Department of Computer Science
Aalborg University
http://www.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
Deriving Subgoals Using Network Distillation

Theme:
Master's Thesis

Project Period:
Spring Semester 2020

Project Group:
MI1017f21

Participant(s):
Aryan Landi
Lars Svane Jensen
Nikolaj Ljørring

Supervisor(s):
Chenjuan Guo

Page Numbers: 9

Date of Completion:
june 11, 2021

Abstract:

Sparsely rewarded environments can be challenging for deep reinforcement learning to understand and even harder to master. Hierarchical reinforcement learning shows promising ways of constructing subgoals, that are more understandable to the agent. Subgoal construction is a slow process to do autonomously, we therefore propose a new method of finding and constructing subgoals. We present a more time-efficient comparison method for subgoal creation. We propose a novel distributed training framework to increase the throughput of the agent. The framework indicates increased data gathering but decreased learning compared to a non-distributed counterpart.

Deriving Subgoals Using Network Distillation

Aryan M. Landi, Lars S. Jensen, Nikolaj. Ljørring
Department of Computer Science
Aalborg University
Denmark

***Abstract*—Sparsely rewarded environments can be challenging for deep reinforcement learning to understand and even harder to master. Hierarchical reinforcement learning shows promising ways of constructing subgoals, that are more understandable to the agent. Subgoal construction is a slow process to do autonomously, we therefore propose a new method of finding and constructing subgoals. We present a more time-efficient comparison method for subgoal creation. We propose a novel distributed training framework to increase the throughput of the agent. The framework indicates increased data gathering but decreased learning compared to a non-distributed counterpart.**

I. Introduction

Many new Deep Reinforcement Learning (DRL) algorithms and methods focus on doing well in both dense and sparse reward games. As mentioned in previous works, the challenge is to construct algorithms that will be able to succeed in both types of games, without hand engineering features or using underlying game data[17].

Two separate ideas for increasing the state space exploration are the hierarchical reinforcement learning (HRL) algorithm Exploration Effort Partition (EEP)[8], and Random Network Distillation (RND)[7]. Since EEP is hierarchical, the main idea is to define meaningful subgoals through training. Solving these subgoals, also called partitions in EEP, should then solve the overall goal. However, a limitation of EEP is that the partitions are determined through a distance measure on states. This distance does not take into account the underlying information in the game, relying only on the pixel data and internally calculated action probabilities between two states. Later on, we will also see that computing this distance is very time consuming. Furthermore, the original EEP algorithm does not provide a framework for distributed environment interactions, something that can increase state space exploration.

RND computes the novelty of a state based on what it has previously seen. Agents taking advantage of this algorithm should then try to explore these novel states further. However, since exploration is guided only by the novelty value of a state, the approach lacks the possibility of creating groups/partitions of states with similar novelty.

Beyond this we observe that several state of the art solutions within the field are dependent on specialized hardware solutions, which in many cases are out of reach. Papers like Agent57 [2], R2D2 [12], and NGU [3] process not only more updates every second but also more environment interactions per actor with higher amounts of actors. Even the transitions they process at higher speeds are of longer sequences and higher complexity, than what our resources allow us to process.

These factors create a disparity between not only competing for state of the art performance but also, hinder the reproducibility of such contributions.

We can use the limitations of EEP as a stepping stone for defining our problem.

First we would like to decrease memory used by the EEP algorithm, by decreasing the size of each entry in the replay memory.

Second, we would like to increase the speed of EEP (decrease the time complexity) by replacing the state distance calculation with a novelty calculation instead.

Lastly we would like to distribute our work in the continued legacy of R2D2, Agent57 etc. to use the advantages that distribution entails.

Combining the above mentioned ideas will result in an efficient, low hardware requirement, yet competitive agent for solving the Atari reinforcement learning test suite with good performance in both reward sparse and dense games.

We expect that the performance of RND alone and partitions from EEP will compliment each other. While the distribution, which is decoupled or wrapped around both RND and EEP, will enable higher throughput without hindrance or overhead.

II. Related Work

The work done in this article relates to subjects such as *active exploration* and *efficient training*.

A. Active exploration

Active exploration is the act of having the agent explore by its own will, this can be done in multiple ways.

Count-exploration is one of the ways to encourage active exploration. By counting the number of times an agent has been in a given state, it can be encouraged to explore rarely visited areas of an environment [6] [14] [23]. This has been done using clustering techniques, where states are clustered together to reduce space complexity[1]. Counting can be based on uncertainty factors to determine the visitation rate of a state [15].

Some methods are more curiosity-driven, where models are made to predict the environment or use the error of models to determine importance. In the work by Yang et al.[24] they calculate optical flow estimation and use the error as a novelty. Within this category of active exploration, there are studies on how to minimize local exploration and encourage agents to find diverse states. [11] The noisy-tv problem can be handled by ensuring that there are several steps between two curiosity monuments as done by Savinov et al. in [20].

We will be using both techniques to create a clustering system, using the error of neural network models as the uncertainty factor and counting the visits to these clusters.

B. Efficient Training

When we develop complex software, we concern ourselves with how fast it can return the right results and how much space is needed to do so. The same is said when training an agent, there is often a tradeoff between these two factors. DQN is fast but allocates lots of memory in its replay memory [17]. A3C is fast and does not use much memory due to its distribution where each actor has the minimal memory needed[16]. Some methods find importance in the memory and prioritize the gathered training data e.g. "Prioritized Experience Replay" by T. schaul et al.[21]. Newer methods focus on speed with a shared prioritized memory. Agent57 and MuZero are the two of the frontrunners by both distributing actors to gather data faster and by prioritizing the data which lets the learner train on the most relevant information[2] [22].

In this article, we will be using multiple actors that hold minimal information, combined with a memory manager that can make some of the simpler calculations of sampling, to the learner, so that that individual workers can focus on their specific task.

C. Hierarchical Reinforcement Learning

Hierarchical reinforcement learning is often used to create subgoals and/or abstactions for the agent and bring a hierarchy to the task at hand. Jean Harb et al.[10] introduce the Asynchronous Advantage Option-Critic (A2OC) algorithm that learns options with multiple parallel agents as in A3C. Harb et al. describe options as temporally abstracted actions with termination conditions. The hierarchy can be thought of as team cooperation, in [25] where multiple agents learned the hierarchy of their high-level combined actions from their low-level autonomous actions and are rewarded for the team effort.

