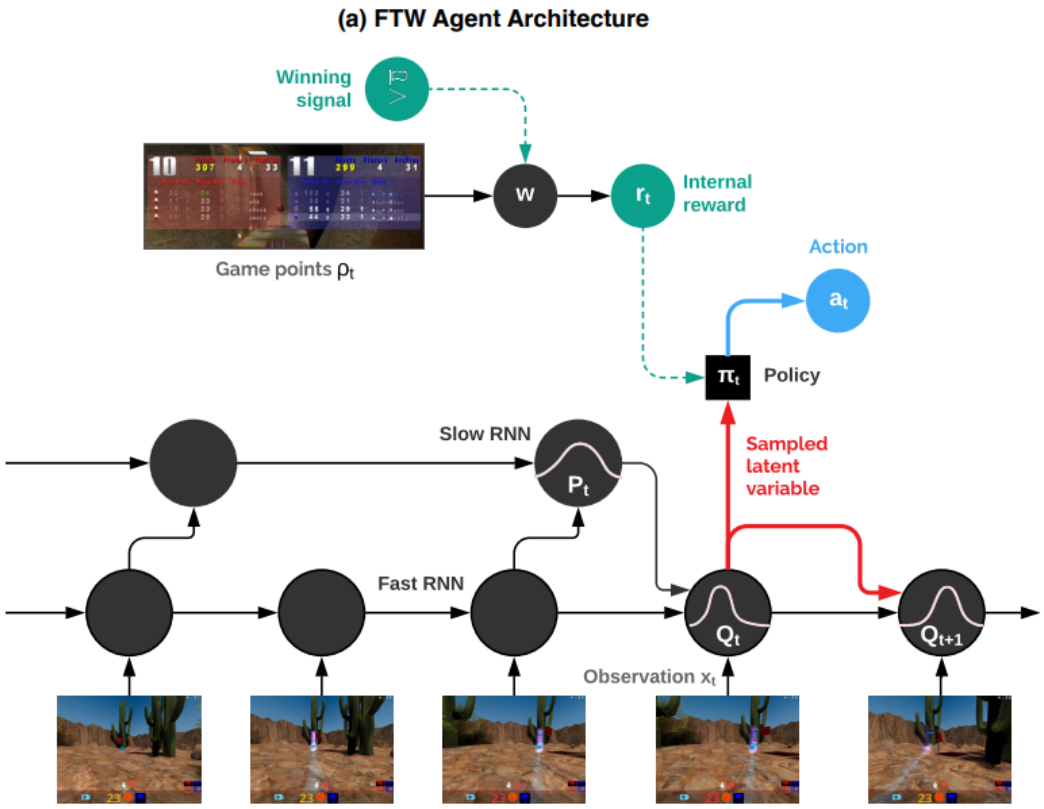


Figure 15: Simplified version of the FTW agents architecture. This shows how the agents processes its temporal sequence of observations on two different time scales, faster at the bottom, slower at the top. A stochastic vector-valued latent variable is sampled at the fast time scale from distribution Q_t based on observations $\mathbf{x}_t$. The policy π_t (action distribution) is sampled conditional on the latent variable at each time step t. The latent variable is regularised by the slow moving prior P_t which helps capture long-range temporal correlations and promotes memory. The network parameters are updated using reinforcement learning based on the agent's own internal reward signal r_t , which is obtained from a learnt transformation $\mathbf{w}$ of game points p_t. $\mathbf{w}$ is optimised for winning probability through population based training, another level of training performed at yet a slower time scale than RL. Image taken from FTW blog site at https://deepmind.com/blog/article/capture-the-flag-science on 28.10.2020. [JCD+19b]

full information of the environment available at each time step. Each agent should then maximize their expected cumulative γ-discounted reward $\mathbb{E}_{\pi(\cdot|x\leq t)}[R_t]$ under a policy conditioned by the agents cumulative history of observations, where $R_t = \sum_{k=0}^{T-t} \gamma^k r_{t+k}$. Due to the ambiguity of the true state given by observation in POMDP, $\mathbb{P}(\mathbf{s}_t|\mathbf{x}_{\leq t})$ represents the current state as a random variable, such that the value fuction $V_t = \mathbb{E}_{\pi(\cdot|\mathbf{x}_{\leq t})}[R_t] = \sum_x \mathbb{P}(\mathbf{s}|\mathbf{x}_{\leq t})\mathbb{E}_{\pi(\cdot|\mathbf{s})}[R_t]$. Using Kullback-Leiber divergence, where policy $\mathbb{Q}$ is regularized against a prior policy $\mathbb{P}$ with a latent variable $\mathbf{z}_t$. $\mathbf{z}_t$ models the dependency on past observations and this leads to an objective function:

$$\mathbb{E}_{\mathbb{Q}(\mathbf{z}_t|C_t^q)}[R_t] - D_{KL}[\mathbb{Q}(\mathbf{z}_t|C_t^q)||\mathbb{P}(\mathbf{z}_t|C_t^p)], \tag{39}$$

where $\mathbb{P}(\mathbf{z}_t|C_t^p)$ is prior distribution on $\mathbf{z}_t$, $\mathbb{Q}(\mathbf{z}_t|C_t^q)$ is variational posterior distributions on $\mathbf{z}_t$ and D_{KL} is the Kullback-Leiber divergence. The sets of conditioning variables C_t^p and C_t^q determine the structure of the probabilistic model of the agent, and can be used to equip model with various representational priors. The conditioning variables C_t^p and C_t^q are chosen such that forward planning and memory are promoted.[JCD+19a]

