# DeepDoom: Visually Navigating 3D Environments Using Distilled Hierarchical Deep Q-Networks

**Rafael Zamora, William Steele, Joshua Hidayat & Lauren An**
Department of Computer Science, Hood College, Frederick, Maryland

***Abstract*—Deep Q-Networks (DQNs) heavily aided in guiding agents to associate differences between rewarding and penalizing actions on both 2D Atari 2600 games and 3D Doom games. To compensate more effectively for the dimensional complexity of 3D environments such as Doom and Minecraft, Hierarchical Deep Q-Networks (h-DQNs) allowed for environment modularization into separate skill-based models—our basis for designing more efficient agents. Building upon previous research, we aimed to implement h-DQNs within Doom to continuously add new knowledge via periodically-distilled h-DQNs in order to achieve multi-skilled functionalities. Our trained h-DQN model provided positive indication of the agent's ability to combine vastly differing skills, and saw both an enhancement in its skill-adaptive capabilities and training times when compared to a single DQN. Additionally, through Policy Distillation, we were able to transfer knowledge acquired from our h-DQN into a single DQN architecture, allowing for an overall reduction in computations and network parameters. This led us to challenge the plausibility of training a distilled h-DQN alongside separate skills to achieve increasingly-complex behaviors.**

***Index Terms*—Visual-based, 3D Environments, First-Person Shooter (FPS), Deep Reinforcement Learning (DRL), Deep Q-Networks (DQN), Double Deep Q-Networks (DDQN), Hierarchical Deep Q-Networks (h-DQN), Policy Distillation**

## I. INTRODUCTION

The design of Artificial Intelligence (AI) within video games is continually improving alongside improvements to video games themselves. However, while ally and enemy NPCs (non-playable characters) are typically designed to keep the human player engaged with a fair challenge, their behaviors can differ drastically from that of human players, namely due to their ability to look "under the hood" and access internal game variables. This holds an unfair advantage over their human counterparts, who may identify this behavior as "cheating." We believe that AIs may behave with greater similarity to human player behavior if they utilize the same available environmental information. While this does not represent the immediate future for game-playing AIs, this design may provide a more balanced and enjoyable experience for players while challenging the possibilities of a human-like behaving agents.

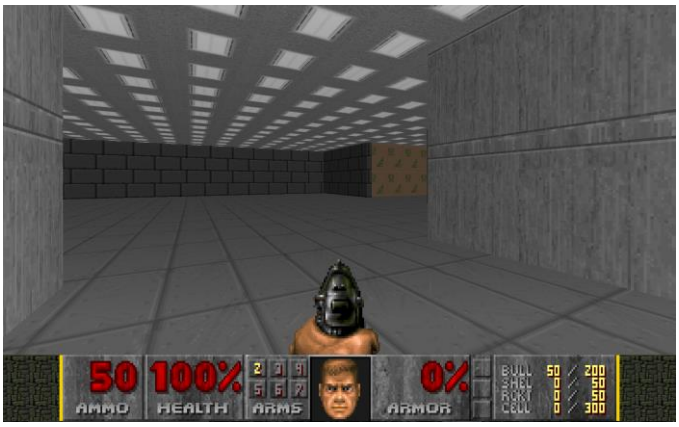


*Figure 1.1 - A sample Doom environment*

Google DeepMind's paper, *Playing Atari with Deep Reinforcement Learning* [1], showed the feasibility of game playing using only visual input. This was done by combining Deep Convolutional Neural Networks (CNNs) with Q-Learning, forming Deep Q-Networks (DQNs). Agents trained via DQNs were able to learn and play two-dimensional (2D) Atari 2600 games such as Pong, Breakout, and Space Invaders. In other words, DQNs guided the agent towards the best policy using positive or negative responses from the game's respective environment. Since then, there has been an interest in researching the applications of these same reinforcement learning techniques to train agents within three-dimensional (3D) environments such as Doom and Minecraft [2].

The Doom series [3] began in 1993 with the release of the first game, DOOM, and it has since sparked countless adaptations of 3D first-person shooter (FPS) games. These environments heightened the video game complexity with the incorporation of depth, providing another factor when designing agents. The relevancy of Doom to Artificial Intelligence can be contributed to the AI research platform ViZDoom [4] alongside the scenario customization software Doom Builder [5] and Slade, which are both compilers for the ACS (Action Code Script) scripting language used by ViZDoom. ViZDoom allows programmers to test reinforcement learning techniques within the environment of Doom. Its Visual Doom AI Competitions [6], as well as

previous research [7], proved the feasibility of teaching agents to sufficiently play Doom using DQNs. However, they mainly focused on the combat aspect of Doom, with only a minor focus on the navigation aspect (via item and health pickups).

For this project, we propose to create an agent capable of solving complex navigational problems using a hierarchical implementation of the Deep Q-Networks [8], expanding on ViZDoom's experimentations but heavily focusing on navigation. We will first utilize DQN models to separately train simple tasks and then integrate these as sub-models in a Hierarchical Deep Q-Network (h-DQN). In addition, we will investigate distilling trained h-DQNs into the simple DQN architecture for a more resource-efficient execution. Increasingly-complex behaviors can then be achieved by incrementally aggregating distilled h-DQNs with more skills into separate Hierarchical-DQNs, thus reflecting our goal of developing a more proficient Doom-playing Artificial Intelligence.

In this paper, we will first look at the related work of the previously-mentioned models, followed by a brief explanation of how we incorporated these models, as well as how we defined some training parameters. We will next detail training the networks on our custom scenarios of Doom and discuss the results of all our experiments, including the hierarchical and distilled experimentation.

## II. RELATED WORK

The inspiration for the use of Deep Reinforcement Learning (DRL) techniques in the context of video games came from Mnih et al. and their work with Atari 2600 games [1], where DQNs (a composition of CNNs and Q-Learning) were used to learn control policies from several games. This demonstrated the applicability of DRL techniques towards learning the various skills needed to play games in a 2D virtual environment. From there, the jump from a 2D to a 3D virtual environment was not inconsiderable, as both environments represented high-dimensional problem spaces.

Mirowski et al. [9] more recently attempted to visually navigate 3D maze-like environments by focusing on color and texture to pilot their control policy, an otherwise similar approach to Mnih et al. [1]. Although this showed the feasibility of training CNNs to navigate 3D environments using only visually-apparent data, it failed to address the inherent complexities of completing a level in a video game such as Doom: the potential need to use multiple task-specific skills in differing scenarios, like operating in-game objects such as switches and doors. In other words, while navigation is certainly possible, the difficulty of training a DQN-agent to play FPS games increases as the amount of skills required to complete levels increase.

