VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

Review 2

SLOT - B1+TB1

Fall Semester 2020-21

Project Title

Training an AI Bot to beat Retro Games using Reinforcement Learning Algorithms

TAARUSSH WADHWA - 18BIT0312
MADHAV RAJESH - 18BIT0325

Artificial Intelligence(ITE-2010)

Faculty: Prof. Ajit Kumar Santra

1.0 Abstract

Reinforcement learning and games have a long and mutually beneficial common history. Deep reinforcement learning has made great achievements since it was initially proposed. Generally, Deep Reinforcement Learning agents receive high-dimensional inputs at each step and take actions according to certain neural network policies. This learning mechanism updates the policy to maximize the accuracy of the output with an end-to-end method. There have been many advancements in Deep Reinforcement Learning methods, including value-based, policy gradient, and model-based algorithms. Deep Reinforcement Learning also plays an important role in in-game artificial intelligence (AI). With the help of Deep Reinforcement Learning methodology, there have been achievements in various video games, including classical Arcade games, first-person perspective games, and multi-agent real-time strategy games, from 2D to 3D, and from single-agent to multi-agent. A large number of video game AIs utilizing Deep Reinforcement Learning have achieved super-human performance, although there are still some challenges in this domain. The recent advances in deep neural networks have led to effective vision-based reinforcement learning methods that have been employed to obtain human-level controllers in Atari 2600 games from pixel data. Atari 2600 games, however, do not resemble real-world tasks since they involve non-realistic 2D environments and the third-person perspective. Here, we propose a novel test-bed platform for reinforcement learning research from raw visual information which employs the first-person perspective in a semi-realistic 3D world. The software, called ViZDoom, is based on the classical first-person shooter video game, Doom. It allows developing bots that play the game using the screen buffer. ViZDoom is lightweight, fast, and highly customizable via a convenient mechanism of user scenarios. In the experimental part, we test the environment by trying to learn bots for two scenarios: a basic move-and-shoot task and a more complex maze-navigation problem. Using convolutional deep neural networks with Q-learning and experience replay, for both scenarios, we were able to train competent bots, which exhibit human-like behaviors. The results confirm the utility of ViZDoom as an AI research platform and imply that visual reinforcement learning in 3D realistic first-person perspective environments is feasible. Keywords: video games, visual-based reinforcement learning, deep reinforcement learning, first-person perspective games, FPS, visual learning, neural networks
In our project, we try to teach an AI bot to play games using these Deep Reinforcement Learning techniques.

2.0 Objective

1. Build our own AI Game bot using Deep Reinforcement Learning.

3.0 Introduction

Artificial intelligence (AI) in video games is a longstanding research area. It studies how to use AI technologies to achieve human-level performance when playing games. More generally, it studies the complex interactions between agents and game environments. Various games provide interesting and complex problems for agents to solve, making video games perfect environments for AI research. These virtual environments are safe and controllable. In addition, these game environments provide an infinite supply of useful data for machine learning algorithms, and they are much faster than real-time. These characteristics make games the unique and favorite domain for AI research. On the other side, AI has been helping games to become better in the way we play, understand, and design them. Broadly speaking, game AI involves the perception and decision-making in various game environments. Deep reinforcement learning continues to be very successful in mastering human-level control policies in a wide variety of tasks such as object recognition with visual attention, high-dimensional robot control, and solving physics-based control problems. In particular, Deep QNetworks (DQN) is shown to be effective in playing Atari 2600 games and more recently, in defeating world-class Go players(Google's AlphaGo)

3.1 Literature Survey

David et al. [1], (2016) High-dimensional observations and complex real-world dynamics present major challenges in reinforcement learning for both function approximation and exploration. We address both of these challenges with two complementary techniques: First, we develop a gradient-boosting style, non-parametric function approximator for learning on Q-function residuals. And second, we propose an exploration strategy inspired by the principles of state abstraction and information acquisition under uncertainty. We demonstrate the empirical effectiveness of these techniques, first, as a preliminary check, on two standard tasks (Blackjack and n-Chain), and then on two much larger and more realistic tasks with high-dimensional observation spaces. Specifically, we introduce two benchmarks built within the game Minecraft where the observations are pixel arrays of the agent's visual field. A combination of our two algorithmic techniques performs competitively on the standard reinforcement-learning tasks while consistently and substantially outperforming baselines on the two tasks with high-dimensional observation spaces. The new function approximator, exploration strategy, and evaluation benchmarks are each of independent interest in the pursuit of reinforcement-learning methods that scale to real-world domains.

Minoru et al. [2], (1996) This paper presents a method of vision-based reinforcement learning by which a robot learns to shoot a ball into a goal. We discuss several issues in applying the reinforcement learning method to a real robot with vision sensor by which the robot can obtain information about the changes in an environment. First, we construct a state space in terms of size, position, and orientation of a ball and a goal in an image, and an action space is designed in terms of the action commands to be sent to the left and right motors of a mobile robot. This causes a "state-action deviation" problem in constructing the state and action spaces that reflect the outputs from physical sensors and actuators, respectively. To deal with this issue, an action set is constructed in a way that one action consists of a series of the same action primitive which is successively executed until the current state changes. Next, to speed up the learning time, a mechanism of Learning from Easy Missions (or LEM) is implemented. LEM reduces the learning

time from exponential to almost linear order in the size of the state space. The results of computer simulations and real robot experiments are given.

Minoru et al. [3], (1994) Because of the recent success and advancements in deep mind technologies, it is now used to train agents using deep learning for first-person shooter games that are often outperforming human players by means of only screen raw pixels to create their decisions. A visual Doom AI Competition is organized each year on two different tracks: limited death-match on a known map and a full death-match on an unknown map for evaluating AI agents because computer games are the best test-beds for testing and evaluating different AI techniques and approaches. The competition is ranked based on the number of frags each agent achieves. In this paper, training a competitive agent for playing Doom's (FPS Game) basic scenario(s) in a semi-realistic 3D world 'VizDoom' using the combination of convolutional Deep learning and Q-learning by considering only the screen raw pixels in order to exhibit agent's usefulness indoor is proposed. Experimental results show that the trained agent outperforms average human player and inbuilt game agents in basic scenario(s) where only move left, right and shoot actions are allowed.

Nicholas et al. [4], (2004) First-person shooter robot controllers (hots) are generally rule-based expert systems written in C/C++. As such, many of the rules are parameterized with values, which are set by the software designer and finalized at compile time. The effectiveness of parameter values is dependent on the knowledge the programmer has about the game. Furthermore, parameters are non-linearly dependent on each other. This paper presents an efficient method for using a genetic algorithm to evolve sets of parameters for bots which lead to their playing as well as hots whose parameters have been tuned by a human with expert knowledge about the game's strategy. This indicates genetic algorithms as being a potentially useful method for tuning hots.

Giuseppe et al. [5], (2011) Neuroevolution, the artificial evolution of neural networks, has shown great promise on continuous reinforcement learning tasks that require memory. However, it is not yet directly applicable to realistic embedded agents using high-dimensional (e.g. raw video images) inputs, requiring very large networks. In this paper, neuroevolution is combined with an unsupervised sensory pre-processor or compressor that is trained on images generated from the environment by the population of evolving recurrent neural network controllers. The compressor not only reduces the input cardinality of the controllers, but also biases the search toward novel controllers by rewarding those controllers that discover images that it reconstructs poorly. The method is successfully demonstrated on a vision-based version of the well-known mountain car benchmark, where controllers receive only single high-dimensional visual images of the environment, from a third-person perspective, instead of the standard two-dimensional state vector which includes information about velocity.

Mark and Richard [6], (2005) One trend in first-person shooter computer games is to increase programmatic access. This allows artificial intelligence researchers to embed their cognitive models into stable artificial characters, whose competitiveness and realism can be evaluated with respect to human players. However, plugging cognitive models in is non-trivial, since games are currently not designed with AI researchers in mind. In this paper we introduce an intermediate architecture that fits between these game and the AI, and assess its feasibility by implementing it within the game Unreal Tournament 2004. Making such an architecture publicly available may potentially lead to improved quality of game AI.

Abdennour [7], (2008) The aim of developing an agent, that is able to adapt its actions in response to their effectiveness within the game, provides the basis for the research presented in this paper. It investigates how adaptation can be applied through the use of a hybrid of AI technologies. The system developed uses the predefined behaviours of a finite-state machine and fuzzy logic system combined with the learning capabilities of a neural computing. The system adapts specific behaviours that are central to the performance of the bot (a computer-controlled player that simulates a human opponent) in the game, with the paper's main focus being on that of the weapon selection behaviour; selecting the best weapon for the current situation. As a development platform, the project makes use of the Quake 3 Arena engine, modifying the original bot AI to integrate the adaptive technologies.

Esparcia-Alcazar et al. [8], (2010) In this paper we employ a steady state genetic algorithm to evolve different types of behaviour for bots in the Unreal Tournament 2004 computer game. For this purpose we define three fitness functions which are based on the number of enemies killed, the lifespan of the bot and a combination of both. Long run experiments were carried out, in which the evolved bots' behaviours outperform those of standard bots supplied by the game, particularly in those cases where the fitness involves a measure of the bot's lifespan. Also, there is an increase in the number of items collected, and the behaviours tend to become less aggressive, tending instead towards a more optimised combat style. Further "short run" experiments were carried out with a further type of fitness function defined, based on the number of items picked. In these cases the bots evolve performances towards the goal they have been aimed, with no other behaviours arising, except in the case of the multiple objective one. We conclude that in order to evolve interesting behaviours more complex fitness functions are needed, and not necessarily ones that directly include the goal we are aiming for.

Chris et al. [9], (2000) Reinforcement learning systems improve behaviour based on scalar rewards from a critic. In this work vision based behaviours, servoing and wandering, are learned through a Q-learning method which handles continuous states and actions. There is no requirement for camera calibration, an actuator model, or a knowledgeable teacher. Learning through observing the actions of other behaviours improves learning speed. Experiments were performed on a mobile robot using a real-time vision system.

Borja et al. [10] (2018) To solve complex real-world problems with reinforcement learning, we cannot rely on manually specified reward functions. Instead, we need humans to communicate an objective to the agent directly. In this work, we combine two approaches to this problem: learning from expert demonstrations and learning from trajectory preferences. We use both to train a deep neural network to model the reward function and use its predicted reward to train an DQN-based deep reinforcement learning agent on 9 Atari games. Our approach beats the imitation learning baseline in 7 games and achieves strictly superhuman performance on 2 games. Additionally, we investigate the fit of the reward model, present some reward hacking problems, and study the effects of noise in the human labels.

Steven Kapturowski et al. [11] (2018) Building on the recent successes of distributed training of RL agents, in this paper we investigate the training of RNN-based RL agents from distributed prioritized experience replay. We study the effects of parameter lag resulting in representational drift and recurrent state staleness and empirically derive an improved training strategy. Using a single network architecture and fixed set of hyper-parameters, the resulting agent, Recurrent

Replay Distributed DQN, quadruples the previous state of the art on Atari-57, and matches the state of the art on DMLab-30. It is the first agent to exceed human-level performance in 52 of the 57 Atari games.

F G Glavin et al. [12], (2015) In current state-of-the-art commercial first person shooter games, computer controlled bots, also known as nonplayer characters, can often be easily distinguishable from those controlled by humans. Tell-tale signs such as failed navigation, “sixth sense” knowledge of human players' whereabouts and deterministic, scripted behaviors are some of the causes of this. We propose, however, that one of the biggest indicators of nonhumanlike behavior in these games can be found in the weapon shooting capability of the bot. Consistently perfect accuracy and “locking on” to opponents in their visual field from any distance are indicative capabilities of bots that are not found in human players. Traditionally, the bot is handicapped in some way with either a timed reaction delay or a random perturbation to its aim, which doesn't adapt or improve its technique over time. We hypothesize that enabling the bot to learn the skill of shooting through trial and error, in the same way a human player learns, will lead to greater variation in game-play and produce less predictable nonplayer characters. This paper describes a reinforcement learning shooting mechanism for adapting shooting over time based on a dynamic reward signal from the amount of damage caused to opponents.

Xavier et al. [13], (2011) While logistic sigmoid neurons are more biologically plausible than hyperbolic tangent neurons, the latter work better for training multi-layer neural networks. This paper shows that rectifying neurons are an even better model of biological neurons and yield equal or better performance than hyperbolic tangent networks in spite of the hard non-linearity and non-differentiability at zero, creating sparse representations with true zeros which seem remarkably suitable for naturally sparse data. Even though they can take advantage of semi-supervised setups with extra-unlabeled data, deep rectifier networks can reach their best performance without requiring any unsupervised pre-training on purely supervised tasks with large labeled datasets. Hence, these results can be seen as a new milestone in the attempts at understanding the difficulty in training deep but purely supervised neural networks, and closing the performance gap between neural networks learnt with and without unsupervised pre-training.

S Hladky and V Bulitko [14], (2008) A well-known Artificial Intelligence (AI) problem in video games is designing AI-controlled humanoid characters. It is desirable for these characters to appear both skillful and believably human-like. Many games address the former objective by providing their agents with unfair advantages. Although challenging, these agents are frustrating to humans who perceive the AI to be cheating. In this paper we evaluate hidden semi-Markov models and particle filters as a means for predicting opponent positions. Our results show that these models can perform with similar or better accuracy than the average human expert in the game Counter-Strike: Source. Furthermore, the mistakes these models make are more human-like than perfect predictions.

Igor V. Karpov [15], (2012) Imitation is a powerful and pervasive primitive underlying examples of intelligent behavior in nature. Can we use it as a tool to help build artificial agents that behave like humans do? This question is studied in the context of the BotPrize competition, a Turing-like test where computer game bots compete by attempting to fool human judges into thinking they are just another human player. One problem faced by such bots is that of human-like navigation within the virtual world. This chapter describes the Human Trace Controller, a component of the UT^2 bot

which took second place in the BotPrize 2010 competition. The controller uses a database of recorded human games in order to quickly retrieve and play back relevant segments of human navigation behavior. Empirical evidence suggests that the method of direct imitation allows the bot to effectively solve several navigation problems while moving in a human-like fashion.

Jan et al. [16], (2014) Dealing with high-dimensional input spaces, like visual in- put, is a challenging task for reinforcement learning (RL). Neuroevolution (NE), used for continuous RL problems, has to either reduce the problem dimensionality by (1) compress- ing the representation of the neural network controllers or (2) employing a pre-processor (compressor) that transforms the high-dimensional raw inputs into low-dimensional fea- tures. In this paper, we are able to evolve extremely small recurrent neural network (RNN) controllers for a task that pre- viously required networks with over a million weights. The high-dimensional visual input, which the controller would normally receive, is first transformed into a compact fea- ture vector through a deep, max-pooling convolutional neu- ral network (MPCNN). Both the MPCNN preprocessor and the RNN controller are evolved successfully to control a car in the TORCS racing simulator using only visual input. This is the first use of deep learning in the context evolutionary RL.

Alex et al. [17], (2012) We trained a large, deep convolutional neural network to classify the 1.2 million high-resolution images in the ImageNet LSVRC-2010 contest into the 1000 dif- ferent classes. On the test data, we achieved top-1 and top-5 error rates of 37.5% and 17.0% which is considerably better than the previous state-of-the-art. The neural network, which has 60 million parameters and 650,000 neurons, consists of five convolutional layers, some of which are followed by max-pooling layers, and three fully-connected layers with a final 1000-way softmax. To make train- ing faster, we used non-saturating neurons and a very efficient GPU implemen- tation of the convolution operation. To reduce overfitting in the fully-connected layers we employed a recently-developed regularization method called “dropout” that proved to be very effective. We also entered a variant of this model in the ILSVRC-2012 competition and achieved a winning top-5 test error rate of 15.3%, compared to 26.2% achieved by the second-best entry.

Sascha Lange and Martin Riedmiller [18], (2010) This paper discusses the effectiveness of deep auto-encoder neural networks in visual reinforcement learning (RL) tasks. We propose a framework for combining the training of deep auto-encoders (for learning compact feature spaces) with recently-proposed batch-mode RL algorithms (for learning policies). An emphasis is put on the data-efficiency of this combination and on studying the properties of the feature spaces automatically constructed by the deep auto-encoders. These feature spaces are empirically shown to adequately resemble existing similarities and spatial relations between observations and allow to learn useful policies. We propose several methods for improving the topology of the feature spaces making use of task-dependent information. Finally, we present first results on successfully learning good control policies directly on synthesized and real images.

M McPartland and M Gallagher [20], (2011) Deep neural network acoustic models pro- duce substantial gains in large vocabu- lary continuous speech recognition systems. Emerging work with rectified linear (ReL) hidden units demonstrates additional gains in final system performance relative to more commonly used sigmoidal nonlinearities. In this work, we explore the use of deep rectifier networks as acoustic models for the 300 hour Switchboard conversational speech recogni- tion task. Using simple training procedures without pretraining, networks with rectifier nonlinearities produce 2% absolute reduc- tions in word error rates over their sigmoidal

counterparts. We analyze hidden layer repre- sentations to quantify differences in how ReL units encode inputs as compared to sigmoidal units. Finally, we evaluate a variant of the ReL unit with a gradient more amenable to optimization in an attempt to further im- prove deep rectifier networks.

Volodymyr Mnih [21], (2015) Reinforcement learning (RL) is a popular machine learning technique that has many successes in learning how to play classic style games. Applying RL to first person shooter (FPS) games is an interesting area of research as it has the potential to create diverse behaviors without the need to implicitly code them. This paper investigates the tabular Sarsa (λ) RL algorithm applied to a purpose built FPS game. The first part of the research investigates using RL to learn bot controllers for the tasks of navigation, item collection, and combat individually. Results showed that the RL algorithm was able to learn a satisfactory strategy for navigation control, but not to the quality of the industry standard pathfinding algorithm. The combat controller performed well against a rule-based bot, indicating promising preliminary results for using RL in FPS games. The second part of the research used pretrained RL controllers and then combined them by a number of different methods to create a more generalized bot artificial intelligence (AI). The experimental results indicated that RL can be used in a generalized way to control a combination of tasks in FPS bots such as navigation, item collection, and combat.

Megan et al. [22], (2007) The theory of reinforcement learning provides a normative account, deeply rooted in psychological and neuroscientific perspectives on animal behaviour, of how agents may optimize their control of an environment. To use reinforcement learning successfully in situations approaching real-world complexity, however, agents are confronted with a difficult task: they must derive efficient representations of the environment from high-dimensional sensory inputs, and use these to generalize past experience to new situations. Remarkably, humans and other animals seem to solve this problem through a harmonious combination of reinforcement learning and hierarchical sensory processing systems, the former evidenced by a wealth of neural data revealing notable parallels between the phasic signals emitted by dopaminergic neurons and temporal difference reinforcement learning algorithms. While reinforcement learning agents have achieved some successes in a variety of domains, their applicability has previously been limited to domains in which useful features can be handcrafted, or to domains with fully observed, low-dimensional state spaces. Here we use recent advances in training deep neural networks to develop a novel artificial agent, termed a deep Q-network, that can learn successful policies directly from high-dimensional sensory inputs using end-to-end reinforcement learning. We tested this agent on the challenging domain of classic Atari 2600 games. We demonstrate that the deep Q-network agent, receiving only the pixels and the game score as inputs, was able to surpass the performance of all previous algorithms and achieve a level comparable to that of a professional human games tester across a set of 49 games, using the same algorithm, network architecture and hyperparameters. This work bridges the divide between high-dimensional sensory inputs and actions, resulting in the first artificial agent that is capable of learning to excel at a diverse array of challenging tasks.

Tony C Smith and Jonathan Miles [23], (2014) In this paper we present RETALIATE, an online reinforcement learning algorithm for developing winning policies in team first- person shooter games. RETALIATE has three crucial characteristics: (1) individual BOT behavior is fixed although not known in advance, therefore individual BOTS work as “plug- ins”, (2) RETALIATE models the problem of learning team tactics through a simple state formulation, (3) discount rates commonly used in Q-learning are not used. As a result of these characteristics, the application of the Q-learning algorithm results in the rapid exploration towards a winning policy against an opponent

team. In our empirical evaluation we demonstrate that RETALIATE adapts well when the environment changes.

Tony C Smith and Jonathan Miles [24], (2014) Machine learning is now widely studied as the basis for artificial intelligence systems within computer games. Most existing work focuses on methods for learning static expert systems, typically emphasizing candidate selection. This paper extends this work by exploring the use of continuous and reinforcement learning techniques to develop fully-adaptive
game AI for first-person shooter bots. We begin by outlining a framework for learning static control models for tanks within the game BZFlag, then extend that framework using continuous learning techniques that allow computer controlled tanks to adapt to the game style of other players, extending overall playability by thwarting attempts to infer the underlying AI. We further show how reinforcement learning can be used to create bots that learn how to play based solely through trial and error, providing game engineers with a practical means to produce large numbers of bots, each with individual intelligences and unique behaviours; all from a single initial AI model.

Chang et al. [25], (2011) The implementation of Artificial Intelligence (AI)in 3-Dimensional (3D) First Person Shooter (FPS) game is quite general nowadays. Most of the conventional AI bots created are mostly from hard coded AI bots. Hence, it has limited the dynamicity of the AI bots and therefore it brings to a fixed strategy for gaming. The main focus of this paper is to discuss the methodologies used in generating the AI bots that is competitive in the FPS gaming. In this paper, a decision making structure is proposed. It has been combined with the Evolutionary Programming in generating the required AI controllers. Hence, there are two methodology discussions involved: (1) the proposed decision making structure and (2)the Evolutionary Programming used. The experiments show highly promising testing results after the generated AI bots have been tested and compared with the conventional ruled based AI bots. It proves that the generated AI bots using the combination of Evolutionary Programming and decision making structure performed better than those AI bots generated using conventional ruled based strategy which is hard coded and time consuming to develop.

David Trenholme and Shamus P Smith [26], (2008) Building realistic virtual environments is a complex, expensive and time consuming process. Although virtual environment development toolkits are available, many only provide a subset of the tools needed to build complete virtual worlds. One alternative is the reuse of computer game technology. The current generation of computer games present realistic virtual worlds featuring user friendly interaction and the simulation of real world phenomena. Using computer games as the basis for virtual environment development has a number of advantages. Computer games are robust and extensively tested, both for usability and performance, work on off-the-shelf systems and can be easily disseminated, for example via online communities. Additionally, a number of computer game developers provide tools, documentation and source code, either with the game itself or separately available, so that end-users can create new content. This short report overviews several currently available game engines that are suitable for prototyping virtual environments.