The hierarchical LSTM's fast timescale generates a hidden state $\mathbf{h}_t^q$ at every time step t, whereas the slow timescale produces an updated hidden state $\mathbf{h}_t^p = \mathbf{h}_{\tau\lfloor\frac{t}{\tau}\rfloor}$ every τ timesteps. Output of the fast LSTM is used as the variational posterior $\mathbb{Q}(\mathbf{z}_t|\mathbb{P}(\mathbf{z}_t), \mathbf{z}_{\leq t}, \mathbf{x}_{\leq t}, a_{\leq t}, r_{\leq t}) = \mathcal{N}(\mu_t^q, \Sigma_t^q)$, where mean μ_t^q and covariance Σ_t^q of the normal distribution are parametrised by the linear transformation $(\mu_t^q, \log\sigma_t^q) = f_q(\mathbf{h}_t^q)$. At each time step a sample $\mathbf{z}_t \sim \mathcal{N}(\mu_t^q, \Sigma_t^q)$ is taken. The slow timescale LSTM output is used for the prior of $\mathbb{P}(\mathbf{z}_t|\mathbf{z}_{\leq\tau\lfloor\frac{t}{\tau}\rfloor}, \mathbf{x}_{\leq\tau\lfloor\frac{t}{\tau}\rfloor}, a_{\leq\tau\lfloor\frac{t}{\tau}\rfloor}, r_{\leq\tau\lfloor\frac{t}{\tau}\rfloor}) = \mathcal{N}(\mu_t^p, \Sigma_t^p)$, where mean μ_t^p and covariance Σ_t^p of the normal distribution are parametrised by the linear transformation $(\mu_t^p, \log\sigma_t^p) = f_p(\mathbf{h}_t^p)$.[JCD+19a]

The fast timescale core of LSTM takes as an input the observation (of the pixels from the screen) that has been decoded by a convolutional neural network, $\mathbf{u}_t = CNN(\mathbf{x}_t)$, the previous action a_{t-1}, previous reward r_{t-1}, the prior parameters μ_t^p and σ_t^p, and the previous sample of the variational posterior $\mathbf{z}_{t-1} \sim \mathcal{N}(\mu_{t-1}^q, \Sigma_{t-1}^q)$. The slow timescale core of LSTM takes in the fast core's hidden state as an input, resulting in recurrent neural network with dynamics of:

$$\mathbf{h}_t^q = g_p(\mathbf{u}_t, a_{t-1}, r_{t-1}, \mathbf{h}_t^p, \mathbf{h}_{t-1}^q, \mu_t^p\Sigma_{t-1}^q, \mathbf{z}_{t-1}) \tag{40}$$

$$\mathbf{h}_t^p = \begin{cases} g_p(\mathbf{h}_{t-1}^q, \mathbf{h}_{t-1}^p) & \text{if } t \mod \tau = 0 \\ \mathbf{h}_{\tau\lfloor\frac{t}{\tau}\rfloor}^p & \text{otherwise,} \end{cases} \tag{41}$$

where g_q and g_p are fast and slow timescale LSTM cores. Stochastic policy, value function, and pixel control signals are obtained from $\mathbf{z_t}$ using further non-linear transformations. This gives the gradient update rule of:

$$\nabla(\mathbb{E}_{\mathbf{z}_t\sim\mathbb{Q}}[-\mathfrak{L}(\mathbf{z}_t, \mathbf{x}_t)] \\ - D_{KL}[\mathbb{Q}(\mathbf{z}_t|\mathbb{P}(\mathbf{z}_t), \mathbf{z}_{\leq t}, \mathbf{x}_{\leq t}, a_{\leq t}, r_{\leq t})||P(\mathbf{z}_t|\mathbf{z}_{\leq\tau\lfloor\frac{t}{\tau}\rfloor}, \mathbf{x}_{\leq\tau\lfloor\frac{t}{\tau}\rfloor}, a_{\leq\tau\lfloor\frac{t}{\tau}\rfloor}, r_{\leq\tau\lfloor\frac{t}{\tau}\rfloor})]), \tag{42}$$

where $C_t^q = \mathbb{P}(\mathbf{z}_t), \mathbf{z}_{\leq t}, \mathbf{x}_{\leq t}, a_{\leq t}, r_{\leq t}$, $C_t^p = \mathbf{z}_{\leq\tau\lfloor\frac{t}{\tau}\rfloor}, \mathbf{x}_{\leq\tau\lfloor\frac{t}{\tau}\rfloor}, a_{\leq\tau\lfloor\frac{t}{\tau}\rfloor}, r_{\leq\tau\lfloor\frac{t}{\tau}\rfloor}$ and $\mathfrak{L}$ represents the objective function composed of terms for multi-step policy gradient and value function optimization, as well as pixel control and reward prediction auxilary tasks. This objective function and gradient update rule tries to capture the idea that slow LSTM core generates a prior on $\mathbf{z}$, which predicts the evolution of $\mathbf{z}$ for the subsequent τ time steps, while fast LSTM core generates a variational posterior on $\mathbf{z}$ that incorporates in new observations, but adheres to the predictions made by the prior. $\mathbf{z}$ must be chosen as a useful representation for maximising reward and auxiliary task performance. This architecture could be expanded to more than two layers, but in this task it was noted to make little difference. [JCD+19a]

