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

Rafael Zamora, William Steele, Joshua Hidayat & Lauren An
Department of Computer Science, Hood College, Frederick, Maryland
May 2017

***Abstract*—Deep Q-Networks (DQNs) heavily aided in guiding agents to associate differences between rewarding and penalizing actions on both the Atari 2600 games and the early Doom games. But to more effectively compensate for the natural intricacy of 3D games such as Doom, the opportune Hierarchical Deep Q-Networks (h-DQN) allowed for environment modularization into separate skill-based models—our basis for designing more efficient agents. The hierarchical methodology was previously employed within the sandbox game Minecraft, although these experimentations merely focused on a few actions following a particular order. We therefore aimed to implement h-DQNs within Doom to one, continuously add new knowledge via periodically distilled h-DQN models, and two, to ensure multi-skilled functionality by randomly-situating these skills in validation maps. An initially-trained h-DQN model provided positive indication of the agent's ability to combine vastly differing skills, both an enhancement in its skill-adaptive capabilities and training times when compared to single DQN models. Advantageously selecting between navigating the level versus operating in-game objects during specific moments (walking and turning, locating exits, and opening doors in our case) led us to challenge the plausibility of training a distilled h-DQN model alongside a separate skill of, say, monster combat. The results were staggering.**

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

I. INTRODUCTION

DESPITE improvements to game-playing Artificial Intelligence (AI), AIs rely on looking "under the hood" of their respective games in order to gain access to the various internal variables, thus holding an unfair advantage over their human counterparts. However, ally and enemy NPCs (non-playable characters) are typically designed to keep the human player engaged with a fair challenge. When set on easier difficulty settings, AIs are typically considered "pushovers," while on harder difficulty settings, AIs are capable of playing near "perfect." In order to design a more balanced and enjoyable experience, it may be more suitable for agents to behave similar to a human player using the same available environmental information.

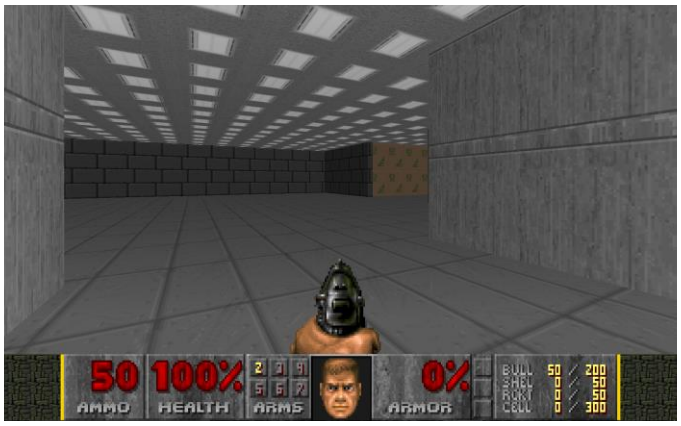

Figure 1.1 - A sample Doom environment

Google DeepMind's paper, *Playing Atari With Deep Reinforcement Learning* [Playing Atari…], 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 ample amount of research into applying these same reinforcement learning techniques to train agents within three-dimensional (3D) environments such as Minecraft [The Malmo…] and Doom.

The Doom series [Doom (series)] 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 level design complexity with the incorporation of depth, providing another factor when designing agents. The relevancy of Doom with Artificial Intelligence can be contributed to the AI research platform ViZDoom [ViZDoom: A…] alongside the scenario customization software Doom Builder and Slade (of which we use Doom Builder). ViZDoom allows programmers to test reinforcement learning techniques within the environment of Doom. Its Visual Doom AI Competitions [ViZDoom], as well as previous research [Playing FPS…], 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 our capstone project, we propose to create an agent capable of solving complex navigational problems using a hierarchical implementation [A Deep…] of the Deep Q-Networks, expanding on ViZDoom's experimentations except with a heavier focus on navigation. We will first utilize DQN models to separately train simple tasks (such as turning and finding exits) and then integrate these as sub-models in a Hierarchical Deep Q-Network (h-DQN) model. In addition, we will investigate distilling, or compressing, trained h-DQN models into the simple DQN architecture for a more resource-efficient execution. Increasingly-complex behaviors can then be achieved by incrementally-aggregating distilled h-DQN models with more skills into other Hierarchical-DQN models, thus reflecting our goal of developing a more proficient Doom-playing Artificial Intelligence.

