Solving Complex Problems with Deep Multi-Level Skill Hierarchies

Nicolaj Casanova Abildgaard, Tobias Lambek Jacobsen
{nabild16, tlja16}@student.aau.dk

Advisor: Chenjuan Guo
cguo@cs.aau.dk

Spring Semester 2021

Abstract

The notion of using pre-trained skills to reduce training time and to facilitate lifelong learning in Deep Reinforcement Learning (DRL) has been around for a long time. However, the number of skills required to work in an environment goes up as the amount of tasks in the environment increases. As a consequence, the complexity of the action space increases and agents will need to train for longer in order to conquer all tasks. In this paper we propose a framework for Deep Multi-Level Skill Hierarchies (D-MuLSH) as a solution to this problem. This framework is an extended version of the Hierarchical Deep Reinforcement Learning Network (H-DRLN) that adds the ability to arrange the skill hierarchy with multiple levels. Simple skills are grouped into complex categories, by use of pre-trained Major Skill Networks (MSN), and agents only need to learn when to use each category, rather than learn when to use each individual skill. We show that D-MuLSH improves training time in the ViZDoom environment compared to the H-DRLN.

1 Introduction

Lifelong learning is concerned with the continued learning of tasks over the course of a lifetime, and is an open problem in the field of general purpose machine learning agents. In a machine learning setting, (Silver et al., 2013) suggests the following essential elements for a lifelong learning agent: (1) the retention of learned knowledge; (2) the selective transfer of prior knowledge when learning new tasks; and (3) a systems approach for the effective and efficient interaction of the retention and transfer elements.

A machine learning approach to lifelong learning will have to overcome the problem of dimensionality, as the state and action spaces of an environment increases as new tasks are introduced. A challenging environment that contains many of the elements related to lifelong learning is the Doom environment. Doom is a First Person Shooter (FPS) game first introduced in 1993 in which the player has to navigate a series of challenging 3D environments whilst fighting enemies, collecting items and avoiding hazardous obstacles. While many machine learning agents are developed within the Doom domain, most of these are focused on solving challenges in specific domains, such as death-match or maze navigation (Wu and Tian (2017) and Parisotto and Salakhutdinov (2018) can be seen as examples of this). Although agents such as these have shown good performances in their respective domains, they lack the structure to retain and reuse the knowledge they obtain from their respective single-task learning settings, in order to perform as a lifelong learning system. Developing a single agent for solving all the challenges the Doom domain has to offer becomes a more complex problem requiring new approaches. A likely solution to this could be a divide-and-conquer approach in which an agent learns to solve sub-problems in the domain and use its combined knowledge to solve more complex problems. In this approach an agent can acquire a knowledge base consisting of different skills that can be reused when encountering different problems throughout its lifetime.

A single level in Doom can be naturally decomposed into a series of repeated sub-problems. An example of this could be a level in which the player has to navigate a maze of tight corridors that leads to an open room with enemies, with the exit located behind a closed door at the far end of the room. This level can be completed by solving the sub-problems of: navigating the corridors, targeting and shooting the enemies, locating the exit and interacting with the door. Representing each sub-problem as a separate skill taught to the agent means that the agent can in the future solve a different level entirely by reusing these skills.

However, learning these skills and understanding when to use them is a non-trivial problem.

On account of significant advances in Deep Reinforcement Learning (DRL) and Deep Q-learning Network (DQN)s in recent times it is not infeasible to imagine Reinforcement Learning (RL) agent that can not only learn to solve these sub-problems, but also learn when to apply different skills based on visual input alone. Previous works exploring these ideas have been published in the past. These include amongst others the FeUdal Networks for Hierarchical Reinforcement Learning Vezhnevets et al. (2017) and the Hierarchical Deep Reinforcement Learning Network (H-DRLN) Tessler et al. (2016). While the FeUdal networks provide an interesting take on sub-goal discovery the H-DRLN is an especially interesting work that tries to tackle the problems of lifelong learning. H-DRLN succeeds in providing proof of concept that a hierarchical approach can solve the challenges of lifelong learning. However, the environments in which they conduct their experiments are in terms of complexity somewhat simple and does not sufficiently represent the complexity of a true lifelong learning system. Our multi-level hierarchical framework contains the necessary structural depth to sufficiently prove that hierarchical reinforcement learning agents are a valid approach in solving the complexity of lifelong learning.

In order to develop RL agents in the Doom domain we can use the **ViZDoom**[1] platform which is a modification of ZDoom, a modern open-source port of the Doom Engine. ViZDoom allows RL agents to step through the game, obtain observation data, perform actions, and receive rewards at each frame. An example frame of the ViZDoom environment can be seen in Figure 1.
Agents in ViZDoom can observe two types of data:

Visual data where resolution of the image can be set by the environment.