The outer meta-optimization process on the other hand requires population based training[JDO+17] (PBT) to optimize hyperparameters ϕ of the agent, learning rate, slow LSTM time scale τ, the weight of D_{KL} term and the entropy cost. This evolutionary process allows a model selection by replacing under-performing agents (under 70% win probability) with mutated versions of themselves and with copies of other the agents. This joint optimization of the agents policy with RL and the optimization of the RL procedure towards a higher level goal proved to be effective. It utilizes the power of RL and evolutionary based methods in large scale system. [JCD+19a]

To evaluate the agents, a tournament with scripted bots and human players with first-person shooter experience was conducted. In each match human players, scripted bots and trained FTW agents were randomly matched up as teammates and opponents in two player teams. FTW agents were able to achieve much higher performance than human players and scripted bots especially in never before seen maps. Team of two game testers were able to win only 25% of the games against FTW agent teams. Interestingly, a agent-human team was able to overperform agent-agent teams. This suggests that agents are able to cooperate with previously unknown

teammates. In the experiments, it was noted that human players were much more proficient at 'tagging' opponents at long range with 17% success rate compared to FTW agents 0.5%. But at the short range, FTW agents dominated human players with accuracy of 80% compared to human players 48%. This suggests that FTW agents learned to utilize their superior short-range accuracy, but even when it was artificially reduced to the same level as humans, FTW agents retained their superior win-rate at matches. This means, that it was not the only contributing factor to their performance and could utilize other features when their accuracy was reduced. Participants in experiment reported that FTW agents were more team-play oriented than human players. This shows emergent complexity in behaviours of agents trained without any prior knowledge of the game with reinforcement learning methods. FTW was particularly effective at executing game actions resulting in scoring more points, eg. capturing the flag. They averaged a much higher count of capture the flag and recover the flag actions than strong human players [JCD+19a].

Agents were able to spontaneously learn a rich representation of the game without being explicitly taught this knowledge by purely RL-based training. This included concepts such as "I have the flag", "I am respawning", "My flag is taken", "Teammate has the flag" etc. As the learning period progressed agents knowledge of the game concepts grew. First, agent learned basics of the game, then navigation, coordination and tagging skills. Finally, agents started to learn strategies of the game. Fundamental game concepts such as defending your own flag and capturing enemy flag are learned during the process. Learned important game behaviours are defending home base, 'camping' at opponents base, waiting for opponent to appear with 'camping' player team's flag or opponents flag to reappear after capture, and following teammates [JCD+19a].

5 Discussion

In recent years, it has been shown that super-human performance in game playing can be achieved with the combination of reinforcement learning and deep networks: deep reinforcement learning. This started in the late 90's when DeepBlue, an AI agent pre-seeded with knowledge of classic opening moves of chess, beat the best human player. Then, in 2015 AlphaGo AI agent demonstrated similar performance in the game of Go with reinforcement learning agent using deep networks and Monte Carlo method. It was also pre-seeded with replays of human play. Then, just two

years later a general purpose deep reinforcement learning agent AlphaZero demonstrated super-human performance in the games of Go, Chess and Shogi. When these classical two player adversarial full information games were solved, eyes turned to online video games. These new types of games from the last few decades offer new challenges: partial observability, multi-agent environments, semi-cooperative gameplay and games taking tens of thousands of actions with long time dependencies.

Progress has been fast and just two years after the AlphaZero, in 2019, competitive professional level performance has been achieved in real-time strategy game of Starcraft 2 and multiplayer online battle arena game of Dota 2. The results and approaches of these games differed in key aspects. In an unrestricted game of Starcraft 2, AI agent called AlphaStar was pre-seeded with replays of strong human player before self-play against each other in a league. The end result was that the agent was able to defeat some of the best professional human players. In Dota2, the game was somewhat restricted, but with just self-play, AI agent called OpenAIFive was able to defeat the reigning world champion team. Additionally, strong human performance was achieved in a restricted version of Quake 3 capture the flag gameplay. This was achieved by FTW agent when learning from the pixel input of the screen and using just self-play. Still, a small step needs to be made to achieve a super-human performance in these games with a general purpose deep reinforcement learning algorithm without prior human knowledge in unrestricted gameplay.

Recent open-ended learning systems utilizing reinforcement learning with self-play and deep networks have achieved impressive results in increasingly dynamic, complex and challenging domains. In game of Starcraft 2 without any limitations, strong Grandmaster level was reached when combined with supervised learning from human play as the starting point. With just self-play, in restricted game of Dota 2 professional team of human players was defeated. Results in these games show that current learning algorithms combined with deep networks may be efficient enough when run on sufficient scale and with a reasonable, fine tuned, way of exploring. Both Alphastar and OpenAI Five were able to learn new strategies and playing styles unlike human play, which showed that like in the case of AlphaGo, these games have strategies previously unexplored by human players. Alphastar was given replays of human play as as starting point, but OpenAI Five was not. Still, even OpenAI Five was able to learn macro-level action chains from micro-level actions with just self-play.