In this paper, we will first look at related work involving the previously-mentioned models, followed by a brief explanation of how we incorporated these models as well as particular parameter selection. We will then detail training models on custom scenarios of Doom and discuss the results of the experiments.

II. RELATED WORK

The inspiration for the use of Deep Reinforcement Learning (DRL) techniques in the context of video games looks to Mnih et al. and the Atari 2600 games, where DQNs (a composition of CNNs and Q-Learning) were used to learn control policies from several games [Playing Atari…]. This demonstrates 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.

A recent attempt at visual navigation by Mirowski et al. [Learning to…] includes a focus on color and texture to navigate maze-like environments using an otherwise similar approach to Mnih et al. This demonstrates that it is feasible to train a CNN capable of navigating complex environments using only visually-apparent data. However, this leaves out some of the inherent complexity of a video game such as Doom, where the player is required to use multiple task-specific skills (like operating in-game objects such as switches and doors) in differing scenarios to reach the end of a level. 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 was created as an experimental platform for the development of visual-input-based agents [ViZDoom: A...]. ViZDoom uses the ZDoom engine to provide an environment to train agents within the context of a first-person shooter game—in this case, Doom and its derivative games. This allows for an extreme amount of customizability in regards to training scenarios because users can create their own levels to train agents on, namely the aforementioned Doom Builder in our case. Doom Builder not only allows for object and texture-specific needs, but also uses ACS Scripts to define reward systems needed for reinforcement learning. To demonstrate the functionality of their platform, ViZDoom created a simple agent using a DQN model again similar to that of Mnih et al.

Lample and Chaplot took this a step further by using the ViZDoom platform to construct an agent capable of Deathmatch scenarios [Playing FPS...]. They decomposed the Deathmatch task into the individual skills of shooting and navigation and trained separate Deep Q-Networks for each task after realizing that a single DQN would be incapable of adequately completing the scenario. Combining these individually-learned skills proved that an agent could be trained to perform in Deathmatch scenarios by simplifying the global task into simpler local skills.

[Add sections on DQN and Multi-Q]

Hierarchical Deep Q-Network (h-DQN) models were shown by Zhou, Kampen, and Chu to demonstrate task decomposition while also helping to accelerate learning and more efficiently transfer knowledge between tasks within the context of flight navigation [Autonomous Navigation...]. Their h-DQN model was able to handle the global task of navigation while a sub-model performed obstacle avoidance. With task decomposition in mind, we looked toward a more robust and adaptive technique, such as h-DQN, as opposed to alterations to the Q-Learning functions.

Additionally, Tessler et al. showed that their Hierarchical Deep Reinforcement Learning Network, which is modification of the h-DQN model, can be trained to selectively use a collection of subnetworks to accomplish high-level tasks in a way that outstrips DQN and Double DQN (DDQN) models in learning performance and convergence rate within the context of Minecraft [A Deep...]. Not only does this demonstrate the applicability of their model to complex 3D environments, but it is also remarkably similar to the kind of decision-making we believe necessary to play Doom, because subnetworks are managed such that they are only used as needed.

Our intent is to incorporate the structural capability of h-DQNs with Distillation, which traditionally allows for the transfer of knowledge from network ensembles or large, regularized network models to smaller networks, as shown by Hinton, Vinyals, and Dean [Distilling the...]. This allows for the use of inefficient, albeit more accurate models during training, while switching over to more compact models for product deployment. Despite the information loss due to compression, they show that a distilled network performs comparably to its larger counterparts.

Rusu et al. shows that policies from single task-oriented DQNs can be successfully distilled or compressed in the context of Atari 2600 games [Policy Distillation]. In this case, task distillation allows a distilled-DQN to play multiple games better than a standard DQN trained to perform multiple tasks. By decomposing gameplay into individual skills in Doom, 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 an h-DQN for use as a sub-model in another h-DQN—thereby treating the cumulative knowledge of an h-DQN as an individual skill.