Our approach will focus more on the progression of the visual aspect of the game, having its hierarchy linked to the progression of what is seen on screen and not on what the specific order of taken actions is.

III. Preliminaries

A. Markov Decision Process

The Markov Decision Process (MDP) is a common mathematical control process that represents the decision process in an environment [9].

An MDP can represent most Reinforcement Learning problems as the tuple (S, A, P, R), where S is the set of states, A is the action set, transition probabilities represented by P and the reward function mapping states and action to rewards $R : S \times A \rightarrowtail \mathbb{R}$ [19]. A state $s \in S$ consists of values that uniquely identify a snapshot of the environment at a given moment. An action $a \in A$ represents interactions with the environment. Taking an action a in a state s will lead to a new state s' in deterministic environments, and to a set of states $\mathbb{S} \subseteq S$ in nondeterministic environments. If circular transitions are part of the environment, the new state s' can be equal to the current state s in the deterministic case, and $s \in \mathbb{S}$ for nondeterministic environments. $P(s'|s, a)$ is the probability of transitioning from state s to s' by taking action a[9][19].

B. The Bellman Equation

The Bellman Equation is a basic principle used when working with reinforcement learning, Equation 1 shows the decomposed form [4].

$$Q(s, a) = \mathbb{E}[r + \gamma \underset{a'}{max} Q^*(s', a'; \theta)|s, a; \theta] \tag{1}$$

The equation describes how to calculate a Q value from a state s and an action a. The Q-value is the known reward r added to the future discounted rewards, calculated using the next state action pair (s', a') and the network with parameters θ [18].

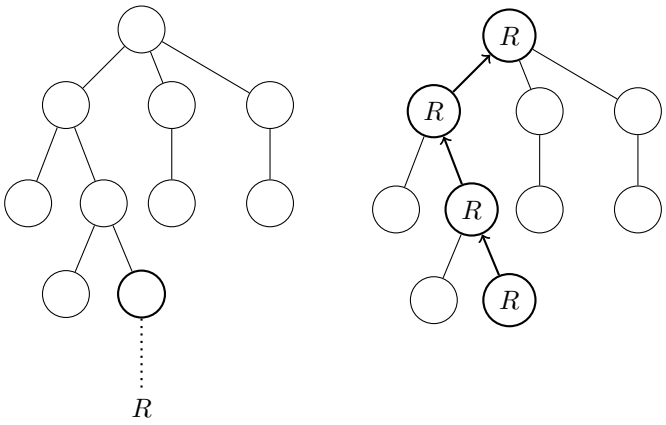
This equation can be used to calculate the Q-target used to backpropagate a network.

C. Mixed Monte Carlo

We use the Mixed Monte Carlo technique when training our networks, because it is a simple addition to the backpropagation technique used by DQN, using more of the sampled information to increase training speed.

The intuition behind this addition is that some of the training can be done similar to how Monte Carlo Tree Search uses its simulation phase where each state-action pair can be thought of as a node in the search tree. The state where a game terminates can be thought of as the leaf node simulating the end result for this branch of (s, a) tuples, as illustrated in Figure 1a [6].

The rewards for each (s, a) are backpropagated to the root/start state, as it would normally be done in Monte Carlo search (Figure 1b) and saved at each state as a Monte Carlo Reward (MCR)[6].

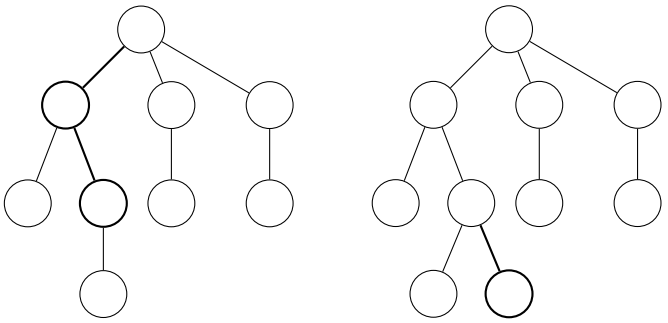


(a) Simulation (b) Backpropagation

Figure. 1: Monte Carlo simulation and backpropagation

A run through the environment from start to termination can be compared to the selection and expansion phases, where

each walk-through will have slight variances, as illustrated in Figure 2. At each walk-through, there is a selection on which action to take. If this action leads to some unknown branch, we expand and add the branch to the tree.



(a) Selection (b) Expansion

Figure. 2: Monte Carlo selection and expansion

We calculate the normal Q-target using the Bellman equation, and combine it with MCR to Mixed Monte Carlo (MMC) target, using a coefficient η. The coefficient η is used to determine the importance of known (MCR) and self-learned results (Qtarget) as seen in Equation 2. By using this technique and combining the Monte Carlo Reward with the more traditional target, the learning yield more information. The additional information comes from the practical rewards given in an episode and backpropagated to the point of training. This practical understanding of the future reward helps the theoretical understanding provided by a target network, giving a better target for the networks' backpropagation[6].

$$MMC_{target} = (1 - \eta) * Qtarget + \eta * MCR \qquad (2)$$

The Mixed Monte Carlo target can then be used against an agent's prediction to calculate loss and backpropagate the agent's neural networks. This method of training makes it easier for the agent to remember rewards over sparsely rewarded environments[6].

IV. Methodology

A. The framework

The distributed framework we propose to use is inspired by Agent57 [2] and Recurrent Replay Distributed DQN [13], with the 3 different sectors, the actor, learner and memory manager as seen on Figure 3. The distribution allows for decoupling of learning and acting, which has been leveraged for performance, throughput and model stability among others, and has been leveraged by many within the field of reinforcement learning.

This framework is intended to have multiple actors, one learner and a memory manager. The actors interact with their own environment to create data that can be used for training. The learner only trains the networks and is able to run through data faster than an agent, doing both acting and learning. The multiple actors are intended to both fill the replay memory fast and provide some stochasticity to the training data by taking paths that are slightly different through the environment. The stochasticity is provided by the agent's ϵ-greedy action choice that is semi-random according to the declining ϵ-value as presented by V. Mnih et al.[17].