Game data which is a set of variables that hold information about the player's health-points, ammunition count, and more.

For actions, ViZDoom provides 43 possible "buttons" that can be pressed by the agent. These include buttons to move, look around, attack, and switch weapons.

The purpose of this paper is to present a novel reinforcement learning agent that is able to complete complicated tasks in a high-dimensional domain, whilst being able to retain and transfer knowledge in a lifelong learning setting.

[1] http://vizdoom.cs.put.edu.pl/

Figure 1: Game sample of a scenario in ViZDoom from the first-person perspective.

Main contributions: (1) Deep Multi-Level Skill Hierarchy (D-MuLSH): A multi-level hierarchical approach for using deep reinforcement learning in the ViZDoom environment. This approach utilizes a hierarchical framework including a controller network as well as reusable Deep Skill Network (DSN)s to solve complicated tasks in the ViZDoom environment. The D-MuLSH framework is designed to function as a truly lifelong learning framework. (2) We show how reusable DSNs can be learned and used as knowledge transfer elements for learning new tasks. (3) We show that by segregating DSNs into sub-hierarchies denoted as Major Skill Network (MSN)s and Simple Skill Network (SSN)s we are able to effectively and efficiently support the retention and transfer of knowledge. (4) We present empirical results demonstrating our framework's performance in both simple and complex subdomains of ViZDoom as compared to a Double Deep Q-learning Network (DDQN) baseline and the H-DRLN.

2 Related Work

Non-hierarchical Approaches: These include agents such as Arnold (Lample and Chaplot, 2018) and F1 (Wu and Tian, 2017) which were both initially developed for the 2016 ViZDoom competition hosted by IEEE CIG[2]. Both of these networks were designed and trained with the purpose of performing effectively in a death-match scenario. Arnold consists of two separate neural networks, one for navigation and one for fighting enemies, while F1 uses an actor-critic model trained using curriculum learning. Both agents performed well in the death-match scenario, but neither addresses the challenges imposed in lifelong learning systems.

Our solution differs from these agents as our framework is subjugated to not only perform in a death-match scenario but also function in broader aspects of the Doom domain with a more diverse problem setting.

[2] ViZDoom competition at IEEE Computational Intelligence and Games Conference http://vizdoom.cs.put.edu.pl/competitions/vdaic-2016-cig.

Hierarchical Approaches:
These solve complex learning tasks using patterns of divide-and-conquer methodology. Notably (Tessler et al., 2016) describes a deep hierarchical approach to teach an agent to do multiple tasks in the game Minecraft. These tasks consists of: navigating a room, breaking and picking up an object and placing an object at a specific location. Each of these tasks are learnt by a separate Deep Skill Network (DSN) and are represented as a skill in the network. A DSN is a DQN with its own policy that has been trained in an environment specifically designed for learning a single task. The learnt skills are then later reused when training the network in a more complex composite environment. A DSN can be activated by the agent to perform actions in the complex environment, based on the agent's own policy. While the paper uses a DQN architecture for the DSNs, they state that the framework can in theory function with other underlying architectures as well. The paper proposes two ways of integrating DSNs into a larger model:

The DSN array is an array of pre-trained DSNs, where each DSN is a separate DQN, trained to perform a single skill. A controller decides when to use a skill, and which DSN should be used.

The Distilled Multi-Skill Network is a single network that has been trained using policy distillation to learn multiple skills. Pre-trained DSNs are used as teacher models for the Multi-Skill Network.

When employing the DSN array the H-DRLN effectively becomes a 2-level hierarchy in which the top level network is supplied with Q-values for all DSNs as well as primitive actions as taken from the Minecraft environment. A controller module is supplied with skill policies from the underlying DSNs and either follows the skill policy of the selected skill or simply takes the primitive action selected by the top level network. The paper empirically demonstrates that this framework functions effectively and efficiently in their established environments as compared to a baseline DDQN. A concern with this approach is that this setup may have difficulties in terms of scaling as more complex environments would need a lot more DSNs in order to complete all potential tasks. This increase of DSNs could severely impact the efficiency of the H-DRLN's performance in more complex environments.

While we draw inspiration from H-DRLN and their hierarchical approach our work differs mainly by being a modular multi-level hierarchical framework in which our hierarchical structure is configured to provide a deeper understanding for the agent in how it should perform in more complex environments.