III. BACKGROUND

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

A. Initial Work

We initially explored using a supervised learning and Monte Carlo Tree Search (MCTS) as it has been shown to be effective in video game environments [Mastering the…]. Our goal was to visually train our agent based solely on human gameplay of Doom, in order for the agent to be able to complete levels designed for human players. In hindsight, this methodology of training the agent was extremely rigid. The agent was limited to simply imitating human actions, having no feedback on whether some action is appropriate or not. Initial agent training on the first level provided mixed results; the agent successfully traversed basic navigational pieces of the level, such as rooms and corridors, but faced slight difficulties recognizing doors. Specifically, the agent would appear to confuse wall textures as doors as the agent would consistently run into some sections of the corridor walls. The complexity of even the first level of Doom proved much for our initial 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

[Cite https://www.nervanasys.com/demystifying-deep-reinforcement-learning/]

Q-Learning is a type of reinforcement learning that defines a function, **Q(s,a)**, which determines the maximum discounted future reward, **R**, at a time step, **t**, of a future state by performing an action, **a**, on a state, **s**. Or:

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

Within the context of Doom, an action, **a**, would be a command input, such as moving forwards, backwards, side-to-side, turning, and activating an object. A state, **s**, would be a frame of visual data. The reward, **R**, is a value we define to be awarded to the bot upon meeting certain conditions, depending on the scenario it is operating in. Lastly, the time, **t**, simply says that a reward corresponds to a particular action and state.

These Q-values can then be used to determine the policy, or best action, ***π***, at a given state, **s**, by,

$$\pi(s) = \mathrm{argmax}_a\, Q(s, a),$$

where **argmax$_a$** is the index of the action, **a**, with the highest Q-value, **Q(s,a)**.

To approximate the value of **Q(s,a)**, we update the Q-function using,

$$Q(s,a)' = Q(s,a) + \alpha(R + g * (\max_{a'} Q(s', a')) - Q(s,a)),$$

Where **Q(s,a)'** is the next Q-value, **R** is the immediate **a'** and **s'** are the next action and state used to determine a Q-value, ***α*** is the Q-learning rate between 0.0 and 1.0, ***g*** is the discount factor between 0.0 and 1.0, and **max$_{a'}$ Q(s')** is the index of the next action, **a'**, with the highest Q-value. For clarity, ***α*** the rate at which the Q-function changes. If $\alpha = 0$, then the Q-function will remain the same. Additionally, the change in ***α***, Δ_α, is determined by,

$$\Delta_\alpha = (\alpha_i - \alpha_f) / (NoE * r_\alpha),$$

where α_i is the initial Q-learning rate, α_f is the final Q-learning rate, **NoE** is the number of total epochs in a training session, and r_α is the rate at which ***α*** decays to its final value.

Furthermore, ***g*** determines how heavily the next highest valued action is considered, and at $g = 0$, the next highest valued action will not be considered at all, where at $g = 1$ the next highest valued action will be heavily considered. While the approximation of **Q(s,a)** may be completely wrong in the early stages of training, it will eventually converge to represent true Q-values. While these Q-values are normally stored in a memory table, there would not be enough room to represent the large problem space of a 3D game.

A DQN resolves this limitation by using multiple hidden layers and the Q-learning function to determine a policy based upon an input of a state. Furthermore, Convolutional Layers (CLs) are used help process visual data by abstracting features from images to approximate the Q-function. This results in the output of an action to be applied to the next state.

Note that in initial stages of training choosing an action can be problematic. This can be resolved by Epsilon-Greedy,

$$P(a|s) = \varepsilon/|A| + (1-\varepsilon) * \mathrm{argmax}_a\, Q(s,a)),$$

where **P** is the probability of an action, **a**, occurring under a state, **s**, **ε** is a value $(1.0 > \varepsilon > 0.0)$, **A** is the length of a set of actions, and **argmax$_a$ Q(s,*a*)** is the index of an action, ***a***, with the highest Q-value, **Q(s,*a*)**. Implemented, Epsilon-Greedy is the probability, **ε**, that an action is randomly selected versus and action determined by the DQN. As training continues, **ε** typically decays such that actions taken are determined by the Q-function, as opposed to being determined randomly. Epsilon decay, Δ_ε, determines the amount the Epsilon value decays per epoch of training, and is calculated by,

$$\Delta_\varepsilon = (\varepsilon_i - \varepsilon_f) / (NoE * r_\varepsilon),$$

where ε_i is the initial Epsilon value, ε_f is the final Epsilon value, **NoE** is the number of total epochs in a training session, and $\mathbf{r}_\varepsilon$ is the Epsilon-rate, which determines how quickly Epsilon decays to its final value.