Mousavi et al [27] (2017) In order to accelerate the learning process in high dimensional reinforcement learning problems, TD methods such as Q-learning and Sarsa are usually combined with eligibility traces. The recently introduced DQN (Deep Q-Network) algorithm, which is a

combination of Q-learning with a deep neural network, has achieved good performance on several games in the Atari 2600 domain. However, the DQN training is very slow and requires too many time steps to converge. In this paper, we use the eligibility traces mechanism and propose the deep Q(λ) network algorithm. The proposed method provides faster learning in comparison with the DQN method. Empirical results on a range of games show that the deep Q(λ) network significantly reduces learning time.

C. J. C. H. Watkins and P. Dayan [29], (1992) Q-learning is a simple way for agents to learn how to act optimally in controlled Markovian domains. It amounts to an incremental method for dynamic programming which imposes limited computational demands. It works by successively improving its evaluations of the quality of particular actions at particular states.
This paper presents and proves in detail a convergence theorem forQ-learning based on that outlined in Watkins (1989). We show thatQ-learning converges to the optimum action-values with probability 1 so long as all actions are repeatedly sampled in all states and the action-values are represented discretely. We also sketch extensions to the cases of non-discounted, but absorbing, Markov environments, and where manyQ values can be changed each iteration, rather than just one.

Mnih et al. [29], (2016) We present the first deep learning model to successfully learn control policies directly from high-dimensional sensory input using reinforcement learning. The model is a convolutional neural network, trained with a variant of Q-learning, whose input is raw pixels and whose output is a value function estimating future rewards. We apply our method to seven Atari 2600 games from the Arcade Learning Environment, with no adjustment of the architecture or learning algorithm. We find that it outperforms all previous approaches on six of the games and surpasses a human expert on three of them.

Guo et al. [30], (2016) The combination of modern Reinforcement Learning and Deep Learning approaches holds the promise of making significant progress on challenging applications requiring both rich perception and policy-selection. The Arcade Learning Environment (ALE) provides a set of Atari games that represent a useful benchmark set of such applications. A recent breakthrough in combining model-free reinforcement learning with deep learning, called DQN, achieves the best real-time agents thus far. Planning-based approaches achieve far higher scores than the best model-free approaches, but they exploit information that is not available to human players, and they are orders of magnitude slower than needed for real-time play. Our main goal in this work is to build a better real-time Atari game playing agent than DQN. The central idea is to use the slow planning-based agents to provide training data for a deep-learning architecture capable of real-time play. We proposed new agents based on this idea and show that they outperform DQN.

Matiisen, B. T. [31], (2016) Today, exactly two years ago, a small company in London called DeepMind uploaded their pioneering paper “Playing Atari with Deep Reinforcement Learning” to Arxiv. In this paper they demonstrated how a computer learned to play Atari 2600 video games by observing just the screen pixels and receiving a reward when the game score increased. The result was remarkable, because the games and the goals in every game were very different and designed to be challenging for humans. The same model architecture, without any change, was used to learn seven different games, and in three of them the algorithm performed even better than a human!

Nachum. O et al. [32] (2018) Hierarchical reinforcement learning (HRL) is a promising approach to extend traditional reinforcement learning (RL) methods to solve more complex tasks. Yet, the majority of current HRL methods require careful task-specific design and on-policy training, making them difficult to apply in real-world scenarios. In this paper, we study how we can develop HRL algorithms that are general, in that they do not make onerous additional assumptions beyond standard RL algorithms, and efficient, in the sense that they can be used with modest numbers of interaction samples, making them suitable for real-world problems such as robotic control. For generality, we develop a scheme where lower-level controllers are supervised with goals that are learned and proposed automatically by the higher-level controllers. To address efficiency, we propose to use off-policy experience for both higher- and lower-level training. This poses a considerable challenge, since changes to the lower-level behaviors change the action space for the higher-level policy, and we introduce an off-policy correction to remedy this challenge. This allows us to take advantage of recent advances in off-policy model-free RL to learn both higher and lower-level policies using substantially fewer environment interactions than on-policy algorithms. We find that our resulting HRL agent is generally applicable and highly sample-efficient. Our experiments show that our method can be used to learn highly complex behaviors for simulated robots, such as pushing objects and utilizing them to reach target locations, learning from only a few million samples, equivalent to a few days of real-time interaction. In comparisons with a number of prior HRL methods, we find that our approach substantially outperforms previous state-of-the-art techniques.

François-Lavet et al [33] (2018) Deep reinforcement learning is the combination of reinforcement learning (RL) and deep learning. This field of research has been able to solve a wide range of complex decision-making tasks that were previously out of reach for a machine. Thus, deep RL opens up many new applications in domains such as healthcare, robotics, smart grids, finance, and many more. This manuscript provides an introduction to deep reinforcement learning models, algorithms and techniques. Particular focus is on the aspects related to generalization and how deep RL can be used for practical applications. We assume the reader is familiar with basic machine learning concepts.

Cobbe K. et al [34] (2019) In this paper, we investigate the problem of overfitting in deep reinforcement learning. Among the most common benchmarks in RL, it is customary to use the same environments for both training and testing. This practice offers relatively little insight into an agent's ability to generalize. We address this issue by using procedurally generated environments to construct distinct training and test sets. Most notably, we introduce a new environment called CoinRun, designed as a benchmark for generalization in RL. Using CoinRun, we find that agents overfit to surprisingly large training sets. We then show that deeper convolutional architectures improve generalization, as do methods traditionally found in supervised learning, including L2 regularization, dropout, data augmentation and batch normalization.

Lample G. et al [35] (2016) Advances in deep reinforcement learning have allowed autonomous agents to perform well on Atari games, often outperforming humans, using only raw pixels to make their decisions. However, most of these games take place in 2D environments that are fully observable to the agent. In this paper, we present the first architecture to tackle 3D environments in first-person shooter games, that involve partially observable states. Typically, deep reinforcement learning methods only utilize visual input for training. We present a method to augment these models to exploit game feature information such as the presence of enemies or

items, during the training phase. Our model is trained to simultaneously learn these features along with minimizing a Q-learning objective, which is shown to dramatically improve the training speed and performance of our agent. Our architecture is also modularized to allow different models to be independently trained for different phases of the game. We show that the proposed architecture substantially outperforms built-in AI agents of the game as well as humans in deathmatch scenarios.

Nair, A.V. et al [36] (2018) For an autonomous agent to fulfill a wide range of user-specified goals at test time, it must be able to learn broadly applicable and general-purpose skill repertoires. Furthermore, to provide the requisite level of generality, these skills must handle raw sensory input such as images. In this paper, we propose an algorithm that acquires such general-purpose skills by combining unsupervised representation learning and reinforcement learning of goal-conditioned policies. Since the particular goals that might be required at test-time are not known in advance, the agent performs a self-supervised "practice" phase where it imagines goals and attempts to achieve them. We learn a visual representation with three distinct purposes: sampling goals for self-supervised practice, providing a structured transformation of raw sensory inputs, and computing a reward signal for goal reaching. We also propose a retroactive goal relabeling scheme to further improve the sample-efficiency of our method. Our off-policy algorithm is efficient enough to learn policies that operate on raw image observations and goals in a real-world physical system, and substantially outperforms prior techniques.

Li, Y. et al [37] (2017) We give an overview of recent exciting achievements of deep reinforcement learning (RL). We discuss six core elements, six important mechanisms, and twelve applications. We start with background of machine learning, deep learning and reinforcement learning. Next we discuss core RL elements, including value function, in particular, Deep Q-Network (DQN), policy, reward, model, planning, and exploration. After that, we discuss important mechanisms for RL, including attention and memory, unsupervised learning, transfer learning, multi-agent RL, hierarchical RL, and learning to learn. Then we discuss various applications of RL, including games, in particular, AlphaGo, robotics, natural language processing, including dialogue systems, machine translation, and text generation, computer vision, neural architecture design, business management, finance, healthcare, Industry 4.0, smart grid, intelligent transportation systems, and computer systems. We mention topics not reviewed yet, and list a collection of RL resources. After presenting a brief summary, we close with discussions.

Racanière, S. et al [38] (2017) We introduce Imagination-Augmented Agents (I2As), a novel architecture for deep reinforcement learning combining model-free and model-based aspects. In contrast to most existing model-based reinforcement learning and planning methods, which prescribe how a model should be used to arrive at a policy, I2As learn to interpret predictions from a trained environment model to construct implicit plans in arbitrary ways, by using the predictions as additional context in deep policy networks. I2As show improved data efficiency, performance, and robustness to model misspecification compared to several strong baselines.

Kaiser, L. et al [39] (2019) Model-free reinforcement learning (RL) can be used to learn effective policies for complex tasks, such as Atari games, even from image observations. However, this typically requires very large amounts of interaction -- substantially more, in fact, than a human would need to learn the same games. How can people learn so quickly? Part of the answer may be that people can learn how the game works and predict which actions will lead to desirable

outcomes. In this paper, we explore how video prediction models can similarly enable agents to solve Atari games with fewer interactions than model-free methods. We describe Simulated Policy Learning (SimPLe), a complete model-based deep RL algorithm based on video prediction models and present a comparison of several model architectures, including a novel architecture that yields the best results in our setting. Our experiments evaluate SimPLe on a range of Atari games in low data regime of 100k interactions between the agent and the environment, which corresponds to two hours of real-time play. In most games SimPLe outperforms state-of-the-art model-free algorithms, in some games by over an order of magnitude.

Lanctot, M. et al [40] (2017) There has been a resurgence of interest in multiagent reinforcement learning (MARL), due partly to the recent success of deep neural networks. The simplest form of MARL is independent reinforcement learning (InRL), where each agent treats all of its experience as part of its (non stationary) environment. In this paper, we first observe that policies learned using InRL can overfit to the other agents' policies during training, failing to sufficiently generalize during execution. We introduce a new metric, joint-policy correlation, to quantify this effect. We describe a meta-algorithm for general MARL, based on approximate best responses to mixtures of policies generated using deep reinforcement learning, and empirical game theoretic analysis to compute meta-strategies for policy selection. The meta-algorithm generalizes previous algorithms such as InRL, iterated best response, double oracle, and fictitious play. Then, we propose a scalable implementation which reduces the memory requirement using decoupled meta-solvers. Finally, we demonstrate the generality of the resulting policies in three partially observable settings: gridworld coordination problems, emergent language games, and poker.

Jaderberg, M. et al [41] (2019) Reinforcement learning (RL) has shown great success in increasingly complex single-agent environments and two-player turn-based games. However, the real world contains multiple agents, each learning and acting independently to cooperate and compete with other agents. We used a tournament-style evaluation to demonstrate that an agent can achieve human-level performance in a three-dimensional multiplayer first-person video game, Quake III Arena in Capture the Flag mode, using only pixels and game points scored as input. We used a two-tier optimization process in which a population of independent RL agents are trained concurrently from thousands of parallel matches on randomly generated environments. Each agent learns its own internal reward signal and rich representation of the world. These results indicate the great potential of multiagent reinforcement learning for artificial intelligence research.

Gupta, J.K. et al [42] (2019) This work considers the problem of learning cooperative policies in complex, partially observable domains without explicit communication. We extend three classes of single-agent deep reinforcement learning algorithms based on policy gradient, temporal-difference error, and actor-critic methods to cooperative multi-agent systems. To effectively scale these algorithms beyond a trivial number of agents, we combine them with a multi-agent variant of curriculum learning. The algorithms are benchmarked on a suite of cooperative control tasks, including tasks with discrete and continuous actions, as well as tasks with dozens of cooperating agents. We report the performance of the algorithms using different neural architectures, training procedures, and reward structures. We show that policy gradient methods tend to outperform both temporal-difference and actor-critic methods and that curriculum learning is vital to scaling reinforcement learning algorithms in complex multi-agent domains.

3.2 Background

Doom is an FPS (First-person shooter) game developed by Id-software. Its first installment was released on December 10, 1993, for the platform of DOS, and its second installment Doom II: Hell on Earth was released in the following year (1994) for Microsoft Windows, play-station, and Xbox-360. Its third installment Doom-3 was released for Microsoft Windows on August 3, 2004, which was later adapted for Linux and MacOSX. Also, later on, „Vicarious Visions" ported the game to the Xbox and released it on April 3, 2005. Now the very recent and latest installment is DOOM developed by id-software and published by Bethesda Softworks. It was released worldwide on Microsoft Windows, play-station 4, and X box-one as well on May 13, 2016. A common screen of the Doom game is shown below.

These days the research community is very active in research on Doom for being a hot area of research using techniques like deep reinforcement learning or visual reinforcement learning. Besides, different Doom-based research platforms like VizDoom, CocoDoom, and ResearchDoom are developed for implementing deep learning techniques or methods. In the same way, every year different visual Doom AI competitions are organized where the agents (bots) are confirmed to exhibit human-like actions and to show that visual reinforcement learning in 3D FPS game environments is feasible. The first paper Playing Atari with Deep Reinforcement Learning addresses how convolutional

neural networks and deep reinforcement learning combine together to accomplish a high-performance AI agent that plays Atari games. The paper analyzes how Reinforcement learning (RL) provides a good solution to game playing problems and also the challenges in Deep Learning brought about by RL from the data representation perspective. The paper proposes deep reinforcement learning on game-playing agents, which is similar to our goal. However, there are differences between our approaches and theirs. One of the differences is that the approach in this paper relies on heavy downsampling images before feeding them into a neural network. Our approach tries to avoid this downsampling procedure in an attempt to produce better data layers for deep Q-learning. The second related paper Deep Learning for Real-Time Atari Game Play Using Offline Monte-Carlo Tree Search Planning proposes another solution to game playing. Compared to the first paper, which uses a model-free Q-learning strategy, this paper tackles the problem using a combination of Monte-Carlo techniques and deep Q-learning, ending up with a much more sophisticated algorithm that adds extra assumptions and considerable complexity. Though we choose the model-free learning approach, this paper still provides us with insights

about deep learning applied to games, such as the preprocessing of raw data and the architecture of convolutional neural networks.

3.3 Motivation

Reinforcement learning algorithms allow (software) agents to learn from experience how to behave optimally in dynamic environments. A real-world example of this setup is when infants learn to walk. Initially, they arbitrarily move their legs and feet and thus may fall or stay upright. The former is undesirable, the latter is desirable. Over many, many trials, this feedback accumulates and the infants learn which states (postures) require which actions (muscle movements) to accumulate high rewards (staying upright). An early breakthrough in human-level AI was a system that used reinforcement learning to learn to play 'Backgammon' and could compete with top players in the early 1990s. More recent breakthroughs in-game AI came in the wake of the deep learning revolution but these, too, crucially rely on reinforcement learning. The recent advances in deep neural networks have led to effective vision-based reinforcement learning methods that have been employed to obtain human-level controllers in Atari 2600 games from pixel data. Atari 2600 games, however, do not resemble real-world tasks since they involve non-realistic 2D environments and the third-person perspective. Here, we propose a novel test-bed platform for reinforcement learning research from raw visual information which employs the first-person perspective in a semi-realistic 3D world. The software, called ViZDoom, is based on the classical first-person shooter video game, Doom. It allows developing bots that play the game using the screen buffer. ViZDoom is lightweight, fast, and highly customizable via a convenient mechanism of user scenarios. In the experimental part, we test the environment by trying to learn bots for two scenarios: a basic move-and-shoot task and a more complex maze-navigation problem. Using convolutional deep neural networks with Q-learning and experience replay, for both scenarios, we were able to train competent bots, which exhibit human-like behaviors. The results confirm the utility of ViZDoom as an AI research platform and imply that visual reinforcement learning in 3D realistic first-person perspective environments is feasible. Keywords: video games, visual-based reinforcement learning, deep reinforcement learning, first-person perspective games, FPS, visual learning, neural networks

4.0 Methodology

Q-Learning for Game Playing

Reinforcement Learning briefly is a paradigm of Learning Process in which a learning agent learns, overtime, to behave optimally in a certain environment by interacting continuously in the environment. The agent during its course of learning experience various different situations in the environment it is in. These are called *states*. The agent while being in that state may choose from a set of allowable actions which may fetch different *rewards*(or penalties). The learning agent overtime learns to maximize these rewards so as to behave optimally at any given state it is in. Q-Learning is a basic form of Reinforcement Learning which uses Q-values (also called action values) to iteratively improve the behavior of the learning agent.

a) Q-Values or Action-Values: Q-values are defined for states and actions. Q(S, A) is an estimation of how good is it to take the action A at the state S. This estimation of Q(S, A) will be iteratively computed using the TD- Update rule which we will see in the upcoming sections.

b) Rewards and Episodes: An agent over the course of its lifetime starts from a start state, makes a number of transitions from its current state to a next state based on its choice of action and also the environment the agent is interacting in. At every step of transition, the agent from a state takes an action, observes a reward from the environment, and then transits to another state. If at any point of time the agent ends up in one of the terminating states that means there are no further transition possible. This is said to be the completion of an episode.

c) Temporal Difference or TD-Update:
The Temporal Difference or TD-Update rule can be represented as follows :

$$Q(S,A) \leftarrow Q(S,A) + \alpha\left(R + \gamma Q(S',A') - Q(S,A)\right)$$

This update rule to estimate the value of Q is applied at every time step of the agents interaction with the environment. The terms used are explained below. :

- S : Current State of the agent.
- A : Current Action Picked according to some policy.
- S' : Next State where the agent ends up.
- A' : Next best action to be picked using current Q-value estimation, i.e. pick the action with the maximum Q-value in the next state.
- R : Current Reward observed from the environment in Response of current action.
- γ(>0 and <=1) : Discounting Factor for Future Rewards. Future rewars are less valuable than current rewards so they must be discounted. Since Q-value is an estimation of expected rewards from a state, discounting rule applies here as well.
- α : Step length taken to update the estimation of Q(S, A).

d) Choosing the Action to take using ε-greedy policy:
ε-greedy policy of is a very simple policy of choosing actions using the current Q-value estimations. It goes as follows :

- With probability (1-ε) choose the action which has the highest Q-value.
- With probability (ε) choose any action at random.

One common way to deal with the game playing problem is to assume a Markov Decision Process (MDP). This is appropriate for Assault because the enemy agents move randomly. An MDP is a model defined by a set of States, Actions, Transitions, and Rewards. In order to train an AI to tackle game playing tasks, reinforcement learning based on Q-learning is a popular choice. In Q-learning, the MDP recurrence is defined as follows:

Q(s ∗ , a) = Es 0 [r + γmaxa 0Q ∗ (s0, a0)|s, a] (1)

where a is the action it takes, s is the current state, r is the reward, and γ is the discount factor. Furthermore, we can use function approximation by parameterizing Q-value. In this way, we can easily adapt linear regression and gradient descent techniques from machine learning. With function approximation, we can calculate the best weights by adapting the update rule:

w ← w − η[Qˆ opt(s, a;w) − (r + γVˆ opt(s 0))]Φ(s, a) (2)

where w is a vector containing weights of each feature and is initialized randomly to avoid getting into the same local optimum in every trial.

Convolutional Neural Networks:
In deep learning, a convolutional neural network (CNN, or ConvNet) is a class of deep neural networks, most commonly applied to analyzing visual imagery. They are also known as shift invariant or space invariant artificial neural networks (SIANN), based on their shared-weights architecture and translation invariance characteristics They have applications in image and video recognition, recommender systems, image classification, medical image analysis, natural language processing, and financial time series. CNNs are regularized versions of multilayer perceptrons. Multilayer perceptrons usually mean fully connected networks, that is, each neuron in one layer is connected to all neurons in the next layer. The "fully-connectedness" of these networks makes them prone to overfitting data. Typical ways of regularization include adding some form of magnitude measurement of weights to the loss function. CNNs take a different approach towards regularization: they take advantage of the hierarchical pattern in data and assemble more complex patterns using smaller and simpler patterns. Therefore, on the scale of connectedness and complexity, CNNs are on the lower extreme.
A convolutional neural network consists of an input and an output layer, as well as multiple hidden layers. The hidden layers of a CNN typically consist of a series of convolutional layers that convolve with a multiplication or other dot product. The activation function is commonly a RELU layer, and is subsequently followed by additional convolutions such as pooling layers, fully connected layers and normalization layers, referred to as hidden layers because their inputs and outputs are masked by the activation function and final convolution.

$$G(X) = g_N(g_{N-1}(...(g_1(X))))$$

When programming a CNN, the input is a tensor with shape (number of images) x (image height) x (image width) x (image depth). Then after passing through a convolutional layer, the image becomes abstracted to a feature map, with shape (number of images) x (feature map height) x (feature map width) x (feature map channels). A convolutional layer within a neural network should have the following attributes: (1) Convolutional kernels defined by a width and height (hyper-parameters). (2) The number of input channels and output channels (hyper-parameter). (3)

The depth of the Convolution filter (the input channels) must be equal to the number channels (depth) of the input feature map. Convolutional layers convolve the input and pass its result to the next layer. This is similar to the response of a neuron in the visual cortex to a specific stimulus. Each convolutional neuron processes data only for its receptive field. Although fully connected feedforward neural networks can be used to learn features as well as classify data, it is not practical to apply this architecture to images. A very high number of neurons would be necessary, even in a shallow (opposite of deep) architecture, due to the very large input sizes associated with images, where each pixel is a relevant variable. For instance, a fully connected layer for a (small) image of size 100 x 100 has 10,000 weights for each neuron in the second layer. The convolution operation brings a solution to this problem as it reduces the number of free parameters, allowing the network to be deeper with fewer parameters. For instance, regardless of image size, tiling regions of size 5 x 5, each with the same shared weights, requires only 25 learnable parameters. By using regularized weights over fewer parameters, the vanishing gradient and exploding gradient problems seen during backpropagation in traditional neural networks are avoided.

$$h_k(x,y) = \sum_{s=-m}^{m} \sum_{t=-}^{n} \sum_{v=-d}^{w} V_k(s,t,v) X(x-s, y-t, z-v)$$

Pooling

Convolutional networks may include local or global pooling layers to streamline the underlying computation. Pooling layers reduce the dimensions of the data by combining the outputs of neuron clusters at one layer into a single neuron in the next layer. Local pooling combines small clusters, typically 2 x 2. Global pooling acts on all the neurons of the convolutional layer. In addition, pooling may compute a max or an average. Max pooling uses the maximum value from each of a cluster of neurons at the prior layer. Average pooling uses the average value from each of a cluster of neurons at the prior layer.