(Vezhnevets et al., 2017) describes a two-level hierarchical neural network named FeUdal Networks (or FuNs) in which the top level network, labeled the Manager, sets goals that the lower level network, the Worker, learns to reach by producing primitive actions. While relatively similar in approach to H-DRLN one interesting novelty in FeUdal Networks is the ability for the Manager to perform sub-goal discovery and learn goal states internally, while H-DRLN is supplied with goal states externally. While sub-goal discovery would be an interesting aspect to consider to take our framework closer to an end-to-end solution, for now we only consider improving our hierarchical framework in terms of modularity and depth while supplying the goal states for our agents externally.

(Song et al., 2019) describes another hierarchical approach named StarNet, that much like H-DRLN consists of a top level Manager and multiple lower level Workers each responsible for completing sub-tasks in the domain. StarNet achieves novelty by introducing Environmental Awareness into their networks. This includes a vision network to help guide the agent in a 3D environment. As StarNet is built with roughly the same foundational structure as H-DRLN our D-MuLSH framework differs as before by allowing for a multi-level hierarchical structure. It is perceivably beneficial to also introduce StarNet's Environmental Awareness for our framework in the future.

3 Background

Markov Decision Process: In a Markov Decision Process (MDP) an agent is interacting with an environment that consists of N states S, and it is in one of these states at each time step t. At each time step the agent can take an action that transitions it into a new state s_{t+1} with some probability. An MDP is defined by a 4-tuple $< S, A, T, R >$ where S is the set of all possible states in the environment, A is the set of all actions, T is the probability of transitioning from state s_t to state s_{t+1}, and R is the reward function given a state-action pair. T and R are defined such that they possess the Markov Property, i.e they are solely dependent on the state at time t and action a_t and not the whole history of states and actions.

Semi Markov Decision Process: In a MDP the state transitions occur at discrete time steps. However, as skills are a temporal abstraction they cannot be planned using MDP theory. Instead we use Semi Markov Decision Process (SMDP)s as they are a generalized version of MDPs that allow state transitions

to occur at irregular times. Using this theory, after an agent takes an action a in state s the environment remains in state s for a time d. In our framework there are only two possible values for d, defined by the set $D = \{1, k\}$ where k is the number of time-steps that skills are active for if they are activated by the agent. d is determined for each chosen action by whether it is a primitive action or a skill:

$$d = \begin{cases} 1 & \text{if } a \text{ is a primitive action} \\ k & \text{otherwise} \end{cases}$$

A SMDP is defined by a 5-tuple $< S, A, T, R, D >$ where T, R and D possess the Markov property and A in our case is a set of both skills and actions.

Reinforcement Learning: The purpose of RL is for an agent to learn to approximate an optimal policy that maximizes the expected return. This policy, defined as $\pi : S \to \Delta A$, is a mapping of states $s \in S$ to a probability distribution over actions A. In basic RL, at time t the agent observes a state $s_t \in S$, selects and action $a_t \in A$ and receives a reward r_t discounted by a discount factor γ.

The action-value function

$$Q^{\pi}(s, a) = \mathbb{E}\left[R_t | s_t = s, a_t = a, \pi\right]$$

represents the expected return of taking action a given state s based on policy π. R_t represents the return as the sum of future discounted rewards at time t defined as $R_t = \Sigma_{k=t}^{\infty} \gamma^{k-t} r_t$. The optimal action-value function Q^* is the action-value function of an optimal policy π^* which always chooses the most optimal action. This action-value function obeys the Bellman equation as defined:

$$Q^*(s_t, a_t) = \mathbb{E}\left[r_t + \gamma \max_{a'} Q^*\left(s_{t+1}, a'\right)\right]$$

Deep Reinforcement Learning: DRL is the combination of RL and Deep Learning. With DRL we can expand Q learning into a DQN that approximate the optimal action-value function by optimizing the network weights θ_i at iteration i such that the Temporal Difference (TD) error of the optimal bellman equation is minimized (Mnih et al., 2015). During training, the optimal target values

$$y^* = r + \gamma \max_{a'} Q^*(s_{t+1}, a')$$

are substituted with approximate target values

$$y = r + \gamma \max_{a'} Q(s_{t+1}, a' : \theta_i^-)$$

using parameters θ_i^- from some previous iteration.

Experience Replay Memory (ER): ER is a technique for offline learning in which the agent's experiences at each time step, $e_t = (s_t, a_t, r_t, s_{t+1})$, is stored in a data set $D_t = \{e_1, ..., e_t\}$. During Q-learning updates these experiences are sampled following some normal distribution. (Tessler et al., 2016) proposes a modified version of Experience Replay in their paper called Skill-Experience Replay (S-ER). The S-ER can store skill trajectories as a transition over k time-steps, from s_t to s_{t+k} along with the discounted reward $\tilde{r}$. Thus, it stores the skill tuple $(s_t, \sigma_t, \tilde{r}_t, s_{t+k})$, where σ_t is the skill executed by the agent at time-step t. Additionally, the S-ER uses Prioritized Experience Replay as described in (Schaul et al., 2016).