C. *Double Deep Q-Networks*

- $Y_t^{DoubleDQN} = R' + g * Q(s', \text{argmax}_{a'} Q(s', a)$
- $Q(s,a) = Q(s,a) + a(R' + g * Q(s', \text{argmax}_a Q(s', a)) - Q(s,a)$

D. *Hierarchical Deep Q-Networks*

- Diagram

At a glance, HDQNs function the same way as a standard DQN. However, instead of outputting a list of actions, HDQNs instead sit upon pre-trained sub-models – DQNs containing skills – and typically output a behavior [A Deep Hierarchical...]. In other words, HDQNs approximate Q-values, contextually **Q(s,b)**, using a slightly modified Bellman Equation,

$$\mathbf{Q(s,b) = Q(s,b) + \alpha(R + g\ (\max_{b'} Q(s')) - Q(s,b))},$$

where **s** is a state, **b** is a behavior, ***a*** is the Q-Learning rate as described above, ***g*** is the discount factor as described above, and $\mathbf{\max_{b'} Q(s')}$ is the index of the next behavior, **b'**, with the highest Q-value [Autonomous...].

This allows HDQNs to learn to use the most suited individual skill it has for a given situation.

E. *Policy Distillation*

Distillation can be used to extract the policy of a trained reinforced agents as shown by(). Distillation is a knowledge transfer method which trains a student network to predict the output a teacher network 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 taking the softmax of the teacher network's predicted Q values with parameter T determining the "temperature" of softmax:

Softmax(q-teacher/t) = e(q-teacher/t)/sum(e(q-teacher/t))

In the case of classification, a higher temperature value is beneficial in transferring knowledge between networks. Softening the distribution allows for more secondary Policy distillation, on the other hand, benefits from lower temperature softmax as Q-values represent not a distribution of probabilities, but the expected future reward of actions. Low temperatures help sharpen the differences between predicted Q-values facilitating the student networks learning. Rusu et al show policy distillation is most effective when training the student network using Kullback-Leibler divergence (KL) as the loss function:

LossKL =

In the context of this project, we used policy distillation as a way of compressing all knowledge contained in a h-DQN into a simple DQN model. The resulting distilled h-DQN model would have... This is done by using the selected submodels output layer to calculate the target values for the student DQN during training. Data is generated by running the h-DQN with an e-greedy exploration policy similar to the one used when training DQNs and h-DQNs. This improved performance of distilled-H

IV. MODELS AND LEARNING PARAMETERS

A. *Preprocessing*

The input of our DQNs consists of a number of frames – 120x160 in resolution – a list of available actions or behaviors, such as directional movement keys or the skill for rigid turning, a depth radius between 0.0 and 1.0 – which determines how far a bot can 'see' – a depth contrast between 0.0 and 1.0 – which determines how well a bot detects textures – and a boolean value - which determines whether or not the DQN will be distilled, and is set to False by default.

B. *DQN and HDQN Model*

Explain our Q-Network with CNN? Input>Convx32>Convx64>FC>Actions; activated using ReLU]

[Describe reasons for choices:

- Replay Memory
- Filter sizes determined by VizDoom: A...
- Number of filters determined by system specs; most w/o impacting training time
- FC Layers 4032 nodes = # of nodes at the end of the last conv layer when flattened
  - Flatten: takes 2d set of nodes, and makes them 1d
- Resolution = 120x160
- ReLU instead of Tanh or Sig
- Mean-Squared Err Loss Function
- Optimizer: RMSprop LR = .0001
- Dropout = .5
- Depth radius and contrast

Our DQNs consist of an Input, two Convolutional Layers, a Fully Connected Layer, and an Output. Additionally, we used Replay Memory to reduce the instability commonly associated with using CNNs to compute Q-values. This involves using a random batch from a list of transitions to train the DQN, while ensuring it does not get stuck at a local minimum. Transitions take the form of,

$$\mathbf{(s,a,r,s')},$$

where **s** is the current state, **a** is the current action, **r** is the current reward, and **s'** is the previous state. Furthermore, we based our CNN filter sizes on the research done by Kempka et al., and our number of filters, 32 and 64 for each Convolutional Layer respectively, on what could be trained in a reasonable time given our system specs [see appendix].

After flattening the nodes in our last 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, which also determines the size of our Fully Connected Layers. Within our Fully Connected Layers we also applied a Dropout value of .5, which randomly removes a node from the network with a probability of .5 to help prevent over training.

Additionally, it is important to note that our layers are activated using Rectified Linear Units, which bring non-linearity into the network by thresholding our activations at zero, and we used the Mean-Squared Error loss function and a RMSProp Learning Rate value of .0001 to optimize our parameter values.

C. *Epsilon Decay*

During training our Epsilon value is initially set to 1.0 and linearly decays to 0.1. Additionally, we define a Wait-value of 10 and an Epsilon-rate of .7, such that over 100 epochs of training the first 10 epochs are completed using only randomly determined actions, and the last 20 epochs are completed at the final Epsilon value. This encourages our bot to use the entirety of its action lists to accurately determine the highest rewarded action for a situation.

