TU DELFT

MSC BIOMECHANICAL ENGINEERING

MASTER THESIS

The effect of sampling methods on Deep Q-Networks in robot navigation tasks

Author
T.J.L de JONG

Supervisor
Dr. Ir. J. BROEKENS

January 14, 2019

Contents

Preface

This thesis is made as a completion of the master education in BioMechanical engineering. Yours truly has a bachelor degree in mechanical engineering from TU Delft and this thesis is the product of the master period. In the master program of BioMechanical engineering several courses are available that connect to machine learning, these fueled my interest in this topic. The machine learning courses are mostly taught by teachers from different faculties, so this research has not been done at the BioMechanical department but instead at the Interactive Intelligence Group.

Two persons have contributed academically, practically and with support to this master thesis. I would therefore like to thank my head supervisor Joost Broekens and co-supervisors Thomas Moerland for their time, valuable input and support throughout the entire research.

Finally, I would like to thank my family and friends for being helpful and supportive during my time studying mechanical engineering at the TU Delft.

Abstract

Enabling mobile robots to autonomously navigate complex environments is essential for real-world deployment in commercial, industrial, military, health care, and domestic settings. Prior methods approach this problem by having the robot maintain an internal map of the world and then use a localization and planning method to navigate through the internal map. However, these approaches often include a variety of assumptions, are computationally intensive, and do not learn from failures. Recent work in deep reinforcement learning shows that navigational abilities could emerge as the by-product of an agent learning a policy that maximizes reward.

Deep Q-Networks (DQN), a reinforcement learning algorithm, uses experience replay to remember and reuse experiences from the past. A sampling technique determents how to sample the experiences that are to be replayed from the experience replay buffer. Here we studied the effect of different sampling techniques on the learning behavior of an agent using DQN in partially observable navigation tasks. In this work five sampling techniques are proposed and compared to the original random sampling technique.

We found that sampling techniques focusing on surprising experiences learn faster than random sampling techniques. Secondly, we found that the final performance of all sampling techniques usually converge to the same policy. Finally, we found the correct use of importance sampling is essential when using prioritized techniques.

1 Introduction

Globally, it is estimated that 1.3 million industrial robots will be added to factories in 2018 [1]. Some call it the "fourth industrial revolution." Regardless the numbers are proof that robots are exponentially being incorporated into our lives. A large part of these robots is mobile. These can be used in many applications, for example transportation tasks, surveillance, cleaning, factory work, entertainment or health care. Mobile robots have three main advantages compared to humans in tasks that require mobility. The first main advantage being reliability, uninterrupted and reliable execution of monotonous tasks such as surveillance. Secondly, safety, inspection of sites that are inaccessible to humans, e.g. tight spaces, hazardous environments or remote sites. Lastly, cost, transportation systems based on autonomous mobile robots can be cheaper than standard track-bound systems.

Recent work showed a mapless motion planner can be trained end-to-end (sensor input to steering control commands) in a virtual environment using *reinforcement learning* [2].

Reinforcement Learning (RL) is a type of machine learning. In reinforcement learning an agent is interacting with the environment and using these interactions to learn an optimal or near-optimal behavior. The idea of reinforcement learning is inspired by the way nature works. The goal of reinforcement learning is to learn good policies for sequential decision problems, by optimizing a cumulative future reward signal.

Deep reinforcement learning or *deep learning*, a subfield of reinforcement learning, uses *neural networks*, an interconnected system of artificial neurons designed to mimic the operation of a brain, as function approximators. Because of this deep learning can automate feature engineering which in turn means reliance on domain knowledge is significantly reduced[3]. Recently there are some exciting achievements of deep learning, benefiting from the exponential growth of stored data, powerful computation, new algorithmic techniques, mature software packages and architectures, and strong financial support[4]. This lead to breakthroughs like learning to play Atari games at human-level performance[5][6] and beating the top human player in the game of Go[7].

Reinforcement learning agents update their policy while they observe a stream of experience. In its most basic form reinforcement learning discards incoming experiences after a single update. This approach has two main problems. First, a common assumption when training a function approximator, such as a neural network, is that the training examples, in this case experiences, are independent and identically distributed (i.i.d.) [8]. While typically in reinforcement learning sequential experiences are often highly correlated and the distribution of these experiences may change as the agent learns new behaviors. This could cause catastrophic forgetting [9] or getting trapped in a poor local minimum. Secondly, all the experience generated is only used once, which is very inefficient data usage.

To solve these problems a technique called *experience replay* is used where the agent's experiences at each time-step are stored in a *replay memory*. This experience replay makes it possible to use experiences multiple times and mix more and less recent experience in a sample. In most applications experience replay will reduce the amount of experience required to learn, which means fewer interactions with the environment. However, it does increase the amount of computation and memory needed, but these are usually less expensive resources.

However, even when using experience replay deep learning can still be data inefficient. For example, one of the state-of-the-art reinforcement learning algorithms [10] using experience replay needs about 18 million frames on average to reach human level performance in 57 Atari games. This corresponds to about 83 hours of game play at a normal 60 frames per seconds, a lot of time for an Atari game that most humans pick up within a few minutes. For games and other environments where simulation is possible, this is not a huge problem. To the contrary, for most real-world settings generating experience takes time and could be costly [11]. This is why efficient use of experiences is essential.

The *Deep Q-Networks* (DQN) algorithm, used to play many Atari games at human-level performance, was the first to use a combination neural networks, experience replay, and Q-learning, a value-based reinforcement learning algorithm. Since then, many extensions have been proposed that enhance its speed or stability [12][13][14].

In DQN the experiences are sampled uniformly from the replay memory, however, other *sampling techniques* are possible. This study investigates the effect of different sampling techniques on the learning behavior of an agent using DQN in navigation tasks. It compares five sampling techniques to the original random sampling technique in a virtual environment. The sampling techniques studied in this thesis are based on the idea that an RL agent can learn more effectively from some experiences than from others. In particular, it is based on the idea that the magnitude of an experience's *temporal-difference (TD) error*, the difference between the estimated value of a state and the better estimate using the feedback from the environment, can indicate the expected learning progress of that experience [13]. In this thesis sampling methods that prioritize experiences with high TD errors are named *prioritized sampling techniques*.

The hypothesis is that RL agents using these prioritized sampling techniques have a better sample efficiency than the original random sampling technique.

In the first section, an overview of the most relevant parts of DQN is given. Then the concept of experience replay is elaborated with an emphasis on sampling techniques and importance sampling. The research question is clarified in section 2. A method to answer the research question is proposed in the next section, this includes a new environment and architecture. Using the method described simulations are done resulting in data about each sampling techniques test performance over time. These results are then compared using performance metrics like the stability and speed of convergence. In the fifth section, this research is discussed and compared to other research. Finally, in the last section, conclusions are drawn from this research and recommendations are done for future work.

1.1 Background

In this section concepts and fundamentals in machine learning, deep learning and reinforcement learning are introduced.

1.1.1 Machine learning

Machine Learning is the science of getting computers to learn and improve their performance on a specific task over time in autonomous fashion. All machine learning algorithms are composed of a dataset, loss function, optimization procedure, and model.

Machine Learning is usually divided into three categories: supervised, unsupervised, and reinforcement learning [15]. Supervised learning is the task of learning a function that maps an input to an output based on example input-output pairs called labeled training data. The goal is to predict the class label for new unseen data based on the relationships it learned from the labeled training data. In unsupervised learning, a model tries to learn relationships between unlabeled examples by looking for hidden structures, patterns or features within those examples. The last category of machine learning, the category this thesis is focused on, reinforcement learning aims at using observations gathered from the interaction with an environment to take actions that would maximize the reward or minimize the risk.

1.1.2 Neural networks

Artificial neural networks, in this thesis called neural networks, consist of a collection of connected units or nodes called artificial neurons, which loosely model the neurons in a biological brain [16]. The artificial neuron receives one or more inputs that are separately weighted. It then sums them to produce an output that is passed through a function known as an activation function. Typically, artificial neurons are aggregated into layers. Different layers may perform different kinds of transformations on their inputs. Signals travel from the first layer (the input layer) towards the last layer (the output layer).

To update the weights in a neural network a method called backpropagation is used to calculate gradients of the weights using the loss function [17]. Then these gradients are used in an iterative optimization algorithm like gradient descent to update the weights. This way neural networks can learn to perform tasks by considering examples, generally without being programmed with any task-specific rules.

1.1.3 Reinforcement learning

Reinforcement learning has a wide range of applications. Games and robotics are two classical RL application areas, but there are other applications like natural language processing, computer vision, neural architecture design, business management, ads, recommendation, customer management, marketing, finance, healthcare, industry, smart grid, intelligent transportation systems, and computer systems [18].

In reinforcement learning an agent interacts with an environment over time in a *Markov Decision Process* (MDP). In an MDP there is a state space $\mathcal{S}$ and an action space $\mathcal{A}$. Each time step t the agent is in a state s_t in $\mathcal{S}$ and then selects an action a_t from $\mathcal{A}$ using a policy $\pi(s_t)$, which represents the agents behavior. Then the agent receives a reward r_t and ends up in the next state s_{t+1}. For a process to be an MDP the next state should only depend only on the current state and action, but not on the past.