Double Deep Q-Learning: The max operator in the DQN algorithm uses the same values to both select and evaluate an action. This leads to overoptimistic value estimates, as the selected values are most likely overestimated. To mitigate this problem, DDQN decouples the selection from evaluation. Action selection is done using the current network parameters θ, and action evaluation is done using the target network parameters θ'. This new target can then be written as:

$$y_t^{DDQN} = R_{t+1} + \gamma \cdot Q\left(s_{t+1}, \max_{a'} Q\left(s_{t+1}, a'; \theta_t\right); \theta_t'\right)$$

as opposed to the original target of:

$$y_t^{DQN} = R_{t+1} + \gamma \cdot \max_{a'} Q\left(s_{t+1}, a'; \theta_i^-\right)$$

(Hado van Hasselt and Silver, 2015).

Skills: (Sutton et al., 1999) defines a skill (also called an option) as an algorithm that is capable of performing temporally extended actions. A skill σ consists of three components: an action policy π, an initiation set $\mathcal{I} \subset S$, and a termination condition $\beta : S^+ \to [0, 1]$. $\mathcal{I}$ is the set of states where the skill can be initiated, and β is the set of states where the skill will be terminated. A skill-based agent performs actions according to a skill policy $\mu : S \to \Delta_{\Sigma}$ which maps states to a probability distribution over skills.
The skill-value function

$$Q(s, \sigma) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R_t | (s, \sigma), \mu\right]$$

represents the expected return of selecting skill σ in state s based on skill policy μ. The skill reward $R_s^{\sigma} = \mathbb{E}\left[r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{k-1} r_{t+k} | s_t = s, \sigma\right]$ is the discounted reward that a skill accumulates over k time-steps.
The optimal skill-value function

$$Q_{\Sigma}^*(s, \sigma) = \mathbb{E}\left[R_s^{\sigma} + \gamma^k \max_{\sigma' \in \Sigma} Q_{\Sigma}^*\left(s', \sigma'\right)\right]$$

is the skill-value function of the optimal skill policy μ^*.

4 Methodology

D-MuLSH architecture: Figure 2 shows the D-MuLSH architecture of the top level of our framework, with DSNs abstracted away by the pink box in the architecture. This architecture takes as input game states from the environment, and provides outputs that correspond to either primitive actions $(a_1, a_2, ..., a_n)$ or DSNs $(DSN_1, DSN_2, ..., DSN_m)$. Each DSN is itself a pre-trained network with its own policies that either execute primitive actions in the environment or activates another DSN situated in a lower level in the hierarchy. The set of primitive actions directly available from the top level Q-network (A_{top}) is a subset of the complete action space A of the environment $(A_{top} \subseteq A)$. A controller unit is responsible for determining the final policy for our agent based on the Q-values of the output layer. As seen in the box labeled *Control* in Figure 2, if the greatest Q-value in the output layer constitutes a primitive action then the final policy is set to execute the action with the highest Q-value by:

$$\pi(s) = max\left(Q(s, a_1), ..., Q(s, a_n)\right).$$

Otherwise, if the greatest Q-value constitutes a DSN, the controller sets the policy π to follow the supplied skill policy of the DSN with the greatest Q-value as defined by:

$$\pi(s) = max\left(Q(s, DSN_1), ..., Q(s, DSN_m)\right).$$

This means that if the greatest Q-value in the output layer is $Q(s, DSN_2)$ then the controller sets the policy $\pi(s)$ to follow the supplied skill policy $\pi_{DSN_2}(s)$.

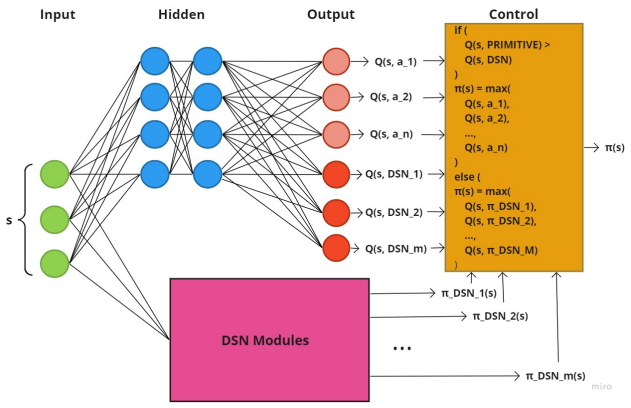


Figure 2: Architecture of the top level of the framework

If the controller determines to execute a primitive action a_t at time step t, then this action is executed directly in the environment for a single time step. If the controller determines to use a DSN at time step t then control of the agent is handed over to the policy of the chosen DSN $(\pi_{DSN_i}(s))$ for k time steps. The DSN modules output their respective skill policies given the game state from the input layer.