Fully Connected

Fully connected layers connect every neuron in one layer to every neuron in another layer. It is in principle the same as the traditional multi-layer perceptron neural network (MLP). The flattened matrix goes through a fully connected layer to classify the images.
In activation layer, different activation functions that can be used:
(i) Sigmoid activation function is given by the equation:

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

It is nonlinear in nature; its combination will also be nonlinear in nature, which gives us the liberty to stack the layers together. Its range is from −2 to 2 on the x-axis and on y-axis it is fairly steep, which shows the sudden changes in the values of y with respect to small changes in the values of x. One of the advantages of this activation function is its output always remains within the range of (0,1).
ii) Tanh function is defined as follows:

$$f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$$

This is also known as the scaled sigmoid function:

$$\tanh(x) = 2sigmoid(2x) - 1$$

Its range is from −1 to 1. The gradient is stronger for the tanh than the sigmoid function.

(iii) Rectified linear unit (ReLU) is the most commonly used activation function, where g denotes pixelwise function, which is nonlinear in nature. That is, it gives the output x, if x is positive and it is 0 otherwise.

$$g(x) = max(0, x)$$

ReLU is nonlinear in nature and its combination is also nonlinear, meaning different layers can be stacked together. Its range is from 0 to infinity, meaning it can also blow up the activation. For the pooling layer, g reduces the size of the features while acting as a layer-wise down-sampling nonlinear function. A fully connected layer has a 1 × 1 convolutional kernel. Prediction layer has a softmax which predicts the probability belonging of Xj to different possible classes.

Convolutional Neural Networks as Function Approximators

We decided to use deep Q-learning instead of ordinary Q-learning because there are too many possible game states. The size of one frame is 216 by 160 pixels, and for each pixel there are 2563 choices of RGB values. Furthermore, a sliding window of k frames leads to 256216∗160∗3∗k possible states in total – this is too large for ordinary Q-learning because it will result in too many rows in our imaginary Q-table. Therefore, we decide to use a neural network to learn these Q values instead. In effect, this neural network ends up operating as a function approximator. The network architecture is currently described as follows:

- Preprocess frames: convert RGB pixels to grayscale and threshold to black or white
- Input layer: takes in the preprocessed frames. Size: [k, 160, 250, 1]
- Hidden convolutional layer 1: kernel size [8, 8, k, 32], strides [1, 4, 4, 1]
- Max pooling layer 1: kernel size [1, 2, 2, 1], strides [1, 2, 2, 1]
- Hidden convolutional layer 2: kernel size [4, 4, 32, 64], strides [1, 2, 2, 1]
- Max pooling layer 2: kernel size [1, 2, 2, 1], strides [1, 2, 2, 1]
- Hidden convolutional layer 3: kernel size [3, 3, 64, 64], strides [1, 1, 1, 1]
- Max pooling layer 3: kernel size [1, 2, 2, 1], strides [1, 2, 2, 1]
- Resize the max pooling outputs to a vector of size [768] and feed to one fully connected layer
- Feed the outputs to a rectified linear activation function
- Collect the final output as 7 Q-values, each corresponding to an action

Though we have 3 max-pooling layers involved in our network architecture, this is not what we had in mind at the beginning. Unlike CNN architectures typically used in computer vision tasks such as image classification, pooling layers in our architecture may not have been desirable for our purposes because we likely do not want to introduce translation invariance since the position of the game entities are important for estimating Q values. However, max-pooling serves as an adequate way to compress our large state space into a vector of size 768, and is the reason why we have been using them. One suggestion to replace these max-pooling layers is to make the strides larger in each hidden convolutional layer – but given that the current strides already have substantial size, we have been hesitant to increase it any further. However, we are not dismissing it as a bad idea and would like to give it a try if we were given an additional month or two to evaluate.

Experience Replay

In general, deep neural networks are difficult to train. In the presence of multiple local optima, gradient descent may end up at a bad local minimum which will lead to poor performance. Initializing the network weights and biases to random values helps a little but is likely not sufficient. We incorporate a technique called experience replay to encourage the algorithm to find better optima instead of getting stuck at some underperforming local optimum. To be more specific, as we run a game session during training, all experiences < s, a, r, s0 > are stored in replay memory. During training, we take random samples from the replay memory instead of always grabbing the most recent transition. By breaking the similarity of subsequent training examples, this trick is likely to prevent the network from diving into some local minimum and will do so in an efficient manner. By using experience replay, the behavior distribution is averaged over many of its previous states, smoothing out learning and avoiding divergence in the parameters.

In our implementation, we simply perform a uniform sample from the bank of observed states (the replay memory) to construct a minibatch of size 100 with which we train on each iteration. Storing all past experiences is impossible due to the humongous state space, so we simply retain the most recent 20,000 observations in the replay memory and sample from that.

A. Why Doom?

Creating yet another 3D first-person perspective environment from scratch solely for research purposes would be somewhat wasteful. Due to the popularity of the first-person shooter genre, we have decided to use an existing game engine as the base for our environment. We concluded that it has to meet the following requirements:

A. based on popular open-source 3D FPS game (ability to modify the code and the publication freedom),
B. lightweight (portability and the ability to run multiple instances on a single machine),
C. fast (the game engine should not be the learning bottleneck),
D. total control over the game's processing (so that the game can wait for the bot decisions or the agent can learn by observing a human playing),
E. customizable resolution and rendering parameters,
F. multiplayer games capabilities (agent vs. agent and agent vs. human),
G. easy-to-use tools to create custom scenarios,
H. ability to bind different programming languages (preferably written in C++),
I. multi-platform.

In order to make the decision according to the above-listed criteria, we have analyzed seven recognizable FPS games: Quake III Arena, Doom 3, Half-Life 2, Unreal Tournament 2004, Unreal Tournament, and Cube. Their comparison is shown in Table I. Some of the features listed in the table are objective (e.g., 'scripting') and others are subjective ("code complexity"). Brand recognition was estimated as the number (in millions) of Google results (as of 26.04.2016) for phrases "game ", where was 'doom', 'quake', 'half-life', 'unreal tournament' or 'cube'. The game was considered as low-resolution capable if it was possible to set the resolution to values smaller than 640 × 480. Some of the games had to be rejected right away in spite of high general appeal. Unreal Tournament 2004 engine is only accessible by the Software Development Kit and it lacks support for controlling the speed of execution and direct screen buffer access. The game has not been prepared to be heavily modified.

Similar problems are shared by Half-Life 2 despite the fact that the Source engine is widely known for modding capabilities. It also lacks direct multiplayer support. Although the Source engine itself

offers multiplayer support, it involves client-server architecture, which makes synchronization and direct interaction with the engine problematic (network communication).
The client-server architecture was also one of the reasons for the rejection of Quake III: Arena. Quake III also does not offer any scripting capabilities, which are essential to make a research environment versatile. The rejection of Quake was a hard decision as it is a highly regarded and playable game even nowadays but this could not outweigh the lack of scripting support.
The latter problem does not concern Doom 3 but its high disk requirements were considered as a drawback. Doom 3 had to be ignored also because of its complexity, Windows-only tools, and OS-dependent rendering mechanisms. Although its source code has been released, its community is dispersed. As a result, there are several rarely updated versions of its sources.
The community activity is also a problem in the case of Cube as its last update was in August 2005. Nonetheless, the low complexity of its code and the highly intuitive map editor would make it a great choice if the engine was more popular.
Unreal Tournament, however popular, is not as recognizable as Doom or Quake but it has been a primary research platform for FPS games. It also has great capabilities. Despite its active community and the availability of the source code, it was rejected due to its high system requirements.
Doom met most of the requirements and allowed to implement features that would be barely achievable in other games, e.g., off-screen rendering and custom rewards. The game is highly recognizable and runs on the three major operating systems. It was also designed to work in 320 × 240 resolution and despite the fact that modern implementations allow bigger resolutions, it still utilizes low-resolution textures. Moreover, its source code is easy-to-understand.
The unique feature of Doom is its software renderer. Because of that, it could be run without the desktop environment (e.g., remotely in a terminal), and accessing the screen buffer does not require transferring it from the graphics card. Technically, ViZDoom is based on the modernized, opensource version of Doom's original engine — ZDoom, which is still actively supported and developed.

B. Application Programming Interface (API)

ViZDoom API is flexible and easy-to-use. It was designed with reinforcement and apprenticeship learning in mind, and therefore, it provides full control over the underlying Doom process. In particular, it allows retrieving the game's screen buffer and make actions that correspond to keyboard buttons (or their combinations) and mouse actions. Some game state variables such as the player's health or ammunition are available directly. ViZDoom's API was written in C++. The API offers a myriad of configuration options such as control modes and rendering options. In addition to the C++ support, bindings for Python and Java have been provided.

C. Features

ViZDoom provides features that can be exploited in different kinds of AI experiments. The main features include different control modes, custom scenarios, access to the depth buffer, and off-screen rendering eliminating the need of using a graphical interface.

i) Control modes: ViZDoom implements four control modes:

- i) synchronous player,
- ii) synchronous spectator,

iii) asynchronous player, and
iv) asynchronous spectator.

In asynchronous modes, the game runs at constant 35 frames per second and if the agent reacts too slowly, it can miss some frames. Conversely, if it makes a decision too quickly, it is blocked until the next frame arrives from the engine. Thus, for reinforcement learning research, more useful are the synchronous modes, in which the game engine waits for the decision-maker. This way, the learning system can learn at its pace, and it is not limited by any temporal constraints. Importantly, for experimental reproducibility and debugging purposes, the synchronous modes run deterministically. In the player modes, it is the agent who makes actions during the game. In contrast, in the spectator modes, a human player is in control, and the agent only observes the player's actions.

In addition, ViZDoom provides an asynchronous multiplayer mode, which allows games involving up to eight players (human or bots) over a network

ii) Scenarios: One of the most important features of ViZDoom is the ability to run custom scenarios. This includes creating appropriate maps, programming the environment mechanics ("when and how things happen"), defining terminal conditions (e.g., "killing a certain monster", "getting to a certain place", "died"), and rewards (e.g., for "killing a monster", "getting hurt", "picking up an object"). This mechanism opens endless experimentation possibilities. In particular, it allows creating a scenario of a difficulty that is on par with the capabilities of the assessed learning algorithms.

The creation of scenarios is possible thanks to easy-to-use software tools developed by the Doom community. The two recommended free tools include Doom Builder 2 and SLADE 3. Both are visual editors, which allow defining custom maps and coding the game mechanics in Action Code Script. They also enable to conveniently test a scenario without leaving the editor. ViZDoom comes with a few predefined scenarios.

iii) Depth Buffer Access: ViZDoom provides access to the renderer's depth buffer, which may help an agent to understand the received visual information. This feature gives an opportunity to test whether the learning algorithms can autonomously learn the whereabouts of the objects in the environment. The depth information can also be used to simulate the distance sensors common in mobile robots.

iv) Off-Screen Rendering and Frame Skipping: To facilitate computationally heavy machine learning experiments, we equipped ViZDoom with off-screen rendering and frame skipping features. Off-screen rendering lessens the performance burden of actually showing the game on the screen and makes it possible to run the experiments on the servers (no graphical interface needed). Frame skipping, on the other hand, allows omitting rendering selected frames at all. Intuitively, an effective bot does not have to see every single frame.

D. ViZDoom's Performance

The main factors affecting ViZDoom performance are the number of the actors (like items and bots), the rendering resolution, and computing the depth buffer. We can see how the number of frames per second depends on these factors. The tests have been made in the synchronous player mode on Linux running on Intel Core i7-4790k. ViZDoom uses only a single CPU core. The performance test shows that ViZDoom can render nearly 7000 low-resolution frames per second. The rendering resolution proves to be the most important factor influencing the processing

speed. In the case of low resolutions, the time needed to render one frame is negligible compared to the backpropagation time of any reasonably complex neural network.

Functioning:

A. Basic Experiment

The primary purpose of the experiment was to show that reinforcement learning from the visual input is feasible in ViZDoom. Additionally, the experiment investigates how the number of skipped frames influences the learning process.

i) **Scenario**: This simple scenario takes place in a rectangular chamber. An agent is spawned in the center of the room's longer wall. A stationary monster is spawned at a random position along the opposite wall. The agent can strafe left and right, or shoot. A single hit is enough to kill the monster. The episode ends when the monster is eliminated or after 300 frames, whatever comes first. The agent scores 101 points for killing the monster, −5 for a missing shot, and, additionally, −1 for each action. The scores motivate the learning agent to eliminate the monster as quickly as possible, preferably with a single shot.

ii) **Deep Q-Learning**: The learning procedure is similar to the Deep Q-Learning introduced for Atari 2600. The problem is modeled as a Markov Decision Process and Q-learning is used to learn the policy. The action is selected by a greedy policy with linear decay. The Q-function is approximated with a convolutional neural network, which is trained with Stochastic Gradient Descent. We also used experience replay but no target network freezing.

iii) **Experimental Setup**:

a) Neural Network Architecture: The network used in the experiment consists of two convolutional layers with 32 square filters, 7 and 4 pixels wide, respectively. Each convolution layer is followed by a max-pooling layer with max-pooling of size 2 and rectified linear units for activation. Next, there is a fully-connected layer with 800 leaky rectified linear units and an output layer with 8 linear units corresponding to the 8 combinations of the 3 available actions (left, right, and shot).

b) Game Settings: A state was represented by the most recent frame, which was a 60 × 45 3-channel RGB image. The number of skipped frames is controlled by the skipcount parameter. We experimented with skipcounts of 0-7, 10, 15, 20, 25, 30, 35, and 40. It is important to note that the agent repeats the last decision on the skipped frames.

c) Learning Settings: We arbitrarily set the discount factor $\gamma = 0.99$, learning rate $\alpha = 0.01$, replay memory capacity to 10 000 elements and mini-batch size to 40. The initial = 1.0 starts to decay after 100 000 learning steps, finishing the decay at = 0.1 at 200 000 learning steps.

Every agent learned for 600 000 steps, each one consisting of performing an action, observing a transition, and updating the network. To monitor the learning progress, 1000 testing episodes were played after each 5000 learning steps. Final controllers were evaluated on 10 000 episodes. The experiment was performed on Intel Core i7-4790k 4GHz with GeForce GTX 970, which handled the neural network.

iv) **Results**:

It demonstrates that although all the agents improve over time, the skips influence the learning speed, its smoothness, as well as the final performance. When the agent does not skip any frames, the learning is the slowest. Generally, the larger the skipcount, the faster and smoother

the learning is. We have also observed that the agents learning with higher skipcounts were less prone to irrational behaviors like staying idle or going the direction opposite to the monster, which results in lower variance on the plots. On the other hand, too large skipcounts make the agent 'clumsy' due to the lack of fine-grained control, which results in suboptimal final scores.
We have also checked how robust to skipcounts the agents are. For this purpose, we evaluated them using skipcounts different from the ones they had been trained with. Most of the agents performed worse than with their "native" skipcounts. The least robust were the agents trained with skipcounts less than 4. Larger skipcounts resulted in more robust agents. Interestingly, for skipcounts greater than or equal to 30, the agents score better on skipcounts lower than the native ones. Our best agent that was trained with skipcount 4 was also the best when executed with skipcount 0.
It is also worth showing that increasing the skipcount influences the total learning time only slightly. The learning takes longer primarily due to the higher total overhead associated with episode restarts since higher skipcounts result in a greater number of episodes.
To sum up, the skipcounts in the range of 4-10 provide the best balance between the learning speed and the final performance. The results also indicate that it would be profitable to start learning with high skipcounts to exploit the steepest learning curve and gradually decrease it to fine-tune the performance.

B. Medikit Collecting Experiment

The previous experiment was conducted on a simple scenario that was closer to a 2D arcade game rather than a true 3D virtual world. That is why we decided to test if similar deep reinforcement learning methods would work in a more involved scenario requiring substantial spatial reasoning.

i) **Scenario**: In this scenario, the agent is spawned in a random spot of a maze with an acid surface, which slowly, but constantly, takes away the agent's life. To survive, the agent needs to collect medikits and avoid blue vials with poison. Items of both types appear in random places during the episode. The agent is allowed to move (forward/backward), and turn (left/right). It scores 1 point for each tick, and it is punished by −100 points for dying. Thus, it is motivated to survive as long as possible. To facilitate learning, we also introduced shaping rewards of 100 and −100 points for collecting a medikit and a vial, respectively. The shaping rewards do not count to the final score but are used during the agent's training helping it to 'understand' its goal. Each episode ends after 2100 ticks (1 minute in real-time) or when the agent dies so 2100 is the maximum achievable score. Being idle results in scoring 284 points.

ii) **Experimental Setup:** The learning procedure was the same as described in A2 with the difference that for updating the weights RMSProp this time.

a) Neural Network Architecture: The employed network is similar to the one used in the previous experiment. The differences are as follows. It involves three convolutional layers with 32 square filters 7, 5, and 3 pixels wide, respectively. The fully-connected layer uses 1024 leaky rectified linear units and the output layer 16 linear units corresponding to each combination of the 4 available actions.

b) Game Settings: The game's state was represented by a 120 × 45 3-channel RGB image, health points, and the current tick number (within the episode). Additionally, a kind of memory was implemented by making the agent use 4 last states as the neural network's

input. The nonvisual inputs (health, ammo) were fed directly to the first fully-connected layer. A Skipcount of 10 was used.

c) Learning Settings: We set the discount factor γ = 1, learning rate α = 0.00001, replay memory capacity to 10 000 elements and mini-batch size to 64. The initial = 1.0 started to decay after 4 000 learning steps, finishing the decay at = 0.1 at 104 000 episodes. The agent was set to learn for 1000 000 steps. To monitor the learning progress, 200 testing episodes were played after each 5000 learning steps. The whole learning process, including the testing episodes, lasted 29 hours.

iii) **Results**: The learning dynamics are shown in. It can be observed that the agents fairly quickly learn to get the perfect score from time to time. Its average score, however, improves slowly reaching 1300 at the end of the learning. Learning dynamics for health gathering scenario. trend might, however, suggest that some improvement is still possible given more training time. The plots suggest that even at the end of learning, the agent for some initial states fails to live more than a random player.

It must, however, be noted that the scenario is not easy and even from a human player, it requires a lot of focus. It is so because the medikits are not abundant enough to allow the bots to waste much time.

Watching the agent play revealed that it had developed a policy consistent with our expectations. It navigates towards medikits, actively, although not very deftly, avoids the poison vials, and does not push against walls and corners. It also backpedals after reaching a dead-end or a poison vial. However, it very often hesitates about choosing a direction, which results in turning left and right alternately on the spot. This quirky behavior is the most probable, direct cause of not fully satisfactory performance.

Interestingly, the learning dynamics consists of three sudden but ephemeral drops in the average and best score. The reason for such dynamics is unknown and it requires further research.

5.0 Platform

Language: Python

Library:

1. **Numpy:** adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.
2. **PyTorch:** open-source machine learning library based on the Torch library, used for applications such as computer vision and natural language processing

Environment: Open AI Gym is a toolkit for developing and comparing reinforcement learning algorithms. It supports teaching agents everything from walking to playing games like Doom.