D. *Alpha Decay*

Like our Epsilon Decay, our $\boldsymbol{a}$ is set to 1.0 and linearly decays to .1 by the end of 100 epochs of training. Like above, we define a Wait-value of 10, and the Alpha-rate is set to .7, such that over 100 epochs of training the first 10 epochs are frequently changing the Q-function, and the last 20 result in few changes to the Q-function.

E. *Frame Skips*

Frame Skips prolong the execution of an action – once selected by a network – over **n** in-game frames to speed up learning at the cost of accuracy, and lighten an otherwise computationally heavy load on systems [VizDoom...]. We arbitrarily chose **n = 4** Frame Skips, which is the minimum of the range recommended by Kempka et al.

V. TRAINING AND EXPERIMENTS

A. *Preface*

All of our scenarios were trained for the same amount of time. Each model was run for 100 Epochs. Each Epoch consisted of 5000 steps in the game. A step is denoted by a state change in the Doom game engine.

B. *Rigid Turning*

The first scenario we began work on is very simple; the player is placed into a hallway with 90° turns, making the corridor S-shaped. The player starts at one end of the corridor and must navigate through the hall to the opposite end. To facilitate in the training process, we defined multiple lines throughout the level which increase a reward when the player crosses over them. These rewards can only be received once per each line definition, in order to prevent reinforcing a behavior of crossing back and forth across a line to maximize the reward. This reward function essentially pushes the agent through the corridor as it learns why it is rewarded. Additionally, there is a function that decreases the agent's reward when it touches a wall. This "reward" can be received indefinitely, and as such holds a relatively low value. As a result, the agent's traversal of the hallway more closely resembles that of a human player, and tends (Not sure if this accurately reflects what this does) to give the agent more information to be used by the hierarchical model. In an effort to reduce the likelihood of over training the model, the textures of the wall, floor, and ceiling textures are randomly chosen from a pre-selected pool of several of the game's included textures.

C. *Exit Finding*

Creating a scenario to facilitate in how to learn to find exits from a room proved more complicated than originally anticipated. The map we are using consists of a large square room, which the player starts in, with a narrow but long corridor leading out from one edge, which the player must enter to end the scenario. Initially, this narrow corridor was not nearly as long, but this resulted in the agent having difficulty when starting training in distinguishing the depth of the corridor from the depth of the wall next to it. We did not feel that keeping the player's starting position at a fixed point would very accurately reflect the skill required for finding an exit out of a room. In such a situation, the network would likely be over trained, suitable only for turning x degrees and moving y units. Instead, we devised a script which would place the player in a random position in the room, facing a random direction.

Rewarding the agent also required slightly more ingenuity to be done accurately. The method of rewarding the agent for crossing over crossing over a line definition was no longer feasible when the starting position of the agent was made random. If the agent were placed in front of the exit, its reward would not be maximized by going directly to the reward, but rather by moving in the room to find the line definitions and then going to the exit. Originally, this was resolved by a continuously running a script that checked if the agent was 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.

The reward function was greatly overhauled upon discovering the above unwanted behavior. In the exit_finding_v2 file, the reward function is applied via discrete line definitions, progressively arching outwards from the exit. These line definitions have attached scripts which borrow from the ideas present in the dynamically calculated rewards. When activated by walking over an associated line definition, the script first checks if the player is looking at the exit by comparing the orientation of the player to the angle of the exit relative to the player. If the player is facing the exit and is moving forward while crossing a line definition, then the agent is rewarded, with the reward increasing as the distance between the agent and the exit decreases. Because having curved line definitions is not possible in the Doom engine, the arches were comprised of multiple line definitions approximating a curve. The staggered nature of the reward value aided in overcoming the problems with having multiple line definitions, and a flag was created to stop rewards for each "tier" after they had been rewarded. When the player has reached the exit, they are given a reward, and the scenario ends.

D. Door Opening

Creating an environment where the agent could learn how to open doors was straightforward at first glance. However, to prevent overtraining from occurring, a degree of randomness is required. 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. 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.

When determining how to reward the agent, the most obvious method resulted in unwanted behavior. 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. Each reward was given only once.

E. Switch Pressing - 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.

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.

