University of Duisburg-Essen
Faculty of Engineering
Department of Computer Science and Applied Cognitive Science
Institute of Intelligent Systems

Master Thesis
in Degree Programm
Computer Engineering - Master

Simulated visual-based reinforcement learning for navigation with Hindsight Experience Replay

Mohamed Nagi

First Reviewer: Prof. Dr. J. Pauli
Second Reviewer: Prof. Dr. J. Krüger
Time Period: 16. October 2019 to 16. April 2020

Abstract

Recently a lot of achievements have fulfilled thanks to the development of neural networks which helps to achieve state of the art learning policies in Reinforcement Learning (RL) for sequential decision making problems. The problem with theses learning policies (Agents),is that they need some kind of reward so they can differentiate between good and bad action.

However this is not always the case specially in robotics and navigation problems where the learning policy rarely get a positive reward, leading to incapability of the agent to learn anything. The problem of sparse reward is addressed either by engineering a shaped reward suited for the given task, or by introduction supervised auxiliary task that provide more feed back to the agent as a reward signal, that aid the agent learning the original task.

Lately a technique namely Hindsight Experience Replay (HER)[2] which allows sample-efficient learning from sparse and binary rewards. In this thesis an adapted version of HER is proposed so that we can train an agent to learn an off-policy using Deep Recurrent Q-network (DRQN)[12] which is trained by feeding a sequence of transitions to address the partial observation problem by introducing a recurrency layer to the vanilla Double Deep Q-network [32] where the agent is trained by feeding a stack of four consecutive frames to address partial observation problem. DRQN enables the agent to remember what it has seen before such that the learned policy is capable of inferring relation between agent actions and the environment states through time. Our approached is combined with an extrinsic shaped reward function in order to evaluate the effect of our proposed HER approach. In this thesis we show that the trained agent using the HER with the shaped reward was able to navigate to multiple goal position in a given environment successfully. Evaluation results shows that training using the shaped reward with HER yields higher success rates than training solely using our modified HER. Surprisingly the shaped reward trained policy shows comparable success rate to trained one with HER and shaped reward. Moreover we found the agent is able to navigate to the goal position by feeding only one frame at time through the agent monocular camera. In other words the agent learnt policy is capable of finding the relation or mapping between the individual frames and their corresponding positions in the environment coordinate system.

AI2-THOR framework [18], which provides an environment with high quality 3D scenes and physics engine is used to train the agent and evaluate its navigation abilities after training. The training of the agent is done and qualitatively evaluated using our modified HER with shaped external reward, with sparse reward, and with shaped and sparse reward only without the utilization of HER. The agent is also trained and its navigation ability is investigated using two different training approaches first by feeding dense latent vector of the agent image and the corresponding agent position

in the environment coordinate system, second using only the dense latent vector of the agent RGB camera frames.

Contents

1. Introduction

1.1. Motivation

In the last decades the number of application, which become dependent on usage of mobile robots has increased drastically. The increasing usage of mobile robot arises from their increased efficiency and accuracy carrying on predefined task which they are programmed to do. Mobile robots normally work in controlled environment where the goals are fixed, as the robot is programmed to navigate following a fixed path to the reach its goal avoiding any obstacles if there are any.

The higher demand for mobile robots to work in more complex environments e.g. hospitals, airport and homes etc is increasing drastically. Because of complexity of those environments, where the environment is not controlled anymore and contain moving objects and obstacle. The approach of programming mobile robot to navigate in such environments can be quite challenging and infeasible task due to unlimited number of states where the robot can be, furthermore the robot should have many sensors and actuators have to used in order for the robot to deal with the environment complexity and navigate through it successfully avoiding collision with obstacles. More sophisticated approaches are required, so that the robot is not only able to navigate in those complex environment but also to adapt to the dynamic and states of those environments.

Recently Reinforcement Learning (RL) showing great success when its applicability is tested and evaluated in different tasks e.g. playing video games where RL agent also called policies are able to not only beat human in those video games but also surpasses a human expert in those video games.

RL agents are are trained in simulated environments and learn complex behaviour by interacting with the environment, for interaction of the robot with the environment, the agent receives a reward from the environment, based on the reward received the agent is able to self assess its own actions. Main goal of RL agent is to maximize the cumulative reward received from the environment.

One of the problems when training RL agents to navigate in complex and dynamic environment, that the agent no matter how much it tries different action combination to achieve its goal, the agent does not perceive any positive reward to assess its own action. This problem is known are sparse reward problem where the reward perceived by the agent is very sparse. Traditional approaches tackling this problem by engineering a reward function that suits the RL agent task.

Lately a technique called Hindsight experience replay (HER) [2] address the sparse reward problem in a different approach.

The objective of this thesis is to investigate training a agent combined with HER to navigate in simulated environment, where the agent perceive a positive reward

only if it achieve the goal successfully avoiding the obstacles on the path to the goal position, otherwise the agent gets a negative reward.

1.2. Problem statement

Applying RL techniques in navigation problems have been successful given the traditional RL framework where the agent is trained by interacting with the environment receive a reward based on the action take by the agent and train [35]. Indeed for any RL policy there is should be a reward function, so the agent could assess its own action. Although engineering a reward function is not trivial for the given task of the agent but still some times it is not enough. Complex environment introduce the problem of sparse reward because of the environment complex dynamics no matter how much the agent tried to reach its goal, the reward it receives is always non-positive. However some methods try overcoming the sparse reward problem by introducing auxiliary tasks which is an additional cost function that the RL agent can predict and observe from the environment in a self-supervised fashion. [22]

HER has been introduced to solve the sparse reward problem which allow sample but efficient learning from rewards which are sparse and therefore avoid the need for complicated reward engineering. [2]

HER approach is to re-examine the failed trajectory of the agent $g \neq s_1, ..., s_T$ where the goal g is not achieved and the agent at each state s_t receives a non-positive reward with a different goal g' achieved in arbitrary state s_t such that $g' = s_t$. HER provides in other word HER applies a clever trick so the agent able to learn from unsuccessful trajectories. Thesis main objective is to apply a modified the HER approach to the navigation problem, where the agent still able to learn from failed trajectories during training feed only raw RGB images received by the agent monocular camera without the need for any auxiliary task.