Deep Skill Networks: Figure 3 illustrates the DSN architecture. Each skill network is given input in the form of game states from the environment. Simple Skill Networks (SSN) use this input to generate simple skill policies $(\pi_{SSN_1}(s))$ with outputs corresponding to a set of primitive actions $(a_1, a_2, ..., a_i)$. Each SSN can have access to a different configuration of primitive actions from the overall environment action space, meaning that the action space of a SSN A_{SSN} is a subset of the complete action space A of the environment $(A_{SSN} \subseteq A)$.

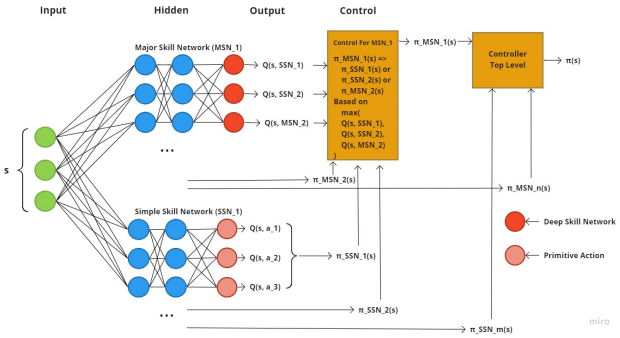


Figure 3: DSN modules architecture

Major Skill Networks (MSN) decide which of their linked DSNs should be given control based on the current input. MSNs can be linked to both SSNs as well as other MSNs. Each MSN contains a controller unit that is supplied with the policies of the relevant DSNs, and uses its own policy to choose which sub-skill to activate. As seen in the box labeled *Control For* MSN_1 in Figure 3, the policy of the MSN (π_{MSN_1}) is set to follow one of the supplied skill policies based on the Q-values of its output layer. The final skill policy π_{MSN_1} is supplied to the controller at the top level of the hierarchy together with policies from other DSNs connected to the top level. The top level controller will itself decide which policy to follow as explained earlier.

MSNs do not have outputs that corresponds to primitive actions, and can only affect the environment through the use of other skill networks. When a controller decides to execute a skill policy, control of the agent is handed over to the DSN for one time step, after which

the policy of the MSN will choose another skill or relinquish control back to the top level controller.

Hierarchy Configuration: A powerful aspect of the D-MuLSH framework is the modularity of its hierarchy configuration. In contrast to H-DRLN which employs a 2-layer hierarchy solely using DSNs equivalent to our SSNs, D-MuLSH employs a highly modular multi-level configuration using both SSNs and MSNs. This difference is visualized in Figure 4 where example configurations for both H-DRLN and D-MuLSH is presented.

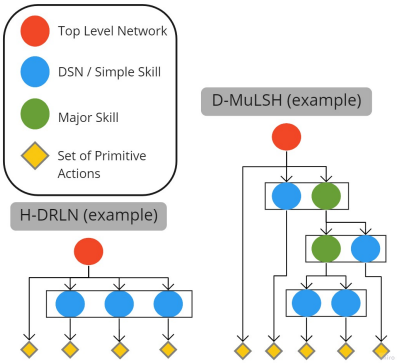


Figure 4: Example of how the hierarchy can be structured in the H-DRLN and the novel architecture. D-MuLSH allows an arbitrary number of levels in the hierarchy.

The configuration of the D-MuLSH framework can by following certain structural rules be customized to fit to an agent's needs. The structural rules are as follows:

- The controller at the top level can be supplied with policies from any type of DSN situated in the layer directly below.
- The top level network can output a subset of primitive actions without accessing a DSN.
- Any MSN can be supplied with policies from any DSN situated in the layer directly below.
- Any SSN can execute a subset of primitive actions in the environment.
- MSNs cannot execute primitive actions.
- SSNs cannot be supplied with policies from other DSNs.

5 Experiments

We train 2 major skills and 6 simple skills. The two major skills are navigation and combat. Navigation governs four simple skills: navigating narrow corridors, navigating open spaces, walking around obstacles, and picking up items. Combat governs the remaining two: shooting enemies and dodging enemy attacks. We choose these particular skills, as they cover many of the tasks necessary to play the original Doom game. Figure 5 shows the specific configuration of the D-MuLSH framework used in experiments.