6.0 Sample Coding

```
# -*- coding: utf-8 -*-
"""Deep Convolutional Q-Learning

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1WsKC2PgRtjEWGy1u0wGCGCh-ZCFmEISA

# Deep Convolutional Q-Learning

### Installing system dependencies for VizDoom
"""

!sudo apt-get update
!sudo apt-get install build-essential zlib1g-dev libsdl2-dev libjpeg-dev nasm tar libbz2-dev libgtk2.0-dev cmake git libfluidsynth-dev libgme-dev libopenal-dev timidity libwildmidi-dev unzip
!sudo apt-get install libboost-all-dev
!apt-get install liblua5.1-dev
!sudo apt-get install cmake libboost-all-dev libgtk2.0-dev libsdl2-dev python-numpy git
!git clone https://github.com/shakenes/vizdoomgym.git
!python3 -m pip install -e vizdoomgym/
!pip install pillow
!pip install scipy==1.1.0

"""### **IMPORTANT NOTE: After installing all dependencies, restart your runtime**

## ------------------------ IMAGE PREPROCESSING (image_preprocessing.py) ------------------------

### Importing the libraries
"""

import numpy as np
from scipy.misc import imresize
from gym.core import ObservationWrapper
from gym.spaces.box import Box

"""### Preprocessing the images"""

class PreprocessImage(ObservationWrapper):

    def __init__(self, env, height = 64, width = 64, grayscale = True, crop = lambda img: img):
        super(PreprocessImage, self).__init__(env)
        self.img_size = (height, width)
        self.grayscale = grayscale
        self.crop = crop
        n_colors = 1 if self.grayscale else 3
        self.observation_space = Box(0.0, 1.0, [n_colors, height, width])

    def observation(self, img):
        img = self.crop(img)
        img = imresize(img, self.img_size)
        if self.grayscale:
            img = img.mean(-1, keepdims = True)
        img = np.transpose(img, (2, 0, 1))
        img = img.astype('float32') / 255.
        return img

"""## ------------------------ EXPERIENCE REPLAY (experience_replay.py) ------------------------

### Importing the libraries
"""
```

```
import numpy as np
from collections import namedtuple, deque

"""### Defining One Step"""

Step = namedtuple('Step', ['state', 'action', 'reward', 'done'])

"""### Making the AI progress on several (n_step) steps"""

class NStepProgress:

    def __init__(self, env, ai, n_step):
        self.ai = ai
        self.rewards = []
        self.env = env
        self.n_step = n_step

    def __iter__(self):
        state = self.env.reset()
        history = deque()
        reward = 0.0
        while True:
            action = self.ai(np.array([state]))[0][0]
            next_state, r, is_done, _ = self.env.step(action)
            reward += r
            history.append(Step(state = state, action = action, reward = r, done = is_done))
            while len(history) > self.n_step + 1:
                history.popleft()
            if len(history) == self.n_step + 1:
                yield tuple(history)
            state = next_state
            if is_done:
                if len(history) > self.n_step + 1:
                    history.popleft()
                while len(history) >= 1:
                    yield tuple(history)
                    history.popleft()
                self.rewards.append(reward)
                reward = 0.0
                state = self.env.reset()
                history.clear()

    def rewards_steps(self):
        rewards_steps = self.rewards
        self.rewards = []
        return rewards_steps

"""### Implementing Experience Replay"""

class ReplayMemory:

    def __init__(self, n_steps, capacity = 10000):
        self.capacity = capacity
        self.n_steps = n_steps
        self.n_steps_iter = iter(n_steps)
        self.buffer = deque()

    def sample_batch(self, batch_size): # creates an iterator that returns random batches
        ofs = 0
        vals = list(self.buffer)
        np.random.shuffle(vals)
        while (ofs+1)*batch_size <= len(self.buffer):
```

```
            yield vals[ofs*batch_size:(ofs+1)*batch_size]
            ofs += 1

    def run_steps(self, samples):
        while samples > 0:
            entry = next(self.n_steps_iter) # 10 consecutive steps
            self.buffer.append(entry) # we put 200 for the current episode
            samples -= 1
        while len(self.buffer) > self.capacity: # we accumulate no more than the capacity (10000)
            self.buffer.popleft()

"""## ---------- AI FOR DOOM (ai.py) ----------

### Importing the libraries
"""

import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.autograd import Variable

"""### Importing the packages for OpenAI and Doom"""

import gym
import vizdoomgym
from gym import wrappers

"""## Part 1 - Building the AI

### Making the Brain
"""

class CNN(nn.Module):

    def __init__(self, number_actions):
        super(CNN, self).__init__()
        self.convolution1 = nn.Conv2d(in_channels = 1, out_channels = 32, kernel_size = 5)
        self.convolution2 = nn.Conv2d(in_channels = 32, out_channels = 32, kernel_size = 3)
        self.convolution3 = nn.Conv2d(in_channels = 32, out_channels = 64, kernel_size = 2)
        self.fc1 = nn.Linear(in_features = self.count_neurons((1, 256, 256)), out_features = 40)
        self.fc2 = nn.Linear(in_features = 40, out_features = number_actions)

    def count_neurons(self, image_dim):
        x = Variable(torch.rand(1, *image_dim))
        x = F.relu(F.max_pool2d(self.convolution1(x), 3, 2))
        x = F.relu(F.max_pool2d(self.convolution2(x), 3, 2))
        x = F.relu(F.max_pool2d(self.convolution3(x), 3, 2))
        return x.data.view(1, -1).size(1)
```

7.0 Test Cases Input

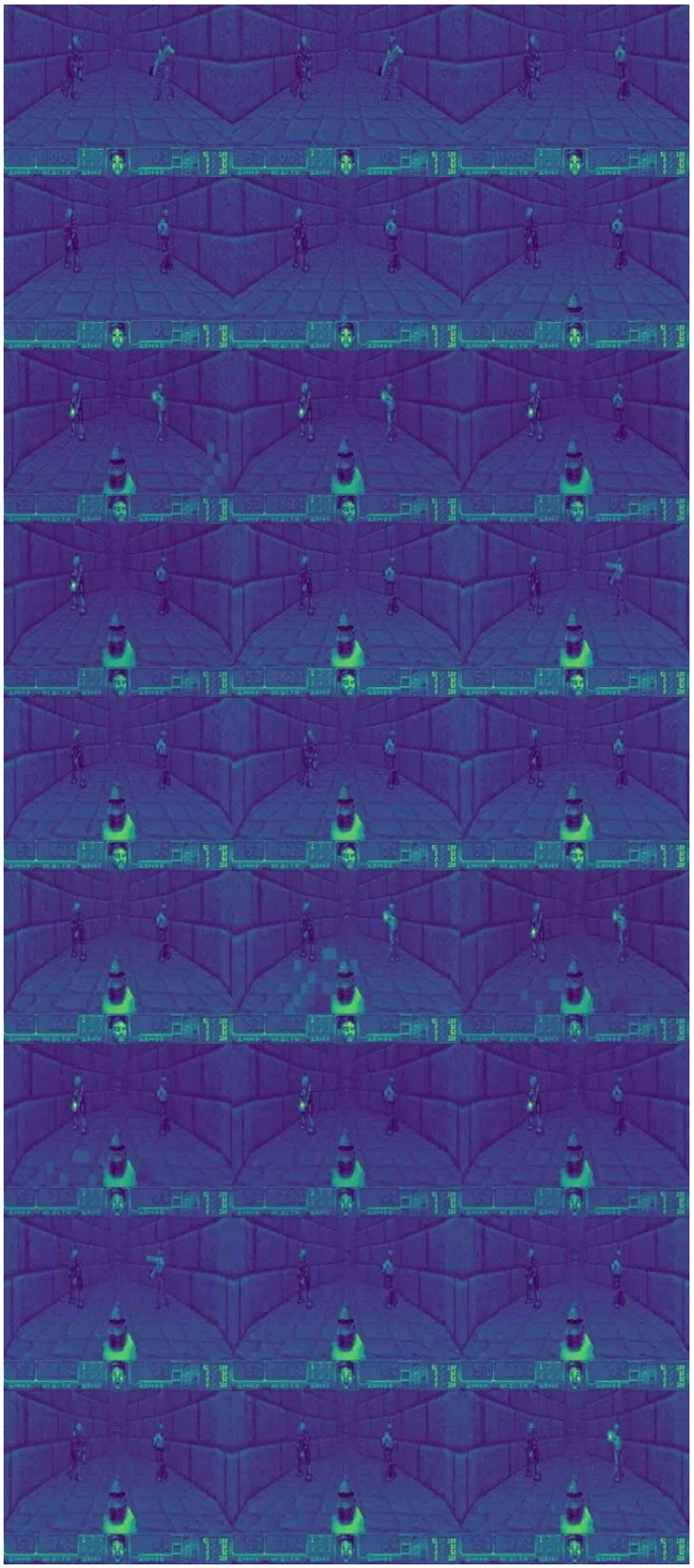
Input is always a 320x320 pixel image of the current display of the video game screen. The agent have the option to the following moves at a given point in time

1. MOVE_LEFT
2. MOVE_RIGHT
3. ATTACK
4. MOVE_FORWARD
5. MOVE_BACKWARD
6. TURN_LEFT
7. TURN_RIGHT

The reinforcement learning reward being as followed

1. +dX for getting closer to the end of the corridor.
2. -dX for getting further from the end of the corridor.
3. -100 death penalty

7.1 Test Case 1 Output (Agent with no Training)

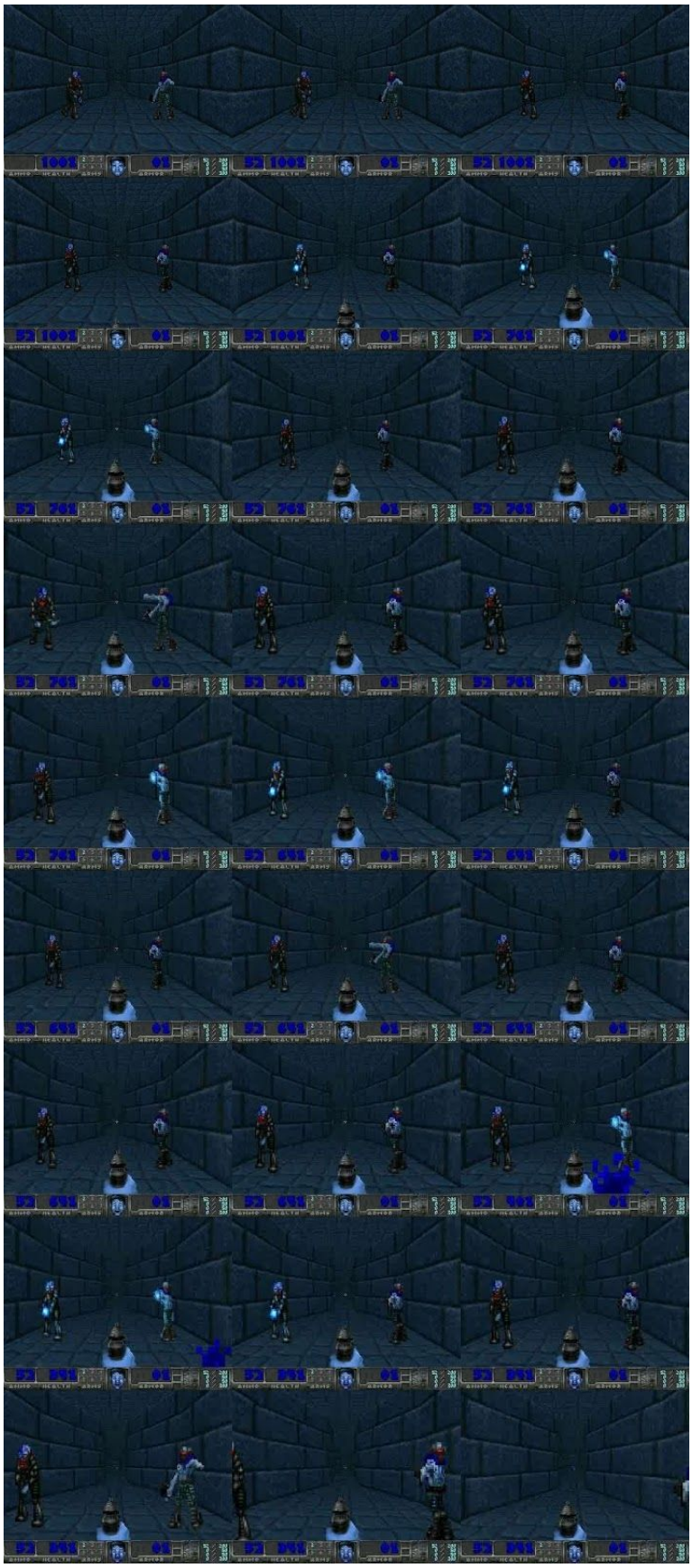


The output for test case 1 clearly shows how incapable the agent is at playing the game without learning. Although the agent learned to take out his gun he doesn't move and receives all gunshot damage from the enemy across the corridor.

Drive Link for the output:

https://drive.google.com/file/d/15OHlPFy5YZMuKvS4C_DsSlgpMjqPgvwC/view?usp=sharing

7.2 Test Case 2 Output (Agent with 6 hours of Training)



The output for test case 2 shows a clear improvement is the agent playing the game. The agent is shown doing MOVE_LEFT and MOVE_RIGHT movement to dodge the enemy bullets and uses MOVE_FORWARD movement to move past the enemy in the corridor.

Drive Link for the output:

https://drive.google.com/file/d/1MEmfqmZwM8I2WVLk5v2u_4LKHzLe5twO/view?usp=sharing

8.0 Discussion

As a preliminary evaluation, we first ran our algorithm (without experience replay) for 5 trials, each with a cut off at the end of 36 hours, using the number of consecutive frames k=3. The final scores for each trial are plotted in the top half of. From the results, we can see that some trials performed pretty badly, and are in fact no better than the baseline Q-learning algorithm. However, other trials performed significantly better than the baseline. We believe these differences in performance among the trials can be explained by the gradient descent approach that we used for training our deep neural network, which is characteristically vulnerable to getting stuck at some under-performing local optimum. We arrived at this explanation because each trial is initialized with random weights and biases, and these trials produce wildly different final scores, so most likely each of them ended up in a different local optimum.

We then tweaked our model parameters (aka hyperparameters) such as the exploitation-exploration parameter, k the number of consecutive frames in consideration, and η the learning rate, in a manner similar to grid search. Since in our situation we do not have a dataset for which to divide into training, validation, and testing sets for the reason that out training comes from operating a dynamic game, we simply repeated training on various values of, k and η and find the combination that gives the best scores. This is straightforward compared to the usual hyperparameter optimization process in machine learning.

We found that setting η to 0.01, k to 4, and to a dynamic one that linearly decreases from 0.8 to 0.05 annealed over 50,000 timesteps gives the best results overall, but we notice that in general varying the hyperparameters does not significantly influence the game agent's performance, so we did not devote too much time trying out different combinations.

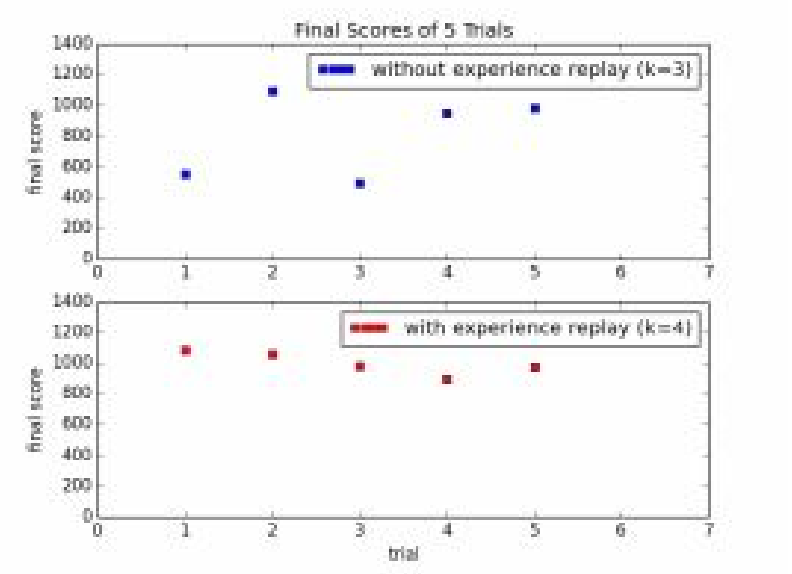


Figure 1: Comparison: performance with and without experience replay

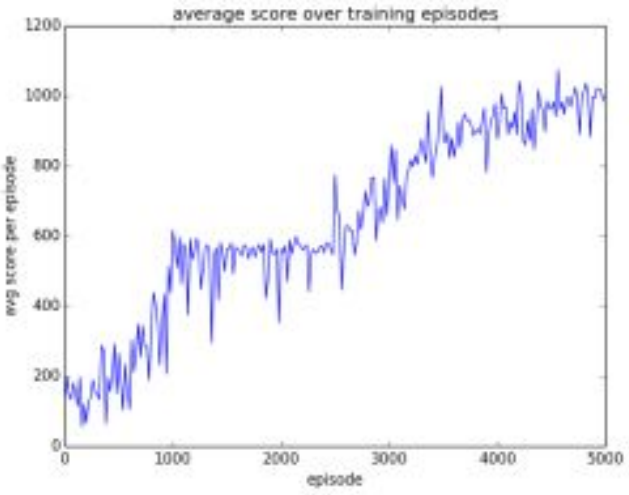


Figure 2: Average scores over training episodes

We obtained our final results by running the agent using the model weights computed at the end of training and noting down the final score repeated for a total of 5 trials, and instead of doing the cutoff based on time, we now terminate training after performing 50,000 iterations. We switched from time-based cutoff to iteration-based cutoff because the time it takes to train a model is mainly a function of the hardware used to train it. Furthermore, reporting the scores obtained from a certain number of episodes makes it more robust against the noise/stochasticity of the OpenAI gym environment. This time we also incorporated experience replay into the training procedure. The scores we obtained are plotted on the bottom half of. Comparing these results with those above, we can see that experience replay produced results that are more consistent and stable. This is because this technique was able to effectively prevent the algorithm from getting stuck at some bad local optimum, and thus help achieve (slightly) higher and more consistent results.
To aid with our analysis we plot the average score per 20 consecutive training episodes over one complete trial in. We can see that in the beginning, the score rises rapidly to around 600 because the game mode stays the same up to this point and so far it is quite easy to get there. After which it appears to get stuck at around 600 for over 1500 episodes, and this is because starting from this point, the game jumps in difficulty – new enemy entities known as ”crawlers” appear on the left and right sides of the agent, in addition to the enemy ships already hovering above. As if it encountered a roadblock, the agent was unable to make much progress past 600 for quite some time, but it did eventually learn to overcome this obstacle. So we conclude that even with this change of difficulty, our agent simply needed some time to adjust and continue to learn. From 600 onwards the agent improves at a rate slower than it did from the start to 600 because the game is no longer as easy as it was in the beginning. Finally, it saturated at around 1000 and this is roughly the final score it was able to achieve.

9.0 Conclusion and Future Scope

ViZDoom is a Doom-based platform for research in vision-based reinforcement learning. It is easy-to-use, highly flexible, multi-platform, lightweight, and efficient. In contrast to the other popular visual learning environments such as Atari 2600, ViZDoom provides a 3D, semi-realistic, first-person perspective virtual world. ViZDoom's API gives the user full control of the environment. Multiple modes of operation facilitate experimentation with different learning paradigms such as reinforcement learning, apprenticeship learning, learning by demonstration, and, even the 'ordinary', supervised learning. The strength and versatility of the environment lie in is customizability via the mechanism of scenarios, which can be conveniently programmed with open-source tools.

We also demonstrated that visual reinforcement learning is possible in the 3D virtual environment of ViZDoom by performing experiments with deep Q-learning on two scenarios. The results of the simple move-and-shoot scenario, indicate that the speed of the learning system highly depends on the number of frames the agent is allowed to skip during the learning. We have found out that it is profitable to skip from 4 to 10 frames. We used this knowledge in the second, more involved, scenario, in which the agent had to navigate through a hostile maze and collect some items and avoid the others. Although the agent was not able to find a perfect strategy, it learned to navigate the maze surprisingly well exhibiting evidence of a human-like behavior

ViZDoom has recently reached a stable 1.0.1 version and has the potential to be extended in many interesting directions. First, we would like to implement an asynchronous multiplayer mode, which would be convenient for self-learning in multiplayer settings. Second, bots are now deaf thus, we plan to allow bots to access the sound buffer. Lastly, interesting supervised learning experiments (e.g., segmentation) could be conducted if ViZDoom automatically labeled objects in the scene.

In this project, we have implemented a game-playing agent for Atari Assault using deep Q-learning. We first implemented ordinary Q-learning to obtain a baseline of score 670.7, then we implemented deep Q-learning by constructing a convolutional neural network using Tensorflow. We obtained promising results after experimenting with and without experience replay. For us, experience replay worked well in helping the agent avoid getting stuck at some bad local optimum. Some of our improvements also came about by extending the training time and tweaking hyperparameters. Our deep Q-learning agent managed to significantly outperform the baseline: the average score it obtained was 980, while the ordinary Q-learning baseline has an average score of 670.7.

Although our agent does significantly better than the baseline, it still does not come close to the oracle. We believe that there are still many places we can try to improve, such as revising the neural network architecture, adding customized feature extractors, and experimenting with more fully connected layers.

VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

Review 3

SLOT - B1+TB1

Fall Semester 2020-21

Project Title

Training an AI Bot to beat Retro Games using Reinforcement Learning Algorithms

TAARUSSH WADHWA - 18BIT0312
MADHAV RAJESH - 18BIT0325

Artificial Intelligence(ITE-2010)

Faculty: Prof. Ajit Kumar Santra

1.0 Abstract

Reinforcement learning and games have a long and mutually beneficial common history. Deep reinforcement learning has made great achievements since it was initially proposed. Generally, Deep Reinforcement Learning agents receive high-dimensional inputs at each step and take actions according to certain neural network policies. This learning mechanism updates the policy to maximize the accuracy of the output with an end-to-end method. There have been many advancements in Deep Reinforcement Learning methods, including value-based, policy gradient, and model-based algorithms. Deep Reinforcement Learning also plays an important role in in-game artificial intelligence (AI). With the help of Deep Reinforcement Learning methodology, there have been achievements in various video games, including classical Arcade games, first-person perspective games, and multi-agent real-time strategy games, from 2D to 3D, and from single-agent to multi-agent. A large number of video game AIs utilizing Deep Reinforcement Learning have achieved super-human performance, although there are still some challenges in this domain. The recent advances in deep neural networks have led to effective vision-based reinforcement learning methods that have been employed to obtain human-level controllers in Atari 2600 games from pixel data. Atari 2600 games, however, do not resemble real-world tasks since they involve non-realistic 2D environments and the third-person perspective. Here, we propose a novel test-bed platform for reinforcement learning research from raw visual information which employs the first-person perspective in a semi-realistic 3D world. The software, called ViZDoom, is based on the classical first-person shooter video game, Doom. It allows developing bots that play the game using the screen buffer. ViZDoom is lightweight, fast, and highly customizable via a convenient mechanism of user scenarios. In the experimental part, we test the environment by trying to learn bots for two scenarios: a basic move-and-shoot task and a more complex maze-navigation problem. Using convolutional deep neural networks with Q-learning and experience replay, for both scenarios, we were able to train competent bots, which exhibit human-like behaviors. The results confirm the utility of ViZDoom as an AI research platform and imply that visual reinforcement learning in 3D realistic first-person perspective environments is feasible. Keywords: video games, visual-based reinforcement learning, deep reinforcement learning, first-person perspective games, FPS, visual learning, neural networks
In our project, we try to teach an AI bot to play games using these Deep Reinforcement Learning techniques.

2.0 Objective

2. Build our own AI Game bot using Deep Reinforcement Learning.

3.0 Introduction

Artificial intelligence (AI) in video games is a longstanding research area. It studies how to use AI technologies to achieve human-level performance when playing games. More generally, it studies the complex interactions between agents and game environments. Various games provide interesting and complex problems for agents to solve, making video games perfect environments for AI research. These virtual environments are safe and controllable. In addition, these game environments provide an infinite supply of useful data for machine learning algorithms, and they are much faster than real-time. These characteristics make games the unique and favorite domain for AI research. On the other side, AI has been helping games to become better in the way we play, understand, and design them. Broadly speaking, game AI involves the perception and decision-making in various game environments. Deep reinforcement learning continues to be very successful in mastering human-level control policies in a wide variety of tasks such as object recognition with visual attention, high-dimensional robot control, and solving physics-based control problems. In particular, Deep QNetworks (DQN) is shown to be effective in playing Atari 2600 games and more recently, in defeating world-class Go players(Google's AlphaGo)

3.1 Literature Survey

David et al. [1], (2016) High-dimensional observations and complex real-world dynamics present major challenges in reinforcement learning for both function approximation and exploration. We address both of these challenges with two complementary techniques: First, we develop a gradient-boosting style, non-parametric function approximator for learning on Q-function residuals. And second, we propose an exploration strategy inspired by the principles of state abstraction and information acquisition under uncertainty. We demonstrate the empirical effectiveness of these techniques, first, as a preliminary check, on two standard tasks (Blackjack and n-Chain), and then on two much larger and more realistic tasks with high-dimensional observation spaces. Specifically, we introduce two benchmarks built within the game Minecraft where the observations are pixel arrays of the agent's visual field. A combination of our two algorithmic techniques performs competitively on the standard reinforcement-learning tasks while consistently and substantially outperforming baselines on the two tasks with high-dimensional observation spaces. The new function approximator, exploration strategy, and evaluation benchmarks are each of independent interest in the pursuit of reinforcement-learning methods that scale to real-world domains.