1.3. Overview

The thesis is structured as follows, in chapter 2.1 the fundamentals of principles and concept of machine learning are presented including multilayer perceptrons 2.1.1 and general neural network training procedure. In subsection 2.1.3 the basics of the convolutional neural networks is presented, it is necessary to understand the developed convolutional auto-encoder that is used to compress the raw pixel RGB images into a dense vector the is fed to our agent to be trained on. In section 2.1.4 the fundamentals of Recurrent neural network is presented as our developed pipeline is based on a modified off-policy namely Deep Recurrent Double Q-network. [12] In Recurrent Neural Network (RNN) is introduced to substitute stacking of raw pixel images in the original double Deep Q-network.[32] The basics of Auto-encoders are introduced in subsection 2.2 as they are used later in the developed architecture to compress the RGB images into a dense latent vector.

The main implemented pipeline architecture is presented in chapter 3 where the different architectural part is presented and showing how they work combined. In section 3.2.3 the modified HER algorithm is illustrated.

Training and evaluation of the our developed agent is presented in chapter 4, where the learned agent policy is assessed qualitatively, quantitatively.

Lastly, in chapter 5 a brief conclusion and ideas for the improvement and enhancement of the developed architecture and modified HER algorithm are suggested for future work.

2. Neural Networks and Reinforcement Learning Fundamentals

This chapter summarizes the relevant fundamentals of the thesis topic for further understanding. In section 2.1 the core concepts of Artificial Neural Networks (ANN) are described, focusing on Convolutional and Recurrent Neural Networks (CNN), (RNN). Section 2.3 covers the fundamentals of Reinforcement learning. Core algorithms e.g Monte Carlo and Temporal Difference Methods are presented. The concepts presented in section 2.1 and section 2.3 are joined in 2.4 for understanding the state of the art Deep Reinforcement Learning algorithms namely, Deep Q-Network (DQN), Deep Recurrent Q-Network (DRQN) and Hindsight Experience Replay (HER) which are the core algorithms of the thesis.

2.1. Artificial Neural Networks

Artificial Neural Networks core concept is an analogy to the biological nervous system. The neurons which are the building blocks of the biological nervous system consists of dendrites, cell body and axon. The dendrites which is the inputs of single neuron receive signals from other neurons and pass them over to the cell body, in turn the cell body processes the inputs and if a certain potential threshold is reached, the cell body fires an electric impulse proportional to the input signal strength through the axon which is the neuron single output. The analogy between the biological and artificial neurons that the axons, dendrites and cell body are equals to connections between nodes and cell body equals to nodes. The connection weight represents input electric impulse and the ANN neuron fires if a threshold is reached to the next neurons as shown in figure 2.1. [4].

The output of a ANN neurons, is calculated according to propagation function 2.1, where x_i components of the input vector $X = (x_1, ..., x_I)$ is multiplied by its respective weight components w_ij of the weight vector $W = (w_{1j}, ..., w_{ij})^\top$ and adding a bias term b. The weight vector components w_{ij} represents weight of edge connecting the ith neuron of the previous layer and jth neuron of the next layer [10].

$$\xi_p = \sum_i W_{ij}^\top X_i + b \tag{2.1}$$

A differentiable activation function f_a is then applied to the weighted sum of ξ_p to calculate neuron final output y_{out} or its activation. [10]

$$y_{out} = f_a(\xi_p) \tag{2.2}$$

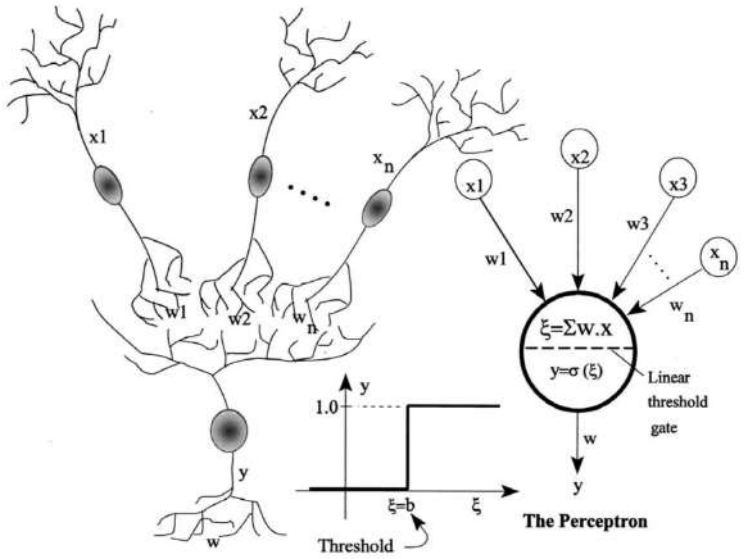

Figure 2.1.: Biological neuron (left) vs. artificial neuron (right). The artificial neuron models the dendrites as weighted inputs and processes the output by summing the dot product of X_i and W_i. [4]

2.1.1. Multilayer Perceptron

The goal of ANN which composed of Multiple Layer of Perceptrons - known as Multilayer Perceptron (MLP) - is to approximate a non-linear function $f^*(x)$. [10]

Artificial neurons of MLP are organized into layers where neurons inputs of each layers are the same. MLP from its name, obviously is composed of multilayer, where the neurons are distributed in consecutive layers namely input, hidden and output layers as dipected in figure 2.2.

2.1.2. Training Procedure

In order for MLP to approximate a function $f^*(X)$, a training procedure where the weight vector $Wij^\top$ need to be updated to find the best approximation [10]. The Training procedure is to find the best parameters $\theta = (W_i, b_i)$, and consists of the following three step:

- **Forward-propagation**: Each hidden layer computes its weighted sum of inputs ξ_i and activation y_i based on the activation of the previous layer y_{i-1} according to equation 2.3, where y_i is the activations of the ith hidden layer neurons, W_i is the matrix of connection weights and b_i donates the bias vector. The input is propagated through the consecutive hidden layer following equations 2.3 and 2.4 to the final layer giving the predict output y_{pred}

$$\xi_i = y_{i-1} W_i + b_i \tag{2.3}$$

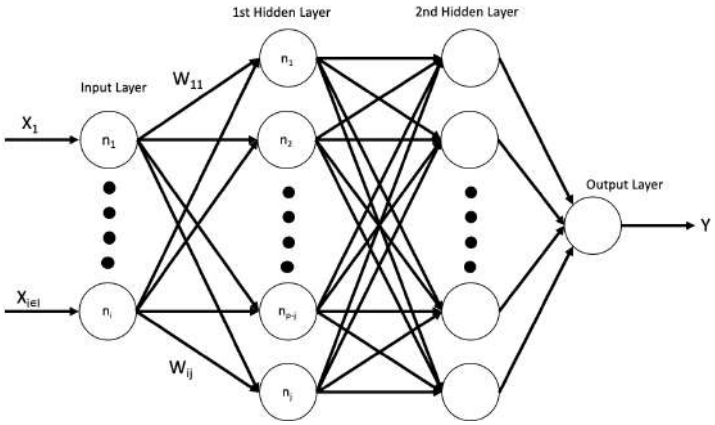

Figure 2.2.: Example of feed forward neural network showing the distribution of the neurons in input layer, two hidden layer and single output layer with the respective connection weight between each neuron in the consecutive layers.

$$y_i = f_a(\xi_i) \tag{2.4}$$

- **Back-propagation**: Based on the predict output y_{pred} of the previous step, ANNs are trained to minimize loss function L which is a function of network parameters $L(\theta)$ between true output y_{true} -know as Labels- and the predicted output y_{pred} using gradient decent methods [10]. The back propagation begins with the gradient loss function with respect to the network's output. Mean squared error, is an example of loss function (See equation 2.6). The global gradient of loss $\nabla L(\theta)$ is propagated back through the network layers in order to adjust the network's parameters. The core concept of the back propagation algorithm introduced by [Rumelhart et al.] is the chain rule. The chain rule is used to propagate the global gradient loss $\nabla L(\theta)$ back through the network. The back propagation step includes the calculation of each neuron's local gradient loss by multiplying the neuron's local derivative with the gradient of loss of the connected neuron in the next layer as shown in figure 2.3 , which in turn back propagate the error to the other preceding neurons.

$$L(\theta) = \frac{1}{n}\sum_{i=1}^{n}(y^i_{true}(\theta) - y^i_{pred}(\theta))^2 \tag{2.5}$$

- **Parameters update**: The network parameters θ are updated in the negative direction of the gradient loss based on chosen of gradient descent algorithm. Given the standard stochastic gradient descent algorithm (SGD), the function approximation approaches closer to the minimum by updating the networks parameters θ with learning rate ω according to the equation 2.6

$$\theta_i \leftarrow \theta_i + \omega\nabla_\theta L(\theta) \tag{2.6}$$

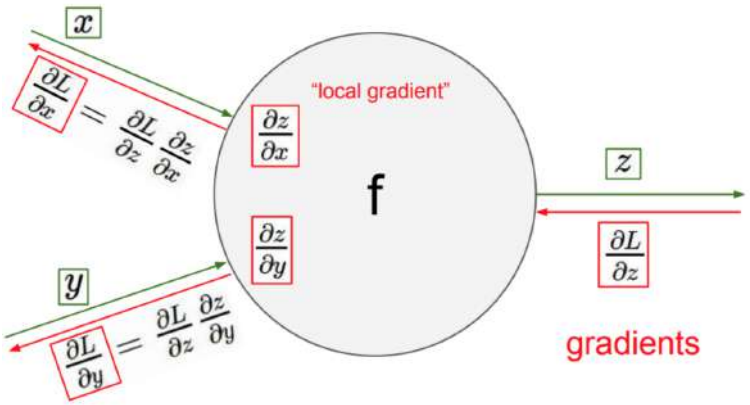

Figure 2.3.: Obtaining local gradient loss according to chain rule. where the gradient loss which is back propagated from the next neuron $\frac{\partial L}{\partial z}$ is multiplied by the local derivative $\frac{\partial z}{\partial x}$ and $\frac{\partial z}{\partial y}$ and back propagated to the preceding neurons. [Agarwal]

2.1.3. Convolutional Neural Networks

The fundamental concept of CNNs is analogous to traditional ANNs. The difference between CNNs and ANNs that the CNNs are mainly used for pattern recognition, feature extraction, compression and segmentation of images or in other words images focused task. As CNNs allow us to encode specific image features into the network architecture overcoming the traditional ANNs problem regarding dealing with computation complexity of image input data size.[24] Next is a brief description of the different layers of CNNs.

Convolutional Layers

CNNs Convolutional layer is composed of two arguments, first is the input which is a multidimensional array representing input image, second argument is also a multi-dimensional array representing the kernel and the output C of convolving the input image with the kernel is also a multi-dimensional array know as feature map, shown in figure 2.4.

The discrete convolution operation is calculated using the formula 2.7, for two-dimensional image I as an input convolved with kernel K. [10] The Kernel K of Convolution layer is consisting of neurons arranged in three dimensions width, height and depth. Kernel neurons will connect only to a local region of the input image. The connection between the kernel K and input image I is local in space along width and height but always full along the entire depth of the input image channels as shown in image 2.4. [10]

$$C(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n) \tag{2.7}$$

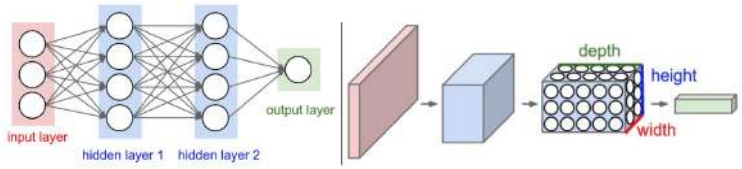

Figure 2.4.: Left: A regular 3-layer Neural Network. Right: A CNN arranges its neurons in three dimensions (width, height, depth), as visualized in one of the layers. Every layer of a CNN transforms the 3D input volume to a 3D output volume of neuron activations. In this example, the red input layer holds the image, so its width and height would be the dimensions of the image, and the depth would be 3 (Red, Green, Blue channels). [10]

.