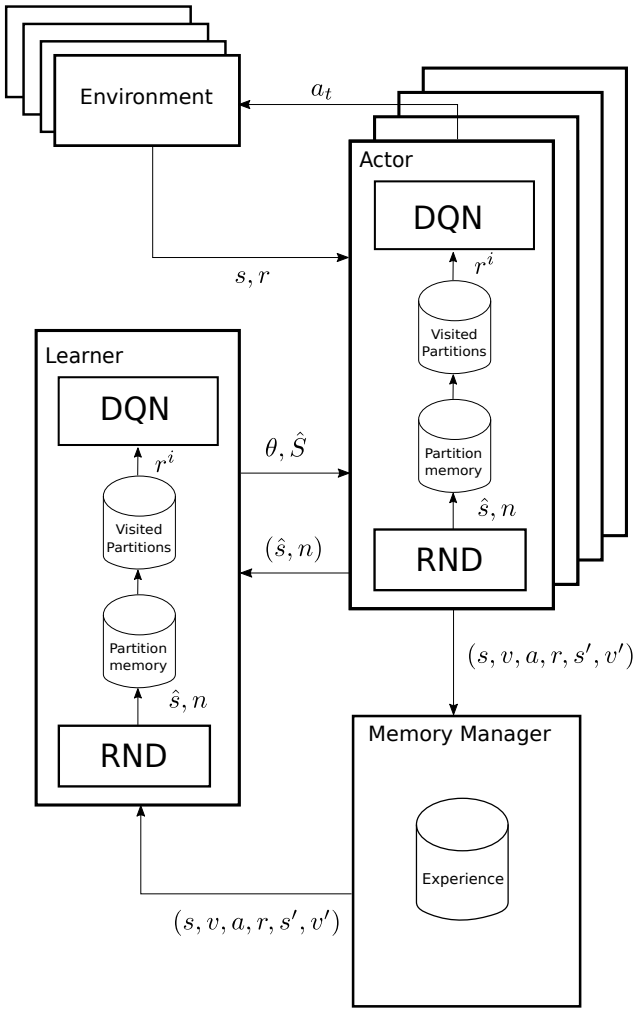


Figure. 3: The distributed framework with RND

The framework allows for multiple instances of the environment, to be interacted with by *actors*. Every experience gained by actors interacting with the environment, is sent to the replay memory and stored by the manager in the form: (s, v, a, r, s', v'). The manager selects and sends the experiences for the learner to train on. The learner uses the experiences to update the agent's networks. Each actor sends a partition candidate along with its novelty value to the learner. The learner uses the novelty values to choose the best partition candidate, adding the candidate with the highest value to the partition memory. The learner then updates the partition memory of the actors as well as their network parameters, represented by $\theta, \hat{S}$ on Figure 3.

B. Partitions

We start this subsection by giving a formal definition of the partitions derived from the agents interacting with the environment:

Definition 1 (Partitions)**.** A partition $p \subset S$ is a set of states that are close together as determined by a distance measure. Each partition p has a representative state $\hat{s_p} \in \hat{S}$. Furthermore, the partition p to which a state belongs is the one whose representative state $\hat{s}_p$ is closest[8].

Also, a partition is a tuple containing the representative state of the partition and a visited counter containing the number of times the partition has been visited.

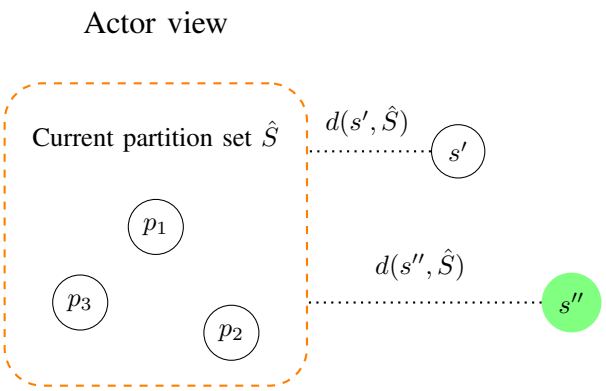


Figure. 4: Partition candidate construction as seen by the actor.

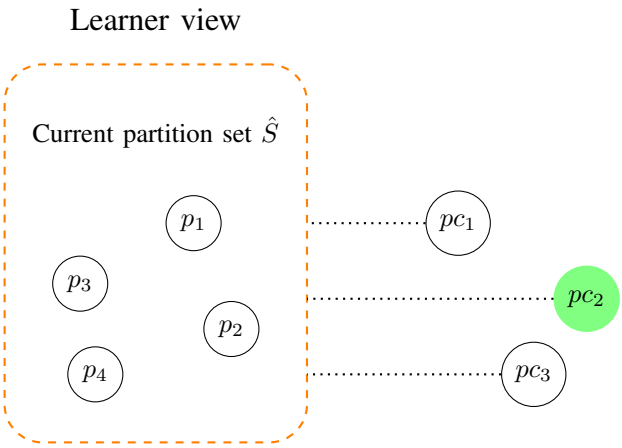


Figure. 5: Construction of partition set by the learner from partition candidates at frequency ϕ. In this example, each partition candidate originates from a different actor.

1) Partition Construction: Partitions are constructed at a frequency ϕ from a set of partition candidates and added to the partition memory. The partition candidate from an actor is the state with the most distance to all other partitions within the partition frequency ϕ. We illustrate partition candidate construction with an example in Figure 4. In this example, the actor visits new states s' and then s''. If the distance from s' to the current partition set $\hat{S}$ is larger than a threshold D_{max}, s' will be added as a partition candidate and $D_{max} = d(s', \hat{S})$. When the actor later visits s'' the partition candidate will be overwritten with s'' and $D_{max} = d(s'', \hat{S})$, since $d(s'', \hat{S}) > d(s', \hat{S})$. Note that for any state s, $d(s, \hat{S}) > D_{max}$ if and only if $\forall \hat{s} \in \hat{S}\ d(s, \hat{s}) > D_{max}$.

In Figure 5 we illustrate what the learner sees when it adds new partitions. In this example, the learner has four partitions in its partition memory, and has received three partition candidates pc_1, pc_2 and pc_3 from three different actors. We see that the learner will add pc_2 as the new partition, since it is furthest away from the current partition-representation set $\hat{S}$.

For each partition that is added to the partition memory, ϕ is increased by a variable percentage value to increase the distance between partitions over time. This ensures more variation in the partitions, increasing the exploration of the environment.

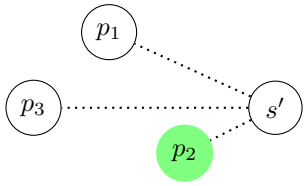


Figure. 6: Partition visitation. In this example, the agent reaches a new state s'. The agent will then calculate the distance between s' and the representative state $\hat{s_p}$ of each partition p, and visit the partition with the minimum distance, which is p_2 in this example.