Minoru et al. [2], (1996) This paper presents a method of vision-based reinforcement learning by which a robot learns to shoot a ball into a goal. We discuss several issues in applying the reinforcement learning method to a real robot with vision sensor by which the robot can obtain information about the changes in an environment. First, we construct a state space in terms of size, position, and orientation of a ball and a goal in an image, and an action space is designed in terms of the action commands to be sent to the left and right motors of a mobile robot. This causes a "state-action deviation" problem in constructing the state and action spaces that reflect the outputs from physical sensors and actuators, respectively. To deal with this issue, an action set is constructed in a way that one action consists of a series of the same action primitive which is successively executed until the current state changes. Next, to speed up the learning time, a mechanism of Learning from Easy Missions (or LEM) is implemented. LEM reduces the learning

time from exponential to almost linear order in the size of the state space. The results of computer simulations and real robot experiments are given.

Minoru et al. [3], (1994) Because of the recent success and advancements in deep mind technologies, it is now used to train agents using deep learning for first-person shooter games that are often outperforming human players by means of only screen raw pixels to create their decisions. A visual Doom AI Competition is organized each year on two different tracks: limited death-match on a known map and a full death-match on an unknown map for evaluating AI agents because computer games are the best test-beds for testing and evaluating different AI techniques and approaches. The competition is ranked based on the number of frags each agent achieves. In this paper, training a competitive agent for playing Doom's (FPS Game) basic scenario(s) in a semi-realistic 3D world 'VizDoom' using the combination of convolutional Deep learning and Q-learning by considering only the screen raw pixels in order to exhibit agent's usefulness indoor is proposed. Experimental results show that the trained agent outperforms average human player and inbuilt game agents in basic scenario(s) where only move left, right and shoot actions are allowed.

Nicholas et al. [4], (2004) First-person shooter robot controllers (hots) are generally rule-based expert systems written in C/C++. As such, many of the rules are parameterized with values, which are set by the software designer and finalized at compile time. The effectiveness of parameter values is dependent on the knowledge the programmer has about the game. Furthermore, parameters are non-linearly dependent on each other. This paper presents an efficient method for using a genetic algorithm to evolve sets of parameters for bots which lead to their playing as well as hots whose parameters have been tuned by a human with expert knowledge about the game's strategy. This indicates genetic algorithms as being a potentially useful method for tuning hots.

Giuseppe et al. [5], (2011) Neuroevolution, the artificial evolution of neural networks, has shown great promise on continuous reinforcement learning tasks that require memory. However, it is not yet directly applicable to realistic embedded agents using high-dimensional (e.g. raw video images) inputs, requiring very large networks. In this paper, neuroevolution is combined with an unsupervised sensory pre-processor or compressor that is trained on images generated from the environment by the population of evolving recurrent neural network controllers. The compressor not only reduces the input cardinality of the controllers, but also biases the search toward novel controllers by rewarding those controllers that discover images that it reconstructs poorly. The method is successfully demonstrated on a vision-based version of the well-known mountain car benchmark, where controllers receive only single high-dimensional visual images of the environment, from a third-person perspective, instead of the standard two-dimensional state vector which includes information about velocity.

Mark and Richard [6], (2005) One trend in first-person shooter computer games is to increase programmatic access. This allows artificial intelligence researchers to embed their cognitive models into stable artificial characters, whose competitiveness and realism can be evaluated with respect to human players. However, plugging cognitive models in is non-trivial, since games are currently not designed with AI researchers in mind. In this paper we introduce an intermediate architecture that fits between these game and the AI, and assess its feasibility by implementing it within the game Unreal Tournament 2004. Making such an architecture publicly available may potentially lead to improved quality of game AI.

Abdennour [7], (2008) The aim of developing an agent, that is able to adapt its actions in response to their effectiveness within the game, provides the basis for the research presented in this paper. It investigates how adaptation can be applied through the use of a hybrid of AI technologies. The system developed uses the predefined behaviours of a finite-state machine and fuzzy logic system combined with the learning capabilities of a neural computing. The system adapts specific behaviours that are central to the performance of the bot (a computer-controlled player that simulates a human opponent) in the game, with the paper's main focus being on that of the weapon selection behaviour; selecting the best weapon for the current situation. As a development platform, the project makes use of the Quake 3 Arena engine, modifying the original bot AI to integrate the adaptive technologies.

Esparcia-Alcazar et al. [8], (2010) In this paper we employ a steady state genetic algorithm to evolve different types of behaviour for bots in the Unreal Tournament 2004 computer game. For this purpose we define three fitness functions which are based on the number of enemies killed, the lifespan of the bot and a combination of both. Long run experiments were carried out, in which the evolved bots' behaviours outperform those of standard bots supplied by the game, particularly in those cases where the fitness involves a measure of the bot's lifespan. Also, there is an increase in the number of items collected, and the behaviours tend to become less aggressive, tending instead towards a more optimised combat style. Further "short run" experiments were carried out with a further type of fitness function defined, based on the number of items picked. In these cases the bots evolve performances towards the goal they have been aimed, with no other behaviours arising, except in the case of the multiple objective one. We conclude that in order to evolve interesting behaviours more complex fitness functions are needed, and not necessarily ones that directly include the goal we are aiming for.

Chris et al. [9], (2000) Reinforcement learning systems improve behaviour based on scalar rewards from a critic. In this work vision based behaviours, servoing and wandering, are learned through a Q-learning method which handles continuous states and actions. There is no requirement for camera calibration, an actuator model, or a knowledgeable teacher. Learning through observing the actions of other behaviours improves learning speed. Experiments were performed on a mobile robot using a real-time vision system.

Borja et al. [10] (2018) To solve complex real-world problems with reinforcement learning, we cannot rely on manually specified reward functions. Instead, we need humans to communicate an objective to the agent directly. In this work, we combine two approaches to this problem: learning from expert demonstrations and learning from trajectory preferences. We use both to train a deep neural network to model the reward function and use its predicted reward to train an DQN-based deep reinforcement learning agent on 9 Atari games. Our approach beats the imitation learning baseline in 7 games and achieves strictly superhuman performance on 2 games. Additionally, we investigate the fit of the reward model, present some reward hacking problems, and study the effects of noise in the human labels.

Steven Kapturowski et al. [11] (2018) Building on the recent successes of distributed training of RL agents, in this paper we investigate the training of RNN-based RL agents from distributed prioritized experience replay. We study the effects of parameter lag resulting in representational drift and recurrent state staleness and empirically derive an improved training strategy. Using a single network architecture and fixed set of hyper-parameters, the resulting agent, Recurrent

Replay Distributed DQN, quadruples the previous state of the art on Atari-57, and matches the state of the art on DMLab-30. It is the first agent to exceed human-level performance in 52 of the 57 Atari games.

F G Glavin et al. [12], (2015) In current state-of-the-art commercial first person shooter games, computer controlled bots, also known as nonplayer characters, can often be easily distinguishable from those controlled by humans. Tell-tale signs such as failed navigation, “sixth sense” knowledge of human players' whereabouts and deterministic, scripted behaviors are some of the causes of this. We propose, however, that one of the biggest indicators of nonhumanlike behavior in these games can be found in the weapon shooting capability of the bot. Consistently perfect accuracy and “locking on” to opponents in their visual field from any distance are indicative capabilities of bots that are not found in human players. Traditionally, the bot is handicapped in some way with either a timed reaction delay or a random perturbation to its aim, which doesn't adapt or improve its technique over time. We hypothesize that enabling the bot to learn the skill of shooting through trial and error, in the same way a human player learns, will lead to greater variation in game-play and produce less predictable nonplayer characters. This paper describes a reinforcement learning shooting mechanism for adapting shooting over time based on a dynamic reward signal from the amount of damage caused to opponents.

Xavier et al. [13], (2011) While logistic sigmoid neurons are more biologically plausible than hyperbolic tangent neurons, the latter work better for training multi-layer neural networks. This paper shows that rectifying neurons are an even better model of biological neurons and yield equal or better performance than hyperbolic tangent networks in spite of the hard non-linearity and non-differentiability at zero, creating sparse representations with true zeros which seem remarkably suitable for naturally sparse data. Even though they can take advantage of semi-supervised setups with extra-unlabeled data, deep rectifier networks can reach their best performance without requiring any unsupervised pre-training on purely supervised tasks with large labeled datasets. Hence, these results can be seen as a new milestone in the attempts at understanding the difficulty in training deep but purely supervised neural networks, and closing the performance gap between neural networks learnt with and without unsupervised pre-training.

S Hladky and V Bulitko [14], (2008) A well-known Artificial Intelligence (AI) problem in video games is designing AI-controlled humanoid characters. It is desirable for these characters to appear both skillful and believably human-like. Many games address the former objective by providing their agents with unfair advantages. Although challenging, these agents are frustrating to humans who perceive the AI to be cheating. In this paper we evaluate hidden semi-Markov models and particle filters as a means for predicting opponent positions. Our results show that these models can perform with similar or better accuracy than the average human expert in the game Counter-Strike: Source. Furthermore, the mistakes these models make are more human-like than perfect predictions.

Igor V. Karpov [15], (2012) Imitation is a powerful and pervasive primitive underlying examples of intelligent behavior in nature. Can we use it as a tool to help build artificial agents that behave like humans do? This question is studied in the context of the BotPrize competition, a Turing-like test where computer game bots compete by attempting to fool human judges into thinking they are just another human player. One problem faced by such bots is that of human-like navigation within the virtual world. This chapter describes the Human Trace Controller, a component of the UT^2 bot

which took second place in the BotPrize 2010 competition. The controller uses a database of recorded human games in order to quickly retrieve and play back relevant segments of human navigation behavior. Empirical evidence suggests that the method of direct imitation allows the bot to effectively solve several navigation problems while moving in a human-like fashion.

Jan et al. [16], (2014) Dealing with high-dimensional input spaces, like visual in- put, is a challenging task for reinforcement learning (RL). Neuroevolution (NE), used for continuous RL problems, has to either reduce the problem dimensionality by (1) compress- ing the representation of the neural network controllers or (2) employing a pre-processor (compressor) that transforms the high-dimensional raw inputs into low-dimensional fea- tures. In this paper, we are able to evolve extremely small recurrent neural network (RNN) controllers for a task that pre- viously required networks with over a million weights. The high-dimensional visual input, which the controller would normally receive, is first transformed into a compact fea- ture vector through a deep, max-pooling convolutional neu- ral network (MPCNN). Both the MPCNN preprocessor and the RNN controller are evolved successfully to control a car in the TORCS racing simulator using only visual input. This is the first use of deep learning in the context evolutionary RL.

Alex et al. [17], (2012) We trained a large, deep convolutional neural network to classify the 1.2 million high-resolution images in the ImageNet LSVRC-2010 contest into the 1000 dif- ferent classes. On the test data, we achieved top-1 and top-5 error rates of 37.5% and 17.0% which is considerably better than the previous state-of-the-art. The neural network, which has 60 million parameters and 650,000 neurons, consists of five convolutional layers, some of which are followed by max-pooling layers, and three fully-connected layers with a final 1000-way softmax. To make train- ing faster, we used non-saturating neurons and a very efficient GPU implemen- tation of the convolution operation. To reduce overfitting in the fully-connected layers we employed a recently-developed regularization method called “dropout” that proved to be very effective. We also entered a variant of this model in the ILSVRC-2012 competition and achieved a winning top-5 test error rate of 15.3%, compared to 26.2% achieved by the second-best entry.

Sascha Lange and Martin Riedmiller [18], (2010) This paper discusses the effectiveness of deep auto-encoder neural networks in visual reinforcement learning (RL) tasks. We propose a framework for combining the training of deep auto-encoders (for learning compact feature spaces) with recently-proposed batch-mode RL algorithms (for learning policies). An emphasis is put on the data-efficiency of this combination and on studying the properties of the feature spaces automatically constructed by the deep auto-encoders. These feature spaces are empirically shown to adequately resemble existing similarities and spatial relations between observations and allow to learn useful policies. We propose several methods for improving the topology of the feature spaces making use of task-dependent information. Finally, we present first results on successfully learning good control policies directly on synthesized and real images.

M McPartland and M Gallagher [20], (2011) Deep neural network acoustic models pro- duce substantial gains in large vocabu- lary continuous speech recognition systems. Emerging work with rectified linear (ReL) hidden units demonstrates additional gains in final system performance relative to more commonly used sigmoidal nonlinearities. In this work, we explore the use of deep rectifier networks as acoustic models for the 300 hour Switchboard conversational speech recogni- tion task. Using simple training procedures without pretraining, networks with rectifier nonlinearities produce 2% absolute reduc- tions in word error rates over their sigmoidal

counterparts. We analyze hidden layer repre- sentations to quantify differences in how ReL units encode inputs as compared to sigmoidal units. Finally, we evaluate a variant of the ReL unit with a gradient more amenable to optimization in an attempt to further im- prove deep rectifier networks.

Volodymyr Mnih [21], (2015) Reinforcement learning (RL) is a popular machine learning technique that has many successes in learning how to play classic style games. Applying RL to first person shooter (FPS) games is an interesting area of research as it has the potential to create diverse behaviors without the need to implicitly code them. This paper investigates the tabular Sarsa (λ) RL algorithm applied to a purpose built FPS game. The first part of the research investigates using RL to learn bot controllers for the tasks of navigation, item collection, and combat individually. Results showed that the RL algorithm was able to learn a satisfactory strategy for navigation control, but not to the quality of the industry standard pathfinding algorithm. The combat controller performed well against a rule-based bot, indicating promising preliminary results for using RL in FPS games. The second part of the research used pretrained RL controllers and then combined them by a number of different methods to create a more generalized bot artificial intelligence (AI). The experimental results indicated that RL can be used in a generalized way to control a combination of tasks in FPS bots such as navigation, item collection, and combat.

Samuel et al. [22] (2018) While deep reinforcement learning (deep RL) agents are effective at maximizing rewards, it is often unclear what strategies they use to do so. In this paper, we take a step toward explaining deep RL agents through a case study using Atari 2600 environments. In particular, we focus on using saliency maps to understand how an agent learns and executes a policy. We introduce a method for generating useful saliency maps and use it to show 1) what strong agents attend to, 2) whether agents are making decisions for the right or wrong reasons, and 3) how agents evolve during learning. We also test our method on non-expert human subjects and find that it improves their ability to reason about these agents. Overall, our results show that saliency information can provide significant insight into an RL agent’s decisions and learning behavior.

Tony C Smith and Jonathan Miles [23], (2014) In this paper we present RETALIATE, an online reinforcement learning algorithm for developing winning policies in team first- person shooter games. RETALIATE has three crucial characteristics: (1) individual BOT behavior is fixed although not known in advance, therefore individual BOTS work as “plug- ins”, (2) RETALIATE models the problem of learning team tactics through a simple state formulation, (3) discount rates commonly used in Q-learning are not used. As a result of these characteristics, the application of the Q-learning algorithm results in the rapid exploration towards a winning policy against an opponent team. In our empirical evaluation we demonstrate that RETALIATE adapts well when the environment changes.

Devdhar Patel et al. [24] (2019) Deep Reinforcement Learning (RL) demonstrates excellent performance on tasks that can be solved by trained policy. It plays a dominant role among cutting-edge machine learning approaches using multi-layer Neural networks (NNs). At the same time, Deep RL suffers from high sensitivity to noisy, incomplete, and misleading input data. Following biological intuition, we involve Spiking Neural Networks (SNNs) to address some deficiencies of deep RL solutions. Previous studies in image classification domain demonstrated that standard NNs (with ReLU nonlinearity) trained using supervised learning can be converted to SNNs with negligible deterioration in performance. In this paper, we extend those conversion results to the domain of Q-Learning NNs trained using RL. We provide a proof of principle of the

conversion of standard NN to SNN. In addition, we show that the SNN has improved robustness to occlusion in the input image. Finally, we introduce results with converting full-scale Deep Q-network to SNN, paving the way for future research to robust Deep RL applications.

Chang et al. [25], (2011) The implementation of Artificial Intelligence (AI)in 3-Dimensional (3D) First Person Shooter (FPS) game is quite general nowadays. Most of the conventional AI bots created are mostly from hard coded AI bots. Hence, it has limited the dynamicity of the AI bots and therefore it brings to a fixed strategy for gaming. The main focus of this paper is to discuss the methodologies used in generating the AI bots that is competitive in the FPS gaming. In this paper, a decision making structure is proposed. It has been combined with the Evolutionary Programming in generating the required AI controllers. Hence, there are two methodology discussions involved: (1) the proposed decision making structure and (2)the Evolutionary Programming used. The experiments show highly promising testing results after the generated AI bots have been tested and compared with the conventional ruled based AI bots. It proves that the generated AI bots using the combination of Evolutionary Programming and decision making structure performed better than those AI bots generated using conventional ruled based strategy which is hard coded and time consuming to develop.

David Trenholme and Shamus P Smith [26], (2008) Building realistic virtual environments is a complex, expensive and time consuming process. Although virtual environment development toolkits are available, many only provide a subset of the tools needed to build complete virtual worlds. One alternative is the reuse of computer game technology. The current generation of computer games present realistic virtual worlds featuring user friendly interaction and the simulation of real world phenomena. Using computer games as the basis for virtual environment development has a number of advantages. Computer games are robust and extensively tested, both for usability and performance, work on off-the-shelf systems and can be easily disseminated, for example via online communities. Additionally, a number of computer game developers provide tools, documentation and source code, either with the game itself or separately available, so that end-users can create new content. This short report overviews several currently available game engines that are suitable for prototyping virtual environments.

Mousavi et al [27] (2017) In order to accelerate the learning process in high dimensional reinforcement learning problems, TD methods such as Q-learning and Sarsa are usually combined with eligibility traces. The recently introduced DQN (Deep Q-Network) algorithm, which is a combination of Q-learning with a deep neural network, has achieved good performance on several games in the Atari 2600 domain. However, the DQN training is very slow and requires too many time steps to converge. In this paper, we use the eligibility traces mechanism and propose the deep Q(λ) network algorithm. The proposed method provides faster learning in comparison with the DQN method. Empirical results on a range of games show that the deep Q(λ) network significantly reduces learning time.

C. J. C. H. Watkins and P. Dayan [29], (1992) Q-learning is a simple way for agents to learn how to act optimally in controlled Markovian domains. It amounts to an incremental method for dynamic programming which imposes limited computational demands. It works by successively improving its evaluations of the quality of particular actions at particular states.
This paper presents and proves in detail a convergence theorem forQ-learning based on that outlined in Watkins (1989). We show thatQ-learning converges to the optimum action-values with

probability 1 so long as all actions are repeatedly sampled in all states and the action-values are represented discretely. We also sketch extensions to the cases of non-discounted, but absorbing, Markov environments, and where manyQ values can be changed each iteration, rather than just one.

Mnih et al. [29], (2016) We present the first deep learning model to successfully learn control policies directly from high-dimensional sensory input using reinforcement learning. The model is a convolutional neural network, trained with a variant of Q-learning, whose input is raw pixels and whose output is a value function estimating future rewards. We apply our method to seven Atari 2600 games from the Arcade Learning Environment, with no adjustment of the architecture or learning algorithm. We find that it outperforms all previous approaches on six of the games and surpasses a human expert on three of them.

Guo et al. [30], (2016) The combination of modern Reinforcement Learning and Deep Learning approaches holds the promise of making significant progress on challenging applications requiring both rich perception and policy-selection. The Arcade Learning Environment (ALE) provides a set of Atari games that represent a useful benchmark set of such applications. A recent breakthrough in combining model-free reinforcement learning with deep learning, called DQN, achieves the best real-time agents thus far. Planning-based approaches achieve far higher scores than the best model-free approaches, but they exploit information that is not available to human players, and they are orders of magnitude slower than needed for real-time play. Our main goal in this work is to build a better real-time Atari game playing agent than DQN. The central idea is to use the slow planning-based agents to provide training data for a deep-learning architecture capable of real-time play. We proposed new agents based on this idea and show that they outperform DQN.

Matiisen, B. T. [31], (2016) Today, exactly two years ago, a small company in London called DeepMind uploaded their pioneering paper “Playing Atari with Deep Reinforcement Learning” to Arxiv. In this paper they demonstrated how a computer learned to play Atari 2600 video games by observing just the screen pixels and receiving a reward when the game score increased. The result was remarkable, because the games and the goals in every game were very different and designed to be challenging for humans. The same model architecture, without any change, was used to learn seven different games, and in three of them the algorithm performed even better than a human!

Nachum. O et al. [32] (2018) Hierarchical reinforcement learning (HRL) is a promising approach to extend traditional reinforcement learning (RL) methods to solve more complex tasks. Yet, the majority of current HRL methods require careful task-specific design and on-policy training, making them difficult to apply in real-world scenarios. In this paper, we study how we can develop HRL algorithms that are general, in that they do not make onerous additional assumptions beyond standard RL algorithms, and efficient, in the sense that they can be used with modest numbers of interaction samples, making them suitable for real-world problems such as robotic control. For generality, we develop a scheme where lower-level controllers are supervised with goals that are learned and proposed automatically by the higher-level controllers. To address efficiency, we propose to use off-policy experience for both higher- and lower-level training. This poses a considerable challenge, since changes to the lower-level behaviors change the action space for the higher-level policy, and we introduce an off-policy correction to remedy this challenge. This allows us to take advantage of recent advances in off-policy model-free RL to learn both higher

and lower-level policies using substantially fewer environment interactions than on-policy algorithms. We find that our resulting HRL agent is generally applicable and highly sample-efficient. Our experiments show that our method can be used to learn highly complex behaviors for simulated robots, such as pushing objects and utilizing them to reach target locations, learning from only a few million samples, equivalent to a few days of real-time interaction. In comparisons with a number of prior HRL methods, we find that our approach substantially outperforms previous state-of-the-art techniques.

François-Lavet et al [33] (2018) Deep reinforcement learning is the combination of reinforcement learning (RL) and deep learning. This field of research has been able to solve a wide range of complex decision-making tasks that were previously out of reach for a machine. Thus, deep RL opens up many new applications in domains such as healthcare, robotics, smart grids, finance, and many more. This manuscript provides an introduction to deep reinforcement learning models, algorithms and techniques. Particular focus is on the aspects related to generalization and how deep RL can be used for practical applications. We assume the reader is familiar with basic machine learning concepts.