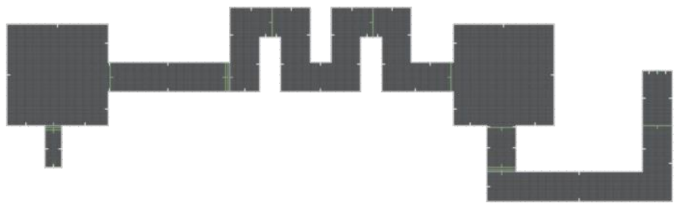

Fig. 5.1 - composite.wad map

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??

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.

B. Hierarchical-DQN vs. DQN??

C. Distillation Benefits??

After a single distillation process, the resulting agent performed similarly to the uncompressed 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. This is in part due to the distilled network no longer exhibiting the behavior which allowed the agent to utilize the door opening skill to press the switch at the end of the scenario. This setback also drastically reduced the maximum reward received by the agent, as shown in figures 6.1 and 6.2.

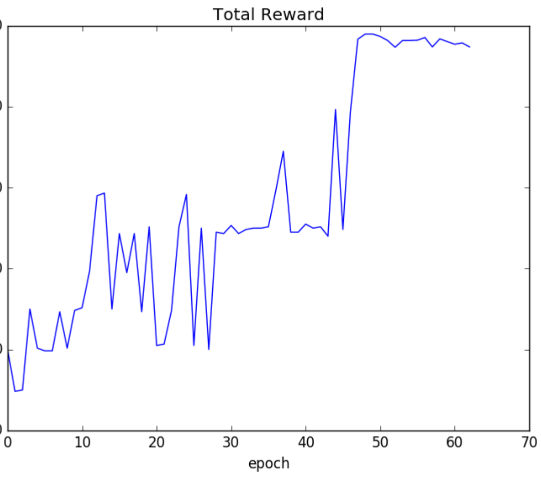

Fig. 6.1 Undistilled HDQN reward received

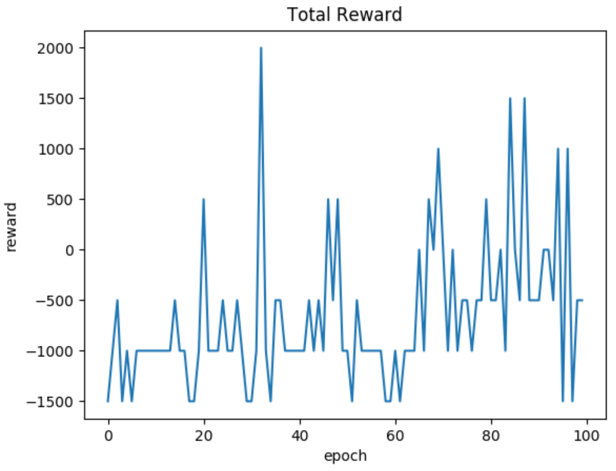

Fig. 6.2 Distilled HDQN reward received

VII. CONCLUSION

A. *Project Thoughts??*

B. *Reward System??*

C. *Future Work??*

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.

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.

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.

ACKNOWLEDGEMENTS

REFERENCES

[1] 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].

[2] 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].

[3] 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].

[4] D. Schlegel. Seminar Talk, Topic: “Deep Machine Learning on GPUs.” Dec. 1, 2015.

[5] 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.

[6] 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].

[7] 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.

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

[9] 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].

[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] 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].

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

[13] Jaromiru. (2016, Nov. 7). “Let’s make a DQN: Double Learning and Prioritized Experience Replay” [Online]. Available: https://jaromiru.com/2016/11/07/lets-make-a-dqn-double-learning-and-prioritized-experience-replay/. [13 Mar. 2017].

[14] 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.

[15] 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 Mar. 2017].

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

[17] 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].

[18] 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.

[19] 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].

[20] P. vd Heiden. (2008). “What is Doom Builder?” [Online]. Available: http://www.doombuilder.com/. [30 March 2017].

[21] P. Veličković. “Deep Reinforcement Learning: DQN and Extensions.” [10 March 2017].

[22] R. Fiszel. (2016, Aug. 24). “Reinforcement Learning and DQN, learning to play from pixels” [Online]. Available: https://rubenfiszel.github.io/posts/rl4j/2016-08-24-Reinforcement-Learning-and-DQN.html. [30 March 2017].

[23] 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].

[24] 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].

[25] Wikipedia. (2017, Feb. 25). “Doom (series)” [Online]. Available: https://en.wikipedia.org/wiki/Doom_(series). [30 March 2017].

[26] 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 Mar. 2017].

[27] 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].