The volume of feature map O resulted from the convolution operation can vary based on other parameters, namely depth, stride and padding is calculated according to 2.8. In 2.8 F refers to depth dimension of the output feature map volume and is corresponding to the number of filter where each filter, Str refers to stride which specifies how the filter is sliding over the input image I, P refers to the padding size which is used to control the spatial size of the feature map output volume O_{conv}. Filters is a concatenation of multiple kernels so for a convolutional layer with kernel K of dimension width w, hight h and depth or channels k the filter dimensions are w*h*k. All filters have the same size but different filter values and applied to the same input image I producing feature map, the final output is a stack of all feature maps. [10].

$$O_{conv} = \frac{I - F + 2P}{Str} + 1 \tag{2.8}$$

Pooling Layers

Pooling layer function is to progressively reduce the input spatial size by applying a downsampling filter to the input. Common pooling filters are max- and average pooling. Max-pooling operation is a filter slides over the input and only the maximum values remains in the output as shown in figure 2.5, meanwhile in average pooling the average of each position in the input and its neighbours is calculated by the filter. [10]

2.1.4. Recurrent Neural Networks

In Section 2.1 the traditional ANNs have been introduced, in section 2.1.2 we showed that training procedure of MLPs consisting of forward propagation, loss or error calculation and back propagation where the weights are updated, the training procedure of MLPs shows that there are not any cyclic connections exist. On the other hand Recurrent Neural Networks (RNNs) are special type of ANNs that allow a cyclic connections or loop where the information passes from one step of the network to the next as depicted in figure 2.6. Having a cyclic connection allow RNNs to map from

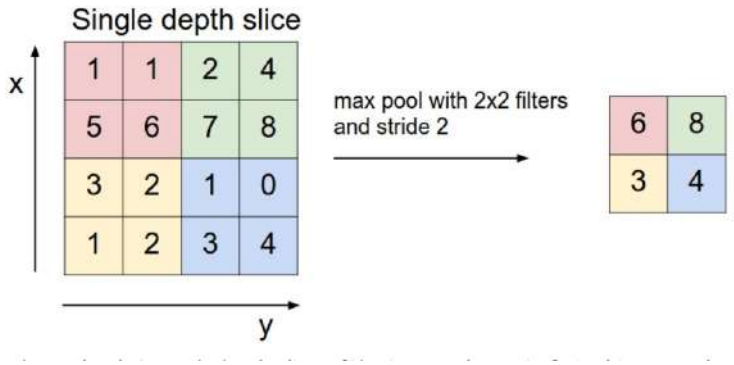

Figure 2.5.: Max pooling filter of size 2x2 is sliding over input with size 4x4 with stride of 2 where only the max values are chosen. [10]

.

the history of previous inputs to each output, where MLPs in section 2.1.1 can only map from the input to output vectors only. Another advantage of RNNs over MLPs that the recurrent connection enables the RNNs to memorize the previous input in the network hidden state which in turn influence the network output.[11] In this section we focus on simple RNN containing self connected hidden layer and special also focusing on Long Short-Term Memory (LSTM) which one of the best practices used RNN architecture because of its robustness and reliability address certain issue that appears when working with traditional RNNs.

RNNs Structure and Training Procedure

RNNs are composed of three layers, the input layer x, the hidden layer h, and the output layer o. Unfolding the RNN loop show that RNNs are multiple copies of the same structure and the hidden state h of each copy is taken as an input to its successor.[9]

Since RNNs are almost similar to the traditional ANNs, where the only difference lies in the additional loop passing the hidden state h to its successor, similarly RNNs training procedure is almost the same as the ANNs, only RNNs final output consider not only input x but also hidden state h. Training procedure of the RNNs consist of the following steps as normal ANNs as follows:

- **Forward-propagation**: In forward pass of RNNs the activations at the hidden layers arrives from both current external input at time t and the previous hidden layer activation at time $t-1$, the hidden layer calculate its activation according to 2.9, where X_i^t is the value of the input i at time t, and a_j^t and b_j^t are respectively the network input to unit j at time t and the activation of

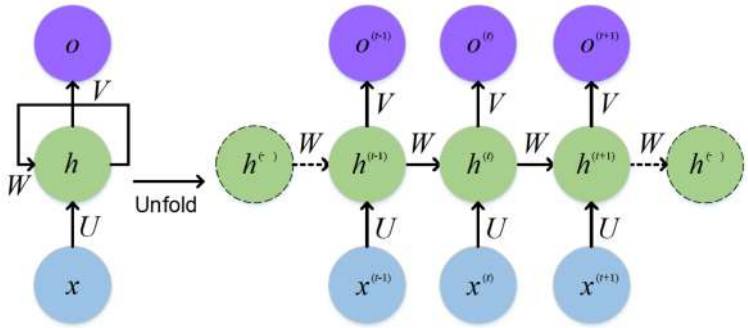

Figure 2.6.: Recurrent Neural Network Structure. The left is the typical RNN structure. The right part is the unfolding version where the previous information is transformed to the later time step.[9]

.

unit j at time t [11]. Weights W_{ih} and $W_{h'h}$ are the weights of input to hidden connection and hidden to hidden connection respectively.[9]

$$a_h^t = \sum_{i=1}^{I} W_{ih} X_i^t + \sum_{h'=1}^{H} W_{h'h} b_{h'}^{t-1} \tag{2.9}$$

Nonlinear, differentiable activation functions are then applied as for an MLP according to 2.10. [11]

$$b_h^t = f_{ah}(a_h^t) \tag{2.10}$$

The complete hidden activation sequence is calculated starting at time $t = 1$ and recursively applying equations 2.9 and 2.10, where b_i^0 should equal zero corresponding to the network states before it receives any information from the data sequence.[11] The loss is then calculated similar to MLPs loss in section 2.1.1.