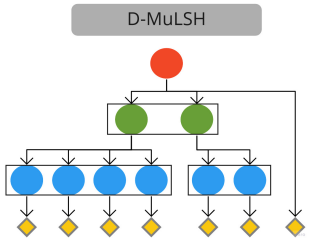
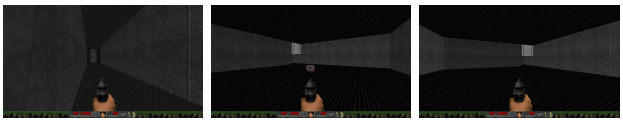
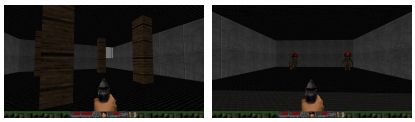


Figure 5: The hierarchy configuration used in experiments.

Simple skill domains: The four simple navigation domains, shown in Figure 6 (a-d) are inspired by the domains detailed in (Tessler et al., 2016). The agent starts in a random spot, facing a random direction and must reach the goal, using only 3 actions: walking forward, turning left and turning right. All four domains give the agent a small penalty at each time step to encourage speed, and a positive reward when the goal is reached.
In the shooting domain (Figure 6e), the agent must turn to face enemies and shoot them. The goal is reached when the agent has killed both enemies. Enemies cannot attack or move, and they die in one hit. The agent is rewarded every time it kills one. There is a small penalty at every step to encourage speed, and every shot that doesn't hit is penalized as well.



(a) Corridor (b) Item pickup (c) Open room



(d) Obstacle room (e) Combat

Figure 6: Environments used for training simple skill networks. Environments (a)-(d) are used for navigation and (e) is used for combat.

The dodging domain uses the same room as the shooting domain. Here the agent must dodge incoming attacks from multiple enemies using only movement

actions (moving forwards, backwards, left and right). The goal is to survive for 100 time steps. The agent is given a small reward at every time step in order to encourage long survival time, and there are penalties for taking damage and dying.

Major skill domains: The complex navigation domain, shown in Figure 7, has the agent run through the four simple domains in sequence. Just like the simple navigation domains, the agent is given a small penalty at every time step, and is rewarded for successfully exiting each room and picking up the item. It must pick up the item before exiting the final room in order for the episode to count as successful.

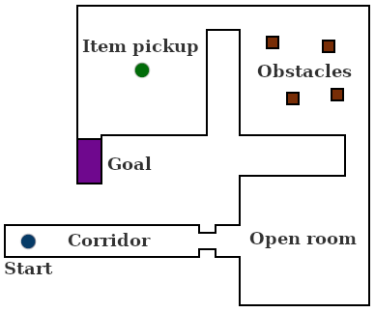


Figure 7: Map of complex navigation domain

The complex combat domain uses the same room as the simple combat domains (Figure 6e), with some exceptions. There is only 1 enemy present at any time. The enemy can move and attack as normal, and it takes several shots to kill. When an enemy dies, another one is spawned in the room. The goal is to kill 3 enemies within a time limit of 375 time steps and without dying or running out of ammunition. The agent is rewarded only when an enemy is killed. There is a small penalty at every time step, and a large penalty for taking damage.

The complex domain: This domain combines the navigation and the combat domains into one, as shown in Figure 8. The goal is to get to the exit after killing all enemies and picking up the item. Two enemies spawn when the agent enters the combat room. They each take multiple shots to kill. When they are killed, the door to the next room appears. The agent is rewarded every time it exits a room, when it kills an enemy and when it picks up the item. There is a penalty for taking damage and for shooting when the enemies are dead. The agent is also penalized at every time step when there are no enemies present.

Evaluation Process: The agents are evaluated during training after each epoch (10000 optimization steps) using the current network weights. During evaluation

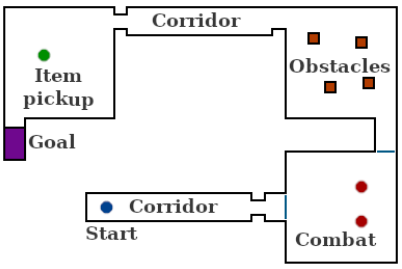


Figure 8: Map of the complex domain

the agent's performance in terms of *success percentage* is averaged over 50 episodes, where success percentage is the percentage (%) of episodes in which the agent completes all sub-tasks and reaches the exit.