The value function $V^\pi(s)$ in reinforcement learning is an estimate of expected return J, i.e. discounted future reward, measuring how "good" each state is:

$$V^\pi(s_t) = \mathbb{E}[J|S_t = s_t] \tag{1}$$

$$J = \sum_{t=1}^{\infty} \gamma^{t-1} r_t \tag{2}$$

with discount factor $\gamma \in (0, 1]$, used to determine the importance of future rewards, and r_t the reward at step t. The agent aims to find the optimal policy π^* that maximizes the expectation of such long term return from each state. An optimal action-value function $Q^*(s, a)$, the value for choosing an action in a state, can be formulated using the Bellman Principle of Optimality that states: "An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision."

$$Q^*(s_t, a_t) = R(s_t, a_t) + \gamma \sum_{s_{t+1}} \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1}) T(s_t, a_t, s_{t+1}) \tag{3}$$

where the transitions between states are modeled as
$T(s_t, a_t, s_{t+1}) = P(s_{t+1}|s, a)$ and $R(s_t, a_t)$ is the reward function depend-

ing on the current state and action. In this thesis the agent does not know apriori what the effects of its actions on the environment are, i.e. state transition and reward models are not known to the agent. This means the action-value function has to be actively learned through interactions with the environment.

There are a wide variety of action-value function based reinforcement learning algorithms. One class of these algorithms that do not need a explicit transition function are the temporal difference (TD) methods. In TD methods an earlier action value is updated by calculating a TD error, the difference between the old estimate and a new estimate of the action-value function. When a value of an action is updated based on the current estimate of that value it is called bootstrapping. This is done in a popular TD learning algorithm Q-learning [19]. It learns the action-value function, with the update rule:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \quad (4)$$

were learning rate $\alpha \in [0, 1]$ is used to progressively approximate the optimal policy and therefore usually decreased over time in stationary environments. The Q-learning algorithm approximates the action-values in order to approach the optimal action-value function. It is guaranteed that the action-values converge to the optimal values in an infinite run by selecting each action in the appropriate state infinitely often [20]. Then the optimal policy is the policy that selects the action with the maximum value in each state.

1.1.4 Deep learning

In small, discrete state and action spaces an action-value function is stored in a tabular form. However when the state or action spaces are too large or continuous function approximation is a way of generalizing these spaces. It is called deep reinforcement learning or deep learning when neural networks are used to approximate the action-value function. In deep learning the neural networks used to approximate the action-value function have one or more hidden layers between the input and output layer. This enables end-to-end training: a learning model that uses raw inputs without manual feature engineering to generate action values for each state and, more importantly, estimate the action values of unseen states based on earlier similar states.

Recently many different RL algorithms that work with deep networks are proposed. Deep Q-Networks, one of the first breakthrough successes in applying deep learning to RL, is sufficient to explore the sampling techniques discussed in section 2.1.

1.2 Deep Q-Network

The first deep reinforcement learning method proposed by DeepMind, and the method this work is building on, is called deep Q-networks (DQN). It showed a way deep neural networks can empower RL to directly deal with high dimensional states like images by integrating supervised learning techniques into RL. Simply replacing a tabular action-value function with a neural network in normal Q-learning leads to unstable or even divergent results. There are two main reasons for this instability. First, the supervised learning techniques used to train the neural network prefer independent and identically distributed (i.i.d) data. This means the training examples used to optimize the neural network need to have similar data distributions and are independent of each other. If the i.i.d. assumption is not met the model could overfit, i.e. the policy will not be generalized. Secondly, supervised learning techniques do not perform well when labels of the same input are changed over time. In RL the labels of state-action pairs, i.e. their estimated action-values, are called targets and these change after every update of the action-value function.

To overcome these problems DQN proposes four important techniques:

1. Experience replay, further elaborated in section 1.3
2. A target network, a periodic copy of the neural network used to estimate the action value which is not directly optimized.
3. Stacking frames, using a history of frames as a state instead of only the last frame.
4. Reward clipping, fixing all positive rewards to 1 and negative to -1 this limits the scale of the error.

To train an agent using DQN there are several steps involved. First, the agent selects an action using a ϵ-greedy selection strategy: with probability $(1 - \epsilon)$ an action is chosen using the current policy $\pi(s)$ or else, with probability ϵ, a random action is chosen. Because the agent can not get a complete state of the environment with only one frame, i.e. the environment is partially observable, it does not comply with the Markov property. This states that the conditional probability distribution of future states of a process must depend only upon present state. One way to alleviate the Markov property is to allow decisions to be based on the history of recent observations [21]. Thus a new state is created by stacking the most recent frame and the last three frames. Another way to do this would be using recurrent architectures to capture arbitrary long-term dependencies [22]. Next, an experience, also called transition, (s_t, a_t, r_t, s_{t+1}) is added to the replay memory. The following step is to sample a batch of random experiences from this replay memory. Then the targets of all the experiences in the batch are calculated using the following equation:

$$Q_{target}(s_{t+1}, r_t) = r_t + \gamma \max_{a_{t+1}} Q_{\bar{\theta}}(s_{t+1}, a_{t+1}) \tag{5}$$

where r_t is the reward of taking action a_t in state s_t. To determine the importance of future rewards the discount factor γ is used like in equation 2. The action-value function approximated by the target network is denoted by $Q_{\bar{\theta}}(s_t, a_t)$, this is a periodic copy of the online network $Q(s_t, a_t)$, i.e. the neural network the current policy uses to estimate the action values. In practice this means the weights of the target neural network $\bar{\theta}$ are periodically replaced by the the weights in the online network θ. In standard DQN the maximum action value in the next state s_{t+1} is used when calculating the target. The next step is to calculate the TD error δ:

$$\delta = Q_{target}(s_{t+1}, r_t) - Q_{\theta}(s_t, a_t) \tag{6}$$

this TD error δ is used in a loss function like mean squared error (MSE):

$$L = \mathbb{E}[\delta^2] \tag{7}$$

The last step is to train the online Q-network Q_θ by minimizing the loss function. This can be done by an iterative method for optimizing like stochastic gradient descent (SDG). SDG simply does away with the expectation in the update and computes the gradient of the weights θ using only a single or a few training examples. Updating the weights is done by:

$$\theta = \theta - \alpha \bigtriangledown_\theta \frac{1}{n} \sum_n L_n(\theta) \tag{8}$$

where n is the number of experiences in a batch and the loss of experience n is denoted by L_n. The learning rate α is a hyperparameter used to control the size of the updates. Choosing the proper learning rate and schedule (i.e. changing the value of the learning rate as learning progresses) can be fairly difficult. SGD can lead to very slow convergence, especially when the learning rate is too low or high. This is why recently more advanced optimization algorithms are used like Adam [23], an algorithm for first-order gradient-based optimization of stochastic objective functions, based on adaptive estimates of lower-order moments. These moments are calculated by keeping track of the velocity v of a weight update:

$$v = \beta \cdot v + \alpha \bigtriangledown_\theta L_n(\theta) \tag{9}$$

where β determines how much of the previous gradients are incorporated into the current update. The main benefits of using Adam are that the magnitudes of parameter updates are invariant to a rescaling of the gradient, its stepsizes are approximately bounded by the learning rate, it does not require a stationary objective, it works with sparse gradients, and it naturally performs a form of learning rate annealing. All these benefits make it a widely used optimization algorithm and the reason it is used in this thesis.

Several limitations of DQN are now known and many extensions have been proposed that enhance its speed or stability. In deep learning algorithm Rainbow [10] six of these extensions are combined. An ablation study is then done that shows the contribution of each component to overall performance. Prioritized experience replay (PER) [13], replaying more often experiences from which there is more to learn, is the extension that proved to be the most crucial component of Rainbow. That inspired the research topic of this thesis.

One of the extensions to DQN used in this thesis is double Q-learning (DDQN) [12], in order to make this study comparable to PER which also uses this extension. Conventional Q-learning is affected by an overestimation bias, due to the maximization step in equation 5. The idea of Double

Q-learning is to reduce overestimations by decomposing the max operation in the target Q_{target} into action selection and action evaluation. This action selection and evaluation is done using the online Q-network. This changes the target, equation 5, to:

$$Q_{target}(s_{t+1}, r_t) = r_t + \gamma Q_{\bar{\theta}}(s_{t+1}, \underset{a}{\operatorname{argmax}}\, Q_{\theta}(s_{t+1}, a)) \tag{10}$$

1.3 Experience Replay

Experience replay is originally proposed in 1993[24]. Since 2015 experience replay enjoys great success in the deep RL community and has become a new norm in many deep RL algorithms. Only learning from the most recent experiences without saving them in a replay memory has several shortcomings discussed in the introduction. In summary, it is data inefficient, causes rapid forgetting, does not produce i.i.d. data to train a neural network and could cause unwanted feedback loops. Experience replay not only provides less correlated data to train a neural network but also significantly improves the data efficiency [21].

In experience replay, experiences (s_t, a_t, r_t, s_{t+1}) are written to a buffer, called the replay memory. To train the Q-network's prediction of the action value a batch of random experiences is sampled from that buffer. This breaks the temporal correlations of the experiences updating the neural network. It increases learning speed and sample efficiency of the algorithm because the use of batches makes for better estimations of the gradients. Finally, it counters catastrophic forgetting. By using experiences multiple times the network is less likely to abruptly forget previously learned information upon receiving new experiences.

There are still several challenges relating to experience replay. How many experiences must be kept in the replay memory? Which experiences should be discarded to make room for new experiences? How many experiences should be in a batch? Are there other more efficient ways to sample experiences from the replay memory for replay than randomly selecting a batch of experiences?

2 Research Question

As stated in the introduction, experience replay significantly improves the data efficiency. There are however some unexplored areas of this experience replay. One design choice that has to be considered is which experiences to replay and how to do so. In other words how to make the most effective use of experience replay for learning.