- **Back-propagation**: Given the partial derivatives of the loss function with respect to the network outputs from the forward propagation step, using Back-propagation through time algorithm (BPTT) [33] is used to update the network weights in the back-propagation step according to 2.11. BPTT uses repeated application of the chain rule similar to MLPs back-propagation step in section 2.1.2, difference is that the loss function depends on the activation of the hidden layer and also its influence on both the hidden layer at the next time step and the output layer. where $\delta_j^t = \frac{\partial L}{\partial a_j^t}$. The complete sequence of δ is calculated

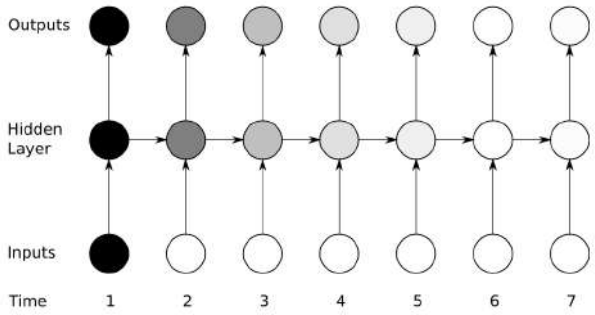

Figure 2.7.: The shading in the network indicates the decaying effect of the inputs at time $t = 1$ on the hidden layer and network output as the new inputs overwrite the activations of the hidden layer, and the network forgets the earlier inputs.[11]

.

starting from $t = T$ and recursively applying equation 2.11 decrementing t at each time step.[11]

$$\delta_h^t = f'_{ah}(a_h^t)\Big(\sum_{k=1}^{K} \delta_k^t w_{hk} + \sum_{h'=1}^{H} \delta_{h'}^{t+1} w_{hh'}\Big) \tag{2.11}$$

Lastly we sum over the whole sequence to get the derivatives with respect to the network weights where the same weights are reused at every time step as shown in equation 2.12.[11]

$$\frac{\partial L}{\partial W_{ij}} = \sum \frac{\partial L}{\partial a_j^t} \frac{\partial a_j^t}{\partial W_{ij}} = \sum \delta_j^t b_i^t \tag{2.12}$$

Long Short-Term Memory (LSTM)

Vanishing gradient problem [15] exist because of the influence of a given input on both the hidden layer and the network output as shown in figure 2.7, either decays or blows up exponentially as it cycles around the recurrent connections. Many approaches had been introduced to address the vanishing gradient problem For RNNs including non-gradient based training algorithms such as simulated annealing and discrete error propagation. [5]

Long Short-Term Memory (LSTM) architecture is one of the best practices used and is also used in the implemented pipeline of this thesis as it will be shown in section 3.2.2 because of its ability dealing with the vanishing gradient problem. In this section a brief illustration of the LSTM architecture would be given.

The architectural difference between the normal RNN and LSTM that the summation units in the hidden layer are replaced by memory blocks, these memory blocks contains one or more self-connected memory cells and three multiplicative units namely the input, output and forget gates as shown in figure 2.8.[11]

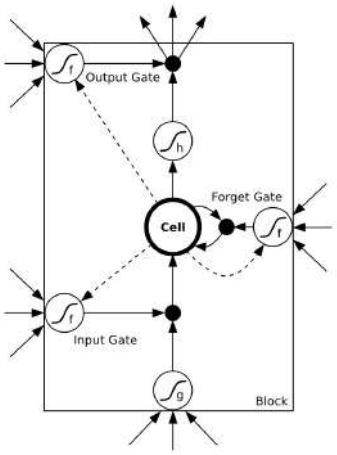

Figure 2.8.: One cell LSTM memory block. The three gates are nonlinear summation units that collect activations from inside and outside the block, and control the activation of the cell via multiplications (small black circles). The input and output gates multiply the input and output of the cell while the forget gate multiplies the cell's previous state. The gate activation function is usually the logistic sigmoid, so that the gate activations are between 0 (closed gate) and 1 (opened gate). The cell input and output activation functions g and h are usually tanh or logistic sigmoid. The dashed line shows the weighted peehole connection from the cell to the gates.[11]

.

The storing and accessing information over long periods of time of LSTM memory cell is controlled by the multiplicative gates. As long as the input gate remains closed, i.e has an activation near 0, the activation of the cell will not be overwritten by the new inputs to the network, and therefore can be made available to the network much later in the sequence, by opening the output gate as shown in figure 2.9. [11]

2.2. Auto-encoders

Auto-encoders are special type of MLP to transform input into output with the least possible distortion. Auto-encoders generally consists of two parts: a feedforward encoder module that maps the input into a dense hidden $\mathbf{z}$ representation in the bottleneck layer and a decoder module that reconstruct the input sample from $\mathbf{z}$ as shown in figure 2.10 . Auto-encoders inputs are batch of samples in matrix χ where each row is an input vector. The auto-encoder output χ' is enforced to equal to χ by minimizing the squared error $||\chi - \chi'||^2$.[34]

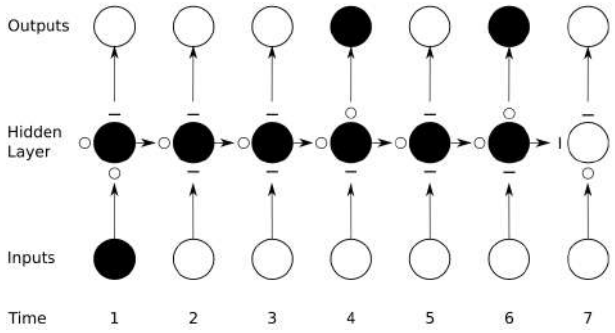

Figure 2.9.: The shading of nodes indicates their sensitivity to the inputs at time $t = 1$, in this case the black nodes are maximally sensitive and the white nodes are entirely insensitive. 'O' and '-' indicates the state of the input, output and forget gates respectively where the gate is either fully open 'O' or fully closed '-' respectively. The output layer sensitivity is controlled by switching on or off the output gate. [31]

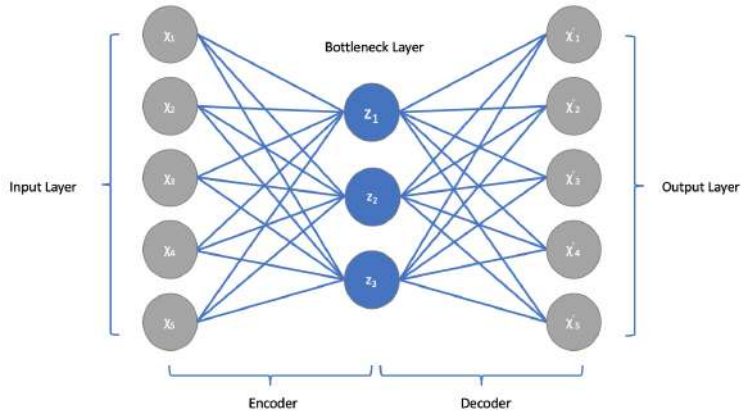

Figure 2.10.: Auto-encoders general architecture.