2) Partition Visitation: As illustrated in Figure 6, a partition is visited when the agent is closest to that partition's representative state. We do this by using Equation 5 to determine the distance between states, visiting the partition that gives the minimum of all the distances.

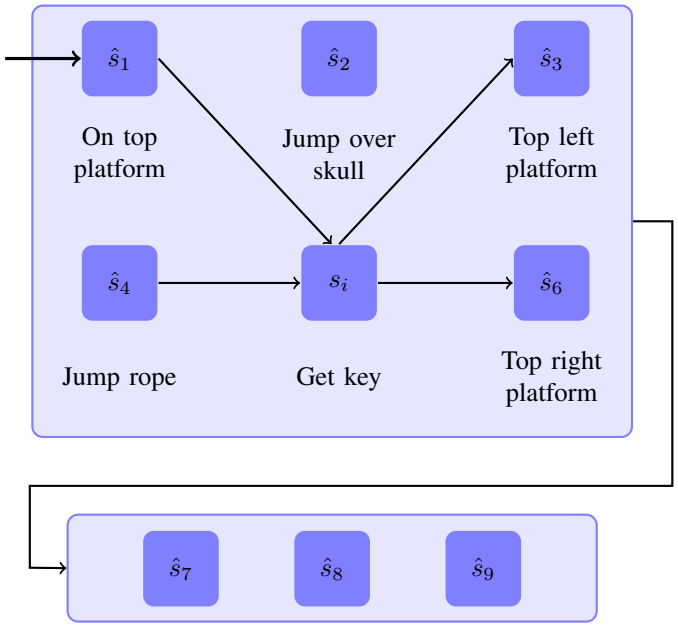


Figure. 7: An example partition hierarchy as seen by the agent when playing *Montezuma's Revenge*. Each level (big surrounding box) includes up to several subgoals/partitions (small boxes) that helps the agent to evaluate which areas to explore in order to solve the overall environmental goal.

We depict the hierarchical partition visitation structure in Figure 7. As the agent begins interacting with the environment,

it starts in an initial state $\hat{s_1}$. Afterwards the agent will randomly explore the environment, adding new partitions as they are observed.

When the agent starts to act based on its Q-network, it should receive rewards for exploring the partitions. At some point the agent should learn to reach the key from either $\hat{s_1}$ or $\hat{s_4}$ and get a small intrinsic reward for reaching either of these partitions. This should lead to a new area for the agent to explore so that it after some time should be able to find the extrinsic reward for finding the key. When the key is found the agent can exit the room through the top left platform $\hat{s_3}$ or top right platform $\hat{s_6}$. Reaching partitions such as $\hat{s}_3$ and $\hat{s}_6$ lets the agent find new areas to explore, and should lead to the agent exiting the level.

C. Distribution, performance and stability

The following will describe the distribution of the framework, which is used to increase throughput of experiences to replay memory and stability via stochasticity.

Different processes communicate using queues, the queues will work according to FIFO. In the final distributed implementation we have nine queues.

Generally there are three types of queues:

- *Network queues* pass network parameters from the learner to the actors, updating their networks to match the learners.
- *Partition queues* pass partitions between actor and learner.
- *Replay queues* pass experiences from actors to the manager.

1) Actors: Each actor will interact with the environment and add sets of experiences, episode buffers, to the replay queue. This is done whenever a state has become terminal and the actor will start a new episode. It should be noted that an episode is deemed terminal when a life is lost, even when the actor has more lives left within the environment. This is done to discourage risky strategies optimizing score over playing the game as intended. At every network update frequency step, the actor will check if anything has been put into the network queues and update accordingly. The agent will also check for selected partitions. Finally, at every ϕ step, the actor will push its current partition candidate as descirbed in subsubsection IV-B1 to the learner.

2) Manager: The manager is a simple middleman between actors and learner. It unwraps and stores the episode-buffer's transitions from actors to the replay-memory, and samples them for consumption by the learner.

3) Learner: The learner has a replay memory managed by the manager. The learner will wait until its replay memory has been filled by the manager at which point it will start to process the transitions. Processing consists of creating batches of experiences, calculating a huber loss between the network predictions and the MMC_{target} and backpropagating the value. Once it has processed the entirety of the memory buffer it will again wait for the buffer to be filled and repeat. Whenever the entirety of the buffer has been processed it will also push the new network weights to the network queues for consumption by actors. Furthermore, when every actor has submitted two partition candidates to the learner, it will evaluate all candidates. The evaluation is based on the candidates' distance to $\hat{S}$ when discovered. When the partition candidate is selected as a partition it is pushed to all actors.

D. RND as a novel distance measure

Our distance measure is inspired by the distance equation by M. Dann et al.[8] and the novelty measure Y. Burda et al. presented in [7].

In [8] they use an exploration effort network to determine the partition s_{p_i} the agent is in, using Equation 3.

$$s_{p_i} = argmin_{s_{p_i} \in \hat{S}}(\max_{\hat{s} \in \hat{S}}(max(||EE^{\pi}_m(\hat{s}, s) - EE^{\pi}_m(\hat{s}, s_{p_i})||, ||EE^{\pi}_m(s, \hat{s}) - EE^{\pi}_m(s_{p_i}, \hat{s})||)), \quad (3)$$

where s is the state the agent is in, $\hat{S}$ is the partition reference states, π is the policy and m is the time limit in steps.

M. Dann et al. uses a replay memory where they store $\{s, v, a, \hat{r}^{\pi}, r, s', v'\}$ for each transition. Additionally to the states and action sets normally stored in this replay memory they store the visited partitions v, the auxiliary reward set $\hat{r}^{\pi}$ according to the current policy π and visited partition after the transition.