2.1 Sampling Techniques

Recent work shows that one of the most effective ways to make learning from experience replay more efficient is by selecting samples for replay that the agent can learn the most from [13]. In biology, neuroscience studies have identified evidence of experience replay in the hippocampus of rodents, suggesting that sequences of prior experience are replayed, either during awake resting or sleep. And in these studies sequences associated with rewards appear to be replayed more frequently [25][26], this indicates more efficient ways of selecting experiences for replay are possible.

In statistics, sampling is the selection of a subset of individuals from within a population to estimate characteristics of the whole population. However, when sampling experiences for reinforcement learning, the goal is to select the experiences that can make experience replay more efficient and effective than if all experiences are replayed uniformly. In this section an overview is given of the six sampling techniques used in this research, starting with the original random sampling. The other five build on ideas from prioritized reinforcement learning [13], further elaborated under "prioritized" below.

All the sampling techniques proposed below are tested three times. First without importance sampling, then with importance sampling and finally with importance sampling and the most recent experiences added to the batch.

2.1.1 Random

This sampling technique was originally used in DQN. It is the most common way of sampling, simple random sampling. Each experience has an equal probability of selection. This minimizes bias and breaks the temporal correlations between the experiences. This important because the algorithms used to update the neural network assumes independent and identically distributed training data, as discussed in section 1.2. However random sampling will inevitably sample experiences from which there is not much to learn: experiences that are less surprising, redundant, or not task-relevant [13].

2.1.2 Prioritized

This sampling techniques is similar to prioritized experience replay [13], with three small changes described at the end of this section. Probability-proportional-to-size sampling is where the selection probability for each experience is set to be proportional to its size measure. This sampling technique can improve efficiency by concentrating samples on large elements that have the greatest impact. However a drawback could be that different portions of the population may be over- or under-represented based on their size measure.

In reinforcement learning the size of an experience is best indicated by the magnitude of an experience's TD error δ, described in equation 6. This sampling technique is used in prioritized experience replay, it replays experiences more frequently relative to the last encountered absolute TD error. The prioritization of experiences with a high magnitude of their TD error has four problems. First, it can lead to a loss of diversity, initially high error experiences get replayed frequently. Secondly, the TD error of every experience changes after an update but to avoid expensive sweeps over the entire replay memory, TD errors are only updated for the experiences that are replayed. Experiences that have a low TD error when they are added to the replay memory may not be replayed for a long time or not at all. Third, it is sensitive to noise because this increases the TD error of irrelevant experiences. Finally, it introduces bias because it changes the distribution of the training data breaking the i.i.d. data assumption needed for stochastic gradient descent, and therefore change the solution that the estimates will converge to.

To solve the lack of diversity a stochastic sampling technique is introduced, it defines the probability of sampling experience i as:

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha} \tag{11}$$

where the exponent α determines how much prioritization is used, with $\alpha = 0$ corresponding to the uniform case. The priority p_i of picking experience i is either one of two variants:

$$p_i = |\delta_i| + \epsilon \qquad p_i = \frac{1}{rank(i)} \tag{12}$$

with ϵ being a small positive constant that prevents the edge-case of experiences not being revisited once their error is zero. In the other variant $rank(i)$ is the rank of experience i when the replay memory is sorted according to $|\delta_i|$. In this thesis, the first variant is used, called proportional prioritization.

To correct the introduced bias prioritized replay experience uses a technique called *importance sampling* (IS), normally used for estimating properties of a particular distribution, while only having samples generated from a different distribution than the distribution of interest. To integrate this in the DQN an importance sampling weight w_i is calculated for each experience in a batch:

$$w_i = \Big(\frac{1}{N} \cdot \frac{1}{P(i)}\Big)^\beta \tag{13}$$

with the number of experiences in the replay memory N and an exponent β to control the amount of compensation for the non-uniform probabilities when selecting experiences, fully compensating with $\beta = 1$. These important sampling weights are then used to scale the TD errors δ, reducing the gradient magnitudes of experiences that are replayed often because of their high TD error. Because the unbiased nature of the updates is most important near convergence at the end of training β is linearly annealed from its initial value β_0 to 1.

There are three differences between the implementation used in the original prioritized replay experience paper and the one used in this thesis based on initial results. These results, displayed in appendix A, showed that the original implementation was instable and has a worse final performance in the environment used in this study. The changes were made to reduce this instability and increase final performance.

First, because a forward pass through the neural network is not that computational intensive with only one hidden layer, the TD error for the experiences in a batch is calculated again after optimization. Then that δ is used for calculating the probability distribution for selecting experiences in the next iteration. This makes it less likely that experiences are picked again when there is not much to learn from them anymore.

Second, no normalization on the importance sampling weights is done. In the original implementation these weights are normalized by scaling them with $1/\max_i w_i$. Because the smallest TD error in the replay memory approaches zero in the later stages of training, the chance of those experiences getting picked approaches zero. Consequently, the maximum importance sampling weight increases so much that the losses are reduced to a really small value. Also, since the Adam optimizer keeps an pair of running averages like mean/variance for the gradients these fluctuating losses do not help.

The last difference is only in the last test, where last experiences are added to the batch. This guarantees all the samples are used at least once.

2.1.3 Age

This sampling technique is a variation of the prioritized sampling technique proposed in this thesis. Because stochastic gradient descent updates may result in significant changes to the policy, it can change the distribution of states observed from the environment. This, in turn, may lead to incorrect gradient estimates. In this sampling method, an age-factor η is proposed. This factor is used in calculating the priority of each experience, lowering the priority of older samples. The idea is that this accounts for and limits the dissimilarity between the current policy and past behaviors in the replay memory. The age-factor is calculated using the age κ_i, the number of experiences added to the replay memory after experience i.

$$\eta_i = \kappa_i^{-0.05} \tag{14}$$

with the oldest age κ being the size of the replay memory. With a replay memory size of 50000 this would mean the age-factor is in the range of 1 to 0,58. The exponent of 0.05 was chosen after a small grid search. This η is then factored into the probability distribution:

$$P(i) = \frac{\eta_i \cdot p_i^\alpha}{\sum_k \eta_k \cdot p_k^\alpha} \tag{15}$$

Keep in mind that by changing the probability distribution a bias is introduced. This bias is compensated in the later stages of the exploitation phase using importance sampling, because the unbiased nature of the updates is most important near convergence [13].

2.1.4 Sequences

This sampling technique, proposed in this thesis, tries to increase the speed of convergence. Most experience replay approaches use individual experiences, however replaying sequences of experiences could offer certain advantages. For example, if an action-value function update results in a relatively large change in the value of the corresponding state-action pair, this change will have a considerable influence on the bootstrapping targets of state-action pairs that led to this experience. If instead of individual experiences, sequences of experiences are replayed, the propagation of this change can be achieved faster. The idea is that these sequences, when replayed, allow the action-value function information to trickle down to larger sections of the state-action space, thereby making the most of the agent's experience. This is similar to eligibility traces, an effective technique to accelerate reinforcement learning by smoothly assigning credit to recently visited states. Experience replay makes the implementation of eligibility traces impossible because states are not processed in the order they are visited [27].

In this sampling technique, half the experiences in a batch are chosen using the prioritized approach. Then the experiences preceding those chosen prioritized experiences are added to the batch. Importance sampling is done only on the experiences selected using the prioritized sampling technique. The other experiences get the same importance sampling weight as their successive prioritized experiences.

As with the other sampling techniques, in the last test, experiences that are new to the replay memory are added to the batch.

2.1.5 Hybrid

This sampling technique, proposed in this thesis, attempts to combine the advantages of random and prioritized sampling. Random sampling in replay memory stabilizes stochastic gradient descent (SGD) by disrupting temporal correlations and extracts information from useful experiences over multiple updates. Prioritized sampling samples more frequently those experiences from which there is more to learn, but thereby changes this distribution of the experiences. In this hybrid sampling technique, half the experiences selected are random samples, the other half are selected using the prioritized technique described above. The idea is that this way advantages of both random and prioritized sampling can be used.

2.1.6 Prioritized memories

This sampling techniques, proposed in this thesis, is based on the idea of awake replay, the sequential reactivation of hippocampal place cells that represent previously experienced behavioral trajectories in the awake state [28]. In normal DQN a single actor interacts with a single environment. This prioritized memories sampling technique requires an extra actor to act in parallel and a backup of the complete state of the environment at every step. This is impossible in a real-world setting, but it can be done in virtual environments. This backup does require a large amount of memory, especially in more complex environments.

It works like this, actor one acts in its own environment and sends its experiences to a central replay memory. The environment of this actor at each step is also saved in the replay memory. Then, every 100 experiences the replay memory receives from this actor, an experience from this actor is selected for replay using the prioritized approach. Actor two then loads the environment just prior to the selected experience, 25 steps earlier. Actor two performs 30 steps in this environment and sends its experiences to the same central replay memory. This should fill that replay memory with experiences around surprising experiences.

Both actors periodically receive the Q-network from a central learner, elaborated in section 3.2. The prioritized sampling technique is used on the central replay memory to pick a batch of experiences for the learner to learn from.

2.2 Question

This study compares these six sampling techniques and tries to answer the research question:

What is the effect of different sampling techniques on the learning behavior of an agent using DQN in a navigation task in a partially observable environment?

The question is based on the idea that learning can be made more efficient by prioritizing updates in an appropriate order. So it is expected that there are techniques that make more effective use of the replay memory for learning. In section 2.1 sampling techniques are discussed two of which have already been studied in previous work, the random and prioritized sampling techniques, while the other four are based on those techniques. The effect of these sampling techniques on the learning behavior will be quantified using three metrics: the maximum performance, learning speed, and stability. These metrics will be elaborated in section 4. The next section will describe the environment and the system architecture used.