Starcraft, a popular online video game of its time and the precursor of Starcraft 2,

has been studied in depth as a research platform for game AI development [OSU+13]. Results show, however, that so far there is no AI agent that would be competitive even with a strong human player, let alone with the best professional players. Many of the best AI agents relied on hard-coded strategies. In addition, in many of these attempts have had to decrease the complexity of this game to make it more feasible with traditional machine learning techniques. This includes allowing AI agents to have full visibility, limiting the available amount of units, structures and limiting the tech three.

To have any chance of developing a feasible AI agent with reinforcement learning, it was generally long believed that new advances, such as hierarchical neural networks [BM03, VOS+17], are necessary for managing long time horizons and partial observability. AlphaStar and OpenAIFive used an LSTM recurrent neural network, FTW agent used multi-timescale recurrent neural network with external memory. The types of neural networks that are available today seem to be capable of handling long time dependencies and partial observability present in these games. One other leap has been the ability to learn from the pixel output of the screen. FTW agent was the only to do so, OpenAIFive and AlphaStar used a representation of the game state as an input. In games of Dota 2 and Starcraft 2 this is very reasonable, but in first person shooter games it is expected for the AI agent to use the pixel input of the screen in the same way as human player. Distributed learning framework with multiple agents learning simultaneously was one key feature in all of the frameworks. This requires setting up leagues for the agents and to have matchmaking scheme to match AI agents against each other in best manner possible. This does lead to more robust agents when they are not learning to play against just the current version of itself but also speeds up the learning. In FTW, stochastic gradient policy update was used to find the optimal policy for the agents. In this case, it seemed to perform well in a restricted Quake 3 ctf environment. OpenAIFive opted to use state of the art PPO, which makes distributed computation, tuning and maintaining of the learning algorithm much easier. Alphastar on the other hand went one step further to improve upon PPO in the environment of Starcraft 2 with Self-Imitation learning. Dota 2 and Starcraft 2 have such vast action spaces and games taking so many actions that basic stochastic gradient update scheme would not have been feasible, but PPO or its variants can help manage sampling. The final key to the puzzle of making deep reinforcement learning feasible in online video games was the introduction of GPU based computation and the increase in computational resources, since deep reinforcement learning algorithms are very computationally expensive.

Open-ended deep learning techniques used in all of these games may not generalize or scale well, as the case of Dota 2 shows. Even if these techniques may be applied to a variety of problems and they work well, this knowledge may not be transferable to even slightly modified environment. Agent-based games played against professional players were tested in games where the number of available heroes was limited to only 18 of 117. Even if deep learning techniques used were able to defeat top human players in restricted game using substantial amount of computational resources, this learned knowledge can't be transferred to using larger subset of heroes, or even the full hero pool. Instead, adding additional heroes from the original mirror match of five fixed heroes to a mix of five heroes from a pool of 18 heroes could only be done at the expense of more computational resources and learning time. Further increasing hero pool would cost more computational resources and more learning time, which would at some point, be infeasible. Increasing hero pool to 25 led to a significant increase in learning time for OpenAI Five showing that mastery of previous heroes did not help in learning additional heroes. When new patches were introduced to the game, making adjustments to maps, items, hero stats and abilities, the previous knowledge from previous version of the game could not be transferred automatically to the new version. Deep learning techniques indeed are very constrained to a specific problem in a specific environment with specific input to output solution using specific parameters and fine tuned attributes to steer the learning. Additionally, the use of special surgery technique was needed to adjust agents to a new version of the game. Otherwise, the learning would have needed to restart from the beginning when a new version of the game was released even if it made just minor adjustments to how the game is played. This could mean very poor generalization. [OB+19]

Encouraging results in the restricted version of Quake 3 ctf gameplay sparks hope that in coming years there could be a possibility to train AI agents in a more complex and very popular online first person shooter game of Counter Strike or Fortnite. A game of Counter Strike is closer to Quake ctf than a battle royale game of Fortnite. A game of Counter Strike would need to have cooperation from five AI agents trying to defeat another team of five players in a slightly more complex game of bomb planting and bomb defuse. Results of FTW agents show that two agent team cooperation works. This could be increased to five players with more training time and computational resources. FTW agents also were able to learn core concepts of the gameplay even when they were just learning from the pixel input from the screen. This gives hope, that in the future, the concepts of more

complicated first-person shooter games would not remain elusive.

During the late stages of writing of this thesis encouraging results from a driving simulator of Gran Turismo sport were released [FSK+20]. They managed to achieve super-human performance in a challenging environment of autonomous high speed car driving with deep neural networks and reinforcement learning. With just 73 hours of training time on 4 Playstation 4 consoles and one desktop pc. This was done without any expert human knowledge, without any explicit path finding on the tracks or any human intervention. The AI agent was able to outperform the best known human lap time on two different tracks and two different cars. Unfortunately, there was not enough time to include these results in more detail in this thesis. If you would like to find out more about this, I encourage you to read paper 'Super-Human Performance in Gran Turismo Sport Using Deep Reinforcement Learning' by Fuchs et al [FSK+20].