Cobbe K. et al [34] (2019) In this paper, we investigate the problem of overfitting in deep reinforcement learning. Among the most common benchmarks in RL, it is customary to use the same environments for both training and testing. This practice offers relatively little insight into an agent's ability to generalize. We address this issue by using procedurally generated environments to construct distinct training and test sets. Most notably, we introduce a new environment called CoinRun, designed as a benchmark for generalization in RL. Using CoinRun, we find that agents overfit to surprisingly large training sets. We then show that deeper convolutional architectures improve generalization, as do methods traditionally found in supervised learning, including L2 regularization, dropout, data augmentation and batch normalization.

Lample G. et al [35] (2016) Advances in deep reinforcement learning have allowed autonomous agents to perform well on Atari games, often outperforming humans, using only raw pixels to make their decisions. However, most of these games take place in 2D environments that are fully observable to the agent. In this paper, we present the first architecture to tackle 3D environments in first-person shooter games, that involve partially observable states. Typically, deep reinforcement learning methods only utilize visual input for training. We present a method to augment these models to exploit game feature information such as the presence of enemies or items, during the training phase. Our model is trained to simultaneously learn these features along with minimizing a Q-learning objective, which is shown to dramatically improve the training speed and performance of our agent. Our architecture is also modularized to allow different models to be independently trained for different phases of the game. We show that the proposed architecture substantially outperforms built-in AI agents of the game as well as humans in deathmatch scenarios.

Nair, A.V. et al [36] (2018) For an autonomous agent to fulfill a wide range of user-specified goals at test time, it must be able to learn broadly applicable and general-purpose skill repertoires. Furthermore, to provide the requisite level of generality, these skills must handle raw sensory input such as images. In this paper, we propose an algorithm that acquires such general-purpose skills by combining unsupervised representation learning and reinforcement learning of goal-conditioned policies. Since the particular goals that might be required at test-time are not known in advance,

the agent performs a self-supervised "practice" phase where it imagines goals and attempts to achieve them. We learn a visual representation with three distinct purposes: sampling goals for self-supervised practice, providing a structured transformation of raw sensory inputs, and computing a reward signal for goal reaching. We also propose a retroactive goal relabeling scheme to further improve the sample-efficiency of our method. Our off-policy algorithm is efficient enough to learn policies that operate on raw image observations and goals in a real-world physical system, and substantially outperforms prior techniques.

Li, Y. et al [37] (2017) We give an overview of recent exciting achievements of deep reinforcement learning (RL). We discuss six core elements, six important mechanisms, and twelve applications. We start with background of machine learning, deep learning and reinforcement learning. Next we discuss core RL elements, including value function, in particular, Deep Q-Network (DQN), policy, reward, model, planning, and exploration. After that, we discuss important mechanisms for RL, including attention and memory, unsupervised learning, transfer learning, multi-agent RL, hierarchical RL, and learning to learn. Then we discuss various applications of RL, including games, in particular, AlphaGo, robotics, natural language processing, including dialogue systems, machine translation, and text generation, computer vision, neural architecture design, business management, finance, healthcare, Industry 4.0, smart grid, intelligent transportation systems, and computer systems. We mention topics not reviewed yet, and list a collection of RL resources. After presenting a brief summary, we close with discussions.

Racanière, S. et al [38] (2017) We introduce Imagination-Augmented Agents (I2As), a novel architecture for deep reinforcement learning combining model-free and model-based aspects. In contrast to most existing model-based reinforcement learning and planning methods, which prescribe how a model should be used to arrive at a policy, I2As learn to interpret predictions from a trained environment model to construct implicit plans in arbitrary ways, by using the predictions as additional context in deep policy networks. I2As show improved data efficiency, performance, and robustness to model misspecification compared to several strong baselines.

Kaiser, L. et al [39] (2019) Model-free reinforcement learning (RL) can be used to learn effective policies for complex tasks, such as Atari games, even from image observations. However, this typically requires very large amounts of interaction -- substantially more, in fact, than a human would need to learn the same games. How can people learn so quickly? Part of the answer may be that people can learn how the game works and predict which actions will lead to desirable outcomes. In this paper, we explore how video prediction models can similarly enable agents to solve Atari games with fewer interactions than model-free methods. We describe Simulated Policy Learning (SimPLe), a complete model-based deep RL algorithm based on video prediction models and present a comparison of several model architectures, including a novel architecture that yields the best results in our setting. Our experiments evaluate SimPLe on a range of Atari games in low data regime of 100k interactions between the agent and the environment, which corresponds to two hours of real-time play. In most games SimPLe outperforms state-of-the-art model-free algorithms, in some games by over an order of magnitude.

Lanctot, M. et al [40] (2017) There has been a resurgence of interest in multiagent reinforcement learning (MARL), due partly to the recent success of deep neural networks. The simplest form of MARL is independent reinforcement learning (InRL), where each agent treats all of its experience as part of its (non stationary) environment. In this paper, we first observe that policies learned

using InRL can overfit to the other agents' policies during training, failing to sufficiently generalize during execution. We introduce a new metric, joint-policy correlation, to quantify this effect. We describe a meta-algorithm for general MARL, based on approximate best responses to mixtures of policies generated using deep reinforcement learning, and empirical game theoretic analysis to compute meta-strategies for policy selection. The meta-algorithm generalizes previous algorithms such as InRL, iterated best response, double oracle, and fictitious play. Then, we propose a scalable implementation which reduces the memory requirement using decoupled meta-solvers. Finally, we demonstrate the generality of the resulting policies in three partially observable settings: gridworld coordination problems, emergent language games, and poker.

Jaderberg, M. et al [41] (2019) Reinforcement learning (RL) has shown great success in increasingly complex single-agent environments and two-player turn-based games. However, the real world contains multiple agents, each learning and acting independently to cooperate and compete with other agents. We used a tournament-style evaluation to demonstrate that an agent can achieve human-level performance in a three-dimensional multiplayer first-person video game, Quake III Arena in Capture the Flag mode, using only pixels and game points scored as input. We used a two-tier optimization process in which a population of independent RL agents are trained concurrently from thousands of parallel matches on randomly generated environments. Each agent learns its own internal reward signal and rich representation of the world. These results indicate the great potential of multiagent reinforcement learning for artificial intelligence research.

Gupta, J.K. et al [42] (2019) This work considers the problem of learning cooperative policies in complex, partially observable domains without explicit communication. We extend three classes of single-agent deep reinforcement learning algorithms based on policy gradient, temporal-difference error, and actor-critic methods to cooperative multi-agent systems. To effectively scale these algorithms beyond a trivial number of agents, we combine them with a multi-agent variant of curriculum learning. The algorithms are benchmarked on a suite of cooperative control tasks, including tasks with discrete and continuous actions, as well as tasks with dozens of cooperating agents. We report the performance of the algorithms using different neural architectures, training procedures, and reward structures. We show that policy gradient methods tend to outperform both temporal-difference and actor-critic methods and that curriculum learning is vital to scaling reinforcement learning algorithms in complex multi-agent domains.

3.2 Background

Doom is an FPS (First-person shooter) game developed by Id-software. Its first installment was released on December 10, 1993, for the platform of DOS, and its second installment Doom II: Hell on Earth was released in the following year (1994) for Microsoft Windows, play-station, and Xbox-360. Its third installment Doom-3 was released for Microsoft Windows on August 3, 2004, which was later adapted for Linux and MacOSX. Also, later on, „Vicarious Visions" ported the game to the Xbox and released it on April 3, 2005. Now the very recent and latest installment is DOOM developed by id-software and published by Bethesda Softworks. It was released worldwide on Microsoft Windows, play-station 4, and X box-one as well on May 13, 2016. A common screen of the Doom game is shown below.

These days the research community is very active in research on Doom for being a hot area of research using techniques like deep reinforcement learning or visual reinforcement learning. Besides, different Doom-based research platforms like VizDoom, CocoDoom, and ResearchDoom are developed for implementing deep learning techniques or methods. In the same way, every year different visual Doom AI competitions are organized where the agents (bots) are confirmed to exhibit human-like actions and to show that visual reinforcement learning in 3D FPS game environments is feasible. The first paper Playing Atari with Deep Reinforcement Learning addresses how convolutional

neural networks and deep reinforcement learning combine together to accomplish a high-performance AI agent that plays Atari games. The paper analyzes how Reinforcement learning (RL) provides a good solution to game playing problems and also the challenges in Deep Learning brought about by RL from the data representation perspective. The paper proposes deep reinforcement learning on game-playing agents, which is similar to our goal. However, there are differences between our approaches and theirs. One of the differences is that the approach in this paper relies on heavy downsampling images before feeding them into a neural network. Our approach tries to avoid this downsampling procedure in an attempt to produce better data layers for deep Q-learning. The second related paper Deep Learning for Real-Time Atari Game Play Using Offline Monte-Carlo Tree Search Planning proposes another solution to game playing. Compared to the first paper, which uses a model-free Q-learning strategy, this paper tackles the problem using a combination of Monte-Carlo techniques and deep Q-learning, ending up with a much more sophisticated algorithm that adds extra assumptions and considerable complexity. Though we choose the model-free learning approach, this paper still provides us with insights

about deep learning applied to games, such as the preprocessing of raw data and the architecture of convolutional neural networks.

3.3 Motivation

Reinforcement learning algorithms allow (software) agents to learn from experience how to behave optimally in dynamic environments. A real-world example of this setup is when infants learn to walk. Initially, they arbitrarily move their legs and feet and thus may fall or stay upright. The former is undesirable, the latter is desirable. Over many, many trials, this feedback accumulates and the infants learn which states (postures) require which actions (muscle movements) to accumulate high rewards (staying upright). An early breakthrough in human-level AI was a system that used reinforcement learning to learn to play 'Backgammon' and could compete with top players in the early 1990s. More recent breakthroughs in-game AI came in the wake of the deep learning revolution but these, too, crucially rely on reinforcement learning. The recent advances in deep neural networks have led to effective vision-based reinforcement learning methods that have been employed to obtain human-level controllers in Atari 2600 games from pixel data. Atari 2600 games, however, do not resemble real-world tasks since they involve non-realistic 2D environments and the third-person perspective. Here, we propose a novel test-bed platform for reinforcement learning research from raw visual information which employs the first-person perspective in a semi-realistic 3D world. The software, called ViZDoom, is based on the classical first-person shooter video game, Doom. It allows developing bots that play the game using the screen buffer. ViZDoom is lightweight, fast, and highly customizable via a convenient mechanism of user scenarios. In the experimental part, we test the environment by trying to learn bots for two scenarios: a basic move-and-shoot task and a more complex maze-navigation problem. Using convolutional deep neural networks with Q-learning and experience replay, for both scenarios, we were able to train competent bots, which exhibit human-like behaviors. The results confirm the utility of ViZDoom as an AI research platform and imply that visual reinforcement learning in 3D realistic first-person perspective environments is feasible. Keywords: video games, visual-based reinforcement learning, deep reinforcement learning, first-person perspective games, FPS, visual learning, neural networks

4.0 Methodology

Q-Learning for Game Playing

Reinforcement Learning briefly is a paradigm of Learning Process in which a learning agent learns, overtime, to behave optimally in a certain environment by interacting continuously in the environment. The agent during its course of learning experience various different situations in the environment it is in. These are called *states*. The agent while being in that state may choose from a set of allowable actions which may fetch different *rewards*(or penalties). The learning agent overtime learns to maximize these rewards so as to behave optimally at any given state it is in. Q-Learning is a basic form of Reinforcement Learning which uses Q-values (also called action values) to iteratively improve the behavior of the learning agent.

a) Q-Values or Action-Values: Q-values are defined for states and actions. Q(S, A) is an estimation of how good is it to take the action A at the state S. This estimation of Q(S, A) will be iteratively computed using the TD- Update rule which we will see in the upcoming sections.

b) Rewards and Episodes: An agent over the course of its lifetime starts from a start state, makes a number of transitions from its current state to a next state based on its choice of action and also the environment the agent is interacting in. At every step of transition, the agent from a state takes an action, observes a reward from the environment, and then transits to another state. If at any point of time the agent ends up in one of the terminating states that means there are no further transition possible. This is said to be the completion of an episode.

c) Temporal Difference or TD-Update:
The Temporal Difference or TD-Update rule can be represented as follows :

$$Q(S,A) \leftarrow Q(S,A) + \alpha\left(R + \gamma Q(S',A') - Q(S,A)\right)$$

This update rule to estimate the value of Q is applied at every time step of the agents interaction with the environment. The terms used are explained below. :

- S : Current State of the agent.
- A : Current Action Picked according to some policy.
- S' : Next State where the agent ends up.
- A' : Next best action to be picked using current Q-value estimation, i.e. pick the action with the maximum Q-value in the next state.
- R : Current Reward observed from the environment in Response of current action.
- γ(>0 and <=1) : Discounting Factor for Future Rewards. Future rewars are less valuable than current rewards so they must be discounted. Since Q-value is an estimation of expected rewards from a state, discounting rule applies here as well.
- α : Step length taken to update the estimation of Q(S, A).

d) Choosing the Action to take using ε-greedy policy:
ε-greedy policy of is a very simple policy of choosing actions using the current Q-value estimations. It goes as follows :

- With probability (1-ε) choose the action which has the highest Q-value.
- With probability (ε) choose any action at random.

One common way to deal with the game playing problem is to assume a Markov Decision Process (MDP). This is appropriate for Assault because the enemy agents move randomly. An MDP is a model defined by a set of States, Actions, Transitions, and Rewards. In order to train an AI to tackle game playing tasks, reinforcement learning based on Q-learning is a popular choice. In Q-learning, the MDP recurrence is defined as follows:

Q(s ∗ , a) = Es 0 [r + γmaxa 0Q ∗ (s0, a0)|s, a] (1)

where a is the action it takes, s is the current state, r is the reward, and γ is the discount factor. Furthermore, we can use function approximation by parameterizing Q-value. In this way, we can easily adapt linear regression and gradient descent techniques from machine learning. With function approximation, we can calculate the best weights by adapting the update rule:

w ← w − η[Qˆ opt(s, a;w) − (r + γVˆ opt(s 0))]Φ(s, a) (2)

where w is a vector containing weights of each feature and is initialized randomly to avoid getting into the same local optimum in every trial.

Convolutional Neural Networks:
In deep learning, a convolutional neural network (CNN, or ConvNet) is a class of deep neural networks, most commonly applied to analyzing visual imagery. They are also known as shift invariant or space invariant artificial neural networks (SIANN), based on their shared-weights architecture and translation invariance characteristics They have applications in image and video recognition, recommender systems, image classification, medical image analysis, natural language processing, and financial time series. CNNs are regularized versions of multilayer perceptrons. Multilayer perceptrons usually mean fully connected networks, that is, each neuron in one layer is connected to all neurons in the next layer. The "fully-connectedness" of these networks makes them prone to overfitting data. Typical ways of regularization include adding some form of magnitude measurement of weights to the loss function. CNNs take a different approach towards regularization: they take advantage of the hierarchical pattern in data and assemble more complex patterns using smaller and simpler patterns. Therefore, on the scale of connectedness and complexity, CNNs are on the lower extreme.
A convolutional neural network consists of an input and an output layer, as well as multiple hidden layers. The hidden layers of a CNN typically consist of a series of convolutional layers that convolve with a multiplication or other dot product. The activation function is commonly a RELU layer, and is subsequently followed by additional convolutions such as pooling layers, fully connected layers and normalization layers, referred to as hidden layers because their inputs and outputs are masked by the activation function and final convolution.

$$G(X) = g_N(g_{N-1}(\ldots(g_1(X))))$$

When programming a CNN, the input is a tensor with shape (number of images) x (image height) x (image width) x (image depth). Then after passing through a convolutional layer, the image becomes abstracted to a feature map, with shape (number of images) x (feature map height) x (feature map width) x (feature map channels). A convolutional layer within a neural network should have the following attributes: (1) Convolutional kernels defined by a width and height (hyper-parameters). (2) The number of input channels and output channels (hyper-parameter). (3)

The depth of the Convolution filter (the input channels) must be equal to the number channels (depth) of the input feature map. Convolutional layers convolve the input and pass its result to the next layer. This is similar to the response of a neuron in the visual cortex to a specific stimulus. Each convolutional neuron processes data only for its receptive field. Although fully connected feedforward neural networks can be used to learn features as well as classify data, it is not practical to apply this architecture to images. A very high number of neurons would be necessary, even in a shallow (opposite of deep) architecture, due to the very large input sizes associated with images, where each pixel is a relevant variable. For instance, a fully connected layer for a (small) image of size 100 x 100 has 10,000 weights for each neuron in the second layer. The convolution operation brings a solution to this problem as it reduces the number of free parameters, allowing the network to be deeper with fewer parameters. For instance, regardless of image size, tiling regions of size 5 x 5, each with the same shared weights, requires only 25 learnable parameters. By using regularized weights over fewer parameters, the vanishing gradient and exploding gradient problems seen during backpropagation in traditional neural networks are avoided.

$$h_k(x,y) = \sum_{s=-m}^{m} \sum_{t=-}^{n} \sum_{v=-d}^{w} V_k(s,t,v) X(x-s, y-t, z-v)$$

Pooling

Convolutional networks may include local or global pooling layers to streamline the underlying computation. Pooling layers reduce the dimensions of the data by combining the outputs of neuron clusters at one layer into a single neuron in the next layer. Local pooling combines small clusters, typically 2 x 2. Global pooling acts on all the neurons of the convolutional layer. In addition, pooling may compute a max or an average. Max pooling uses the maximum value from each of a cluster of neurons at the prior layer. Average pooling uses the average value from each of a cluster of neurons at the prior layer.

Fully Connected

Fully connected layers connect every neuron in one layer to every neuron in another layer. It is in principle the same as the traditional multi-layer perceptron neural network (MLP). The flattened matrix goes through a fully connected layer to classify the images.

In activation layer, different activation functions that can be used:

(i) Sigmoid activation function is given by the equation:

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

It is nonlinear in nature; its combination will also be nonlinear in nature, which gives us the liberty to stack the layers together. Its range is from −2 to 2 on the x-axis and on y-axis it is fairly steep, which shows the sudden changes in the values of y with respect to small changes in the values of x. One of the advantages of this activation function is its output always remains within the range of (0,1).

ii) Tanh function is defined as follows:

$$f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$$

This is also known as the scaled sigmoid function:

$$\tanh(x) = 2sigmoid(2x) - 1$$

Its range is from −1 to 1. The gradient is stronger for the tanh than the sigmoid function.

(iii) Rectified linear unit (ReLU) is the most commonly used activation function, where g denotes pixelwise function, which is nonlinear in nature. That is, it gives the output x, if x is positive and it is 0 otherwise.

$$g(x) = max(0, x)$$

ReLU is nonlinear in nature and its combination is also nonlinear, meaning different layers can be stacked together. Its range is from 0 to infinity, meaning it can also blow up the activation. For the pooling layer, g reduces the size of the features while acting as a layer-wise down-sampling nonlinear function. A fully connected layer has a 1 × 1 convolutional kernel. Prediction layer has a softmax which predicts the probability belonging of Xj to different possible classes.

Convolutional Neural Networks as Function Approximators

We decided to use deep Q-learning instead of ordinary Q-learning because there are too many possible game states. The size of one frame is 216 by 160 pixels, and for each pixel there are 2563 choices of RGB values. Furthermore, a sliding window of k frames leads to 256216∗160∗3∗k possible states in total – this is too large for ordinary Q-learning because it will result in too many rows in our imaginary Q-table. Therefore, we decide to use a neural network to learn these Q values instead. In effect, this neural network ends up operating as a function approximator. The network architecture is currently described as follows:

- Preprocess frames: convert RGB pixels to grayscale and threshold to black or white
- Input layer: takes in the preprocessed frames. Size: [k, 160, 250, 1]
- Hidden convolutional layer 1: kernel size [8, 8, k, 32], strides [1, 4, 4, 1]
- Max pooling layer 1: kernel size [1, 2, 2, 1], strides [1, 2, 2, 1]
- Hidden convolutional layer 2: kernel size [4, 4, 32, 64], strides [1, 2, 2, 1]
- Max pooling layer 2: kernel size [1, 2, 2, 1], strides [1, 2, 2, 1]
- Hidden convolutional layer 3: kernel size [3, 3, 64, 64], strides [1, 1, 1, 1]
- Max pooling layer 3: kernel size [1, 2, 2, 1], strides [1, 2, 2, 1]
- Resize the max pooling outputs to a vector of size [768] and feed to one fully connected layer
- Feed the outputs to a rectified linear activation function
- Collect the final output as 7 Q-values, each corresponding to an action

Though we have 3 max-pooling layers involved in our network architecture, this is not what we had in mind at the beginning. Unlike CNN architectures typically used in computer vision tasks such as image classification, pooling layers in our architecture may not have been desirable for our purposes because we likely do not want to introduce translation invariance since the position of the game entities are important for estimating Q values. However, max-pooling serves as an adequate way to compress our large state space into a vector of size 768, and is the reason why we have been using them. One suggestion to replace these max-pooling layers is to make the strides larger in each hidden convolutional layer – but given that the current strides already have substantial size, we have been hesitant to increase it any further. However, we are not dismissing it as a bad idea and would like to give it a try if we were given an additional month or two to evaluate.

Experience Replay

In general, deep neural networks are difficult to train. In the presence of multiple local optima, gradient descent may end up at a bad local minimum which will lead to poor performance. Initializing the network weights and biases to random values helps a little but is likely not sufficient. We incorporate a technique called experience replay to encourage the algorithm to find better optima instead of getting stuck at some underperforming local optimum. To be more specific, as we run a game session during training, all experiences < s, a, r, s0 > are stored in replay memory. During training, we take random samples from the replay memory instead of always grabbing the most recent transition. By breaking the similarity of subsequent training examples, this trick is likely to prevent the network from diving into some local minimum and will do so in an efficient manner. By using experience replay, the behavior distribution is averaged over many of its previous states, smoothing out learning and avoiding divergence in the parameters.

In our implementation, we simply perform a uniform sample from the bank of observed states (the replay memory) to construct a minibatch of size 100 with which we train on each iteration. Storing all past experiences is impossible due to the humongous state space, so we simply retain the most recent 20,000 observations in the replay memory and sample from that.