We devise a novel distance measure using the error from RND. We calculate the loss between the target and the predictor as shown in Equation 4, and use it as the novelty measure for a state. As such, the bigger the calculated error is, the more novel is the state that was given as input to the target and predictor networks. We use this measure as the distance, so the more novel a state is, the farther away it is from a representative state.

$$MSE(x) = ||\hat{f}(x; \theta) - f(x)||^2 \quad (4)$$

In accordance with the principles of the partitioning scheme from [8], the EE network calculates the distance between state s and a state in the partition memory s' as seen in Equation 3. As presented in [7], RND does not find the distance between two states, but finds the novelty of a state. We propose to use the novelty metric from RND as the distance measure for generating partitions in EEP. The benefit of the proposed method is that it requires less memory, since auxiliary rewards are not stored. A secondary benefit of using the RND instead of the EE is that the output features do not need to be tied to the action-space of an environment. This allows the agent to calculate novelty or distance from more than what is seen, also from some of the implied rules of the observed environment. Furthermore it should achieve at least the same scores as the original EEP algorithm and some similar partitions. Exchanging the distance measure in Equation 3 with RND results in Equation 5:

$$s_{p_i} = argmin_{s_{p_i} \in \hat{S}}(max(||\hat{f}(s, s'; \theta) - f(s, s')||^2, ||\hat{f}(s', s; \theta) - f(s', s)||^2)) \quad (5)$$

Since RND works towards a static target, the reference point $\hat{s}$ can be omitted, as done in Equation 5, by removing the nested maximum loop. The target network acts as the arbitrary reference point that s and s' relates to. This omission simplifies distance calculation and the related calculations because the reference set $\hat{S}$ is not used in the calculation.

This change in networks also reduces the needed items in the replay memory. Since the predictor network trains towards a static target there is no need to save the auxiliary rewards. Using Equation 5 instead of Equation 3 to find the partitions should therefore be faster and more memory efficient.

Theorem 1 (Memory space reduction of RND)**.** Replacing EE with RND in EEP leads to the reduction in memory usage $\mathcal{R}$ as a linear function with regards to the set of transitions $\mathcal{T}$.

Proof. By replacing EE with RND in EEP, the set of auxiliary rewards for each transition does not need to be stored. Each auxiliary reward set is a set of float values, one value for each possible action. As such, $\mathcal{R}$ is defined as

$$\mathcal{R}(|\mathcal{T}|) = (|\hat{r}|\cdot\mathcal{F}) \cdot |\mathcal{T}|, \tag{6}$$

where $\mathcal{F}$ is the size of float on the given machine. It is clear that $\mathcal{R}$ is a linear function of $|\mathcal{T}|$. □

V. Experiment

In this section we present our experiment setting and environment together with our results. The results are divided into two parts corresponding to the contribution. The first part for RND as the distance measure and the second part for the distributed framework.

The Setting: The approach we have suggested be applied to 3 sparse- and 2 dense rewarded Atari games. We use the sparse games Freeway, Montezuma's Revenge and Venture. The dense games are Battlezone and Robot Tank. The games have been chosen to show the new approaches capability both in traditionally hard and easy to learn environments. The experiments will be run on the AAU CLAAUDIA cluster on a virtual system with one Nvidia Tesla v100 GPU and dual Intel Xeon Platinum 8168, 2.7 GHz, 24-cores CPU.

We benchmark our method against the EEP method presented by M. Dann et al. in [8]. The two methods use the same hyper parameter setup. Due to the RND using the training continuously to calculate novelty, the RND predictor network had continuous training, whereas the EE network was stopped after 2 million steps as done in the original paper [8].

Environment: For the environment we use OpenAI's gym Atari which builds upon the arcade learning environment by Bellemare et al.[5]. We use a *repeat action probability* of 1.00 meaning our selected actions stick for the 4 frames which are skipped. Furthermore we make it so that a life loss is deemed terminal, meaning that states where we lose life are considered terminal states by the learner. Lastly we clip our rewards between 1 and -1, meaning that positive rewards are upper bound to 1, negative lower bound to -1 and all else to 0. For more details on hyperparameters see Appendix A.

During our experiments we measure the step count for each episode, the total score, time pr. step, current partition number, the average score pr. 100 steps and the epsilon value. However, some of this information will not be used as a comparative measure, but was gathered to observe internal behavior. As seen in Figure 8 and Figure 9, we use the step count and average score as the evaluation metrics when comparing our results on different games.

A. Results

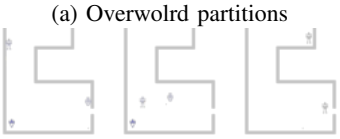
The graphs show the average score for the agent in some environments, unfortunately, the graphs show no results produced by an agent using the distributed framework, further explained in subsection V-B.

The graphs in Figure 8 and Figure 9 show the scores for each game as obtained by our implementation of the original EE and our RND solution. These scores are low compared to the results presented by M. Dann et al.[8].

However contrary to many of the other games, we see that we consistently underperform compared to the baseline in Venture as seen in Figure 8a, rather than following it.

This may be surprising considering that, when we look at our partitions for RND (Figure 11) compared to the baseline (Figure 10), we see that the baseline explores fewer rooms.

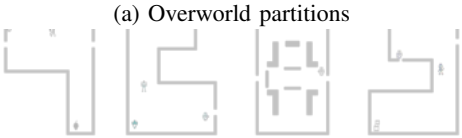
(a) Overwolrd partitions



(b) Single room being explored partitions

Figure. 10: A selection of partitions illustrating the EE agents exploring only a single room, although more thoroughly than NRD as seen in Figure 11