3 Method

To validate the sampling techniques proposed in this study to real-world settings, an experiment using a mobile robot is preferred. However, to find the effect of sampling techniques on deep learning algorithm DQN a simpler virtual world will suffice. That is why the tests in this study are done in a simulation that can be adapted to a real robot with a small amount of fine-tuning. A lot of frameworks have been proposed to create accurate simulations of navigation tasks in real-world environments. For example, the THOR framework[29], which provides an environment with high-quality 3D scenes and physics engine, or Labyrinth a first-person 3D game platform extended from OpenArena. These environments, however, have rich visuals, realistic physics, textures are often dynamic so as to convey a game world where walls and floors shimmer and pulse, adding significant complexity to the perceptual task. This, in turn, adds a lot of simulation and training time.

One problem of using reinforcement learning in navigation tasks is that rewards are often sparsely distributed in the environment. This means a lot of exploration to find the goal location, especially dynamic goal locations. Previous work solves this problem by supplying the robot with a goal location with respect to the robot [2] or by augmenting the loss with auxiliary tasks, for example predicting if the current location has been visited before [30]. In this thesis, the effect of sampling techniques on learning performance in navigation tasks is investigated.
In the following sections the environment is described and then there is a section about the architecture used for simulation.

3.1 Environment

The environment created in this study is inspired by Freeway the video game designed by David Crane for the Atari 2600 video game console. In this game, one or two players control chickens who can be made to run across a ten lane highway filled with traffic in an effort to "get to the other side."

There are however five key differences between the simulation in this study and Freeway the game. First, the robot in this simulation is allowed to travel in all directions, not only up or down. Secondly, there is only one car, if it is hit the simulated robot re-spawns on its start location. Third, contrary to the two minutes, sixteen seconds in the original game there is no time limit in this simulation. Fourth, when the other side is reached the robot does not re-spawn at the bottom but instead has to cross the road in the other direction. Lastly, there is no cluck sound when a chicken/robot is struck by a car. A screenshot of the simulation used in this study can be seen in figure 1.

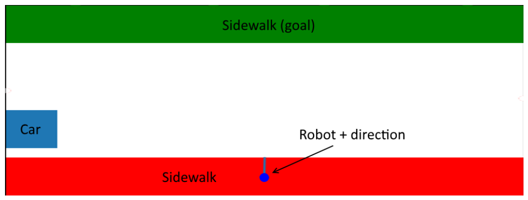


Figure 1: Road environment.

There are three main elements in this environment: the robot, car, and sidewalks. All sizes are expressed in meters and modeled on their size in the real-world.

To make the simulation resemble a real-world scenario a stretch of road with a length of 40 meters is simulated. The road is 6 meters wide and has two sidewalks with a width of 2 meters next to it. One of the sidewalks, represented in green, is the goal sidewalk. When the road is crossed the other sidewalk becomes the goal sidewalk.

The car is on the right side of the road moving from left to right on the screen. It is 4 meters long and 2 meters wide. Because this is a crossing the car has a slow walking pace of 1.4 m/s, this remains constant throughout the simulation. When the car crosses the border on the right side of the screen the car re-spawns on the left side.

The robot spawns in the middle of the sidewalk on the bottom of the screen. The speed and turn speed are derived from the preliminary research into the real-world robot Pepper. This robot has a top speed of 0.83 (3 km/h) and turn speed of $\frac{\pi}{2}$ rad/s (90° per second), these also remain constant during the simulation. Each step the robot can choose one of three actions: forward, left or right. The agent observes its environment using simulated laser range sensors, based on the laser sensors used for collision avoidance in Pepper. There are 20 lasers spread out evenly from -120° to 120° with respect to the orientation of the robot, notice that there is no laser pointing directly forward. These simulated lasers have a maximum range of 15 meters, this means the robot is always in range to detect at least one of the borders of the environment. When by moving forward one of these borders is crossed the car does not move instead. Unlike a real robot, this simulated robot has no size, it is represented by a single point and angle.

The agent receives a reward of 1 if the road is crossed and -1 if a car is hit, called extrinsic rewards. An additional reward of 0.01 is given for moving forward this is called an intrinsic reward and is a common way to encourage exploration [31]. Trying to cross the border results in a reward of 0.

When testing the amount of laser range sensor readings per second Pepper could send an average of about 4 readings per second was found. This is why one discrete time step in this simulation is 0.25 s. The amount of readings per second has a large effect on the size of the state space, the amount of exploration needed to find the goal, the difficulty of evading the car, and the ratio between intrinsic and extrinsic rewards. In practice, this time step means all speeds are divided by four to get the speed per step.

3.2 Architecture

With the environment in place now the architecture used in this thesis can be discussed. The architecture used in this study is inspired by A3C, mentioned in section 1.2, and the Gorila architecture [32], the first massively distributed architecture for deep reinforcement learning. In this research DQN also acts and learns in parallel, using a distributed replay memory and distributed neural network. This leads to a massive reduction in training time when training on a single CPU with multiple cores. This architecture divides the algorithm into three parts: Actor, Learner and Replay memory. Figure 2 is an overview of this architecture. The solid arrows show the data streams in its basic form, while the dotted lines are data streams that are added when more advanced sampling techniques are used.

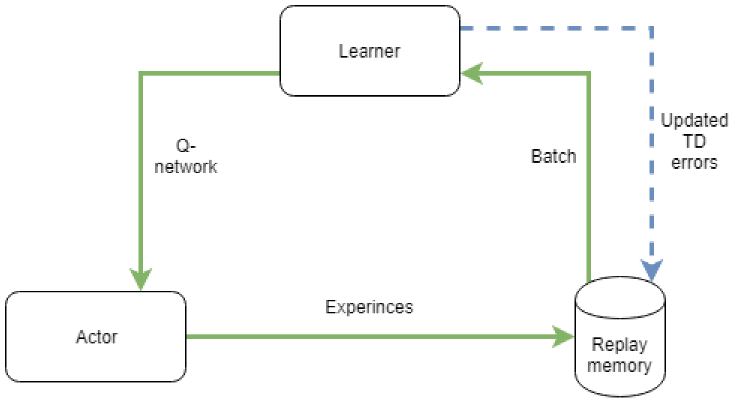


Figure 2: Overview of the architecture

Mainly by separating the learning and acting part in different processes acting on separate cores within the CPU a speedup by a factor two was achieved. The only thing the actor has to do is generate experiences and periodically update its policy with the latest policy received from the learner. While the learner's task only consists of optimizing the policy.

3.2.1 Actor

Each step the agent in the actor process must select actions a_t to apply in its environment. To do this the agent uses a behavior strategy called ϵ-greedy, described in section 1.2. A random action is chosen for a proportion ϵ of the steps, the other steps an optimal action is chosen according to the latest policy. This policy is received from the learner in the form of a Q-network, which is used to determine the action with the highest value. The weights in this Q-network are updated with values received from the learner.

The input of the policy is the 20 lasers range readings. These only provide the distance to a obstacle, not if that obstacle is a car or wall. Two steps of preprocessing are done. First, the readings are normalized, this way all input values are a value between 0 and 1 depending on the distance to an obstacle, where 1 is the maximum distance of the sensors. Secondly, the last four readings are stacked to represent a state s_t, resulting in a 80 dimensional input vector. This effectively gives the algorithm a measure of velocity for moving objects. The output of the Q-network are values for the three actions: forward, left and right. Either a random action is chosen or the action with the highest value, using the ϵ-greedy strategy.

Once an action is chosen the actor interacts with the environment and receives a reward and perceives the next state. Another variable, collision a boolean, is used to keep track of the experiences a collision with the car occurred. When the target of an experience with a collision is calculated only its reward, -1, is considered not the value of the next state, because the next state is a crash. The state, action, reward, collision and next state are then sent to the replay memory as an experience. Lastly, the next state is used as the current state and the cycle is repeated. If there is a collision the environment is reset and a new state is created by performing four random actions. In figure 3 an overview of the process of an agent.

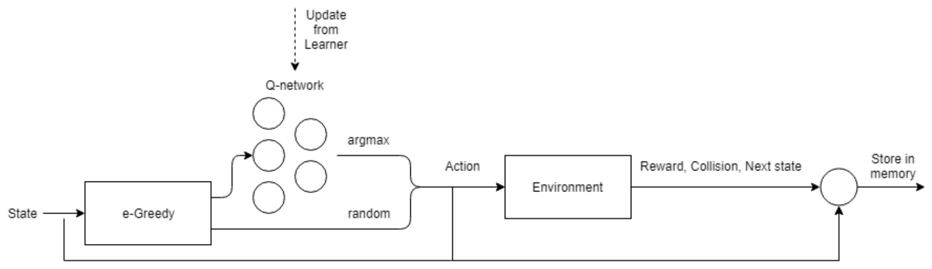


Figure 3: Overview of the actor

3.2.2 Memory

The experiences generated by the actor are stored in a replay memory. It stores the latest 50000 experiences including their collision value discarding the oldest experiences in favor of new ones. Then, at regular intervals, it samples a batch of 32 experiences using one of the six sampling techniques described in section 2.1 and sends them to the learner.

When using a prioritized approach a TD error is associated with every experience. The newest experiences are always sent to the learner to obtain a TD error for each experience. Each time an experience is sent to the learner its TD errors is updated using their latest TD error returned by the learner.