Fortunately, ViZDoom [4] was created as an experimental platform for the development of visual-input-based agents within the context of a first-person shooter game—in this case, Doom I and II. Utilizing the ZDoom engine as its environment to train agents allowed for an extreme amount of customization; in regard to training scenarios, users could create their own levels with which to guide and train agents in, using the Doom Builder software. Doom Builder [5] not only provided object and texture-specific needs, but also allowed ACS scripting language to define reward systems needed for reinforcement learning. To demonstrate the functionality of their platform, ViZDoom created a simple agent using a single DQN model, again similar to that of Mnih et al. [1].

Lample and Chaplot [7] took this a step further by using the ViZDoom platform to construct an agent capable of Deathmatch scenarios. After realizing that a single DQN would be incapable of adequately completing this type of a task, Lample and Chaplot instead chose to decompose the Deathmatch scenario into the individual skills of shooting and navigation, each trained on a separate Deep Q-Network. Decomposing the global task into simpler, local skills demonstrated its multi-model adaptability to complex environments, but not without its flaws. The limitations of using just one model per a given moment—the shooting model for combat situations and the navigation model for all other situations—slightly hindered the realism behind learning different skills, or lacking the sense of fluidly-combined networks of knowledge.

There existed various extensions of Deep Q-Networks that also helped to resolve some of its other inadequacies, such as the Double Q-Learning algorithm created by van Hasselt et al. [10] to primarily prevent overestimation; it was also shown to be applicable to large-scale function approximation within Atari 2600 games. van Hasselt et al. [10] revealed that the prevalence of overestimation in DQN models negatively affected performance, and by extension, the scores achieved in these games. Furthermore, they showed that it was possible to implement the Double Q-Learning algorithm without having to change the architecture of the already-existent DQN, creating Double Deep Q-Networks (DDQN). This extension of the DQN architecture found better policies and improved learning in comparison to standard DQNs, representing the model with which we will be using to train individual scenarios.

Hierarchical Q-Learning was shown by Zhou, van Kampen, and Chu [11] to demonstrate task decomposition while also helping to accelerate learning and more efficiently transfer knowledge between tasks within the context of flight navigation. Their first level model handled the global task of navigating a maze to reach some target area, while their second level sub-model focused on optimizing the path taken as well as performing various obstacle avoidance. With task decomposition in mind, we perceived this concept as an ideal method of training agents, as its ability to meld differing skills together seemingly reflected the learning process of a human

being. Thus, we looked toward the more robust and adaptive technique of h-DQN models to integrate our individually-trained DQN models together, as opposed to using an alteration to the Q-Learning function in training one large, generalized situation.

Additionally, Tessler et al. [8] showed that their Hierarchical Deep Reinforcement Learning Network, a variation of our h-DQN model, could be trained to selectively use a collection of sub-networks to accomplish high-level tasks in a sandbox game like Minecraft. Training and combining three separate tasks into one network helped to further illustrate the advantages of an h-DQN model, outstripping both DQN and DDQN models in learning performance and convergence rate. Although their sub-network concept remarkably paralleled the kind of decision-making process we believed necessary to play Doom, their hierarchical behavior was constructed by linearly-linking their individual subtasks—and thus failed to show using sub-networks in a more dynamic, "as-needed" basis. Our research sought to generalize the h-DQN to be able to freely use its sub-behaviors given the appropriate situation.

Our intent to reduce the number of required resources involves incorporating the structural capability of h-DQNs with the concept of distillation, which was traditionally designed to transfer knowledge from network ensembles, or large, regularized network models, into smaller networks, as shown by Hinton, Vinyals, and Dean [12]. This allowed for the use of inefficient, albeit more accurate models during training as well as the switch over to more compact models for product deployment. Despite the information loss due to distillation, Hinton et al. [12] showed that a distilled network still performed comparably to its larger counterpart.

Rusu et al. [13] showed that policies from a single, task-oriented DQN model was able to be successfully distilled within the context of Atari 2600 games. In these cases, Policy Distillation exhibited the effectiveness of a distilled DQN model at playing multiple games over a standard DQN model trained to perform multiple tasks within these same games. By decomposing the Doom gameplay into individual skills, we aim to apply Policy Distillation in a similar manner, but with a focus on the modularity afforded to us by the hierarchical model. This should allow us to effectively distill our h-DQN for use as a sub-model in separate Hierarchical-DQN —thereby treating the cumulative knowledge of the distilled model as an individual skill.

## III. BACKGROUND

Below we give a brief description of our initial work and the general algorithms of DQNs, h-DQNs, and Policy Distillation.

### A. *Initial Work*

Initially, we explored using a supervised learning algorithm in Monte-Carlo Tree Search (MCTS) to solve single-player levels of Doom, as it had been previously shown by Silver et al. [14] to be effective in the environment of Go. Our methodology consisted of visually training an agent using both an exploratory and objective human gameplay of Doom in order for it to learn an effective policy for each level. Agents trained on the first level of Doom II provided mixed results; while the agent successfully traversed the basic navigational aspects of corridors and large rooms, it faced slight difficulties recognizing doors and monster enemies. Specifically, the agent would appear to confuse wall textures for doors, as it consistently ran into some sections of the corridor walls.

In hindsight, this direction of training an agent was extremely rigid. First, the scope of our agent's knowledge was limited to a few sets of human actions. Imitating these actions proved futile, as it lacked the feedback of whether some action taken was appropriate or not. Second, the scope of a 3D video game such as Doom contains a much larger game space when compared to a board game like Go's. While Go is limited by the size of its game board, therefore a finite problem, Doom is theoretically unlimited in its gameplay, never forcing the player to complete a level. Guo et al. [15] used a similar approach in the context of Atari 2600 games and found it unrealistic for an MCTS agent to function in real-time on visual domains. Thus, the complexity of even the first level of Doom was infeasible for this supervised model, which led us to seek more effective methods of building an agent, namely Deep Q-Networks.