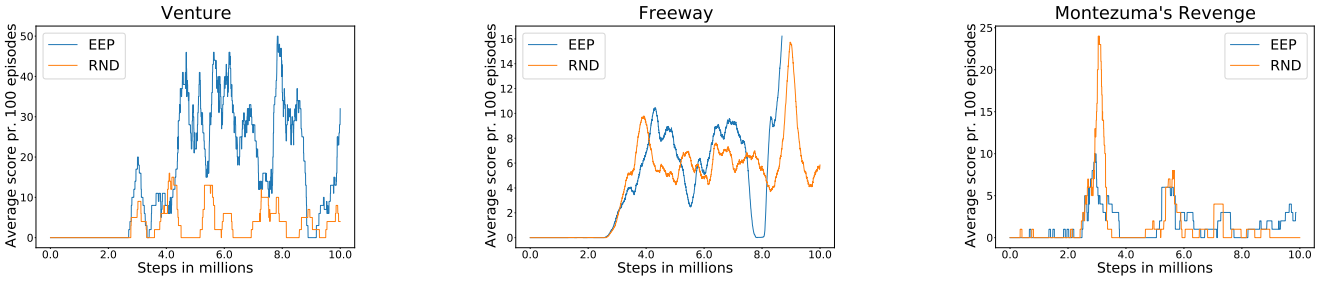
(a) Overworld partitions



(b) More varied rooms explored by RND

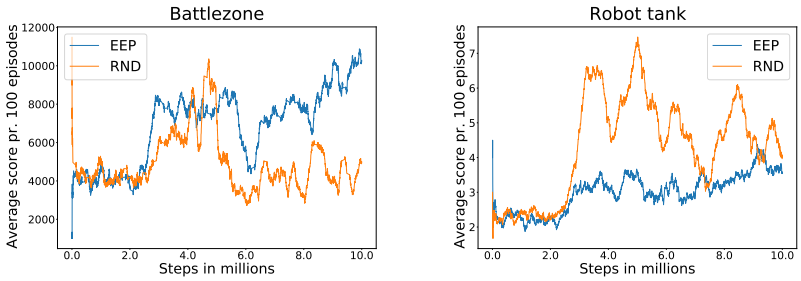
Figure. 11: A selection of partitions illustrating the RND agents exploring many different rooms. Most rooms only have a single partition, thus the agent visits the room without exploring it as thoroughly as with EE in Figure 10

Thus it would seem that the EE agent, in Venture, experiences less rooms than RND. But since the objective for extrinsic rewards are within the rooms, not just at the entrance, we see that EE still performs better. This could again be



(a) Venture (b) Freeway (c) Montezuma's revenge

Figure. 8: Average score for the last 100 episodes over steps taken for three sparse reward games by non-distributed agents



(a) Battlezone (b) Robot Tank

Figure. 9: Average score for the last 100 episodes over steps taken for two dense rewarded games by non-distributed agents

related to the noisy-tv problem. Rather than getting stuck, we prioritize finding rooms, rather than learning them and getting their respective rewards.

Interestingly though we see an, albeit relatively weak but nonetheless consistent, pattern that could indicate negative learning. We see in Figure 8a that the peaks are consistently becoming lower. Even though the data for RND contains a slightly higher episode top score, we are much less consistent, contributing to the low average score.

On Figure 8b we see that the partitions encourage the agent to explore. The agent using RND as distance measure is more stable in the middle of its training but spikes at the end. The sudden fall afterward seems to be because of the agents' two newest partitions where the game character got run over. These new partitions produce relative high rewards and will therefore be explored for a bit. They will however be ignored over time, due to overall loss resulting from termination.

Both models presented in Figure 8c are able to get scores in Montezuma's Revenge, which is one of the hardest games in the Atari 2600 test suite. But they could use some more hyper parameter tuning to increase learning in the later stages of training. Partitions generated using RND and EEP seem similar, both generating good and bad partitions that leads to extrinsic rewards.

The two graphs on dense reward games on Figure 9 indicate improvement from the beginning to the middle of the training period and negative learning for the rest of the training. This is the case for both Battlezone and Robot Tank. We argue that this is because the partitions made in these games mostly seem to be a few states away from a terminating state. These types of partitions indicate an issue related to the noisy tv-problem because the agent is presented with a lot of visual noise at states right before termination. This visual noise will be interpreted as novel by the RND network.

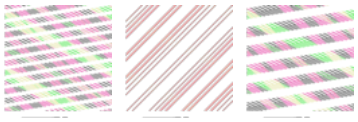


Figure. 12: A partition made in Robot Tank

About 90% of the generated states are variations of Figure 12. Because the agent seeks the partitions to gain points, it will be in a state of no return when such a state is found. The agent using the exploration effort to make partitions is not as explosive in its result, but has a more stable increase.

All the agents show that there is room for improvement in both hyper parameter tuning and some more conditions on when to make partitions. They are able to get better scores than just taking random actions. The partitions made using RND instead of EE use a smaller amount of memory dependent on the action space and the data type used to store the auxiliary rewards used to train the EE calculation as shown in Theorem 1. In the end, we also manage a memory saving in the range of 2 bits to 1152 bytes pr. transition, depending on whether the auxiliary reward is saved as bits or float64 for the 2 to 18 actions in the environment.

B. Distribution results

Preliminary results from the distribution are unfortunately worrying.

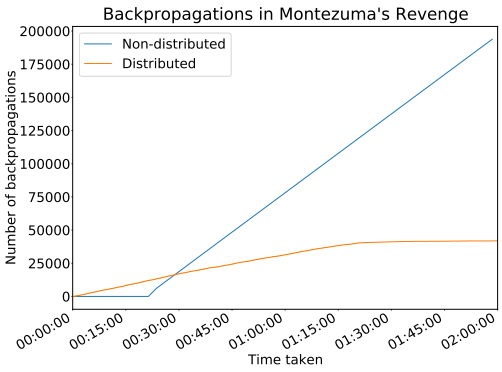


Figure. 13: The amount of backpropagations as a function over time, performed by the distributed and non distributed framework

Our intuition as to why the backpropagation is slower compared to non-distributed, considering implemented functions are identical, is related to processor scheduling, this can be seen on Figure 13. The results of this test show the difference in amount of backpropagation batches the agent runs through as a function of time. Even though calculations are done on GPU, the method for doing so and fetching of data etc. is CPU bound. Meaning the speed still relies on process time. Furthermore, considering the amount of data fetches may also impact its process priority due to I/O waits. The process may very well have low priority compared to actors with low I/O.

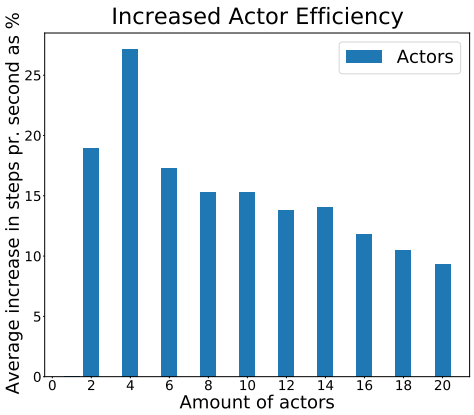


Figure. 14: The efficiency of each actor, as measured by how many frames they can process each second compared to a single actor.