Both Alphastar and OpenAI five show that deep learning techniques are very resource and/or data hungry. Using self-play for learning costs a lot of computational resources and time. Learning can be sped up using recorded human play to set up the neural network layers before starting self play. But, this needs a lot of data, tens of thousands hours of game play from thousands of different players. Results in games like Starcarft 2 and Dota 2 show that one of the biggest constraints in these strategic video games still lie in computational resources, where deep learning techniques are effective. AlphaZero, a general deep learning framework used about 1900 petaflop/s-days in training compute[AH19] when training to defeat the best GO player in the world. Sufficient scale in Open AI five was using about 800 petaflop/s-days in training compute [OB+19], which is still significantly less than in a 'simpler' game of GO. In AlphaStar, each of the agents in the league was trained using 32 third-generation tensor processing units over 44 days [OB+19]. These are significant uses of computational resources that do not come cheap in money or natural resources. Case of Gran Turismo on the other hand, shows that results can be achieved with just 73 hours of training time on 4 Playstation 4 consoles and one desktop pc in certain environments.[FSK+20] Is it worth it and can we afford economically and environmentally to use these amounts of computational resources to solve these kinds of complex problems? Do we still need to do research to find more inexpensive methods or just wait until we have even more computational power to solve even more complex problems? It is estimated that AI computational resources double every 3.5 months[AH19]. If we can put our limited resources to use in improving our lives and our surroundings, maybe it will be worth it.

Another issue with current deep learning techniques is that it may be incredibly difficult to fine-tune their hyper parameters to a certain task without any generalization to even slightly variant task. This also complicates the reproducibility of these results and application of them to new environments. It remains to be seen when we can apply deep learning techniques to real world applications. Computer vision[VDDP18] has taken leaps and bounds ahead, self-driving cars[BTD+16] are getting ready to the real world, but not yet ready to be fully autonomous. The general-purpose deep learning algorithm used in Dota 2 has already been successfully applied in control of a robotic arm [ABC+20]. Results produced by deep reinforcement learning techniques cannot be denied, but they do come with their caveats, at least for now.

6 Conclusion

This thesis briefly introduced the concepts of reinforcement learning, deep networks, deep reinforcement learning and how to solve reinforcement learning problems in partially observable environments with deep reinforcement learning by combining deep networks with reinforcement learning techniques. This included the standard reinforcement learning model, convolutional network, recurrent neural network and LTSM network examples of deep neural networks along with stochastic gradient descent update rule for optimizing networks performance. The appliction of these deep networks to reinforcement learning leads to various deep reinforcement learning algorithms capable of handling partially observable environments. DQN, actor-critic model and PPO were introduced.

After the classical games of Go, Chess and Shogi were solved, new frontier for game AI research needed to be looked into. Three competitive online video games of different types (moba, rts, fps), Defence of the Ancients 2 (Dota 2), Starcraft 2, and Quake 3, were looked into as deep reinforcement learning research platforms. These games brought to the table new challenges: with vast action and observation spaces, fast paced gameplay, long term and sparse reward assignment, and partially observable complex game environments with multiple agents.

The different state of the art deep network architectures and reinforcement learning algorithms utilizing them were looked into in this thesis and their results in these three games were impressive. Previously, AI agents could not compete with even good human players, let alone professional players in video games and they were

limited to scripted AI's or given unfair advantages. Three AI agents looked into shared some features but differed in others, but the principles were fairly close to each other. FTW agent in Quake 3 utilized a more traditional approach of policy gradient updates, but also adding a meta-optimization layer with population based methods. OpenAI Five in Dota 2 opted to use PPO for policy optimization and Starcraft 2 used a novel UPGO algorithm building upon actor-critic model and V-trace for policy updates. Each of these utilized a LTSM neural network, highly useful recurrent neural network for long time dependencies and partially observable environments. All of these solution also had a league set up to self-play agents against each other in the learning period with various methods for matchmaking and evaluating agents performance. A highly competitive level of AI was developed in Quake 3 and in games of Starcraft 2 and Dota 2, even the best professional players were defeated. Alphastar used human knowledge to kick-start learning like was done in AlphaGo, but OpenAI Five and Dota 2 opted to go with straight RL method. This meant that the games needed to be somewhat restricted in scale.

These results were then discussed with the strengths, limitations and the future applications of these solutions, and with their computational cost in mind. Success in this new field was expected to a certain degree, but as to how fast those results were achieved after the results in classic board games was unexpected. Limitations and approaches were very understandable, since the environments of video games are many magnitudes more complex than of classic board games. With the latest algorithms and architectures, and enough computational power, in the coming years we might see general purpose deep reinforcement learning system like AlphaZero was to classic board games to competitive online video games or application in the real world.

References

ABC+20 Andrychowicz, O. M., Baker, B., Chociej, M., Jozefowicz, R., McGrew, B., Pachocki, J., Petron, A., Plappert, M., Powell, G., Ray, A. et al., Learning dexterous in-hand manipulation. *The International Journal of Robotics Research*, 39,1(2020), pages 3–20.

ADBB17 Arulkumaran, K., Deisenroth, M. P., Brundage, M. and Bharath, A. A., Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, 34,6(2017), pages 26–38.

AH19 Amodei, D. and Hernandez, D., Ai and compute, 2019. URL `https://openai.com/blog/ai-and-compute/`. Accessed on 20.10.2020.