### B. *Q-Learning and Deep Q-Networks*

Q-Learning is a type of reinforcement learning algorithm used to approximate the maximum discounted future reward $R$ given the current state $s$ and a set of available actions $a$, as described by Matiisen [16]. In other words, the algorithm provides a way of determining the total end game reward by performing the best set of actions on the first state, and acting optimally there-after. The Q-value, or "quality," of a given state-action pair is given by (3.1).

$$Q(s_t, a_t) = max\, R_{t+1} \tag{3.1}$$

Within the context of Doom, a preprocessed frame of visual data provided by the game engine and a set of command inputs (such as moving forwards and backwards, turning side-to-side, activating objects, and attacking) combine to form a reward $R$ at the next time step $t + 1$, assuming this becomes the next step taken. The value awarded to the agent is dependent upon it meeting certain conditions as well as the scenario it is operating in, all defined by user needs. (Note, we will denote any future variable from this point on using the prime notation $'$, imitating the format employed by the various sources used).

Q-values are typically chosen using a greedy algorithm. As shown by (3.2), the policy $\pi$, given the current state, selects the set of actions that will produce the highest Q-value (denoted $argmax_a\, Q(s, a)$).

$$\pi(s) = argmax_a\ Q(s,a) \tag{3.2}$$

The Bellman Equation then combined the ideas of both (3.1) and (3.2) to form the new equation (3.3). It approximates its Q-values by summing together the reward obtained with the current state and set of actions and the highest Q-value at the next state $s'$ and best set of actions $a'$.

$$Q(s,a) = r + \gamma * max_{a'}\ Q(s',a') \tag{3.3}$$

But a more recently-updated Q-function to the Bellman Equation is shown by (3.4), which approximates its Q-values through an iterative process.

$$Q(s,a) = Q(s,a) + \alpha(r + \gamma * max_{a'}\ Q(s',a')) - Q(s,a)) \tag{3.4}$$

First, $\alpha$ represents the Q-Learning rate value between 0.0 and 1.0, determining how heavily the Q-function incorporates differences between current and future Q-values. When $\alpha = 0$, the Q-function remains the same; when $\alpha = 1$, the Q-function updates with the future Q-value, as the two $Q(s,a)$ terms will cancel. Second, $\gamma$ represents the discount factor value, again between 0.0 and 1.0. It determines how heavily the next highest-valued Q-value is considered. With $\gamma = 0$, the future Q-value is disregarded, while with $\gamma = 1$, the future Q-value is heavily considered.

While the approximation of $Q(s,a)$ may be inaccurate in the early stages of training, it eventually converges to the true Q-values for any finite Markov Decision Processes (MDPs). The dilemma of choosing an action in the initial stages of training is resolved by an $\varepsilon$-Greedy Exploration Policy, described by Veličković [17] with (3.5).

$$\mathbb{P}(a|s) = \frac{\varepsilon}{|A|} + (1-\varepsilon) * argmax_a\ Q(s,a) \tag{3.5}$$

The probability $\mathbb{P}$ of performing an action from a list of actions of length $A$ is determined by the epsilon value $\varepsilon$, valued between 0.0 and 1.0. When implemented, the $\varepsilon$-Greedy Policy dictates the probability that an action is randomly selected ($\varepsilon = 1$) versus an action that is determined by the Q-function ($\varepsilon = 0$). During training, $\varepsilon$ typically decays such that actions are initially randomly-chosen to promote exploration, then solely determined by the Q-function to exploit its explored Q-values.

The original Q-Learning algorithm employs a table method to track the changing Q-values of each state-action pair. However, this method is infeasible for high-dimensional domains such as Doom. Deep Q-Networks resolve this limitation by using a Deep Convolutional Neural Network (CNN) to approximate the Q-Learning function using the visual input provided from the game engine [7]. Convolutional layers have been highly successful in the domain of image recognition and are used in DQNs to learn abstract features from the visual input to better approximate Q-values. Thus, the goal is to find the weights $\theta$ such that $Q_\theta(s,a) \approx Q(s,a)$, leading to (3.6).

$$Loss(\theta) = \sum(y - Q_\theta(s,a))^2 \tag{3.6}$$

A replay memory is used in order to reduce the instability commonly associated with using CNNs to compute Q-values. This involves selecting a random batch from a list of transitions to train the DQN, ultimately ensuring it does not get stuck at a local minimum; these transitions take the form of (3.7).

$$(s,a,r,s') \tag{3.7}$$

While $s, a,$ and $r$ all represent the current state parameters (the state, action taken, and reward given respectively), $s'$ represents the future state given $s$ and $a$.

### C. *Double Deep Q-Networks*

Double Deep Q-Networks (DDQN) is an approach to Deep Q-Learning that seeks to prevent overestimated Q-values, by partially decoupling action selection from action evaluation using two value functions—and thus two networks. This is introduced by van Hasselt el al. [10] with (3.8).

$$Q_{online}(s,a) = r + \gamma * Q_{target}(s', argmax_a\ Q_{online}(s',a)) \tag{3.8}$$

The online and target Q-functions represent the two value functions, where the online network is used to select the best action and the target network is used to generate new Q-values, denoted $Q_{target}(s', argmax_a\ Q_{online}(s',a))$. In comparison, regular Q-Learning—and by extension DQNs—uses the same network to both select and evaluate an action.

The Q-function has also been updated to approximate its Q-values through an iterative process. In this case, the target network is periodically updated with the online network during training, as shown by (3.9).

$$Q_{online}(s,a) = Q_{online}(s,a) + \alpha(r + \gamma * Q_{target}(s', argmax_a\ Q_{online}(s',a)) - Q_{online}(s,a)) \tag{3.9}$$

### D. *Hierarchical Deep Q-Networks*

At a glance, Hierarchical Deep Q-Networks (h-DQNs) function the same way as a standard DQN. However, instead of outputting a list of actions, a meta-DQN instead sits upon pre-trained sub-models—DQNs containing individually-trained skills—and typically output a behavior [8] (see Figure 3.1).

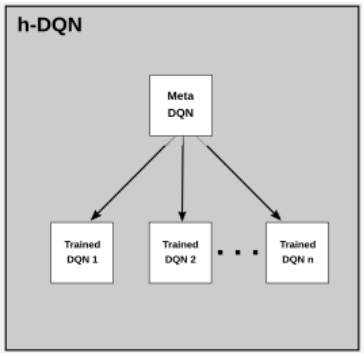


*Figure 3.1 – Diagram of the h-DQN, where the meta-DQN selects behaviors.*

In other words, h-DQNs approximate Q-values, contextually $Q(s,b)$, using a slightly modified Bellman Equation, as described by Zhou, van Kampen, and Chu [10] with (3.10).

$$Q(s,b) = Q(s,b) + \alpha(r + \gamma(max_{b'}\, Q(s',b')) - Q(s,b)) \quad (3.10)$$

The major difference involves having a behavior $b$ and the need to select the highest Q-value with the best behavior $b'$ at the next state $s'$, or $max_{b'}\, Q(s',b')$. This allows h-DQNs to learn to use the best-suited individual skill for any given situation. When training the h-DQN, the goal is to find the weights $\theta$ for the h-DQN such that $Q_\theta(s,b) \approx Q(s,b)$, similar to (3.6).