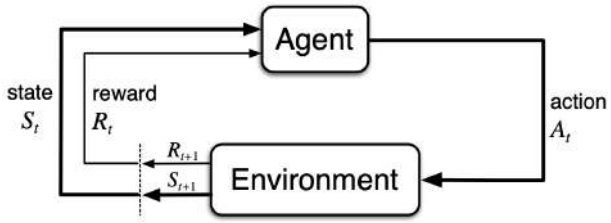

Figure 2.11.: The agent interacts with the environment to learn from experiences. At each time step t the agent is in a certain state $s_t \in S$ and takes action $a_t \in A(s)$. As result, it switches to a new state s_{t+1} and receives a reward R_{t+1}.[31]

.

2.3. Reinforcement Learning

The main concept of RL is learning from experience. The agent collect these experiences by interacting with the environment in order to learn a specific behaviour by trial-and-error. The agent experiences consisted of environment state $s_t \in S$ that the agent receives at time t, then the agent takes action $a_t \in A(S)$, based on the taken action, the agent receives a new environment state $s_{t+1} \in S$ and a reward $r_{t+1} \in R \subset \mathbb{R}$ assess how good or bad the action taken. Figure 2.11 shows the core concept of RL. [31]

Markov Decision Processes

Markov Decision Processes (MDP) describes the robots interaction with environment as a trajectory that begins like this $s_0, a_0, r_1, s_1, a_1, r_1, ...$ where the discrete probability distributions of state s_{t+1} and reward r_{t+1} dependent only on the previous state s_t and action a_t and not at all on earlier states and action according to MDP. The state s_t must include information about all aspects of the past agent-environment interaction in order to have the *Markov property*. The discrete probability distribution is defined according to 2.13 for all $s', s \in S, r \in R$ and $a \in A(s)$. Given the previous definition of the discrete probability, the MPD is formulated as an ordinary deterministic function of four arguments $p : S$ x R x S x $A \rightarrow [0, 1]$ for finite MDP. From the dynamic equation p the state transition probabilities is computed according to equation 2.14. [31]

$$p(s', r|s, a) \doteq Pr\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\} \tag{2.13}$$

$$p(s'|s, a) \doteq Pr\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\} = \sum_{r \in R} p(s', r|s, a) \tag{2.14}$$

Goals and Rewards

An important concept of RL frameworks is the reward. Informally the agents goal is to maximize the cumulative reward it receives in the long run. The use of reward signal, is to formalize the idea of goal for a RL policy. In the simplest case, the return is the sum of the rewards and is formulated according to 2.15, where T is the final time step. While T is for the applications which there is a natural notion of final time step. An episodes represents those applications where each episode ends in a special state called the *terminal state*. On the other hand for *continuing task* where the agent-environment interaction does not break in identifiable episodes such that the final time step $T = \infty$. The cumulative reward for *continuing task* is not the sum of cumulative rewards, but the discounted cumulative rewards. In particular the agent chooses action a_t to maximize the expected discounted return according to 2.16 where γ is the reward discount factor also called *discount rate* between $0 \leq \gamma \leq 1$. If $\gamma = 1$ all rewards of the future are having the same weight, on the other hand if $\gamma = 0$ just the immediate reward is taken into account [31].

$$G_t \doteq R_{t+1} + R_{t+2} + ... + R_T \tag{2.15}$$

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+2} + ... + R_T = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.16}$$

Policy and Value Functions

As stated before in subsection 2.3 the goal of the agent is to maximize the cumulative rewards, such that the reward r_{t+1} received at time $t+1$ is used to tell how good or bad it is for the agent to be in a given state s_{t+1}. Indeed the received reward and the new state of agent are dependant on what actions it will take. Estimating how good or bad for the agent to be in a give state, or how good it is to perform an action in a given state is determined using *value functions* with respect policy. Value function is either a function of state or state-action pairs. The agent policy is a mapping from states to probabilities of selecting each possible action, formally if the agent follows policy π at time t then $\pi(a|s)$ is the probability that $A_t = a$ if $S_t = s$. RL methods specify how the agents policy is changed as a result of its experience. The agent *state-value function for policy* π is defined formally for MPDs according to equation 2.17, where $\mathbb{E}_\pi[.]$ is the expected cumulative return, when starting in state s following a policy π for all $s \in S$ [31].

$$v_\pi(s) = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s\right] \tag{2.17}$$

If the value function for policy π estimates not only the state values, but also the value of taking action a in a state s it is called *action-value function* and donated according to 2.18

$$q_\pi(s,a) = \mathbb{E}_\pi\Big[\sum_{k=0}^{\infty}\gamma^k R_{t+k+1}\Big|S_t = s, A_t = a\Big] \tag{2.18}$$

The relationship between the value of a state s and the values of its possible successor states s' is donated by the *Bellman equation* 2.19, satisfying a fundamental property of value functions namely recursive relationship and it is used throughout reinforcement learning. The Bellman equation 2.19 firstly averages over all possibilities, weighting each by its probability of occurring, second it states that the value of state s must equal the discounted value of the expected next state s' plus the reward r expected along the way.

$$v_\pi(s) = \sum_a \pi(a|s)\sum_{s',r} p(s',r|s,a)[r + \gamma v_\pi(s')] \tag{2.19}$$

RL aims to find and *optimal policy* π_*. A policy π is better or equal to a policy π' if its expected return is better than or equal to policy π', in other words $\pi \geq \pi^{'}$ if and only if $v_\pi(s) \geq v_{\pi'}(s)$ for all $s \in S$. There is at least one policy that is better than or equal to all other policies. This is an *optimal policy*, and there may be more than one. They share the same state-value function, called the *optimal state-value function* according to 2.20 for $s \in S$ and also share the same *optimal action-value function* defined as in 2.21 for all $s \in S$ and $a \in A(s)$. [31]

$$v_*(s) = \max_\pi v_\pi(s) \tag{2.20}$$

$$q_*(s,a) = \max_\pi q_\pi(s,a) \tag{2.21}$$

Temporal Difference Methods

Temporal Difference TD is favoured over other RL methods, because they are applicable to online learning and continuous RL-Tasks. TD based policies does not need for episode to complete and the policy is updated at each time step in contrast to other RL methods e.g Monte Carlo that need a completed episode to update its target $V(S_t)$. Moreover TD method does not need a model of the environment dynamics compared to other methods e.g. Dynamic Programming (DP). The true expected return G_t for TD is approximated with the TD-target. TD-target is determined by taking the sum of the immediate return R_{t+1} and the expected value of the next state $V(S_{t+1})$ discounted by $\gamma \in [0,1]$ and is donated by 2.22, where α is a constant representing the step size and changes from time step to time step. The algorithm 1 shows the concept of TD methods. An agent following a policy π, takes action A_t given initial state S_t, the agent execute action A_t, then the agent observe the reward R_{t+1} and transit to next state S_{t+1}. Lastly the value-function $V(S_t)$ according to 2.22. When talking about the TD learning method the differentiation between on

and off-policy should be made clear. An off-policy method learns the value of the optimal policy independently of the agents actions e.g. Q-learning as shown in section 2.3. On the other hand an on-policy method learns the value of the policy being carried out by agent e.g. SARSA. [26]