3.2.3 Learner

The learner is tasked with computing the desired changes to the parameters of the Q-network. First, it receives a batch of experiences from the replay memory. From each experience the learner begins with processing the state, next state and reward to calculate targets using equation 10. When an experience contains a collision only the reward of -1 is used as target. The next step is to find the current action value $Q(s, a)$ using the state and action in an experience from the batch. This current action value is subtracted from the target, this results in the TD error δ. This TD error is used in a loss function called Huber loss:

$$L(\delta) = \begin{cases} \frac{1}{2}\delta^2 & |\delta| \leq 1 \\ |\delta| - \frac{1}{2} & |\delta| > 1 \end{cases} \tag{16}$$

where c is 1. Huber loss is used because it is less sensitive to outliers, experiences with large TD errors scale linearly. This, unfortunately, reduces the effect of sampling techniques. However, Huber loss is needed to limit the gradient of the loss and thereby stabilize the learning behavior. In case of using importances sampling this loss is weighted by the importance sampling weight of each experience. The loss is then used to update the weights of the Q-network using back-propagation in combination with an advanced stochastic gradient decent algorithm like Adam to minimize the loss.

The Q-network used in this study only has one hidden layer with 79 neurons, this size is chosen to have enough hidden neurons to learn a decent policy while also keeping the number of neurons between the number of input and output values [33]. This hidden layer uses a rectified linear unit (ReLU) as activation function, this means the output of a neuron is 0 if the sum of the weighted inputs is negative and the value of that sum otherwise. As described in section 1.2 there are actually two networks with the same structure: the online and target network. The online network is used to select actions and the target network is a periodic copy of the online network which is not directly optimized. The use of a target network enables relatively stable learning of the action value function.

After the weights are updated and if a prioritized sampling technique is used the TD error of each state is calculated again in order to send the most recent value back to the experience replay. Lastly, the learner sends the weights of the Q-network periodically to the actor. An overview of the learning process is given in figure 4.

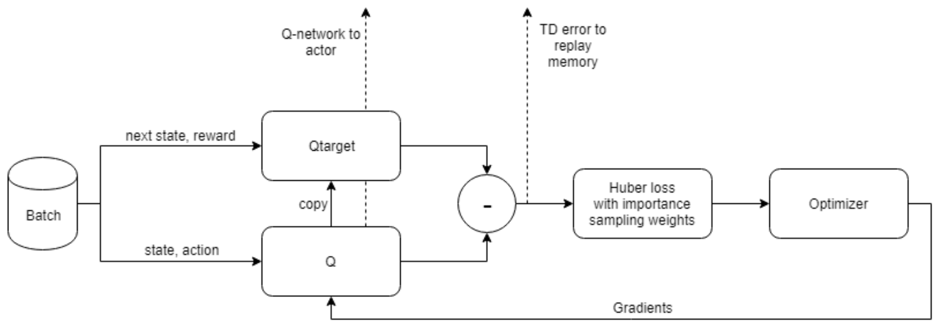


Figure 4: Overview of the learner

3.3 Measures

Testing using the performance of the training agent is problematic. It is difficult to differentiate between an increase in performance due to a better policy or because of a decrease in exploration. This is why a separate actor process is created on a different CPU core where a test agent runs in parallel on its own environment. This test agent receives the same Q-network from the learner but sends no experiences to the replay memory. There are three main differences between the train and test agent. The test agent has an ϵ-greedy policy with $\epsilon = 0.001$, this means it practically only takes the action with the highest action value in the current policy. The small value of 0.001 was chosen to model some noise on the action space that could be present in a real world system. Secondly, the test run resets every 1000 steps to prevent it from getting stuck due to a suboptimal temporary policy. Lastly, the test run only counts the reward for getting to the other side of the road, not the intrinsic reward for moving forward, because this is the true performance of the agent.

The performance metric used in this thesis to quantify an agents performance μ is the moving average of the reward as a fraction of the theoretical maximum reward. The theoretical maximum reward is theoretical maximum performance in the environment without the car. It can be calculated because the fastest way to the other side is obvious and the rewards for this route are known. Four aspects of the performance are important: final performance, learning stability, maximum performance and learning speed.

Final performance μ_r^{final} is calculated using the average of the mean performance per step over the last 100000 steps of a test. This metric expresses the average quality of a policy that the sampling technique used in that test converges to.

The stability of the learning process σ_r^{final} is calculated using the standard deviation of the moving average in the final step of a test. Even when a good policy has already been learned the learning process can become unstable and the final performance can vary significantly, a low σ_r^{final} is therefore desired.

The maximum performance μ_r^{max} is the maximum performance of all runs in a test. While stability is desired, it is important not to forget that reinforcement learning is about finding the true optimal policy, i.e. the policy with the highest returns.

The last metric sampling techniques are judged on is learning speed $\mu_r^{0.8}$. This metric will be expressed in rise time 0.8, i.e. the step at which the performance is 0.8 * μ_{max}^{final} divided by the total steps in the test. Where μ_{max}^{final} is the maximum final performance, μ_r^{final}, of all the sampling techniques. This metric can only be calculated after all final performances are known and the metric is infinite for tests that never reach the 0.8 * μ_{max}^{final} threshold.

3.4 Experimental configuration

The hardware and software configurations used for testing will be explained below.

3.4.1 Hardware

Testing was done on the INSY cluster, a collection of large computing resources. The compute servers are all multiprocessor machines running Linux. All machines have multiple central processing units (CPUs) that perform all the computations. Each CPU can process one thread (i.e. a separate string of computer code) at a time. This means the actor, learner and replay memory each need there own CPU, as well as a few other parts of the program like the main program starting all these processes and collecting their data. This is why 8 CPUs are requested for each test. Tests for every sampling technique are done in parallel, this means using 48 CPUs simultaneously. A test consists of 50 times one run, i.e. 800000 steps, each time resetting all parameters. The training time of one run is about 45 min, making the time to test all sampling methods once 37 hours and 30 minutes.

3.4.2 Software

To test the effect of different sampling techniques all hyper-parameters have to be set to the same values for each test. In table 1 all these hyper-parameters are presented with a some of the reasoning behind them.

Table 1: Hyper-parameters

Hyper-Parameter	**Value**	**Explanation**
Steps	800000	Length of a run
ϵ_0	1.0	Start value of ϵ
ϵ_{end}	0.01	Exploration linearly decreased to this value
Exploration steps	200000	Step where $\epsilon = \epsilon_{end}$
Q-network update rate	4	Balance computational cost with accuracy
γ	0.99	Discount factor
Learning rate	0.001	Based on a small grid search
Target network update rate	100	Based on a small grid search
Size replay memory	50000	Based on previous work
Batch size	32	Balance computational cost with accuracy
Sample rate	4	Balance computational cost with accuracy
Prioritized α	0.6	Based on previous work
Prioritized β_0	0.4	Based on previous work
Prioritized β_{end}	1.0	Based on previous work

4 Results

With all the most important concepts in place, the results of the tests will now be elaborated in this section.

4.1 Baselines

In figure 5 all the sampling techniques are tested without importance sampling and without the addition of the most recent experiences.

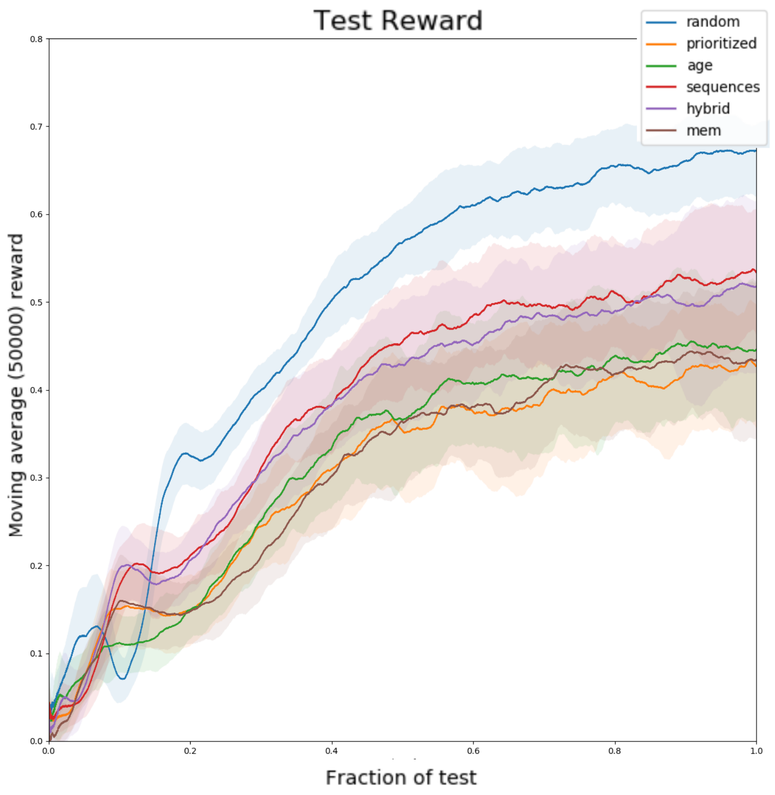


Figure 5: Performance of the test agent, all lines averaged over 50 runs. On the vertical axis the moving average of rewards as a fraction of maximum reward. On the horizontal axis the step of a test run as a fraction of the amount of steps that test agent could perform in the time the training agent needed to perform 800000 steps.

The random technique has the highest performance. This is expected because this technique feeds the optimizer more independent and identically distributed (i.i.d.) data. This means the experiences sampled have no correlation other than they are in the replay memory and each sample has the same probability distribution as the others. Because the random technique samples experiences across the whole state-action space slightly more steps are needed before its performance increases, but when it does increase it surpasses all other techniques.