### *E. Policy Distillation*

Policy Distillation was used to extract the policy of trained reinforced agents, as shown by Rusu et al. [13]. It is a type of knowledge transfer method that trains a student network (like a DQN) to predict the outputs of a teacher network (like an h-DQN) via supervised learning. When transferring policies between DQNs, we mainly focus on retaining ordinal information found in the teacher's predicted Q-values. Thus, the target values for the student network are obtained by using the softmax function (3.11) as presented by Hinton et al. [12], where $q_{teacher}$ is the predicted Q-values of the teacher network and $\tau$ is the temperature of the softmax function.

$$softmax\left(\frac{q_{teacher}}{\tau}\right) = \frac{e^{\frac{q_{teacher}}{\tau}}}{\sum e^{\frac{q_{teacher}}{\tau}}} \quad (3.11)$$

In the case of classification, a higher temperature value is beneficial in transferring knowledge between networks. Policy Distillation, on the other hand, benefits from a lower temperature value as Q-values represent not a distribution probability, but the expected future reward of each action combination. Low temperatures help sharpen the differences between predicted Q-values, thereby facilitating learning by the student network. Rusu et al. [13] show Policy Distillation to be most effective when training the student network using a Kullback-Leibler divergence (KL) as the loss function in (3.12).

$$Loss_{KL}(\theta) = \sum softmax\left(\frac{q_{teacher}}{\tau}\right) \ln \frac{softmax\left(\frac{q_{teacher}}{\tau}\right)}{softmax(q_{student})} \quad (3.12)$$

We used Policy Distillation as a way of compressing all knowledge contained in an h-DQN model into a simple DQN architecture. The resulting distilled network retained the policy of the original h-DQN while reducing overall computations and network parameters. This was done by using the teacher DQN's selected sub-model to calculate the target values for the student DQN during training. Data for the distillation process was generated by running the h-DQN in a specified scenario with an $\varepsilon$-Greedy Policy, similar to the one used when training DQNs and h-DQNs (see Figure 3.2).

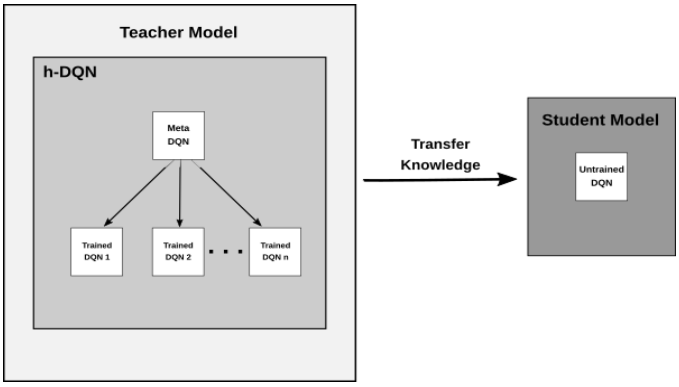


*Figure 3.2 – Diagram of the distillation process, where a teacher network (h-DQN) teaches a student network (untrained DQN).*

## IV. MODELS AND LEARNING PARAMETERS

In this section, we will discuss the network architecture and hyperparameter configurations we used in training our agent.

### *A. Preprocessing*

The ViZDoom engine provides user access to the in-game visual buffer, depth buffer, and internal game variables (of which were not exploited for the purposes of our research). To prepare the data for the DQN model training, the RGB visual buffer image was greyscaled and then combined with the depth buffer using (4.1), where $C$ represents the contrast between the grayscaled buffer and the depth buffer.

$$Buffer_{processed} = (1-C)Buffer_{grayscaled} + (C)Buffer_{depth} \quad (4.1)$$

Varying the amount of depth buffer present in the processed image helped to train the agent on different scenarios. Scenarios

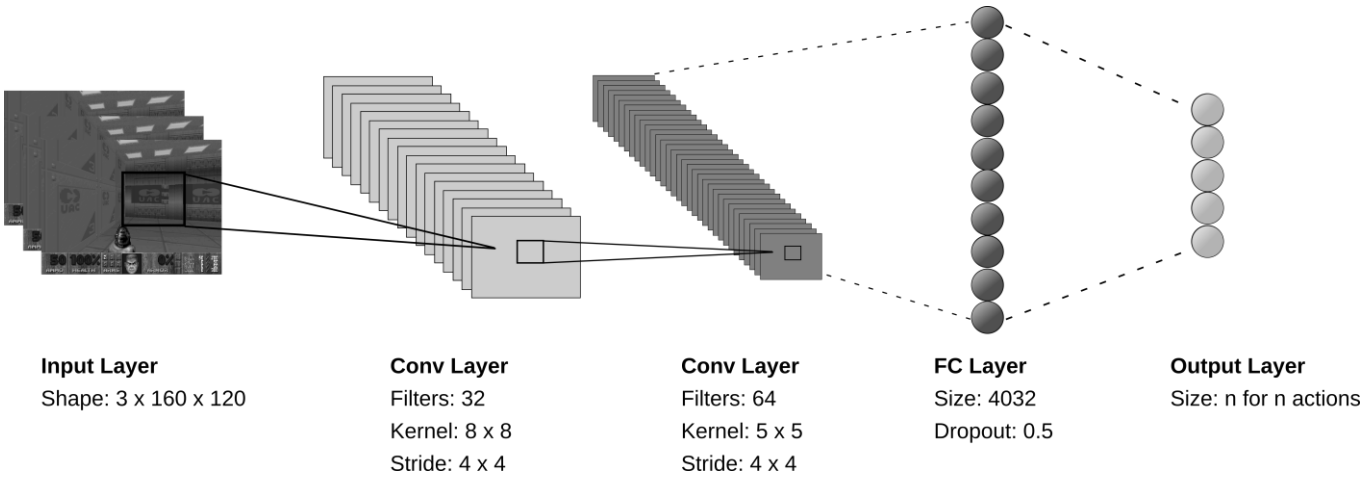


*Figure 4.1 – A diagram of our network architecture used for both DQN and h-DQN. The input consists of three frames of processed images, followed by two convolutional layers. The output of the convolutional layers is flattened and fed into a fully-connected layer; this fully-connected layer has a dropout of 0.5 to minimize overtraining. The final layer represents Q-value outputs for any given action.*

where the behavior is dependent on depth perception of the level benefited from high $C$ values (like with walking and turning through corridors), while scenarios where the agent must be able to distinguish between differing textures benefited from a low $C$ value (like with opening doors).