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)] \tag{2.22}$$

Data: The policy π, $\alpha \in (0, 1]$
initialize $V(s)$, for all $s \in S$, arbitrarily except that $V(terminal) = 0$;
for *each episode* **do**
  Initialize S_t;
  while S *is not terminal* **do**
    $A \leftarrow$ action given by π for S_t;
    Take action A, observe R_{t+1}, S_{t+1};
    $V(S_t) \leftarrow V(S_t) + \alpha[\underbrace{\underbrace{R_{t+1} + \gamma V(S_{t+1})}_{\text{TD-target}} - V(S_t)}_{\text{TD-error}}]$;
    $S_t \leftarrow S_{t+1}$;
  end
end

Algorithm 1: General algorithm of TD methods: The TD-Target $R_{t+1} + \gamma V(S_{t+1})$ is the approximation of the expected return G_t. [31]

Q-Learning

Q-learning is an off-policy TD control method. Q-learning does not update the value function of the state $V(S_t)$ rather the target of the Q-learning update rule is based on the action-value function $Q(S_t, A_t)$. The learned action-value function Q directly approximates the optimal action-values function in 2.21. The Q-learning update rule is defined by 2.23. The defined Q-learning update rule state that the Q-learning target is determined based on sum of the immediate return R_{t+1} and the expected action-value function of the next state and action pair $Q(S_{t+1}, A_t)$ discounted by $\gamma \in [0, 1]$ rather than state-value function $V(S_t)$. In algorithm 2 the agent takes action $a \in A$ following a policy π such a policy would always choose the actions that have maximum value given by action-value function, these policies called *greedy policy*. Greedy policies exploit the current knowledge they learned; the agent can get stuck as only actions with the highest action-value is taken and end up in a non-optimal policy. To avoid such a situation, the agent take action according to *ε-greedy* policy where the agent take explorative actions with probability ε, while the greedy actions are taken with probability $(1 - \varepsilon)$. [31]

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \tag{2.23}$$

Data: Step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
initialize $Q(s, a)$, for all $s \in S$, and $a \in A(s)$ arbitrarily except that $Q(terminal, .) = 0$;
for *each episode* **do**
  Initialize S_t;
  while S *is not terminal* **do**
    choose action $a \in A$ from S_t using policy derived from Q (e.g., ε-greedy);
    Take action a, observe R_{t+1}, S_{t+1};
    $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$;
    $S_t \leftarrow S_{t+1}$;
  end
end

Algorithm 2: General algorithm of Q-learning method. The agent choose the action $a \in A$ following a policy π e.g.ε-greedy policy given a state S_t.[31]

2.4. Deep Reinforcement Learning

The basic idea behind many reinforcement learning algorithms is to estimate the action-value function by using the Bellman equation 2.23 as an iterative update. Such value iteration algorithms converge to the optimal action-value function $Q_i \rightarrow Q^*$ as $i \rightarrow \infty$. In practice, this basic approach is impractical, because the action-value function is estimated separately for each sequence, without any generalization. Instead, it is common to use a function approximator to estimate the action-value function $Q(s, a; \theta) \approx Q^*(s, a)$. [23]

ANNs are used because of their ability to approximate non-linear function. ANNs are used as function approximator to extract the relevant features from raw inputs make it possible to generalize over unseen states. We refer to ANN approximator with weights θ as a Q-network. In this section core TD based RL algorithms are presented namely Deep Q-network (DQN), [23] Deep Recurrent Q-Network (DRQN) [12] and Hindsight experience Replay (HER). [2]

The implemented agent is developed based on DRQN which is a improved version of DQN to address the problem of Partial Observable Markov Decision Process (POMP), the later HER algorithm is main objective of this thesis and it is used to address a core problem of RL which is sparse reward problem.

Deep Q-network (DQN)

DQN has been introduced by DeepMind group where they have combined a the model-free, off-policy Q-learning method with Deep ANN. The developed end-end architecture is able to extract relevant feature by itself from raw RGB frames. Based on the extracted relevant features and the received reward from the environment which in this case was an 2600 Atari games [6], the outputs correspond to the predicted Q-values $q(s_t, a)$ of the individual actions $a \in A(s)$ for the input state $s_t \in S$. The

main advantage of DQN architecture is the ability to compute Q-values for all possible actions in a given state with only a single forward pass. through the network.

initialize replay memory D to capacity N;
initialize action-value function parameters Q with random weights θ;
initialize target action-value function $\hat{Q}$ with random weights $\hat{\theta} = \theta$;
for *episode* $= 1, M$ **do**
  Initialize sequence $s = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$;
  for *t=1,T* **do**
    With probability ε select a random action a_t;
    othewise select $a_t = \arg\max_a Q(\phi(s_t), a; \theta)$;
    Execute action a_t and observe reward r_t and image x_{t+1};
    Set $s_{t+1} = s_t$and preprocess $\phi_{t+1} = \phi(st_{t+1})$;
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D;
    Sample random mini-batch of transitions $(\phi_t, a_t, r_t, \phi_{t+1})$ from D;
    Set $y_j =$
    $$\begin{cases} \mathrm{r}_j, & \text{if episode terminates at step j+1} \\ \mathrm{r}_j + \gamma \arg\max_{a'} \hat{Q}(\phi_{j+1}, a'; \hat{\theta}), & \text{otherwise} \end{cases}$$
    Perform a gradient step descent step on $(y_j - Q(\phi_j, a_j, \theta))^2$ with respect to network parameters θ;
    Every N steps reset $\hat{Q} = Q$
  end
end