The sampling techniques most dependent on the TD error such as prioritized, age and prioritized memories without importance sampling clearly perform significantly worse. In these techniques, the beneficial re-sampling of the state-action space distribution is not compensated by the unwanted re-sampling of the environment dynamics and reward distributions. Without importance sampling the need for experience diversity and i.i.d. data is not fulfilled, the function approximation is trying to optimize mostly on experiences with a crash or where the sidewalk is reached as these tend to be the most surprising. Interesting is that these techniques increase their performance first during exploration. As exploration decreases there are more crashes and sidewalks reached, this in turn skews the environment dynamics and reward distribution in the batch.

The two techniques that seem to be the most resistant to this prioritized sampling are the more random oriented techniques hybrid and sequences. These techniques both choose half of the experiences in their batch using the TD error. Because of this, they seem less affected by the chance of environment dynamics and reward distributions. Whether the other half of the experiences is strongly correlated to the first half, the preceding experiences, or random experiences does not seem to have any influence.

4.2 Performance with importance sampling

In figure 6 the performances of different sampling techniques using importance sampling are compared to the random sampling baseline.

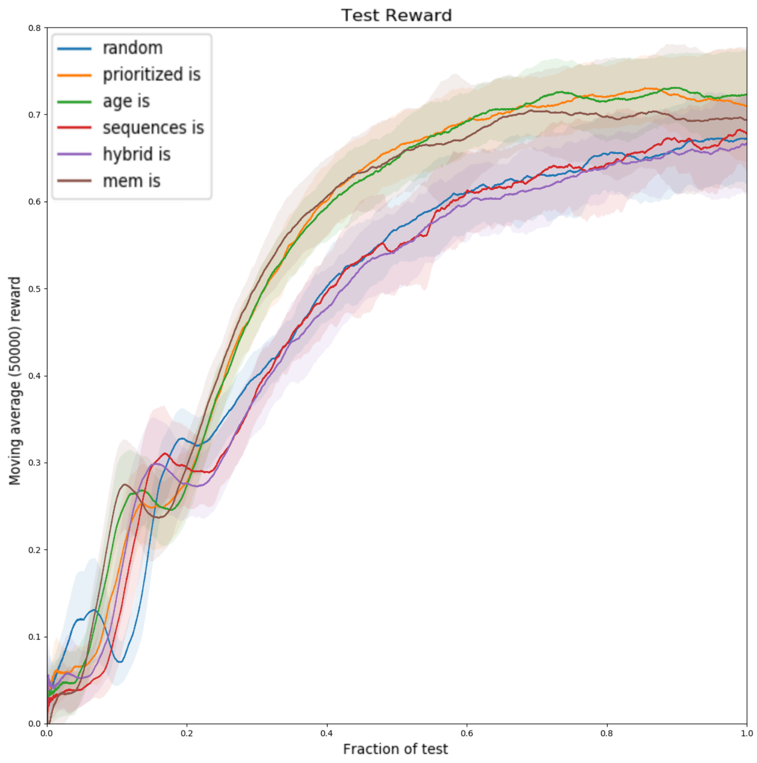


Figure 6: Performance of the test agent, all lines averaged over 50 runs. On the vertical axis the moving average of rewards as a fraction of maximum reward. On the horizontal axis the step of a test run as a fraction of the amount of steps that test agent could perform in the time the training agent needed to perform 800000 steps.

All performances increase as exploration decreases, but some increase their test performance before others. The more prioritized based techniques increase their performance first with the more random oriented techniques following. The first technique to increase is prioritized memories with importance sampling and last are the more random techniques.

All lines display a decrease in performance after an initial increase. The whole task of getting to the other side can be split up in three subtasks it has to learn. First, the robot needs to find the other sidewalk and turn around. The next task is to evade the car. And lastly it needs to find the optimal route to evade the car and get to the other side. The shape of the lines clearly shows these task being learned in succession. As the robot makes more greedy action choices on the first task, to get to the other side, the more likely it is to crash into the car. Then after crashing into the car a couple of times the next task, evading the cars, is learned. Then slowly the last task of optimizing the path is done.

The techniques that use a more prioritized based technique peak first but lower than random ones. Because of the decrease in exploration more collisions are made, this, in turn, increases the number of experiences with a high change of being sampled. This means the progress optimizing a path to the other side is slowed down to focus on the task of evading the car. This is especially apparent in the prioritized memories sampling technique, it does increase its performance relatively early in the exploration phase but when exploration is decreased more crashes are made. This, in turn, means more crash experiences are added skewing the samples. When more importance sampling is introduced by increasing β this is compensated and the performance starts to increase again after 0.2 fraction of test. With a memory full of surprising experiences the prioritized memories technique can improve its performance slightly earlier than both the prioritized and age techniques.

After the exploration phase prioritized based techniques with importance sampling are better at finding that optimal path compared to random oriented techniques. Because of the importance sampling, they perform a lot of small updates on states with large errors, this leads to a faster increase in performance. The more random oriented techniques all follow the same, slower, learning trajectory.

In the end all sampling techniques converge to the roughly the same performance with a slight edge to the prioritized methods.

4.3 Adding recent experiences

A final test was done to check how adding the latest experiences to a batch influences the performance of the different sampling techniques. This means instead of the usual 32 experiences chosen for a batch 28 experiences are chosen and the four most recent experiences are added to that batch. In figure 7 test runs are plotted with importance sampling and adding the last experiences.

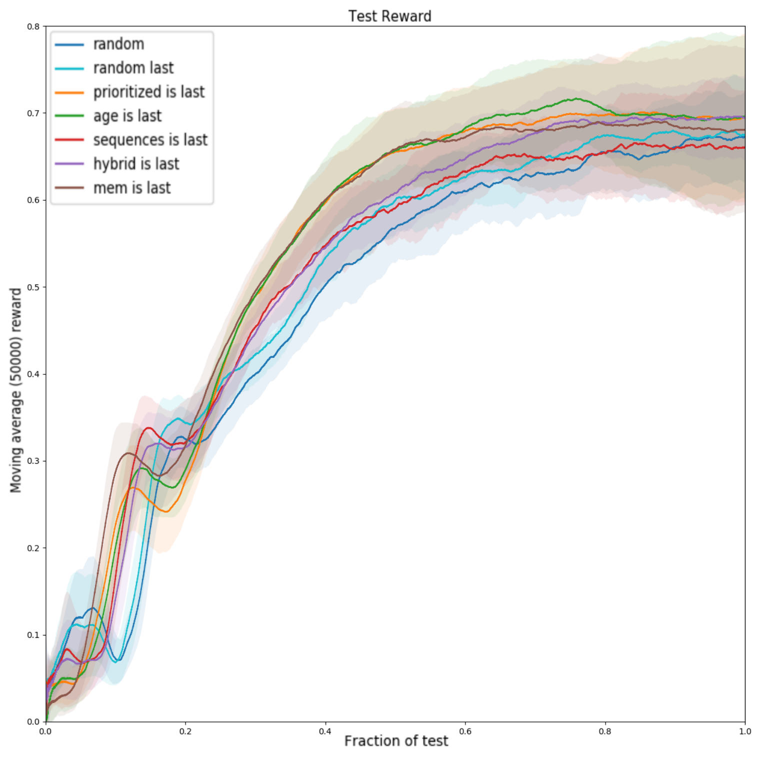


Figure 7: Performance of the test agent, all lines averaged over 50 runs. On the vertical axis the moving average of rewards as a fraction of maximum reward. On the horizontal axis the step of a test run as a fraction of the amount of steps that test agent could perform in the time the training agent needed to perform 800000 steps.

These results are almost the same as results of the last test. There are however two slight differences.

After the exploration phase the techniques with half the experiences in a batch prioritized and the random technique with the most recent experiences added increase their performance marginally faster than pure random sampling. The addition of the most recent experiences could be helping those techniques because these experiences are not compensated by importance sampling since their TD error is not known yet. This, in turn, could be helping to create a batch representing the current environment dynamics and reward distribution more accurate.

In the end, all sampling techniques converge to almost the same performance. However, the final performances of those techniques with the most recent experiences added do not outperform the performances without the most recent experiences added.

4.4 Performance Metrics

In the following table, the performance measurements, as described in section 3.3, are shown. The **bold** numbers are the best values in the column. For μ_r^{final} and μ_r^{max} the best value is their highest value, representing the best final and maximum performance. For σ_r^{final} and $\mu_r^{0.8}$ the best value is their lowest value, representing the lowest variance and most rapid increase in performance.

Table 2: Performance metrics

Method	μ_r^{final}	σ_r^{final}	μ_r^{max}	$\mu_r^{0.8}$
Random	0.6688	0.0517	0.7704	0.5233
Random+last	0.6747	0.0637	0.7667	0.4671
Prioritized	0.4256	0.0711	0.6415	∞
Prioritized+is	0.7168	0.0636	0.8341	0.3728
Prioritized+is+last	0.6944	0.0975	0.8156	0.3747
Age	0.4490	0.0776	0.6607	∞
Age+is	**0.7237**	0.0521	0.8178	0.3799
Age+is+last	0.6943	0.0926	0.8037	0.3773
Sequences	0.5267	0.0719	0.6956	∞
Sequences+is	0.6540	0.0737	0.7511	0.4588
Sequences+is+last	0.6618	0.0673	0.7970	0.4706
Hybrid	0.5075	0.0982	0.6844	∞
Hybrid+is	0.6582	0.0574	0.7570	0.5563
Hybrid+is+last	0.6938	**0.0495**	0.7911	0.4352
Prioritized_mem	0.4363	0.0890	0.6481	∞
Prioritized_mem+is	0.6945	0.0839	**0.8348**	**0.3603**
Prioritized_mem+is+last	0.6819	0.0950	0.8156	0.3755

In this table, it becomes clear where adding the last experiences helps. The more random oriented techniques (random, sequences, and hybrid) benefit from the addition of the newest experiences while adding them to the more prioritized techniques (prioritized, age, and prioritized memories) hurts the performance of these techniques.