In order to capture temporal-dependent features, the DQN model first processes the current frame as well as a number of previous frames. After the preprocessing procedure is complete, our DQN model receives three frames of 160pt x 120pt processed images as its input (see Figure 4.1).

### *B. DQN and h-DQN*

Our policy models are implemented using the Double Deep Q-Network methodology and consist of the following layers, depicted in Figure 4.1: an Input Layer, two Convolutional Layers, a Fully-Connected Layer, and an Output Layer. Furthermore, our chosen CNN filter sizes of thirty-two and sixty-four, respectively, were inspired by the research done by Kempka et al. [4] as well as constrained by what could be feasibly trained in a reasonable time given our system specs (see Appendix I).

After flattening the nodes in our second convolutional layer, which involves taking what is in essence a 2D matrix of nodes and converting them into a 1D array, there are 4032 nodes present, dictating the size of our fully-connected layer. Within our fully-connected layer, we also apply a dropout value of 0.5, which uses a 0.5 probability of removing a node from the network to help prevent overtraining.

On a more technical note, our layers are activated using a Rectified Linear Unit (ReLU) function, which bring non-linearity into the network by thresholding our activations at zero; we also used the mean squared error (MSE) loss function and a RMSProp optimizer with a learning rate value of 0.0001 to update network weight values. Q-learning updates where preformed using a random batch of forty transitions from a replay memory of ten-thousand transitions.

The h-DQNs are similarly designed, using the same network architecture as a DQN for the meta-DQN with the difference of outputting the index of the desired sub-DQN model as opposed to specific actions. Once selected, the sub-DQN processes the image input and produces the best action according to its pre-trained behavior. It is important to note that image inputs used by sub-DQNs must be processed according to the parameters used during their training.

### *C. Epsilon Decay for Exploration Policy*

The change in the $\varepsilon$ value $\Delta_{\varepsilon}$, or an $\varepsilon$ decay, determines the amount this value decreases per epoch of training, and is calculated by (4.3), where $\varepsilon_i$ is the initial $\varepsilon$ value, $\varepsilon_f$ is the final $\varepsilon$ value, $NoE$ is the total number of epochs per training session, and $r_{\varepsilon}$ is the $\varepsilon$-rate, which determines how quickly $\varepsilon$ decays to its final value.

$$\Delta_{\varepsilon} = \frac{\varepsilon_i - \varepsilon_f}{NoE * r_{\varepsilon}} \tag{4.3}$$

During training, our $\varepsilon$ value is initially set to 1.0 and linearly decays to its final value of 0.1 over the one hundred epochs of training. For the first ten epochs, an $\varepsilon$ value of 1.0 allows our agent to explore the contents of the scenario by randomly selecting some action combination. Over the next seventy epochs, the linear decay from 1.0 to 0.1 still continues to promote exploration, but more so allows the agent to begin creating associations within the specified scenario. The final twenty epochs of $\varepsilon$ value 0.1 is targeted to fully exploit the agent previously using the entirety of its action list in an effort to accurately determine the highest-rewarding action combination for any given situation.

### D. *Alpha Decay for Learning Rate*

Additionally, the change in the $\alpha$ value $\Delta_\alpha$ is calculated by (4.4), where $\alpha_i$ is the initial Q-Learning rate, $\alpha_f$ is the final Q-Learning rate, $NoE$ is again the total number of epochs per training session, and $r_\alpha$ is the rate at which $\alpha$ decays to its final value.

$$\Delta_\alpha = \frac{\alpha_i - \alpha_f}{NoE * r_\alpha} \tag{4.4}$$

Like our $\varepsilon$ decay, our $\alpha$ value is initially set to 1.0 and linearly decays to 0.1 by the end of the one hundred epochs of training. Specifically, the $\alpha$ value exactly matches the ten-seventy-twenty epoch segments used for the $\varepsilon$ decay. In this case, the first ten epochs frequently change the Q-function—as higher $\alpha$ values heavily consider the rewards of future states—while the last twenty epochs result in few changes to the Q-function. This helps to stabilize learning in highly deterministic scenarios.

### E. *Frame Skips*

Frame skips prolong the execution of an action—once selected by the network—over $n$ in-game frames to speed up learning at the cost of accuracy, as well as lessening an otherwise computationally-heavy load on the computer system, as shown by Kempka et al. [4]. So while a frame skip value of two garners better precision at the cost of computing the best action every two frames, a frame skip value of ten lessens the amount of computations needed at the cost of less accurate judgments. We arbitrarily chose $n = 4$ frame skips, which was the recommended value from a range of frame skip values researched by Kempka et al. [4].

## V. TRAINING AND EXPERIMENTS

In this section, we will discuss the different scenarios used to train singular skills, and the composite scenario used to test whether our distilled h-DQN implementation was able to effectively combine the skills to complete a new problem.

### A. *Training Setup*

All training scenarios were trained for one hundred epochs using our Double Deep Q-Network model, while each epoch consisted of five thousands steps in the game. A step is denoted by a state change in the Doom game engine, where the agent receives an input frame and selects the best action based on its current policy. Additionally, each epoch of training was proceeded by thirty tests, where we used the average test reward to determine the relative knowledge gathered by the agent.

In the case of the distilled network, it is important to note that each epoch consisted of one game instance played by the trained h-DQN.

As for the actual scenarios themselves, each one has a living reward function, defined within the configuration file. This living reward grants the agent a penalty of one point for every frame in order to encourage it to complete the scenario as quickly as possible. Furthermore, while navigation in Doom can be broken down into numerous, differing skillsets, we reduce the number of behaviors to navigating corridors, locating exits, and opening doors.

### B. *Navigating Corridors*

The first scenario we began work on was a simple task. The agent is placed into one end of an S-shaped hallway with 90° turns and must navigate its way through the corridors to the opposite end (see Figure 5.1).