Although we see an increase in throughput for every actor we add to the system, the gain does not scale linearly. With the distributed EEP implementation we tested the throughput as frames processed pr. second for different amounts of actors. From the test results in Figure 14 we see that the efficiency from adding actors peaks earlier than expected, at around four actors, while Figure 15 show that total throughput peaks at fourteen. The results were gathered using the CLAAUDIA servers as mentioned earlier and efficiency tests were run for fifteen minutes before actors were expected to have filled

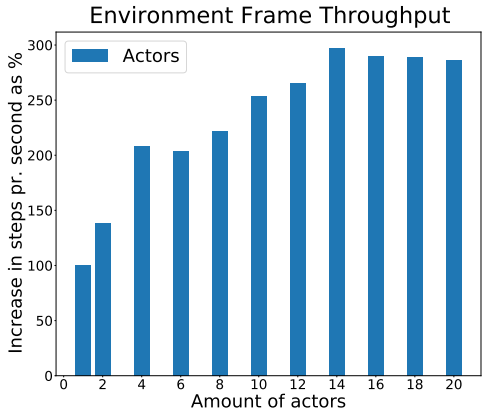


Figure. 15: The total throughput of all actors combined, compared to that of a single actor.

replay memory. Although we still get more throughput from adding actors, we see them gradually become less efficient, as we add actors.

However, we also encountered many hard- and software related problems.

First off, we have experienced numerous issues with regards to shared memory management when running code on the CLAAUDIA server cluster.

Beyond this, we also had problems with utilization, at seemingly random times we would encounter, what we theorize to be a livelock in our code. Our repeated code analysis could not identify a situation that should allow for this, however, we would consistently encounter the error. In the situation, the manager and learner would stop progressing, seemingly in a livelock, with each other, while actors could unhindered continue to progress.

Considering results from non-distributed section and preliminary non-conclusive results from the distributed section we would expect test performance to be the same. However, we cannot say anything conclusive about the throughput performance of our implementation.

Lastly, we have collected some issue trackers and information on the problems, see Appendix B

VI. Conclusion

We have shown that using Random Network Distillation as a novel distance measure to make partitions did not significantly improve test scores over the exploration effort implementation. Contrary in Venture and Battle Zone the game scores were worse than the baseline. However, using RND results in fewer network propagations in the training. Using the novelty measure as the distance formula instead of the exploration effort also decreased the complexity of the distance formula used to find and make partitions. During our experiments, RND almost halved the training time when compared to exploration effort. Using RND as distance also seemed to worsen the agents performance over time, while the agent using EE was stable and increasing in performance.

The distribution framework did likewise show an increase in environment interactions, however, as described in subsection V-B presented its own suite of problems and complexity to the solution as a whole. One of the biggest problems being the decreased amount of backpropagations. Though some of these may prove trivial to solve with more experience

A. Future Work

A future improvement could be having multiple learners with a shared model such as PPO or A3C does, to have both multiple actors and multiple learners. In addition to this, a meta controller or process manager could be implemented to better balance the load between manager, learner(s) and actors.

References

[1] David Abel et al. "Exploratory gradient boosting for reinforcement learning in complex domains". In: *arXiv preprint arXiv:1603.04119* (2016).

[2] Adrià Puigdomènech Badia et al. "Agent57: Outperforming the atari human benchmark". In: *International Conference on Machine Learning*. PMLR. 2020, pp. 507–517.

[3] Adrià Puigdomènech Badia et al. "Never give up: Learning directed exploration strategies". In: *arXiv preprint arXiv:2002.06038* (2020).

[4] EN Barron and H Ishii. "The Bellman equation for minimizing the maximum cost". In: *Nonlinear Analysis: Theory, Methods & Applications* 13.9 (1989), pp. 1067–1090.

[5] M. G. Bellemare et al. "The Arcade Learning Environment: An Evaluation Platform for General Agents". In: *Journal of Artificial Intelligence Research* 47 (June 2013), pp. 253–279.

[6] Marc G Bellemare et al. "Unifying count-based exploration and intrinsic motivation". In: *arXiv preprint arXiv:1606.01868* (2016).

[7] Yuri Burda et al. "Exploration by Random Network Distillation". In: *CoRR* abs/1810.12894 (2018). arXiv: 1810.12894. URL: http://arxiv.org/abs/1810.12894.

[8] Michael Dann, Fabio Zambetta, and John Thangarajah. "Deriving Subgoals Autonomously to Accelerate Learning in Sparse Reward Domains". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 33.01 (July 2019), pp. 881–889. DOI: 10.1609/aaai.v33i01.3301881. URL: https://ojs.aaai.org/index.php/AAAI/article/view/3876.

[9] Frédérick Garcia and Emmanuel Rachelson. "Markov Decision Processes". In: *Markov Decision Processes in Artificial Intelligence*. John Wiley and Sons, Ltd, 2013. Chap. 1, pp. 1–38. ISBN: 9781118557426. DOI: https://doi.org/10.1002/9781118557426.ch1. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781118557426.ch1. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/9781118557426.ch1.

[10] Jean Harb et al. "When Waiting is not an Option : Learning Options with a Deliberation Cost". In: *CoRR* abs/1709.04571 (2017). arXiv: 1709.04571. URL: http://arxiv.org/abs/1709.04571.

[11] Zhang-Wei Hong et al. "Diversity-driven exploration strategy for deep reinforcement learning". In: *arXiv preprint arXiv:1802.04564* (2018).

[12] Steven Kapturowski et al. "RECURRENT EXPERIENCE REPLAY IN DISTRIBUTED REINFORCEMENT LEARNING". In: *ICLR 2019* (2019), pp. 1–19.

[13] Steven Kapturowski et al. "Recurrent experience replay in distributed reinforcement learning". In: *International conference on learning representations*. 2019.

[14] Marlos C Machado, Marc G Bellemare, and Michael Bowling. "Count-based exploration with the successor representation". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 34. 04. 2020, pp. 5125–5133.

[15] Jarryd Martin et al. "Count-based exploration in feature space for reinforcement learning". In: *arXiv preprint arXiv:1706.08090* (2017).

[16] Volodymyr Mnih et al. "Asynchronous Methods for Deep Reinforcement Learning". In: *CoRR* abs/1602.01783 (2016). arXiv: 1602.01783. URL: http://arxiv.org/abs/1602.01783.

[17] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *nature* 518.7540 (2015), pp. 529–533.

[18] Brendan O'Donoghue et al. "The uncertainty bellman equation and exploration". In: *International Conference on Machine Learning*. 2018, pp. 3836–3845.

[19] Martin L. Puterman. "Chapter 8 Markov decision processes". In: *Stochastic Models*. Vol. 2. Handbooks in Operations Research and Management Science. Elsevier, 1990, pp. 331–434. DOI: https://doi.org/10.1016/S0927-0507(05)80172-0. URL: https://www.sciencedirect.com/science/article/pii/S0927050705801720.