The prioritized and age techniques are clearly the best performing techniques, however, the other performance metrics show this is only in terms of final performance. Hybrid with importance sampling and the last experiences added has decent final performance and the lowest σ_r^{final}, this means more runs actually achieve that performance.

The prioritized memories sampling technique has both the best run and the fastest rise time. This shows the potential of this technique, however, it suffers from some instability. It also needs to be noted that this technique adds extra experiences to the replay memory so the memory fills up in less training agent steps, this could be the reason for the fast rise time.

5 Discussion

In this work, existing and novel sampling techniques are investigated by applying them to a navigational problem. By doing so a search is done to find sampling techniques that make experience replay more efficient and effective than simple random sampling in navigation tasks of mobile robots with range sensors. In this section the fidelity of the simulations will be discussed, then this study is compared to recent work and some recommendations for future work are done.

5.1 Simulation

The difficulty of transferring simulated experiences into the real-world is often called the "reality gap." The reality gap is a subtle but important discrepancy between reality and simulation that prevents the simulated robotic experience from directly enabling effective real-world performance. The challenge with simulated training is that even the best available simulators do not perfectly capture reality. Models trained purely on synthetic data fail to generalize to the real-world, as there is a discrepancy between simulated and real environments, in terms of both visual and physical properties. The simulated environment used in this study has three main differences with the real world.

First, the environment is perfectly symmetrical, the walls around the road are straight lines. This means there are a lot of duplicate states. This increases the difficulty of a navigation task since there is less information to learn from.

The second issue is with the laser sensors, in the simulation, these only have a small amount of artificial noise added. Experiments using a real robot showed however that the noise of some of these laser range finder sensors is not following the standard normal noise distribution used in the simulation. Instead, the sensors either showed the maximum or minimum range if a faulty measurement was done.

Finally, there also is the issue of re-spawning, placing the environment in its initial state when the robot gets hit by a car. This is trivial in a simulated environment, but re-spawning a real robot is often more difficult.

5.2 Comparison to recent work

In this section, the study is linked to existing research on the topic of experience replay and navigation.

5.2.1 Prioritized replay experience

Most of the sampling techniques researched in this study are based on the paper about prioritized replay experience [13]. It would only be appropriate to try to make a comparison between these works. There are some differences in the environment as discussed in section 3.1, however, the task is almost the same, i.e. to get to the other side of the road. In figure 8 learning curves (in raw score) for Double DQN (uniform baseline, in black), with rank-based prioritized replay (red), proportional prioritization (blue), for the freeway game of the Atari benchmark suite. Each curve corresponds to a single training run over 200 million unique frames.

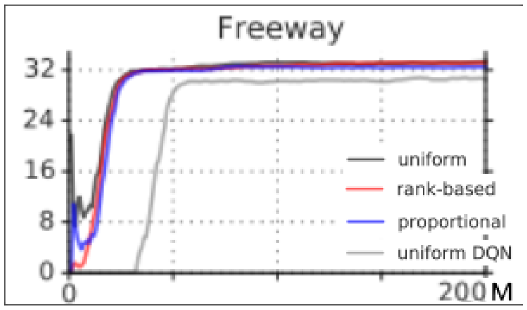
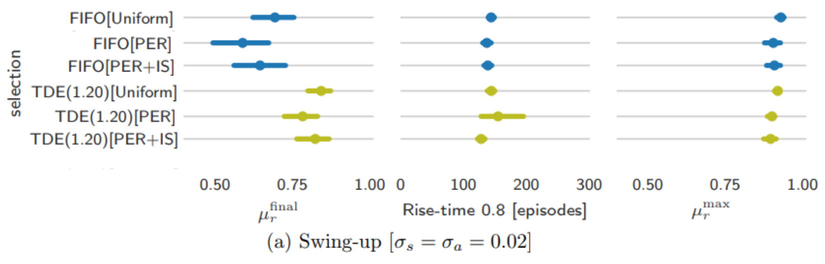


Figure 8: Learning curves from the prioritized replay experience paper for the freeway game. On the vertical axis the raw score of a game. On the horizontal axis the amount of frames.

The figure shows that using pixel input in the freeway game there is no significant difference in using prioritized replay experience and not using it. While this thesis found that in a similar task using local navigation, i.e. using range sensors as input, there is a difference between sampling uniform and prioritizing certain experiences.

5.2.2 Experience retention

In this recent paper [34] methods for experience retention are discussed. When the experience memory is full, some metric should decide which experiences should be overwritten. In that study, they consider three criteria for overwriting experiences: age, surprise, and exploration. Two benchmark problems are considered in that paper. In the pendulum task, an underactuated pendulum needs to be swung up and balanced in the upright position by controlling the torque applied by a motor. In the magnetic manipulation (magman) task, a steel ball (top) needs to be positioned by controlling the currents through four electromagnets. A selection of the results relevant for comparison to this research is shown in figure 9.



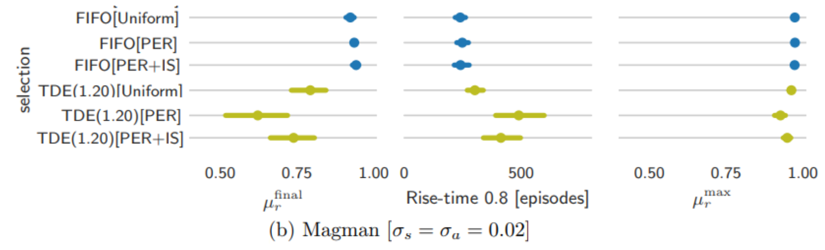


Figure 9: Performance of representative experience selection methods in [34] with and without importance sampling on the benchmarks with sensor and actuator noise.

FIFO, first in first out, is the method of retention used in this study. The most recent experiences are added to the replay memory and oldest are thrown out, this is the method used in this thesis. TDE is based on the temporal difference error: the memories with the lowest temporal difference are thrown out. In brackets is the sampling technique used, PER is the same prioritized technique used in this study and IS means importance sampling.

The effect of different retention methods in the two benchmarks is clearly visible in the results. They conclude that the need for diverse states and actions largely depends on the ease and importance of generalizing across the state-action space, which is benchmark dependent. On the swing up task, TDE is the better method while the FIFO method of retention is better for the magman task. This result makes it difficult to compare the results to this study. Still, there are two interesting comparisons to be made between the two studies.

First, in both benchmarks, there is no difference in rise-time 0.8 over all FIFO tests, while in this thesis a clear difference is shown. In this thesis PER without IS performing worst and PER with IS having a faster rise-time 0.8 than the uniform method. This is mainly because the two benchmarks, swing-up and magman, are fully observable and have a continuous reward signal. This enables any sampling technique to learn near optimal policies relatively quickly.

Secondly, the TDE[PER + IS] method, comparable to the prioritized memories technique, hurts the final performance of the magman task while in the results of this thesis the technique does not hurt performance and even has the fastest rise-time 0.8. This could be because the two benchmarks have more noise added to the sensors and actuators. This, in turn, increases the tendency of TD error based techniques to seek out noisy experiences and that is hurting performance.

5.3 Recommendations for future work

Interesting directions for future research are proposed in this section.

5.3.1 Noise

Recent work by Open AI shows that adding noise to every aspect of the simulation helps to transfer in simulation learned behavior to physical robots [35]. In this study, only a small amount of noise is used on the laser range readings, many more aspects of the agent and environment could be randomized.

5.3.2 More actors

Using a lot of parallel agents each generating experience in their own environment and sending information to a central learner is a popular research direction recently[36][37]. The benefits are clear, more less correlated experience is created. In the prioritized memories sampling technique proposed in this research one extra actor is created. This approach could be scaled to as many actors as there are free CPUs.

5.3.3 Network architecture

Another important topic open for discussion is neural network architecture. In this study this part is kept as simple as possible using an architecture with just one hidden layer. There are however many other options in areas for example number of layers, number of hidden units per layer, activation function, optimizer, and learning rate. With navigational problems, in particular, the option of recurrent neural networks is an interesting direction for future research. Although in this study four concurrent laser sensor readings are concatenated and used as a state, a recurrent architecture could find connections between states over longer periods increasing the performance.

An advantage and actor-critic setup, used in A3C [36], setup could also improve results and help decide what samples should be replayed. The advantage judges actions based on how much better they turned out to be than expected. An actor-critic setup separates the value estimate from the policy in two final fully connected layers. This changes the TD error and could help the prioritized sampling techniques what experiences to replay. An extra benefit of this architecture is that it allows for a continuous action space better suited for robotic control.

5.3.4 Reward

The spare extrinsic rewards and constant intrinsic reward used in this study can be improved upon. For instance, by adding auxiliary tasks [30], in the road environment this could be predicting the goal location (up or down), or by replacing the rewards by curiosity-driven learning [38], that use a reward signal based on how hard it is for the agent to predict the consequences of its own actions. By improving the reward signal the TD error changes helping the prioritized sampling techniques to choose the optimal experiences to improve learning.