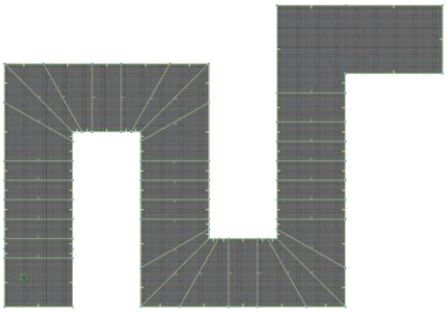

*Figure 5.1 – Map of the navigating corridors behavior. This S-shaped hallway teaches walking and turning.*

To facilitate in the training process, we define multiple line definitions throughout the level that reward the agent when it crosses over them; the agent gains twenty points for each straightaway line and sixty points for each turning line. These rewards can only be received once per line definition, in order to prevent reinforcing a behavior where the agent crosses back and forth to maximize its reward. The basic idea of this reward function is to essentially push the agent through the corridor, generating both a forward-moving and turning behaviors. In addition, there is a penalty function to dissuade the agent from touching the walls, idealistically to prevent the agent from using the wall as its moving guidance. This penalty can be received indefinitely, and as such holds a relatively low value of negative ten points. Reaching the end of the hallway gives the agent one hundred points.

As a result, the agent's traversal of the hallway more closely resembles that of a human player, with the intent of providing the hierarchical model a more generalized moving behavior. The addition of a moving reward function may have improved this behavior, which is designed to negate the living reward for simply moving (thus, a plus one reward for each step taken). In an effort to reduce the likelihood of overtraining the agent, the textures of the walls, floors, and ceilings are randomly chosen per epoch from a pre-selected pool of the more distinct game textures (currently three textures for each part).

### C. *Locating Exits*

On the other hand, creating a scenario to teach an agent to locate the exit of a room proved more complicated than originally anticipated. The agent starts in a large, square room and must "enter" the long corridor leading out from one side of the square room to end the scenario (see Figure 5.2).

*Figure 5.2 – Map of the locating exits behavior. This large room and corridor is designed to emulate "exiting" a room.*

Rewarding the agent required slightly more ingenuity to generate an accurately-represented behavior. The basic method of rewarding the agent for crossing over a line definition is not feasible for a randomly-placed agent. If the agent is placed in front of the exit, its reward would not be maximized by going directly to the reward, but rather by moving throughout the room to find the line definitions and then going to the exit. This is resolved by continuously running a script that checks if the agent is moving towards the exit based on the distance between the point at the center of the exit and the *x* and *y* coordinates of the player. While this reward did guide the agent towards the exit, it inadvertently rewarded the agent for moving as slowly as it was capable, as this would result in more frames that the reward function would be satisfied, therefore inflating the reward received.

Initially, this corridor was much shorter, resulting in the agent having difficulty distinguishing the depth of the corridor from the depth of the walls next to it. We also did not feel that keeping the player's starting position at a fixed point very accurately reflected the skill required for locating the exit of a room; in other words, the network would likely be overtrained, suitable only for repeating the exact movements necessary to exit the room. We therefore devised a script which places the player in a random position in the room, facing a random direction in an attempt to generalize this behavior.

### D. *Opening Doors*

The map for this scenario has a very simple layout, being a single corridor with nine doors placed equidistant from each other along the length of the hallway. To prevent overtraining from occurring, a degree of randomness is required. The textures of the doors are assigned randomly at the time the level loads from a pool of pre-selected door textures. Likewise, the walls, floor, and ceiling are textured in the same manner.

Our first revision of the reward function rewarded the agent for activating/opening a door, but this resulted in the agent repeatedly pressing the "use" action on the door, constantly opening and closing it, instead of progressing towards the end of the scenario. To eliminate this behavior, the reward function was made closer to the reward function of the rigid turning scenario. To promote advancing through the scenario, the agent was rewarded for crossing line definitions placed in doorways, and for crossing lines placed between two doors to promote moving towards the next door. Like in the navigating corridors scenario, rewards are received only once per line definition.

### E. *Pressing Switches - Deceased*

Upon deciding to implement sub models for individual skills, we decided that recognition and use of switches would require its own model, but this was ultimately not necessary. Due to the reward function of the composite scenario, the agent learned to adapt the door opening skill to press switches. This realization rendered the switch pressing model obsolete, and thus it was not trained. The proposed map for this model is similar to the exit finding scenario. The player is placed in a small square room, with a switch placed in the middle of the south wall. Upon the player entering the map, the player's position and orientation is randomly assigned, and the textures of the walls, ceiling, floor, and the switch are selected randomly from a pre-determined pool of textures.

To reward the agent, line definitions are placed arching around the switch, spaced 32 units apart. The script for these line definitions is identical to those of the exit finding scenario, only providing their reward when the player is both facing and moving towards the switch. Like in exit finding, the reward is increased as distance between the player and switch decreased, and a specific reward value is only given once.

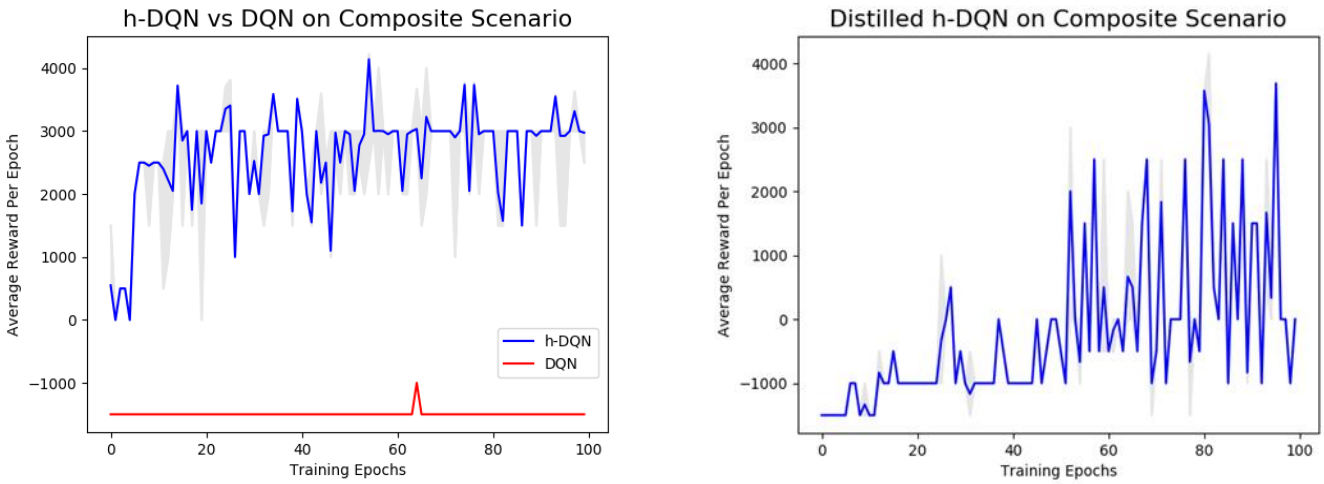


*Figure 6.1 – Plot of average training reward for (a) h-DQN and DQN on composite scenario and (b) distilled h-DQN on composite scenario. The range of reward per epoch is shown by grey shading on plot.*

### *F. **Composite Skills***

The combined skills map is designed to train the higher-level model. The map consists of connected instances of the other maps, which test the higher-level model's ability to select the correct skill at a given state. Unlike the other maps, which provided rewards for progress made in completing a skill, this map only provides rewards when the agent has successfully navigated through a segment of the map which denotes a given skill.