A. Why Doom?

Creating yet another 3D first-person perspective environment from scratch solely for research purposes would be somewhat wasteful. Due to the popularity of the first-person shooter genre, we have decided to use an existing game engine as the base for our environment. We concluded that it has to meet the following requirements:

J. based on popular open-source 3D FPS game (ability to modify the code and the publication freedom),
K. lightweight (portability and the ability to run multiple instances on a single machine),
L. fast (the game engine should not be the learning bottleneck),
M. total control over the game's processing (so that the game can wait for the bot decisions or the agent can learn by observing a human playing),
N. customizable resolution and rendering parameters,
O. multiplayer games capabilities (agent vs. agent and agent vs. human),
P. easy-to-use tools to create custom scenarios,
Q. ability to bind different programming languages (preferably written in C++),
R. multi-platform.

In order to make the decision according to the above-listed criteria, we have analyzed seven recognizable FPS games: Quake III Arena, Doom 3, Half-Life 2, Unreal Tournament 2004, Unreal Tournament, and Cube. Their comparison is shown in Table I. Some of the features listed in the table are objective (e.g., 'scripting') and others are subjective ("code complexity"). Brand recognition was estimated as the number (in millions) of Google results (as of 26.04.2016) for phrases "game ", where was 'doom', 'quake', 'half-life', 'unreal tournament' or 'cube'. The game was considered as low-resolution capable if it was possible to set the resolution to values smaller than 640 × 480. Some of the games had to be rejected right away in spite of high general appeal. Unreal Tournament 2004 engine is only accessible by the Software Development Kit and it lacks support for controlling the speed of execution and direct screen buffer access. The game has not been prepared to be heavily modified.

Similar problems are shared by Half-Life 2 despite the fact that the Source engine is widely known for modding capabilities. It also lacks direct multiplayer support. Although the Source engine itself

offers multiplayer support, it involves client-server architecture, which makes synchronization and direct interaction with the engine problematic (network communication).
The client-server architecture was also one of the reasons for the rejection of Quake III: Arena. Quake III also does not offer any scripting capabilities, which are essential to make a research environment versatile. The rejection of Quake was a hard decision as it is a highly regarded and playable game even nowadays but this could not outweigh the lack of scripting support.
The latter problem does not concern Doom 3 but its high disk requirements were considered as a drawback. Doom 3 had to be ignored also because of its complexity, Windows-only tools, and OS-dependent rendering mechanisms. Although its source code has been released, its community is dispersed. As a result, there are several rarely updated versions of its sources.
The community activity is also a problem in the case of Cube as its last update was in August 2005. Nonetheless, the low complexity of its code and the highly intuitive map editor would make it a great choice if the engine was more popular.
Unreal Tournament, however popular, is not as recognizable as Doom or Quake but it has been a primary research platform for FPS games. It also has great capabilities. Despite its active community and the availability of the source code, it was rejected due to its high system requirements.
Doom met most of the requirements and allowed to implement features that would be barely achievable in other games, e.g., off-screen rendering and custom rewards. The game is highly recognizable and runs on the three major operating systems. It was also designed to work in 320 × 240 resolution and despite the fact that modern implementations allow bigger resolutions, it still utilizes low-resolution textures. Moreover, its source code is easy-to-understand.
The unique feature of Doom is its software renderer. Because of that, it could be run without the desktop environment (e.g., remotely in a terminal), and accessing the screen buffer does not require transferring it from the graphics card. Technically, ViZDoom is based on the modernized, opensource version of Doom's original engine — ZDoom, which is still actively supported and developed.

B. Application Programming Interface (API)

ViZDoom API is flexible and easy-to-use. It was designed with reinforcement and apprenticeship learning in mind, and therefore, it provides full control over the underlying Doom process. In particular, it allows retrieving the game's screen buffer and make actions that correspond to keyboard buttons (or their combinations) and mouse actions. Some game state variables such as the player's health or ammunition are available directly. ViZDoom's API was written in C++. The API offers a myriad of configuration options such as control modes and rendering options. In addition to the C++ support, bindings for Python and Java have been provided.

C. Features

ViZDoom provides features that can be exploited in different kinds of AI experiments. The main features include different control modes, custom scenarios, access to the depth buffer, and off-screen rendering eliminating the need of using a graphical interface.

i) Control modes: ViZDoom implements four control modes:

i) synchronous player,
ii) synchronous spectator,
iii) asynchronous player, and
iv) asynchronous spectator.

In asynchronous modes, the game runs at constant 35 frames per second and if the agent reacts too slowly, it can miss some frames. Conversely, if it makes a decision too quickly, it is blocked until the next frame arrives from the engine. Thus, for reinforcement learning research, more useful are the synchronous modes, in which the game engine waits for the decision-maker. This way, the learning system can learn at its pace, and it is not limited by any temporal constraints. Importantly, for experimental reproducibility and debugging purposes, the synchronous modes run deterministically. In the player modes, it is the agent who makes actions during the game. In contrast, in the spectator modes, a human player is in control, and the agent only observes the player's actions.

In addition, ViZDoom provides an asynchronous multiplayer mode, which allows games involving up to eight players (human or bots) over a network

ii) Scenarios: One of the most important features of ViZDoom is the ability to run custom scenarios. This includes creating appropriate maps, programming the environment mechanics ("when and how things happen"), defining terminal conditions (e.g., "killing a certain monster", "getting to a certain place", "died"), and rewards (e.g., for "killing a monster", "getting hurt", "picking up an object"). This mechanism opens endless experimentation possibilities. In particular, it allows creating a scenario of a difficulty that is on par with the capabilities of the assessed learning algorithms.

The creation of scenarios is possible thanks to easy-to-use software tools developed by the Doom community. The two recommended free tools include Doom Builder 2 and SLADE 3. Both are visual editors, which allow defining custom maps and coding the game mechanics in Action Code Script. They also enable to conveniently test a scenario without leaving the editor. ViZDoom comes with a few predefined scenarios.

iii) Depth Buffer Access: ViZDoom provides access to the renderer's depth buffer, which may help an agent to understand the received visual information. This feature gives an opportunity to test whether the learning algorithms can autonomously learn the whereabouts of the objects in the environment. The depth information can also be used to simulate the distance sensors common in mobile robots.

iv) Off-Screen Rendering and Frame Skipping: To facilitate computationally heavy machine learning experiments, we equipped ViZDoom with off-screen rendering and frame skipping features. Off-screen rendering lessens the performance burden of actually showing the game on the screen and makes it possible to run the experiments on the servers (no graphical interface needed). Frame skipping, on the other hand, allows omitting rendering selected frames at all. Intuitively, an effective bot does not have to see every single frame.

D. ViZDoom's Performance

The main factors affecting ViZDoom performance are the number of the actors (like items and bots), the rendering resolution, and computing the depth buffer. We can see how the number of frames per second depends on these factors. The tests have been made in the synchronous player mode on Linux running on Intel Core i7-4790k. ViZDoom uses only a single CPU core. The performance test shows that ViZDoom can render nearly 7000 low-resolution frames per second. The rendering resolution proves to be the most important factor influencing the processing speed. In the case of low resolutions, the time needed to render one frame is negligible compared to the backpropagation time of any reasonably complex neural network.

Functioning:

A. Basic Experiment

The primary purpose of the experiment was to show that reinforcement learning from the visual input is feasible in ViZDoom. Additionally, the experiment investigates how the number of skipped frames influences the learning process.

i) **Scenario**: This simple scenario takes place in a rectangular chamber. An agent is spawned in the center of the room's longer wall. A stationary monster is spawned at a random position along the opposite wall. The agent can strafe left and right, or shoot. A single hit is enough to kill the monster. The episode ends when the monster is eliminated or after 300 frames, whatever comes first. The agent scores 101 points for killing the monster, −5 for a missing shot, and, additionally, −1 for each action. The scores motivate the learning agent to eliminate the monster as quickly as possible, preferably with a single shot.

ii) **Deep Q-Learning**: The learning procedure is similar to the Deep Q-Learning introduced for Atari 2600. The problem is modeled as a Markov Decision Process and Q-learning is used to learn the policy. The action is selected by a greedy policy with linear decay. The Q-function is approximated with a convolutional neural network, which is trained with Stochastic Gradient Descent. We also used experience replay but no target network freezing.

iii) **Experimental Setup**:

a) Neural Network Architecture: The network used in the experiment consists of two convolutional layers with 32 square filters, 7 and 4 pixels wide, respectively. Each convolution layer is followed by a max-pooling layer with max-pooling of size 2 and rectified linear units for activation. Next, there is a fully-connected layer with 800 leaky rectified linear units and an output layer with 8 linear units corresponding to the 8 combinations of the 3 available actions (left, right, and shot).

b) Game Settings: A state was represented by the most recent frame, which was a 60 × 45 3-channel RGB image. The number of skipped frames is controlled by the skipcount parameter. We experimented with skipcounts of 0-7, 10, 15, 20, 25, 30, 35, and 40. It is important to note that the agent repeats the last decision on the skipped frames.

c) Learning Settings: We arbitrarily set the discount factor $\gamma = 0.99$, learning rate $\alpha = 0.01$, replay memory capacity to 10 000 elements and mini-batch size to 40. The initial = 1.0 starts to decay after 100 000 learning steps, finishing the decay at = 0.1 at 200 000 learning steps.

Every agent learned for 600 000 steps, each one consisting of performing an action, observing a transition, and updating the network. To monitor the learning progress, 1000 testing episodes were played after each 5000 learning steps. Final controllers were

evaluated on 10 000 episodes. The experiment was performed on Intel Core i7-4790k 4GHz with GeForce GTX 970, which handled the neural network.

iv) **Results**:
It demonstrates that although all the agents improve over time, the skips influence the learning speed, its smoothness, as well as the final performance. When the agent does not skip any frames, the learning is the slowest. Generally, the larger the skipcount, the faster and smoother the learning is. We have also observed that the agents learning with higher skipcounts were less prone to irrational behaviors like staying idle or going the direction opposite to the monster, which results in lower variance on the plots. On the other hand, too large skipcounts make the agent 'clumsy' due to the lack of fine-grained control, which results in suboptimal final scores.
We have also checked how robust to skipcounts the agents are. For this purpose, we evaluated them using skipcounts different from the ones they had been trained with. Most of the agents performed worse than with their "native" skipcounts. The least robust were the agents trained with skipcounts less than 4. Larger skipcounts resulted in more robust agents. Interestingly, for skipcounts greater than or equal to 30, the agents score better on skipcounts lower than the native ones. Our best agent that was trained with skipcount 4 was also the best when executed with skipcount 0.
It is also worth showing that increasing the skipcount influences the total learning time only slightly. The learning takes longer primarily due to the higher total overhead associated with episode restarts since higher skipcounts result in a greater number of episodes.
To sum up, the skipcounts in the range of 4-10 provide the best balance between the learning speed and the final performance. The results also indicate that it would be profitable to start learning with high skipcounts to exploit the steepest learning curve and gradually decrease it to fine-tune the performance.

B. Medikit Collecting Experiment
The previous experiment was conducted on a simple scenario that was closer to a 2D arcade game rather than a true 3D virtual world. That is why we decided to test if similar deep reinforcement learning methods would work in a more involved scenario requiring substantial spatial reasoning.

i) **Scenario**: In this scenario, the agent is spawned in a random spot of a maze with an acid surface, which slowly, but constantly, takes away the agent's life. To survive, the agent needs to collect medikits and avoid blue vials with poison. Items of both types appear in random places during the episode. The agent is allowed to move (forward/backward), and turn (left/right). It scores 1 point for each tick, and it is punished by −100 points for dying. Thus, it is motivated to survive as long as possible. To facilitate learning, we also introduced shaping rewards of 100 and −100 points for collecting a medikit and a vial, respectively. The shaping rewards do not count to the final score but are used during the agent's training helping it to 'understand' its goal. Each episode ends after 2100 ticks (1 minute in real-time) or when the agent dies so 2100 is the maximum achievable score. Being idle results in scoring 284 points.

ii) **Experimental Setup:** The learning procedure was the same as described in A2 with the difference that for updating the weights RMSProp this time.

a) Neural Network Architecture: The employed network is similar to the one used in the previous experiment. The differences are as follows. It involves three convolutional layers with 32 square filters 7, 5, and 3 pixels wide, respectively. The fully-connected layer uses 1024 leaky rectified linear units and the output layer 16 linear units corresponding to each combination of the 4 available actions.
b) Game Settings: The game's state was represented by a 120 × 45 3-channel RGB image, health points, and the current tick number (within the episode). Additionally, a kind of memory was implemented by making the agent use 4 last states as the neural network's input. The nonvisual inputs (health, ammo) were fed directly to the first fully-connected layer. A Skipcount of 10 was used.
c) Learning Settings: We set the discount factor γ = 1, learning rate α = 0.00001, replay memory capacity to 10 000 elements and mini-batch size to 64. The initial = 1.0 started to decay after 4 000 learning steps, finishing the decay at = 0.1 at 104 000 episodes. The agent was set to learn for 1000 000 steps. To monitor the learning progress, 200 testing episodes were played after each 5000 learning steps. The whole learning process, including the testing episodes, lasted 29 hours.

iii) **Results**: The learning dynamics are shown in. It can be observed that the agents fairly quickly learn to get the perfect score from time to time. Its average score, however, improves slowly reaching 1300 at the end of the learning. Learning dynamics for health gathering scenario. trend might, however, suggest that some improvement is still possible given more training time. The plots suggest that even at the end of learning, the agent for some initial states fails to live more than a random player.
It must, however, be noted that the scenario is not easy and even from a human player, it requires a lot of focus. It is so because the medikits are not abundant enough to allow the bots to waste much time.
Watching the agent play revealed that it had developed a policy consistent with our expectations. It navigates towards medikits, actively, although not very deftly, avoids the poison vials, and does not push against walls and corners. It also backpedals after reaching a dead-end or a poison vial. However, it very often hesitates about choosing a direction, which results in turning left and right alternately on the spot. This quirky behavior is the most probable, direct cause of not fully satisfactory performance.
Interestingly, the learning dynamics consists of three sudden but ephemeral drops in the average and best score. The reason for such dynamics is unknown and it requires further research.

5.0 Platform

Language: Python

Library:

3. **Numpy:** adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.
4. **PyTorch:** open-source machine learning library based on the Torch library, used for applications such as computer vision and natural language processing

Environment: Open AI Gym is a toolkit for developing and comparing reinforcement learning algorithms. It supports teaching agents everything from walking to playing games like Doom.

6.0 Sample Coding

```
# -*- coding: utf-8 -*-
"""Deep Convolutional Q-Learning

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1WsKC2PgRtjEWGy1u0wGCGCh-ZCFmEISA

# Deep Convolutional Q-Learning

### Installing system dependencies for VizDoom
"""

!sudo apt-get update
!sudo apt-get install build-essential zlib1g-dev libsdl2-dev libjpeg-dev nasm tar libbz2-dev libgtk2.0-dev cmake git libfluidsynth-dev libgme-dev libopenal-dev timidity libwildmidi-dev unzip
!sudo apt-get install libboost-all-dev
!apt-get install liblua5.1-dev
!sudo apt-get install cmake libboost-all-dev libgtk2.0-dev libsdl2-dev python-numpy git
!git clone https://github.com/shakenes/vizdoomgym.git
!python3 -m pip install -e vizdoomgym/
!pip install pillow
!pip install scipy==1.1.0

"""### **IMPORTANT NOTE: After installing all dependencies, restart your runtime**

## ------------------------ IMAGE PREPROCESSING (image_preprocessing.py) ------------------------

### Importing the libraries
"""

import numpy as np
from scipy.misc import imresize
from gym.core import ObservationWrapper
from gym.spaces.box import Box

"""### Preprocessing the images"""

class PreprocessImage(ObservationWrapper):

    def __init__(self, env, height = 64, width = 64, grayscale = True, crop = lambda img: img):
        super(PreprocessImage, self).__init__(env)
        self.img_size = (height, width)
        self.grayscale = grayscale
        self.crop = crop
        n_colors = 1 if self.grayscale else 3
        self.observation_space = Box(0.0, 1.0, [n_colors, height, width])

    def observation(self, img):
        img = self.crop(img)
        img = imresize(img, self.img_size)
        if self.grayscale:
            img = img.mean(-1, keepdims = True)
        img = np.transpose(img, (2, 0, 1))
        img = img.astype('float32') / 255.
        return img

"""## ------------------------ EXPERIENCE REPLAY (experience_replay.py) ------------------------

### Importing the libraries
"""
```

```
import numpy as np
from collections import namedtuple, deque

"""### Defining One Step"""

Step = namedtuple('Step', ['state', 'action', 'reward', 'done'])

"""### Making the AI progress on several (n_step) steps"""

class NStepProgress:

    def __init__(self, env, ai, n_step):
        self.ai = ai
        self.rewards = []
        self.env = env
        self.n_step = n_step

    def __iter__(self):
        state = self.env.reset()
        history = deque()
        reward = 0.0
        while True:
            action = self.ai(np.array([state]))[0][0]
            next_state, r, is_done, _ = self.env.step(action)
            reward += r
            history.append(Step(state = state, action = action, reward = r, done = is_done))
            while len(history) > self.n_step + 1:
                history.popleft()
            if len(history) == self.n_step + 1:
                yield tuple(history)
            state = next_state
            if is_done:
                if len(history) > self.n_step + 1:
                    history.popleft()
                while len(history) >= 1:
                    yield tuple(history)
                    history.popleft()
                self.rewards.append(reward)
                reward = 0.0
                state = self.env.reset()
                history.clear()

    def rewards_steps(self):
        rewards_steps = self.rewards
        self.rewards = []
        return rewards_steps

"""### Implementing Experience Replay"""

class ReplayMemory:

    def __init__(self, n_steps, capacity = 10000):
        self.capacity = capacity
        self.n_steps = n_steps
        self.n_steps_iter = iter(n_steps)
        self.buffer = deque()

    def sample_batch(self, batch_size): # creates an iterator that returns random batches
        ofs = 0
        vals = list(self.buffer)
        np.random.shuffle(vals)
        while (ofs+1)*batch_size <= len(self.buffer):
```

```
            yield vals[ofs*batch_size:(ofs+1)*batch_size]
            ofs += 1

    def run_steps(self, samples):
        while samples > 0:
            entry = next(self.n_steps_iter) # 10 consecutive steps
            self.buffer.append(entry) # we put 200 for the current episode
            samples -= 1
        while len(self.buffer) > self.capacity: # we accumulate no more than the capacity (10000)
            self.buffer.popleft()

"""## ---------- AI FOR DOOM (ai.py) ----------

### Importing the libraries
"""

import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.autograd import Variable

"""### Importing the packages for OpenAI and Doom"""

import gym
import vizdoomgym
from gym import wrappers

"""## Part 1 - Building the AI

### Making the Brain
"""

class CNN(nn.Module):

    def __init__(self, number_actions):
        super(CNN, self).__init__()
        self.convolution1 = nn.Conv2d(in_channels = 1, out_channels = 32, kernel_size = 5)
        self.convolution2 = nn.Conv2d(in_channels = 32, out_channels = 32, kernel_size = 3)
        self.convolution3 = nn.Conv2d(in_channels = 32, out_channels = 64, kernel_size = 2)
        self.fc1 = nn.Linear(in_features = self.count_neurons((1, 256, 256)), out_features = 40)
        self.fc2 = nn.Linear(in_features = 40, out_features = number_actions)

    def count_neurons(self, image_dim):
        x = Variable(torch.rand(1, *image_dim))
        x = F.relu(F.max_pool2d(self.convolution1(x), 3, 2))
        x = F.relu(F.max_pool2d(self.convolution2(x), 3, 2))
        x = F.relu(F.max_pool2d(self.convolution3(x), 3, 2))
        return x.data.view(1, -1).size(1)
```

7.0 Test Cases Input

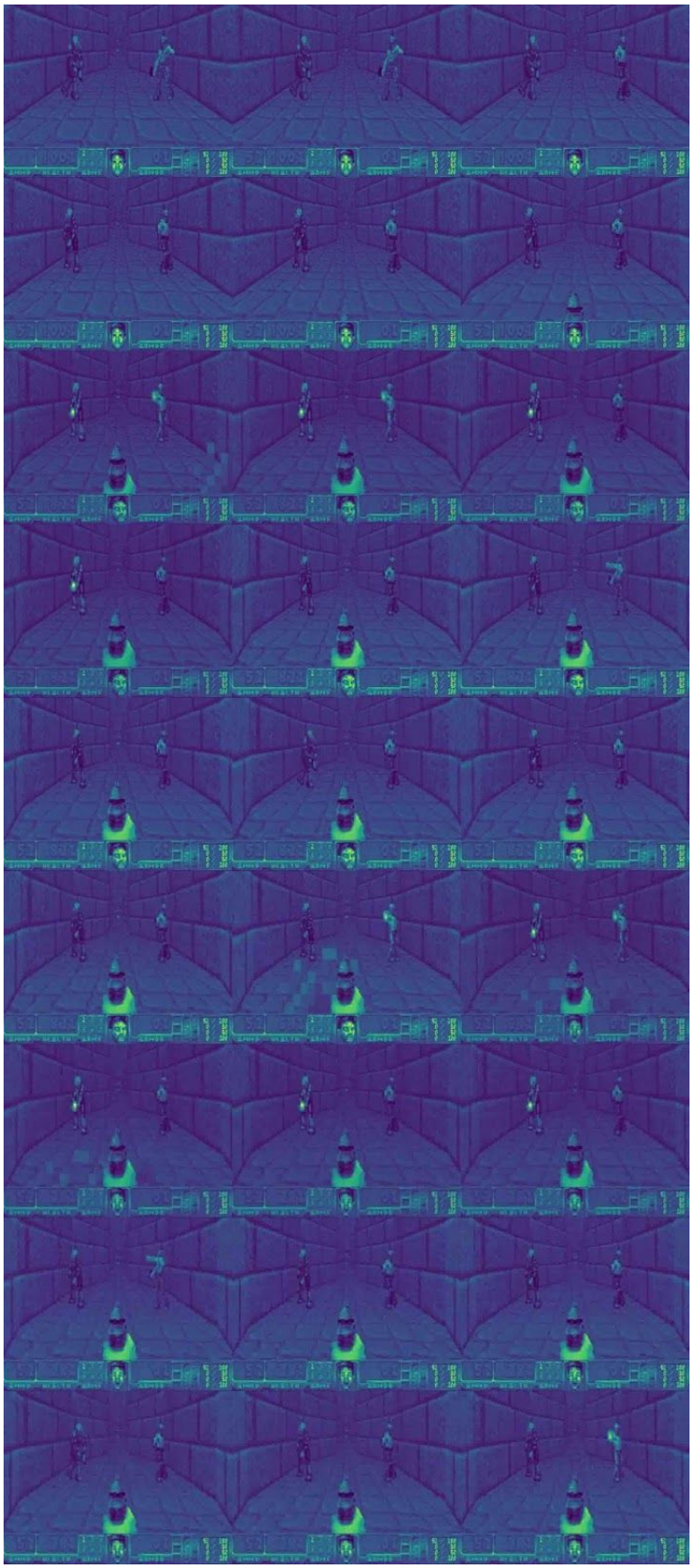
Input is always a 320x320 pixel image of the current display of the video game screen. The agent have the option to the following moves at a given point in time