Training Simple Skill Networks: To train the different SSNs we use the vanilla DDQN algorithm (Hado van Hasselt and Silver, 2015). We modify the original hyper-parameter settings to include less exploration (epsilon decreased from 1.0 to 0.1 over 100K time-steps) and a smaller experience replay memory size of 75000 tuples[3]. The SSNs are trained in their respective environments (listed in Figure 6) with both combat related SSNs using the same environment. Within 100 epochs (1M optimization steps) all agents achieve success percentages around 100%, as seen in Table 1.

Domain	Success Percentage	Epoch
Corridor Navigation	100	31
Open Navigation	100	45
Obstacle Navigation	96	91
Item Navigation	100	15
Combat (Shooting)	100	7
Combat (Dodging)	100	80

Table 1: SSN training results. Lists success percentages for each skill network as well as in which epoch highest percentage was first reported.

Training Major Skill Networks: MSNs were trained in the complex domains as specified using only the pre-trained SSNs as available actions. The major skill of navigation was trained in the complex navigation domain using all four navigational SSNs for a total of 200 epochs. A graph of the agent's evaluation scores is seen in Figure 9. The agent is able to achieve a success percentage of **40%** already after the very first epoch, and achieves its highest success percentage of **98%** after 150 epochs. Although it takes a relatively

[3]The replay memory size reduction was a necessity to accommodate training on our own lower-end machines.

long training time to reach its highest score, it manages to achieve a **96%** already around the 50 epoch mark. In comparison, a vanilla DDQN baseline was trained in the same environment for 50 epochs without achieving a single successful episode.

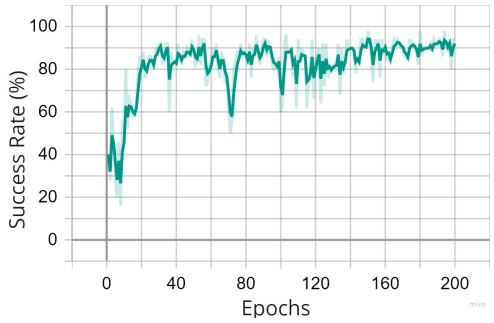


Figure 9: Success percentages for navigation MSN during evaluations.

The major skill of combat was trained in the complex combat environment using the two combat related SSNs for 40 epochs. The graph in Figure 10 shows that the agent achieved its highest success percentage of **96%** at epoch 20. The graph also demonstrated a highly unstable learning process. This instability may in part be caused by the inconsistencies of the Doom AI, as the aggressiveness of the enemies change between episodes. A longer training time may have led to a more consistent success rate. A vanilla DDQN baseline that was trained in the same environment for 100 epochs managed to achieve a success percentage of **4%**.

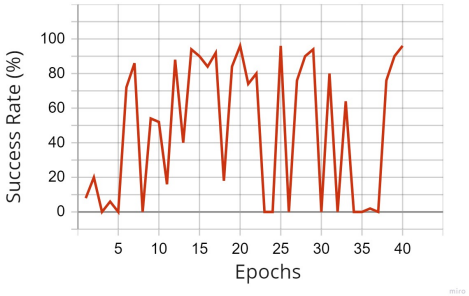


Figure 10: Success percentages for combat MSN during evaluations.

Training a D-MuLSH: The D-MuLSH agent was trained in the complex domain using both MSNs as well as the primitive actions of: moving forward, turning to either side and shooting. The agent was trained for 100 epochs, in which it reached its first successful episode at epoch 10, and its highest success percentage of **96%** at epoch 73. In comparison, an agent using H-DRLN that trained in the same environment for an equal 100 epochs achieved its highest success percentage of **50%** at epoch 88, and its first successful episode at epoch 40. A DDQN baseline failed in achieving a single successful episode throughout the 100 epochs. The graph in Figure 11 shows the evaluation scores of all agents during their respective training sessions.

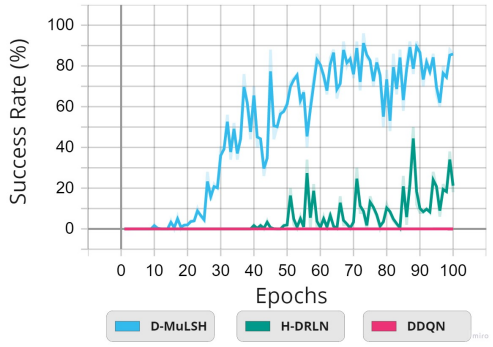


Figure 11: Success percentages for the final environment.