The walls of each skill, separated in figure 5.1 by line definitions, were each given a unique texture in an attempt to further facilitate associating an image/state with a particular skill. These textures were chosen for being greatly different from each other. The door opening skill was given the crate texture, exit finding was given the grey brick texture, rigid turning was assigned a concrete wall texture, and the button pressing skill has a red snake-like texture.

## VI. RESULTS

### *A. **Custom Scenarios***

### *B. **Hierarchical-DQN vs. DQN***

- DQN is very apt to learning a well-defined task
- A trained DQN is not well suited to performing tasks other than what it has been trained for
- (new paragraph ?) HDQN remedies some of the shortcomings of DQNs
- Instead of actions, HDQNs output DQNs

### *C. **Distillation Benefits***

Our approach of using compartmentalized skills seems to offer a great amount of expandability while also being accurate enough to navigate through our simulated levels. That our higher-level model is capable of navigating through a map where it must choose between multiple skills appears to justify our choices in designing our custom scenarios. However, the distillation does result in more noise, due to the data lost in the process.

After a single distillation process, the resulting agent performed similarly to the un-distilled network. An increase of variance in overall rewards was observed in the distilled network, with the distilled network not showing signs of convergence within 100 epochs. It is important to note that during network distillation, one epoch consists of a single game instance, so the distilled network was completed much faster than training a new network.

When a hierarchical network is distilled, it is compressed to the size of a single model. In our case, we went from the size of four networks, to the size of one network. Through careful monitoring of the distillation process, we were able to select a distilled model that received an average reward close to the converged reward of the non-distilled HDQN.

## VII. DISCUSSION

### *A. **Reward System***

One of the major hurdles of this project was shaping rewards to achieve desired behaviors in training. Our initial attempts to do so for some of the more complex skills, such as exit finding, ultimately resulted in having to remake the map and redefine how our bot should be rewarded. However, this and other behaviors highlighted the difference between explicitly and implicitly defined rewards.

Explicitly defined rewards were those we purposefully hard-coded into each scenario. They had the most pronounced effect on behavior, but did not account for some of the eccentricities we observed during and after training – which seemed to remain after minor tweaks. Implicitly defined rewards, however, were associations between behaviors and points that the network made in its attempts to accrue points. They were things we did not think of at the time of at the time of designing scenarios, and sometimes necessitated a complete redesign of the method by which we granted rewards. As such, reward shaping is something we should more carefully consider in future work.

### *B. Contributions & Future Work*

We showed that DQNs are suitable only to one explicitly defined task, and that decomposition of a problem into easily solvable sub-problems is feasible within the context of reinforcement learning through the use of H-DQNs. Furthermore, H-DQNs can be distilled while retaining some functionality, which not only resolves the issue of increasing size as more sub-models are attached to them, but also allows them to be incorporated into other H-DQNs as sub-models – thus allowing for increasingly complex skillsets.

Future work on this project will likely consist of discovering the point at which our methods are no longer useful. For example, in our work with distilling hierarchical networks, we did observe some loss in accuracy. In our experience, this proved to have little impact on the ability of the agent to navigate through a map, though the impact may be greater when distilling a hierarchical network which already makes use of a distilled network. It would also be interesting to investigate how using multiple distilled hierarchical models as sub-models in a hierarchical model effects performance. We also feel it may be prudent to investigate how reward shaping influences skill choice, which would allow us to more efficiently train networks. Lastly, investigation into frame skip decay would provide great insight into balancing the rapid training provided with high frame skips with the high accuracy displayed by models trained with small frame skips.

At the time of this writing, we are working to incorporate a new skill into the network to further prove our claims of expandability of the network.

## ACKNOWLEDGEMENTS

## REFERENCES


[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. (2013, Dec.). "Playing Atari with Deep Reinforcement Learning." Computing Research Repository. [Online]. Available: https://arxiv.org/abs/1312.5602v1. [9 March 2017].

[2] M. Johnson, K. Hofmann, T. Hutton, and D. Bignell. "The Malmo Platform for Artificial Intelligence Experimentation." Presented at Twenty-Fifth International Joint Conference on Artificial Intelligence. [Online]. Available: https://www.ijcai.org/Proceedings/16/Papers/643.pdf. [13 March 2017].

[3] Wikipedia.org. "Doom (series)." Internet: https://en.wikipedia.org/wiki/Doom_(series), Feb. 25, 2017. [30 March 2017].

[4] M. Kempka, M. Wydmuch, G. Runc, J. Toczek, and W. Jaśkowski. (2016, May). "ViZDoom: A Doom-based AI Research Platform for Visual Reinforcement Learning." Computing Research Repository. [Online]. Available: https://arxiv.org/abs/1605.02097v2. [9 March 2017].

[5] P. vd Heiden. "What is Doom Builder?" Internet: http://www.doombuilder.com/, 2008. [30 March 2017].

[6] M. Kempka, G. Runc, J. Toczek, M. Wydmuch, and W. Jaśkowski. "ViZDoom." [Online]. Available: http://vizdoom.cs.put.edu.pl/authors. [13 March 2017].

[7] G. Lample and D. S. Chaplot. (2016, Sep.). "Playing FPS Games with Deep Reinforcement Learning." Computing Research Repository. [Online]. Available: https://arxiv.org/abs/1609.05521v1. [10 March 2017].

[8] C. Tessler, S. Givony, T. Zahavy, D. J. Mankowitz, and S. Mannor. (2016, Nov.). "A Deep Hierarchical Approach to Lifelong Learning in Minecraft." Computing Research Repository. [Online]. Available: https://arxiv.org/abs/1604.07255v3. [10 March 2017].

[9] P. Mirowski, R. Pascanu, F. Viola, H. Soyer, A. J. Ballard, A. Banino, M. Denil, R. Goroshin, L. Sifre, K. Kavucuoglu, D. Kumaran, and R. Hadsell. (2017, Jan.). "Learning to Navigate in Complex Environments." Computing Research Repository. [Online]. Available: https://arxiv.org/abs/1611.03673v3. [10 March 2017].