Bel66 Bellman, R., Dynamic programming. *Science*, 153,3731(1966), pages 34–37.

Bli Blizzard, Starcraft ii official game site. URL `https://starcraft2.com/en-us/`. Accessed on 20.10.2020.

BM03 Barto, A. G. and Mahadevan, S., Recent advances in hierarchical reinforcement learning. *Discrete event dynamic systems*, 13,1-2(2003), pages 41–77.

Bro51 Brown, G. W., Iterative solution of games by fictitious play. *Activity analysis of production and allocation*, 13,1(1951), pages 374–376.

BTD+16 Bojarski, M., Testa, D. D., Dworakowski, D., Firner, B., Flepp, B., Goyal, P., Jackel, L. D., Monfort, M., Muller, U., Zhang, J., Zhang, X., Zhao, J. and Zieba, K., End to end learning for self-driving cars, arXiv:1604.07316v1 [cs.CV], 2016.

CB98 Claus, C. and Boutilier, C., The dynamics of reinforcement learning in cooperative multiagent systems. *AAAI/IAAI*, 1998, pages 746–752.

CHJH02 Campbell, M., Hoane Jr, A. J. and Hsu, F.-h., Deep blue. *Artificial intelligence*, 134,1-2(2002), pages 57–83.

DHK13 Deng, L., Hinton, G. and Kingsbury, B., New types of deep neural network learning for speech recognition and related applications: An overview. *2013 IEEE international conference on acoustics, speech and signal processing*. IEEE, 2013, pages 8599–8603.

ESM+18 Espeholt, L., Soyer, H., Munos, R., Simonyan, K., Mnih, V., Ward, T., Doron, Y., Firoiu, V., Harley, T., Dunning, I., Legg, S. and Kavukcuoglu, K., Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures, 2018.

FSK+20 Fuchs, F., Song, Y., Kaufmann, E., Scaramuzza, D. and Duerr, P., Super-human performance in gran turismo sport using deep reinforcement learning, arXiv:2008.07971v1 [cs.AI], 2020.

GBC16 Goodfellow, I., Bengio, Y. and Courville, A., *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

GM10 Guo, H. and Meng, Y., Distributed reinforcement learning for coordinate multi-robot foraging. *Journal of intelligent & robotic systems*, 60,3-4(2010), pages 531–551.

GWR+16 Graves, A., Wayne, G., Reynolds, M., Harley, T., Danihelka, I., Grabska-Barwińska, A., Colmenarejo, S. G., Grefenstette, E., Ramalho, T., Agapiou, J. et al., Hybrid computing using a neural network with dynamic external memory. *Nature*, 538,7626(2016), pages 471–476.

Hal13 Halmos, P. R., *Measure theory*, volume 18. Springer, 2013.

Hau00 Hauskrecht, M., Value-function approximations for partially observable markov decision processes. *Journal of Artificial Intelligence Research*, 13,1(2000), page 33–94. URL `http://dx.doi.org/10.1613/jair.678`.

HS97 Hochreiter, S. and Schmidhuber, J., Long short-term memory. *Neural computation*, 9,8(1997), pages 1735–1780.

HS17 Hausknecht, M. and Stone, P., Deep recurrent q-learning for partially observable mdps, arXiv:1507.06527v4 [cs.LG], 2017.

HZRS15 He, K., Zhang, X., Ren, S. and Sun, J., Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *Proceedings of the IEEE international conference on computer vision*, 2015, pages 1026–1034.

ID ID, Quake iii wikipedia entry. URL `https://en.wikipedia.org/wiki/Quake_III_Arena`. Accessed on 20.10.2020.

IL67 Ivakhnenko, A. G. and Lapa, V. G., *Cybernetics and forecasting techniques*. Mod. Analytic Comput. Methods Sci. Math. North-Holland, New York, NY, 1967. URL `https://cds.cern.ch/record/209675`. Trans. from the Russian, Kiev, Naukova Dumka, 1965.

JCD+19a Jaderberg, M., Czarnecki, W. M., Dunning, I., Marris, L., Lever, G., Castañeda, A. G., Beattie, C., Rabinowitz, N. C., Morcos, A. S., Ruderman, A., Sonnerat, N., Green, T., Deason, L., Leibo, J. Z., Silver, D., Hassabis, D., Kavukcuoglu, K. and Graepel, T., Human-level performance in 3d multiplayer games with population-based reinforcement learning. *Science*, 364,6443(2019), pages 859–865. URL `https://science.sciencemag.org/content/364/6443/859`.

JCD+19b Jaderberg, M., Czarnecki, W. M., Dunning, I., Marris, L., Tracey, B., Lever, G., Castaneda, Antonio, G., Beattie, C., Rabinowitz, N., Morcos, A., Ruderman, A., Sonnerat, N., Green, T., Deason, L., Leibo, Joel, Z., Silver, D., Hassabis, D., Kavukcuoglu, K. and Graepel, T., Capture the flag: the emergence of complex cooperative agents, 2019. URL `https://deepmind.com/blog/article/capture-the-flag-science`. Accessed on 30.10.2020.