6 Conclusion

To our knowledge, this is the first study that systematically investigates a significant set of sampling strategies in a partially observable environment. A number of sampling techniques are presented and explored. All these sampling techniques are tested in a simulated environment with and without importance sampling. Based on the results three conclusions can be drawn and two recommendations are done.

First, prioritizing all experiences in a batch with larger TD errors improves the speed of learning, demonstrated in the prioritized, age, and prioritized memories sampling techniques. Contrary, more random based sampling techniques, like hybrid, sequences, and random, increase their performance slower.

Secondly, when given enough training time all sampling techniques converge to almost the same performance. In the later stages of training less high TD error experiences are encountered making the difference between sampling techniques smaller.

Finally, it can be concluded that importance sampling has a significant influence on the performance. When used correctly it can increase final performance and stabilize learning, especially for the more prioritized based techniques. Without importance sampling, to correct for the skewing of the state and reward distributions, all prioritized sampling techniques perform worse than simple random sampling.

To facilitate quick and stable learning of a policy with good performance the following recommendation can be done. First, it is best to use a sampling technique based on prioritizing experiences with higher TD errors. Secondly, when doing this, it is imperative to use importance sampling.

Appendix A. Original and proposed PER

In figure 10 the original prioritized experience replay (PER)[13] and the prioritized method proposed in this thesis are plotted.

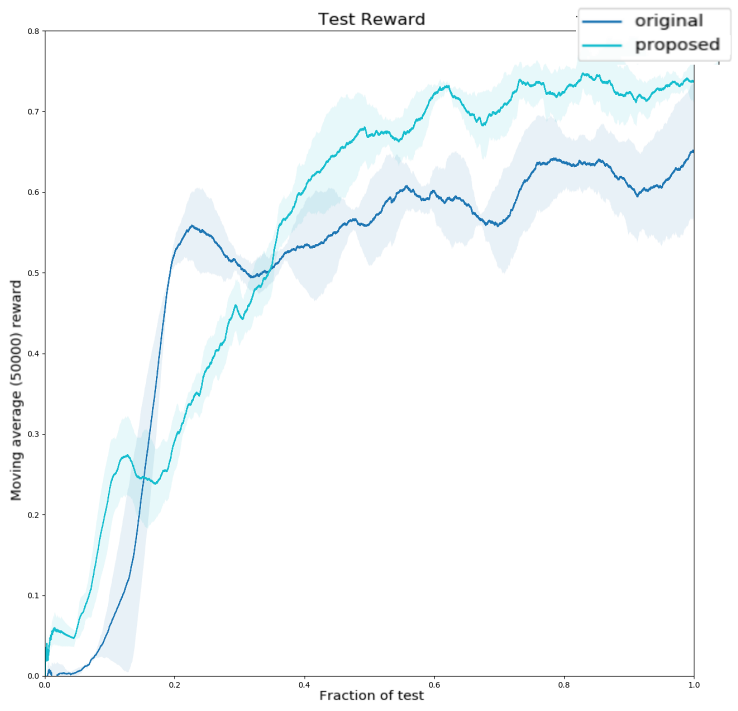


Figure 10: Performance of the test agent, all lines averaged over 3 runs. On the vertical axis the moving average of rewards as a fraction of maximum reward. On the horizontal axis the step of a test run as a fraction of the amount of steps that test agent could perform in the time the training agent needed to perform 800000 steps.

The original prioritized replay technique has a quick increase in performance in the second half of the exploration phase. However, the method proposed in this thesis increases its performance first and has a better final performance. It is also clear that this final performance is more stable. Thus the proposed method is used in this study.

References

[1] Ifr, "Survey: 1.3 million industrial robots to enter service by 2018."

[2] L. Tai, G. Paolo, and M. Liu, "Virtual-to-real deep reinforcement learning: Continuous control of mobile robots for mapless navigation," in *Intelligent Robots and Systems (IROS), 2017 IEEE/RSJ International Conference on*, pp. 31–36, IEEE, 2017.

[3] 953383504824263, "Why deep learning over traditional machine learning?," Mar 2018.

[4] D. Vena, "3 top deep-learning stocks to buy now," Dec 2017.

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[6] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, p. 529, 2015.

[7] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[8] T. Darrell, M. Kloft, M. Pontil, G. Rätsch, and E. Rodner, "Machine learning with interdependent and non-identically distributed data (dagstuhl seminar 15152)," in *Dagstuhl Reports*, vol. 5, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[9] R. M. French, "Catastrophic forgetting in connectionist networks," *Trends in cognitive sciences*, vol. 3, no. 4, pp. 128–135, 1999.

[10] M. Hessel, J. Modayil, H. Van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver, "Rainbow: Combining improvements in deep reinforcement learning," *arXiv preprint arXiv:1710.02298*, 2017.

[11] J. Kober, J. A. Bagnell, and J. Peters, "Reinforcement learning in robotics: A survey," *The International Journal of Robotics Research*, vol. 32, no. 11, pp. 1238–1274, 2013.

[12] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning.," in *AAAI*, vol. 16, pp. 2094–2100, 2016.

[13] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *arXiv preprint arXiv:1511.05952*, 2015.

[14] Z. Wang, T. Schaul, M. Hessel, H. Van Hasselt, M. Lanctot, and N. De Freitas, "Dueling network architectures for deep reinforcement learning," *arXiv preprint arXiv:1511.06581*, 2015.

[15] P. Lison, ""an introduction to machine learning," 2015.

[16] R. Lippmann, "An introduction to computing with neural nets," *IEEE Assp magazine*, vol. 4, no. 2, pp. 4–22, 1987.

[17] P. J. Werbos, "Backpropagation through time: what it does and how to do it," *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1550–1560, 1990.

[18] Y. Li, "Deep reinforcement learning: An overview," *arXiv preprint arXiv:1701.07274*, 2017.

[19] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[20] F. S. Melo, "Convergence of q-learning: A simple proof," *Institute Of Systems and Robotics, Tech. Rep*, pp. 1–4, 2001.

[21] L.-J. Lin and T. M. Mitchell, *Memory approaches to reinforcement learning in non-Markovian domains*. Citeseer, 1992.

[22] F. A. Oliehoek, C. Amato, *et al.*, *A concise introduction to decentralized POMDPs*, vol. 1. Springer, 2016.

[23] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[24] L.-J. Lin, "Reinforcement learning for robots using neural networks," tech. rep., Carnegie-Mellon Univ Pittsburgh PA School of Computer Science, 1993.

[25] L. A. Atherton, D. Dupret, and J. R. Mellor, "Memory trace replay: the shaping of memory consolidation by neuromodulation," *Trends in neurosciences*, vol. 38, no. 9, pp. 560–570, 2015.

[26] H. F. Ólafsdóttir, C. Barry, A. B. Saleem, D. Hassabis, and H. J. Spiers, "Hippocampal place cells construct reward related sequences through unexplored space," *Elife*, vol. 4, 2015.

[27] B. Daley and C. Amato, "Efficient eligibility traces for deep reinforcement learning," *arXiv preprint arXiv:1810.09967*, 2018.

[28] M. F. Carr, S. P. Jadhav, and L. M. Frank, "Hippocampal replay in the awake state: a potential substrate for memory consolidation and retrieval," *Nature neuroscience*, vol. 14, no. 2, p. 147, 2011.

[29] Y. Zhu, R. Mottaghi, E. Kolve, J. J. Lim, A. Gupta, L. Fei-Fei, and A. Farhadi, "Target-driven visual navigation in indoor scenes using deep reinforcement learning," in *Robotics and Automation (ICRA), 2017 IEEE International Conference on*, pp. 3357–3364, IEEE, 2017.

[30] P. Mirowski, R. Pascanu, F. Viola, H. Soyer, A. J. Ballard, A. Banino, M. Denil, R. Goroshin, L. Sifre, K. Kavukcuoglu, *et al.*, "Learning to navigate in complex environments," *arXiv preprint arXiv:1611.03673*, 2016.

[31] N. Chentanez, A. G. Barto, and S. P. Singh, "Intrinsically motivated reinforcement learning," in *Advances in neural information processing systems*, pp. 1281–1288, 2005.

[32] A. Nair, P. Srinivasan, S. Blackwell, C. Alcicek, R. Fearon, A. De Maria, V. Panneershelvam, M. Suleyman, C. Beattie, S. Petersen, *et al.*, "Massively parallel methods for deep reinforcement learning," *arXiv preprint arXiv:1507.04296*, 2015.

[33] J. Heaton, *Introduction to neural networks with Java*. Heaton Research, Inc., 2008.

[34] T. De Bruin, J. Kober, K. Tuyls, and R. Babuška, "Experience selection in deep reinforcement learning for control," *The Journal of Machine Learning Research*, vol. 19, no. 1, pp. 347–402, 2018.

[35] M. Andrychowicz, B. Baker, M. Chociej, R. Jozefowicz, B. McGrew, J. Pachocki, A. Petron, M. Plappert, G. Powell, A. Ray, *et al.*, "Learning dexterous in-hand manipulation," *arXiv preprint arXiv:1808.00177*, 2018.

[36] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International Conference on Machine Learning*, pp. 1928–1937, 2016.

[37] L. Espeholt, H. Soyer, R. Munos, K. Simonyan, V. Mnih, T. Ward, Y. Doron, V. Firoiu, T. Harley, I. Dunning, *et al.*, "Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures," *arXiv preprint arXiv:1802.01561*, 2018.

[38] D. Pathak, P. Agrawal, A. A. Efros, and T. Darrell, "Curiosity-driven exploration by self-supervised prediction," in *International Conference on Machine Learning (ICML)*, vol. 2017, 2017.