The graph in Figure 12 depicts the usage (%) of skills as opposed to primitive actions during evaluations of both D-MuLSH (blue) and H-DRLN (green). We can observe that the D-MuLSH agent exhibits a spike in skill usage in the first 10 epochs and afterwards levels out to a steady usage of about 25%. This observation indicates that using only skills to solve new environments would lead to sub-optimal solutions, caused by the skills themselves being sub-optimal in unseen environments. In our framework the agent is able to use skills in an exploratory fashion because of the epsilon-greedy approach used by our network. This exploration with skills allows our agent to lean on the pre-trained skills for a faster convergence towards an optimal solution. Once the algorithm allows for more exploitation the agent learns to use primitive actions to refine the skills to better suit the new environment. In contrast H-DRLN starts with almost no skill usage, and steadily increases usage to match ours at the end of training. This may be a cause for H-DRLN's slow start to learning in this environment.

6 Discussion

We have presented D-MuLSH, an extended version of the H-DRLN. Our framework contains all of the building blocks necessary for a truly lifelong learning framework, including: Efficient knowledge retention through Simple and Major Skill Networks; Selective transfer of knowledge using reusable skills; The effective and efficient retention and transfer of knowledge through a

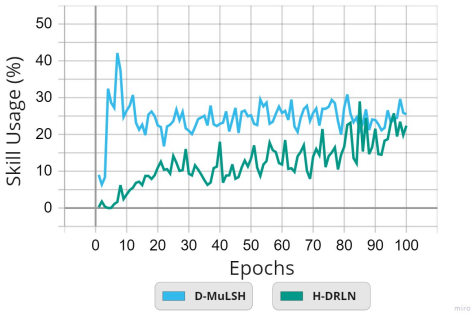


Figure 12: Skill usage during evaluations.

modular multi-level skill hierarchy with multiple levels of control units.

While the D-MuLSH framework does increase the overall time needed to pre-train skills, our experiments show that it is only a small increase. The overall time needed to train both the Navigation and the Combat MSNs to at least 90% success rate was a little under 1 hour and 30 minutes. Conversely, it also decreases the time needed to train the Q-network at the top level significantly. Both D-MuLSH and H-DRLN needed about 4 hours and 40 minutes[4] to train for 100 epochs, and the trend shown in Figure 11 suggests that the H-DRLN would need several more training-hours to reach a similar success rate to D-MuLSH.

Interesting property of MSNs: Both H-DRLN and D-MuLSH hands over control to a skill network for k time steps when they activate a skill. However, if a MSN is activated by D-MuLSH, the MSN will remain in control, rather than relegating control to one of its sub-skills. This means that the MSN can switch which sub-skill it uses during the k time steps.

The H-DRLN cannot do this, as its skills do not have sub-skills. This may be part of the reason for why D-MuLSH performs better than the H-DRLN. For example, one activation of the combat MSN in D-MuLSH can potentially be used to both dodge an attack and shoot at an enemy within k time steps, whereas the equivalent approach for the H-DRLN would be to first use k steps to dodge and then another k steps to shoot.

7 Future work

It may be beneficial to investigate the effect of updating network weights on skills. As of writing, we only update weights for the Q-network at the top level. That means that MSNs and SSNs do not update during training. However, this should make skills better at adapting to new environments, making it easier for the agent to generalize what it has learned.

[4]Both agents were trained on the same machine.

Bibliography

Hado van Hasselt, A. G. and Silver, D. (2015), 'Deep reinforcement learning with double q-learning', *Association for the Advancement of Artificial Intelligence* .

Lample, G. and Chaplot, D. S. (2018), 'Playing fps games with deep reinforcement learning'.

Mnih, V. et al. (2015), 'Human-level control through deep reinforcement learning', *Nature* .

Parisotto, E. and Salakhutdinov, R. (2018), 'Neural map: Structured memory for deep reinforcement learning'.

Schaul, T., Quan, J., Antonoglou, I. and Silver, D. (2016), 'Prioritized experience replay'.

Silver, D., Yang, Q. and Li, L. (2013), Lifelong machine learning systems: Beyond learning algorithms.

Song, S. et al. (2019), 'Playing fps games with environment-aware hierarchical reinforcement learning'.

Sutton, R. S., Precup, D. and Singh, S. (1999), 'Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning', *Artificial Intelligence* **112**(1), 181–211.

Tessler, C., Givony, S., Zahavy, T., Mankowitz, D. J. and Mannor, S. (2016), 'A deep hierarchical approach to lifelong learning in minecraft'.

Vezhnevets, A. S. et al. (2017), 'Feudal networks for hierarchical reinforcement learning'.

Wu, Y. and Tian, Y. (2017), 'Training agent for first-person shooter game with actor-critic curriculum learning'.