JDO+17 Jaderberg, M., Dalibard, V., Osindero, S., Czarnecki, W. M., Donahue, J., Razavi, A., Vinyals, O., Green, T., Dunning, I., Simonyan, K., Fernando, C. and Kavukcuoglu, K., Population based training of neural networks, arXiv:1711.09846v2 [cs.LG], 2017.

JSJ95 Jaakkola, T., Singh, S. P. and Jordan, M. I., Reinforcement learning algorithm for partially observable markov decision problems. *Advances in neural information processing systems*, 1995, pages 345–352.

KB17 Kingma, D. P. and Ba, J., Adam: A method for stochastic optimization, arXiv:1412.6980, 2017.

KLC98 Kaelbling, L. P., Littman, M. L. and Cassandra, A. R., Planning and acting in partially observable stochastic domains. *Artificial intelligence*, 101,1-2(1998), pages 99–134.

KLM96 Kaelbling, L. P., Littman, M. L. and Moore, A. W., Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4,1(1996), pages 237–285.

KRS+19 Kurach, K., Raichuk, A., Stańczyk, P., Zając, M., Bachem, O., Espeholt, L., Riquelme, C., Vincent, D., Michalski, M., Bousquet, O. and Gelly, S., Google research football: A novel reinforcement learning environment, arXiv:1907.11180v2 [cs.LG], 2019.

KT00 Konda, V. R. and Tsitsiklis, J. N., Actor-critic algorithms. *Advances in neural information processing systems*, 2000, pages 1008–1014.

KWR+16 Kempka, M., Wydmuch, M., Runc, G., Toczek, J. and Jaśkowski, W., Vizdoom: A doom-based ai research platform for visual reinforcement learning. *2016 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE, 2016, pages 1–8.

LBD+89 LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W. and Jackel, L. D., Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1,4(1989), pages 541–551.

LC17 Lample, G. and Chaplot, D. S., Playing fps games with deep reinforcement learning. *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

Li18 Li, Y., Deep reinforcement learning: An overview, arXiv:1701.07274v6 [cs.LG], 2018.

Lit94 Littman, M. L., Markov games as a framework for multi-agent reinforcement learning. *ICML*, volume 94, 1994, pages 157–163.

Mat94 Mataric, M. J., Learning to behave socially. *Third international conference on simulation of adaptive behavior*, volume 617. Citeseer, 1994, pages 453–462.

Mat97 Matarić, M. J., Reinforcement learning in the multi-robot domain. In *Robot colonies*, Springer, 1997, pages 73–83.

MBM+16 Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D. and Kavukcuoglu, K., Asynchronous methods for deep reinforcement learning. *International conference on machine learning*, 2016, pages 1928–1937.

MHN13 Maas, A. L., Hannun, A. Y. and Ng, A. Y., Rectifier nonlinearities improve neural network acoustic models. *Proc. icml*, volume 30, 2013, page 3.

MIJD17 Metz, L., Ibarz, J., Jaitly, N. and Davidson, J., Discrete sequential prediction of continuous actions for deep rl, arXiv:1705.05035v3 [cs.LG], 2017.

Mil19 Millington, I., *AI for Games*. CRC Press, 2019.

MKS+13 Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. and Riedmiller, M., Playing atari with deep reinforcement learning, arXiv:1312.5602v1 [cs.LG], 2013.

MKS+15 Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G. et al., Human-level control through deep reinforcement learning. *Nature*, 518,7540(2015), page 529.

Mye13 Myerson, R. B., *Game theory*. Harvard university press, 2013.

N+50 Nash, J. F. et al., Equilibrium points in n-person games. *Proceedings of the national academy of sciences*, 36,1(1950), pages 48–49.

Nie15 Nielsen, M., *Neural networks and deep learning*. Determination press, 2015. http://neuralnetworksanddeeplearning.com accessed on 30.08.2020.

OB+19 OpenAI, :, Berner, C., Brockman, G., Chan, B., Cheung, V., Dębiak, P., Dennison, C., Farhi, D., Fischer, Q., Hashme, S., Hesse, C., Józefowicz, R., Gray, S., Olsson, C., Pachocki, J., Petrov, M., de Oliveira Pinto, H. P., Raiman, J., Salimans, T., Schlatter, J., Schneider, J., Sidor, S., Sutskever, I., Tang, J., Wolski, F. and Zhang, S., Dota 2 with large scale deep reinforcement learning, arXiv:1912.06680v1 [cs.LG], 2019.

OGSL18 Oh, J., Guo, Y., Singh, S. and Lee, H., Self-imitation learning, arXiv:1806.05635v1 [cs.LG], 2018.

Ope18 OpenAI, Openai five, https://blog.openai.com/openai-five/, 2018. Accessed on 30.10.2020.

OSU+13 Ontanón, S., Synnaeve, G., Uriarte, A., Richoux, F., Churchill, D. and Preuss, M., A survey of real-time strategy game ai research and competition in starcraft. *IEEE Transactions on Computational Intelligence and AI in games*, 5,4(2013), pages 293–311.

PL05 Panait, L. and Luke, S., Cooperative multi-agent learning: The state of the art. *Autonomous Agents and Multi-Agent Systems*, 11,3(2005), pages 387–434.

RHW86 Rumelhart, D. E., Hinton, G. E. and Williams, R. J., Learning representations by back-propagating errors. *nature*, 323,6088(1986), pages 533–536.