[10] H. van Hasselt, A. Guez, and D. Silver. (2015, Dec.). "Deep Reinforcement Learning with Double Q-learning." Computing Research Repository. [Online]. Available: https://arxiv.org/abs/1509.06461v3. [10 March 2017].

[11] Y. Zhou, E. van Kampen, and Q. P. Chu. "Autonomous Navigation in Partially Observable Environments Using Hierarchical Q-Learning." Presented at the International Micro Air Vehicle Conference and Competition, Beijing, China, 2016. Available: http://www.imavs.org/papers/2016/70_IMAV2016_Proceedings.pdf. [5 March 2017].

[12] G. Hinton, O. Vinyals, and J. Dean. (2015, Mar.). "Distilling the Knowledge in a Neural Network." Computing Research Repository. [Online]. Available: https://arxiv.org/abs/1503.02531v1. [20 March 2017].

[13] A. A. Rusu, S. G. Colmenarejo, C. Gülçehre, G. Desjardins, J. Kirkpatrick, R. Pascanu, V. Mnih, K. Kavukcuoglu, and R. Hadsell. (2016, Jan.). "Policy Distillation." *Computing Research Repository*. [Online]. Available: https://arxiv.org/abs/1511.06295v2. [28 March 2017].

[14] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. (2016, Jan.). "Mastering the game of Go with deep neural networks and tree search." *Nature*. [Online]. vol. 529, pp. 485-489. Available: https://storage.googleapis.com/deepmind-media/alphago/AlphaGoNaturePaper.pdf. [8 April 2017].

[15] X. Guo, S. Singh, H. Lee, R. Lewis, and X. Wang. "Deep Learning for Real-Time Atari Game Play Using Offline Monte-Carlo Tree Search Planning." In *Advances in neural information processing systems*, pp. 3338-3346. 2014. [21 April 2017].

[16] T. Matiisen. "Guest Post (Part I): Demystifying Deep Reinforcement Learning." Internet: https://www.nervanasys.com/demystifying-deep-reinforcement-learning/, Nov. 21, 2015. [8 April 2017].

[17] P. Veličković. Nvidia Deep Learning Day, Topic: "Deep Reinforcement Learning: DQN and Extensions." University of Cambridge, England, June 30, 2016. Available: https://www.cl.cam.ac.uk/~pv273/slides/RL-PetarV-Presentation.pdf. [10 March 2017].


## ADDITIONAL RESOURCES


[1] D. S. Chaplot, G. Lample, K. M. Sathyendra, and R. Salakhutdinov. "Transfer Deep Reinforcement Learning in 3D Environments: An Empirical Study." Presented at Conference on Neural Information Processing Systems, Barcelona, Spain, 2016. [Online]. Available: http://www.cs.cmu.edu/~rsalakhu/papers/DeepRL_Transfer.pdf. [10 March 2017].

[2] D. Schlegel. Seminar Talk, Topic: "Deep Machine Learning on GPUs." Dec. 1, 2015. Available: http://www.ziti.uni-heidelberg.de/ziti/uploads/ce_group/seminar/2014-Daniel_Schlegel.pdf. [10 March 2017].

[3] E. Duryea, M. Ganger, and W. Hu. "Exploring Deep Reinforcement Learning with Multi Q-Learning." Intelligent Control and Automation, vol. 7, pp. 129-144, Nov. 2016. [10 March 2017].

[4] E. Park, X. Han, T. L. Berg, and A. C. Berg. "Combining Multiple Sources of Knowledge in Deep CNNs for Action Recognition." Presented at Winter Conference on Applications of Computer Vision, New York, USA, 2016. [Online]. Available: http://www.cs.unc.edu/~eunbyung/papers/wacv2016_combining.pdf. [10 March 2017].

[5] E. Schlessinger, P. J. Bentley, and R. B. Lotto. "Modular Thinking: Evolving Modular Neural Networks for Visual Guidance Agents." In Proc. GECCO, 2006, pp. 215-22. [10 March 2017].

[6] G. Lample and D. S. Chaplot. (2016, Sep.). "Playing FPS Games with Deep Reinforcement Learning." Computing Research Repository. [Online]. Available: https://arxiv.org/abs/1609.05521v1. [10 March 2017].

[7] J. Oh, X. Guo, H. Lee, R. Lewis, and S. Singh. (2015, Dec.). "Action-Conditional Video Prediction using Deep Networks in Atari Games." Computing Research Repository. [Online]. Available: https://arxiv.org/abs/1507.08750v2. [10 March 2017].

[8] J. R. Kok and N. Vlassis. "Sparse Cooperative Q-learning." In Proc. ICML, 2004. Available: http://www.machinelearning.org/icml2004_proc.html. [5 March 2017].

[9] J. Janisch. "Let's make a DQN: Double Learning and Prioritized Experience Replay." Internet: https://jaromiru.com/2016/11/07/lets-make-a-dqn-double-learning-and-prioritized-experience-replay/, Nov. 7, 2016. [13 March 2017].

[10] K. Tumer, A. K. Agogino, and D. H. Wolpert. "Learning Sequences of Actions in Collectives of Autonomous Agents." In Proc. AAMAS, 2002, pp. 378-385. [10 March 2017].

[11] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting." Journal of Machine Learning Research, vol. 15, pp. 1929-1958, June 2014. [10 March 2017].

[12] R. Fiszel. "Reinforcement Learning and DQN, learning to play from pixels." Internet: https://rubenfiszel.github.io/posts/rl4j/2016-08-24-Reinforcement-Learning-and-DQN.html, Aug. 24, 2016. [30 March 2017].

[13] T. Barron, M. Whitehead, and A. Yeung. "Deep Reinforcement Learning in a 3-D Blockworld Environment." Presented at Deep Reinforcement Learning: Frontiers and Challenges Workshop, International Joint Conference on Artificial Intelligence, New York, USA, July 2016. [Online]. Available: https://drive.google.com/file/d/0B68wM6S3qQtfU182dXo2bndKSWM/view. [10 March 2017].

[14] Y. Zhu, R. Mottaghi, E. Kolve, J. J. Lim, A. Gupta, L. Fei-Fei, and A. Farhadi. (2016, Sep.). "Target-driven Visual Navigation in Indoor Scenes using Deep Reinforcement Learning." Computing Research Repository. [Online]. Available: https://arxiv.org/abs/1609.05143v1. [10 March 2017].