8. MOVE_LEFT
9. MOVE_RIGHT
10. ATTACK
11. MOVE_FORWARD
12. MOVE_BACKWARD
13. TURN_LEFT
14. TURN_RIGHT

The reinforcement learning reward being as followed

4. +dX for getting closer to the end of the corridor.
5. -dX for getting further from the end of the corridor.
6. -100 death penalty

7.1 Test Case 1 Output (Agent with no Training)

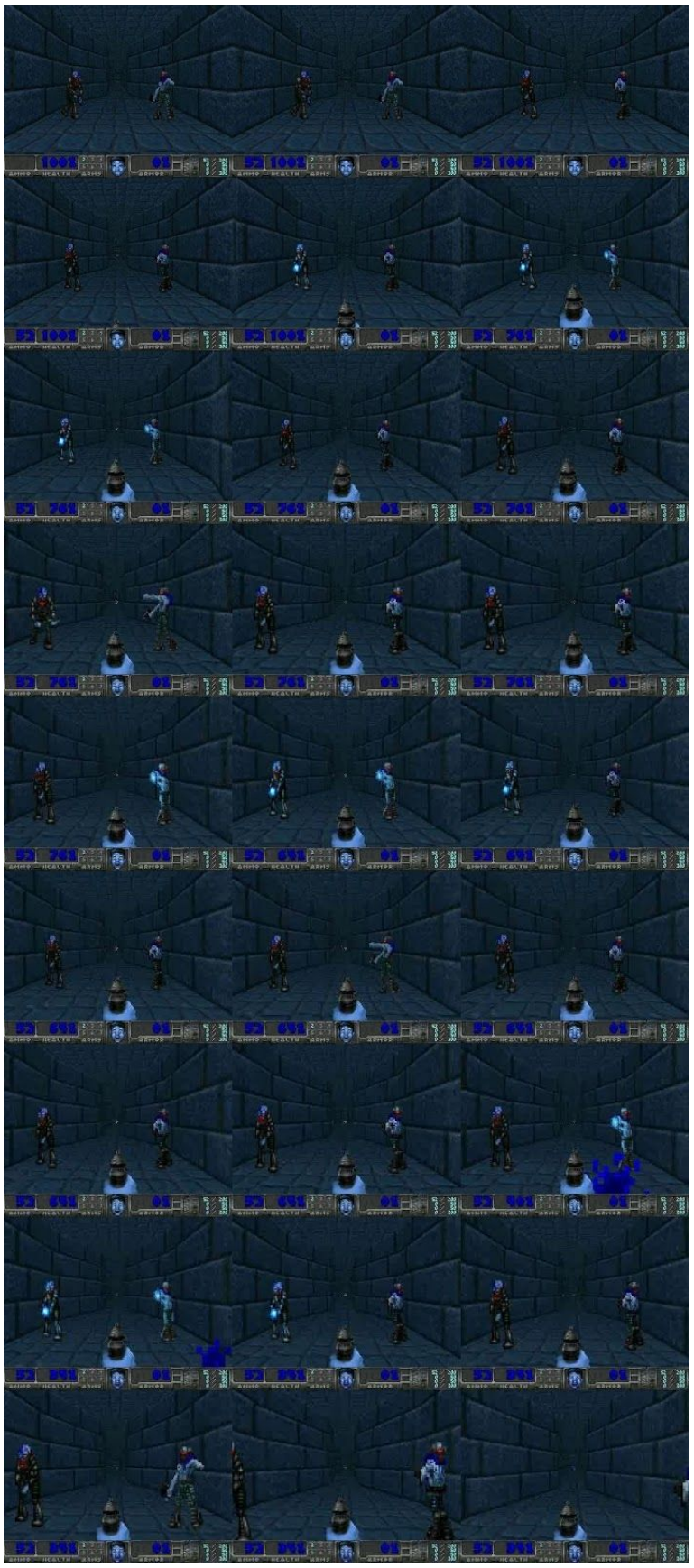


The output for test case 1 clearly shows how incapable the agent is at playing the game without learning. Although the agent learned to take out his gun he doesn't move and receives all gunshot damage from the enemy across the corridor. The agent does occasional ATTACK.

Drive Link for the output:

https://drive.google.com/file/d/15OHlPFy5YZMuKvS4C_DsSlgpMjqPgvwC/view?usp=sharing

7.2 Test Case 2 Output (Agent with 6 hours of Training)



The output for test case 2 shows a clear improvement is the agent playing the game. The agent is shown doing MOVE_LEFT and MOVE_RIGHT movement to dodge the enemy bullets and uses MOVE_FORWARD movement to move past the enemy in the corridor.

Drive Link for the output:

https://drive.google.com/file/d/1MEmfqmZwM8I2WVLk5v2u_4LKHzLe5twO/view?usp=sharing

8.0 Discussion

As a preliminary evaluation, we first ran our algorithm (without experience replay) for 5 trials, each with a cut off at the end of 36 hours, using the number of consecutive frames k=3. The final scores for each trial are plotted in the top half of. From the results, we can see that some trials performed pretty badly, and are in fact no better than the baseline Q-learning algorithm. However, other trials performed significantly better than the baseline. We believe these differences in performance among the trials can be explained by the gradient descent approach that we used for training our deep neural network, which is characteristically vulnerable to getting stuck at some under-performing local optimum. We arrived at this explanation because each trial is initialized with random weights and biases, and these trials produce wildly different final scores, so most likely each of them ended up in a different local optimum.

We then tweaked our model parameters (aka hyperparameters) such as the exploitation-exploration parameter, k the number of consecutive frames in consideration, and η the learning rate, in a manner similar to grid search. Since in our situation we do not have a dataset for which to divide into training, validation, and testing sets for the reason that out training comes from operating a dynamic game, we simply repeated training on various values of, k and η and find the combination that gives the best scores. This is straightforward compared to the usual hyperparameter optimization process in machine learning.

We found that setting η to 0.01, k to 4, and to a dynamic one that linearly decreases from 0.8 to 0.05 annealed over 50,000 timesteps gives the best results overall, but we notice that in general varying the hyperparameters does not significantly influence the game agent's performance, so we did not devote too much time trying out different combinations.

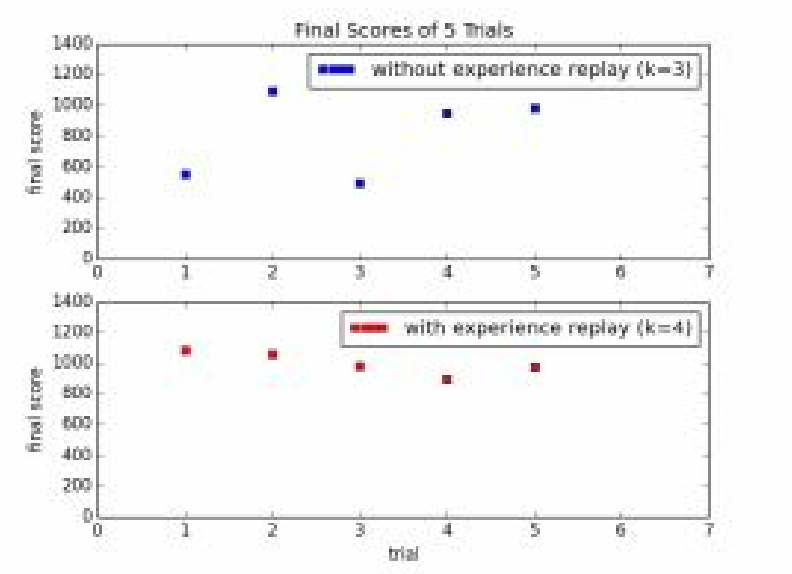


Figure 1: Comparison: performance with and without experience replay

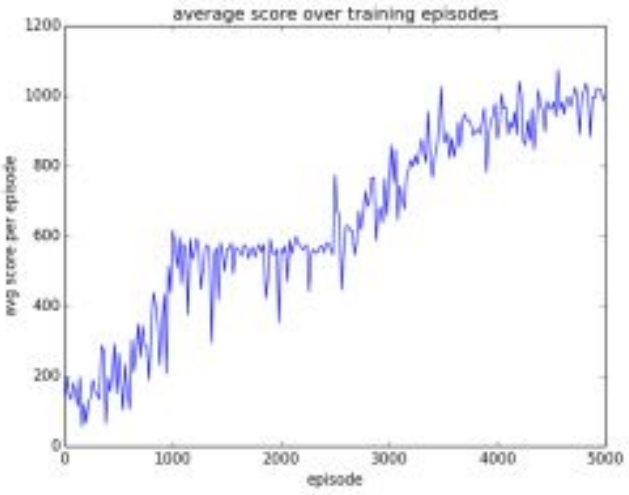


Figure 2: Average scores over training episodes

We obtained our final results by running the agent using the model weights computed at the end of training and noting down the final score repeated for a total of 5 trials, and instead of doing the cutoff based on time, we now terminate training after performing 50,000 iterations. We switched from time-based cutoff to iteration-based cutoff because the time it takes to train a model is mainly a function of the hardware used to train it. Furthermore, reporting the scores obtained from a certain number of episodes makes it more robust against the noise/stochasticity of the OpenAI gym environment. This time we also incorporated experience replay into the training procedure. The scores we obtained are plotted on the bottom half of. Comparing these results with those above, we can see that experience replay produced results that are more consistent and stable. This is because this technique was able to effectively prevent the algorithm from getting stuck at some bad local optimum, and thus help achieve (slightly) higher and more consistent results.
To aid with our analysis we plot the average score per 20 consecutive training episodes over one complete trial in. We can see that in the beginning, the score rises rapidly to around 600 because the game mode stays the same up to this point and so far it is quite easy to get there. After which it appears to get stuck at around 600 for over 1500 episodes, and this is because starting from this point, the game jumps in difficulty – new enemy entities known as ”crawlers” appear on the left and right sides of the agent, in addition to the enemy ships already hovering above. As if it encountered a roadblock, the agent was unable to make much progress past 600 for quite some time, but it did eventually learn to overcome this obstacle. So we conclude that even with this change of difficulty, our agent simply needed some time to adjust and continue to learn. From 600 onwards the agent improves at a rate slower than it did from the start to 600 because the game is no longer as easy as it was in the beginning. Finally, it saturated at around 1000 and this is roughly the final score it was able to achieve.

9.0 Conclusion and Future Scope

ViZDoom is a Doom-based platform for research in vision-based reinforcement learning. It is easy-to-use, highly flexible, multi-platform, lightweight, and efficient. In contrast to the other popular visual learning environments such as Atari 2600, ViZDoom provides a 3D, semi-realistic, first-person perspective virtual world. ViZDoom's API gives the user full control of the environment. Multiple modes of operation facilitate experimentation with different learning paradigms such as reinforcement learning, apprenticeship learning, learning by demonstration, and, even the 'ordinary', supervised learning. The strength and versatility of the environment lie in is customizability via the mechanism of scenarios, which can be conveniently programmed with open-source tools.

We also demonstrated that visual reinforcement learning is possible in the 3D virtual environment of ViZDoom by performing experiments with deep Q-learning on two scenarios. The results of the simple move-and-shoot scenario, indicate that the speed of the learning system highly depends on the number of frames the agent is allowed to skip during the learning. We have found out that it is profitable to skip from 4 to 10 frames. We used this knowledge in the second, more involved, scenario, in which the agent had to navigate through a hostile maze and collect some items and avoid the others. Although the agent was not able to find a perfect strategy, it learned to navigate the maze surprisingly well exhibiting evidence of a human-like behavior

ViZDoom has recently reached a stable 1.0.1 version and has the potential to be extended in many interesting directions. First, we would like to implement an asynchronous multiplayer mode, which would be convenient for self-learning in multiplayer settings. Second, bots are now deaf thus, we plan to allow bots to access the sound buffer. Lastly, interesting supervised learning experiments (e.g., segmentation) could be conducted if ViZDoom automatically labeled objects in the scene.

In this project, we have implemented a game-playing agent for Atari Assault using deep Q-learning. We first implemented ordinary Q-learning to obtain a baseline of score 670.7, then we implemented deep Q-learning by constructing a convolutional neural network using Tensorflow. We obtained promising results after experimenting with and without experience replay. For us, experience replay worked well in helping the agent avoid getting stuck at some bad local optimum. Some of our improvements also came about by extending the training time and tweaking hyperparameters. Our deep Q-learning agent managed to significantly outperform the baseline: the average score it obtained was 980, while the ordinary Q-learning baseline has an average score of 670.7.

Although our agent does significantly better than the baseline, it still does not come close to the oracle. We believe that there are still many places we can try to improve, such as revising the neural network architecture, adding customized feature extractors, and experimenting with more fully connected layers.

10.0 References

[1] David Abel, Alekh Agarwal, Fernando Diaz, Akshay Krishnamurthy, and Robert E. Schapire(2016). Exploratory gradient boosting for reinforcement learning in complex domains. *Clinical Orthopaedics and Related Research*, 16, 210-252.

[2] Minoru Asada, Shoichi Noda, Sukoya Tawaratsumida, and Koh Hosoda(1996). Purposive behavior acquisition for a real robot by vision-based reinforcement learning. *Recent Advances in Robot Learning*. 2, 163–187.

[3] Minoru Asada, Eiji Uchibe, Shoichi Noda, Sukoya Tawaratsumida, and Koh Hosoda(1994). A vision-based reinforcement learning for coordination of soccer playing behaviors. *AI and A-life and Entertainment*. 1, 16–21.

[4] Nicholas Cole, Sushil J Louis, and Chris Miles(2004). Using a genetic algorithm to tune first-person shooter bots. *Evolutionary Computation*. 1, 139–145..

[5] Giuseppe Cuccu, Matthew Luciw, Jurgen Schmidhuber, and Faustino ¨ Gomez(2011). Intrinsically motivated neuroevolution for vision-based reinforcement learning. *Development and Learning, IEEE International Journal*. 2, 1–7.

[6] Mark Dawes and Richard Hall(2005). Towards using first-person shooter computer games as an artificial intelligence testbed. *KnowledgeBased Intelligent Information and Engineering Systems*. 1, 276–282.

[7] Abdennour El Rhalibi and Madjid Merabti(2008). A hybrid fuzzy ANN system for agent adaptation in a first person shooter. *International Journal of Computer Games Technology.* 2, 12-21.

[8] A I Esparcia-Alcazar, A Martinez-Garcia, A Mora, J J Merelo, and P Garcia-Sanchez(2010). Controlling bots in a First Person Shooter game using genetic algorithms. *Evolutionary Computation*. 1, 1–8.

[9] Chris Gaskett, Luke Fletcher, and Alexander Zelinsky.(2000) Reinforcement learning for a vision based mobile robot. *Intelligent Robots and Systems.* 1, 403–409.

[10] Ibarz, B., Leike, J., Pohlen, T., Irving, G., Legg, S. and Amodei, D. (2018). Reward learning from human preferences and demonstrations in Atari. *Advances in neural information processing systems*. 1, 8011-8023.

[11] Kapturowski, S., Ostrovski, G., Quan, J., Munos, R. and Dabney, W. (2018), September. Recurrent experience replay in distributed reinforcement learning. *Journal on learning representations*. 1, 1-7.

[12] F G Glavin and M G Madden(2015). Adaptive Shooting for Bots in First Person Shooter Games Using Reinforcement Learning. *Computational Intelligence and AI Games, IEEE Transactions*. 7(2):180–192.

[13] Xavier Glorot, Antoine Bordes, and Yoshua Bengio(2011). Deep sparse rectifier neural networks. *Journal of Machine Learning Research - Workshop and Conference Proceedings.* 15, 315– 323.

[14] S Hladky and V Bulitko(2008). An evaluation of models for predicting opponent positions in first-person shooter video games. *Computational Intelligence and Games.* 1, 39–46.

[15] Igor V. Karpov, Jacob Schrum, and Risto Miikkulainen(2012). Believable Bot Navigation via Playback of Human Traces. *Springer Berlin Heidelberg.* 1, 151–170.

[16] Jan Koutn´ık, Jurgen Schmidhuber, and Faustino Gomez(2014). Evolving deep ¨ unsupervised convolutional networks for vision-based reinforcement learning. *Genetic and evolutionary computation.* 1, 541–548.

[17] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton(2012). Imagenet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems.* 25, 1097–1105..

[18] Sascha Lange and Martin Riedmiller(2010). Deep auto-encoder neural networks in reinforcement learning. *International Journal on Neural Networks.* 1, 1–8.

[19] Andrew L. Maas, Awni Y. Hannun, and Andrew Y. Ng(2013). Rectifier nonlinearities improve neural network acoustic models. *International Journal on Machine Learning.* 1, 2-7.

[20] M McPartland and M Gallagher(2011). Reinforcement Learning in First Person Shooter Games. Computational Intelligence and AI in Games, IEEE Transactions. 3(1):43–56.

[21] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis(2005). Human-level control through deep reinforcement learning. *Nature, 518(7540*). 2, 529–533,

[22] Greydanus, S., Koul, A., Dodge, J. and Fern, A. (2018). Visualizing and understanding atari agents. *Journal of Machine Learning*. 1, 1792-1801.

[23] Tony C Smith and Jonathan Miles(2014). Continuous and Reinforcement Learning Methods for First-Person Shooter Games. *Journal on Computing (JoC).* 1, 1.

[24] Patel, D., Hazan, H., Saunders, D.J., Siegelmann, H.T. and Kozma, R. (2019). Improved robustness of reinforcement learning policies upon conversion to spiking neuronal network platforms applied to Atari Breakout game. *Neural Networks*. *120*, 108-115.

[25] Chang Kee Tong, Ong Jia Hui, J Teo, and Chin Kim On(2011). The Evolution of Gamebots for 3D First Person Shooter (FPS). *BioInspired Computing: Theories and Applications. 1,* 21–26.

[26] David Trenholme and Shamus P Smith(2008). Computer game engines for developing first-person virtual environments. *Virtual reality.* 12(3), 181– 187.

[27] Mousavi, S.S., Schukat, M., Howley, E. and Mannion, P. (2017). Applying q (λ)-learning in deep reinforcement learning to play atari games. *Adaptive Learning Agents*. 1, 2-9.

[28] C. J. C. H. Watkins and P. Dayan(1992). Q-learning. *Machine Learning.* 8(3), 279–292.

[29] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Wierstra, D., & Riedmiller, M. (2016). Playing Atari with Deep Reinforcement Learning. *Evolutionary Computing.* 1, 23-29.

[30] Guo, X., Singh, S., Lee, H., Lewis, R., & Wang, X. (n.d.)(2016). Deep Learning for Real-Time Atari Game Play Using Offline Monte-Carlo Tree Search Planning. *IEEE Transactions*. 2, 34-41.

[31] Matiisen, B. T. (n.d.)(2016). Demystifying Deep Reinforcement Learning. *Advances in neural information processing systems. 1, 3540-3551.*

[32] Nachum, O., Gu, S.S., Lee, H. and Levine, S. (2018). Data-efficient hierarchical reinforcement learning. *Advances in Neural Information Processing Systems*. 1, 3303-3313.

[33] François-Lavet, V., Henderson, P., Islam, R., Bellemare, M.G. and Pineau, J. (2018). An introduction to deep reinforcement learning. Evolutionary Computing*. 1, 560-564*.

[34] Cobbe, K., Klimov, O., Hesse, C., Kim, T. and Schulman, J. (2019). Quantifying generalization in reinforcement learning. *International Journal on Machine Learning*. 1, 1282-1289.

[35] Lample, G. and Chaplot, D.S.(2016). Playing FPS games with deep reinforcement learning. *Advances in Neural Networks. 1, 521-530*.

[36] Nair, A.V., Pong, V., Dalal, M., Bahl, S., Lin, S. and Levine, S. (2018). Visual reinforcement learning with imagined goals. *Advances in Neural Information Processing Systems*. 1, 9191-9200.

[37] Li, Y.(2017). Deep reinforcement learning: An overview. *IEEE Journal for Innovative AI. 1, 274-280*.

[38] Racanière, S., Weber, T., Reichert, D., Buesing, L., Guez, A., Rezende, D.J., Badia, A.P., Vinyals, O., Heess, N., Li, Y. and Pascanu, R. (2017). Imagination-augmented agents for deep reinforcement learning. *Advances in neural information processing systems*. 1, 5690-5701.

[39] Kaiser, L., Babaeizadeh, M., Milos, P., Osinski, B., Campbell, R.H., Czechowski, K., Erhan, D., Finn, C., Kozakowski, P., Levine, S. and Mohiuddin, A. (2019). Model-based reinforcement learning for atari. *Evolutionary Computing. 1, 374-380*.

[40] Lanctot, M., Zambaldi, V., Gruslys, A., Lazaridou, A., Tuyls, K., Pérolat, J., Silver, D. and Graepel, T. (2017). A unified game-theoretic approach to multiagent reinforcement learning. *Advances in neural information processing systems. 1,* 4190-4203.

[41] Jaderberg, M., Czarnecki, W.M., Dunning, I., Marris, L., Lever, G., Castaneda, A.G., Beattie, C., Rabinowitz, N.C., Morcos, A.S., Ruderman, A. and Sonnerat, N. (2019). Human-level performance in 3D multiplayer games with population-based reinforcement learning. *Innovative Sciences*, *364*(6443), 859-865.

[42] Gupta, J.K., Egorov, M. and Kochenderfer, M. (2017). Cooperative multi-agent control using deep reinforcement learning. *Autonomous Agents and Multiagent Systems*. 1, 66-83.

~Thank You~