SB18 Sutton, R. S. and Barto, A. G., *Reinforcement learning: An introduction*. MIT press, 2018.

SBS+18 Sünderhauf, N., Brock, O., Scheirer, W., Hadsell, R., Fox, D., Leitner, J., Upcroft, B., Abbeel, P., Burgard, W., Milford, M. et al., The limits and potentials of deep learning for robotics. *The International Journal of Robotics Research*, 37,4-5(2018), pages 405–420.

SHM+16 Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M. et al., Mastering the game of go with deep neural networks and tree search. *nature*, 529,7587(2016), pages 484–489.

SHS+18 Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T. et al., A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362,6419(2018), pages 1140–1144.

SLBS10 Singh, S., Lewis, R. L., Barto, A. G. and Sorg, J., Intrinsically motivated reinforcement learning: An evolutionary perspective. *IEEE Transactions on Autonomous Mental Development*, 2,2(2010), pages 70–82.

SML+18 Schulman, J., Moritz, P., Levine, S., Jordan, M. and Abbeel, P., High-dimensional continuous control using generalized advantage estimation, arXiv:1506.02438v6 [cs.LG], 2018.

SSK05 Stone, P., Sutton, R. S. and Kuhlmann, G., Reinforcement learning for robocup soccer keepaway. *Adaptive Behavior*, 13,3(2005), pages 165–188.

Sut88 Sutton, R. S., Learning to predict by the methods of temporal differences. *Machine learning*, 3,1(1988), pages 9–44.

SWD+17 Schulman, J., Wolski, F., Dhariwal, P., Radford, A. and Klimov, O., Proximal policy optimization algorithms, arXiv:1707.06347v2 [cs.LG], 2017.

Tes95 Tesauro, G., Temporal difference learning and td-gammon. *Communications of the ACM*, 38,3(1995), pages 58–68.

Thr00 Thrun, S., Monte carlo pomdps. *Advances in neural information processing systems*, 2000, pages 1064–1070.

Val Valve, Dota ii official game site. URL `https://www.dota2.com/play/`. Accessed on 20.10.2020.

VBC+ Vinyals, O., Babuschkin, I., Chung, J., Mathieu, M., Jaderberg, M., Czarnecki, W., Dudzik, A., Huang, A., Georgiev, P., Powell, R., Ewalds, T., Horgan, D., Kroiss, M., Danihelka, I., Agapiou, J., Oh, J., Dalibard, V., Choi, D., Sifre, L., Sulsky, Y., Vezhnevets, S., Molloy, J., Cai, T., Budden, D., Paine, T., Gulcehre, C., Wang, Z., Pfaff, T., Pohlen, T., Yogatama, D., Cohen, J., McKinney, K., Smith, O., Schaul, T., Lillicrap, T., Apps, C., Kavukcuoglu, K., Hassabis, D. and Silver, D., AlphaStar: Mastering the Real-Time Strategy Game StarCraft II, https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii.

VBC+19 Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., Chung, J., Choi, D. H., Powell, R., Ewalds, T., Georgiev, P. et al., Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575,7782(2019), pages 350–354.

VDDP18 Voulodimos, A., Doulamis, N., Doulamis, A. and Protopapadakis, E., Deep learning for computer vision: A brief review. *Computational intelligence and neuroscience*, 2018.

VEB+17 Vinyals, O., Ewalds, T., Bartunov, S., Georgiev, P., Vezhnevets, A. S., Yeo, M., Makhzani, A., Küttler, H., Agapiou, J., Schrittwieser, J., Quan, J., Gaffney, S., Petersen, S., Simonyan, K., Schaul, T., van Hasselt, H., Silver, D., Lillicrap, T., Calderone, K., Keet, P., Brunasso, A., Lawrence, D., Ekermo, A., Repp, J. and Tsing, R., Starcraft ii: A new challenge for reinforcement learning, arXiv:1708.04782v1 [cs.LG], 2017.

VFJ15 Vinyals, O., Fortunato, M. and Jaitly, N., Pointer networks. In *Advances in Neural Information Processing Systems 28*, Cortes, C., Lawrence, N. D., Lee, D. D., Sugiyama, M. and Garnett, R., editors, Curran Associates, Inc., 2015, pages 2692–2700, URL `http://papers.nips.cc/paper/5866-pointer-networks.pdf`.

VNM47 Von Neumann, J. and Morgenstern, O., *Theory of games and economic behavior, 2nd rev.* Princeton University Press, 1947.

VOS+17 Vezhnevets, A. S., Osindero, S., Schaul, T., Heess, N., Jaderberg, M., Silver, D. and Kavukcuoglu, K., Feudal networks for hierarchical reinforcement learning. *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, 2017, pages 3540–3549.

WD92 Watkins, C. J. and Dayan, P., Q-learning. *Machine learning*, 8,3-4(1992), pages 279–292.

WFPS07 Wierstra, D., Foerster, A., Peters, J. and Schmidhuber, J., Solving deep memory pomdps with recurrent policy gradients. *International Conference on Artificial Neural Networks*. Springer, 2007, pages 697–706.

YHPC18 Young, T., Hazarika, D., Poria, S. and Cambria, E., Recent trends in deep learning based natural language processing. *ieee Computational intelligenCe magazine*, 13,3(2018), pages 55–75.