[20] Nikolay Savinov et al. "Episodic curiosity through reachability". In: *arXiv preprint arXiv:1810.02274* (2018).

[21] Tom Schaul et al. "Prioritized experience replay". In: *arXiv preprint arXiv:1511.05952* (2015).

[22] Julian Schrittwieser et al. "Mastering atari, go, chess and shogi by planning with a learned model". In: *Nature* 588.7839 (2020), pp. 604–609.

[23] Alexander L Strehl and Michael L Littman. "An analysis of model-based interval estimation for Markov decision processes". In: *Journal of Computer and System Sciences* 74.8 (2008), pp. 1309–1331.

[24] Hsuan-Kung Yang et al. "Exploration via Flow-Based Intrinsic Rewards". In: *arXiv preprint arXiv:1905.10071* (2019).

[25] Jiachen Yang, Igor Borovikov, and Hongyuan Zha. "Hierarchical cooperative multi-agent reinforcement learning with skill discovery". In: *arXiv preprint arXiv:1912.03558* (2019).

APPENDIX A
HYPERPARAMETER TABLES

Hyperparameter table		
Training steps	10.000.00	The amount of steps an agent is allowed to run before termination
Partition discovery frequency	20.000	The amount of steps before a new partitio nwill be added (base)
Partition discovery frequency multiplier	0.2	The multiplier for steps before a partition is added (multiplied at every partition addition
Partition start	2.000.000	Delay in frames before we start to create partitions
Partition memory size	100	The upper bound amount of partitions that can be present in agents memory
Replay memory size	1.000.000	Upper bound for transitions in replay memory
Discount factor	0.99	Future reward multiplier controlling that due to uncertainty we cannot weigh predicted rewards as much as current
Batch size	32	Transistions taken from memory passed to network*
Distribution specific parameters		
Partition candidate push frequency	10.000	The freqency (in steps) at which actors push partition candidates to learner
Weight push frequency	1.000	How many transitions are trained before new network weights are pushed to actors
Non Distribution specific parameters		
Update frequency	1.000	Steps before we update target networks

Convelutional network specifications									
	Layer 1			Layer 2			Layer 3		
	input→output	kernel	stride	input→output	kernel	stride	input→output	kernel	stride
Qnet	1→32	8	4	32→64	4	2	64→64	3	1
EEnet	1→16	8	4	16→16	4	2	16→16	3	1

TABLE I: Note that Qnet and RNDnet share convelutional network structures so only 1 is listed

APPENDIX B
PYTORCH IMPORTANT ISSUES

A series of linked issues and open problems with the framework, in no particular order: https://github.com/pytorch/pytorch/issues/17199 https://github.com/pytorch/pytorch/issues/41486 https://github.com/pytorch/pytorch/issues/57401 https://github.com/pytorch/pytorch/issues/53178 https://github.com/pytorch/pytorch/issues/48382

Even with the great help of local helpdesk we could not find any root causes for these runtime complications. Which include the following:

- `RuntimeError: unable to open shared memory object </torch_29919_1396182366>`
  - Long stading issue with continually closed and reopen status
  - https://github.com/pytorch/pytorch/issues/1355
  - ulimit er ikke testedt mere endnu
- Generic RunTimeError codes
  - 2, no such file or dir
    - May be related to filesharing startegy which is dependant on file system file handles
    - https://pytorch.org/docs/stable/multiprocessing.html#sharing-strategies
  - 5, queues read write same time even though safe
    - May be related to queues, though multiprocessing.Queues is documented to be thread and process safe, and we followed the multiprocessing guidelines and best practice dexcribed here https://pytorch.org/docs/stable/notes/multiprocessing.html and cuda sharing described here https://pytorch.org/docs/stable/multiprocessing.html#multiprocessing-cuda-sharing-details
  - 1455, shared file mapping
    - May be related to error code 2, a missing file descriptor handle can cause garbage collection to clean up shared memory resulting in invalid shared file mappings
- ThByte storage linux python OS error
  -

APPENDIX C
PSEUDOCODE

Algorithm 1 Actor Pseudocode

procedure MAIN()

Reset variables, s, s', v

// Add representative state for first partition
$s_{p_1} \leftarrow s$
$R \leftarrow \{s_{p_1}\}$
$t_{partition} \leftarrow 0$

for Training length **do**
Select $a = \text{argmax}_a(Q(s, a))$
Take action a, observe r, s'

// Determine the current partition acording to the Equation 5
$s_{p_c} \leftarrow argmin_{s_{p_i} \in R} d(s', s_{p_i})$

// Update the set of visited partitions
$v' \leftarrow v \cup s_{p_c}$

Store transition $\{s, v, a, r, s', v'\}$ in episode buffer

if $s'$is terminal **then**
Send episode buffer to manager
Update all partitions' visit counts based on v
Reset variables, s, s', v
end if

// Update the best candidate
if $d(s', s_{p_c}) > D_{max}$ **then**
$\tilde{s}_{p_{n+1}} \leftarrow s'$
$D_{max} \leftarrow d(s', s_{p_c})$
end if

// Add a new rep. state every Tadd steps
$t_{partition} \leftarrow t_{partition} + 1$
if $t_{partition} > T_{add}$ **then**
$SendPartitionToLearner(\tilde{s}_{p_{n+1}})$
$Dmax \leftarrow 0$
$t_{partition} \leftarrow 0$
end if

$s \leftarrow s'$
if Network Update Frequency **then**
Update network parameters with learner parameters
Add selected partition form learner to partition memory
end if
end for
end procedure

Algorithm 2 Memory Maneger Pseudocode

```
procedure MANAGE()
    while Actors Live do
        if len(RelayMemory) < minmemory or queue to learner is full then
            Process episode buffers from actors to replay memory
        end if

        if len(RelayMemory) > minmemory and queue to leraner is Empty then
            Fill queue to learner with transitions from replay memory
        end if
    end while
end procedure
```

Algorithm 3 Learner Pseudocode

```
procedure LEARN()
    while Actors Live do
        if queue from manager is full then
            Empty queue to local memory
            if patition queue from actor is full then
                Select furthest partition and send to actors
            end if
            Qlearn()
            EElearn()
            if target network unpdate frequency then
                Q-taget ← Q-network
            end if
            Send networks to actors
        end if
    end while
end procedure
```