Algorithm 3: Pseudo algorithm of DQN. [23]

In algorithm 3, first the agent initialize a replay memory D to store the consecutive transitions consisted of $\phi_t(s)$ representing current preprocessed environment state, current action a_t, current expected return r_t and the next preprocessed state of the environment. Replay memory allows for greater data efficiency as each step of experience is used in many weights update, also breaks the correlations between consecutive samples by updating the agent using random mini-batch sampled from the replay memory, lastly memory replay the behaviour distribution is averaged over many of its previous states which leads to smooth learning and parameter oscillations or divergence.[23]

The agent inputs are preprocessed states. At first the preprocessing of each state, resize and normalize the value of each pixel between zero and one of the input raw RGB frames to reduce the input dimensionality and then stack four consecutive frames to produce the input to the DQN[23]. MDPs assumes that the agent can observe environment states fully, this assumption does not reflect the nature of real or many simulated environments in which the agent only observes a partial information of the environment states. POMDP is used to represent the agent interaction with such observable environments. Maintaining a belief distribution over unobservable state is one technique used to address POMDP environments, such a belief distribution is

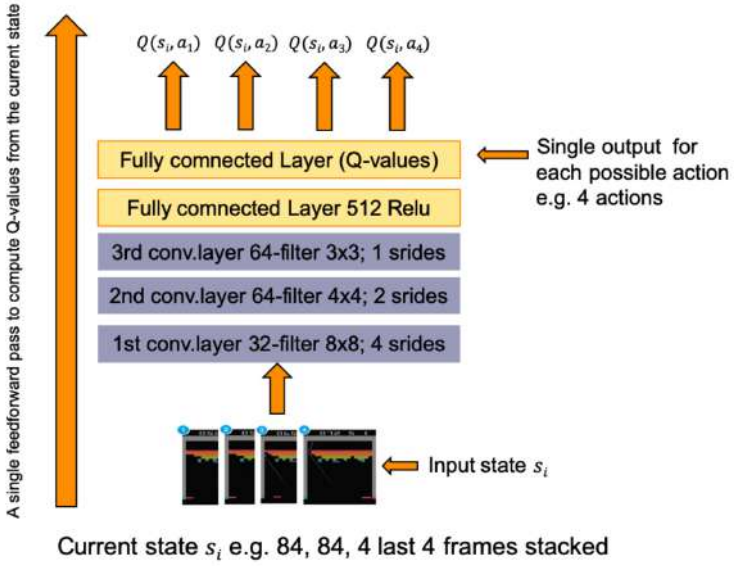

Figure 2.12.: Block diagram showing the architecture of the Deep Q-Network [23]

.

realized by stacking multiple frames as input to the DQN to address the POMDP problem.[20]

In equation 2.24, an additional Q-network is used known as *frozen target network* that is a clone of the original Q-network architecture but with different weights are used: θ for the Q-network and $\hat{\theta}$ for the target Q-network. Regularly updating the Q-network according to the loss function from equation 2.24, while the target network is updated by copying the parameters of the Q-network to the target Q-network $\hat{\theta} = \theta$ every C time steps. Thus, the weights of the target Q-network $\hat{\theta}$ are held frozen for C time steps. It smooths training oscillations and leads to more stable learning.

$$L_i(\theta_i) = \mathbb{E}_{s,a,r}[(r_{j+1} + \gamma \arg\max_{a'} Q(S_{j+1}, a'; \hat{\theta}_j) - Q(S_j, a_j, \theta_j))^2] (2.24)$$

Lastly the DQN network is updated according to the loss function from eq 2.24, by taking the squared TD-error. Figure 2.12 show the complete architecture of the DQN The first hidden layer convolves 32 filters of 8×8 with stride 4 with the input image and applies a rectifier nonlinearity. The second hidden layer convolves 64 filters of 4×4 with stride 2, again followed by a rectifier nonlinearity. This is followed by a third convolutional layer that convolves 64 filters of 3×3 with stride 1 followed by a rectifier. The final hidden layer is fully-connected and consists of 512 rectifier units. The output layer is a fully-connected linear layer with a single output Q-values for each possible action.

Deep Recurrent Q-network (DRQN)

In [12] presented a modified version of the original DQN presented in the previous section to deal with POMDP. DQN stacks a number of consecutive frames as input to the agent to provide a belief to the agent about the system state. The presented DRQN provide such a belief to address the POMDP problem by using a recurrent neural network, in particular LSTM as shown in figure 2.13. The input states to DRQN are a single preprocessed frame at each time step. The values of each pixel are normalized between 1 and 0 but as a preprocessing step but not stacked as the original DQN. The preprocessed frames are then processed by three convolutional layers extracting the relevant feature for the agent. Convolutional layers output are fed to the fully connected LSTM layer. Finally, a linear layer outputs a Q-value for each possible action exactly as the DQN. The author suggest two different approaches to ensure a stable recurrent updates in particular *bootstrapped sequential updates* and *bootstrapped random updates*. Both approaches select random episodes from the replay memory, in the first approach the updates begin at the beginning of the episode and proceed forward to the conclusion of the episode, the later approach the updates begins at random points in the episode and proceed for only *unrolled iteration* time steps. Both approaches generate the target Q-values for the target Q-network at each time step, additionally the LSTM initial state is zeroed at the start of the update. [12]

Sequential updates have the advantage of carrying the LSTM's hidden state forward from the beginning of the episode. Sequential updates violates the DQN's random sampling concept. Random updates supports the DQN random sampling concept but the LSTM hidden state must be zeroed at the start of each update which makes it harder for the lSTM to learn function that lasts longer time scales than the number of the time steps reached b the back propagation through time.[12]

Hindsight Experience Replay (HER)

Large and complex environment dynamics introduces the problem of sparse reward. Sparse reward problem is generally address in particular for robotics by engineering a reward function that not only reflects the task at hand but is also carefully shaped to guide agent policy optimization. Other techniques dealing with the sparse reward problem by augmenting the sparse extrinsic reward signal from the environment by additional reward signals to aid the agents learning[16]. Augments the whole training process by introducing additional supervised tasks that augment the original sparse reward signal with three additional reward signals. In [25] the sparse extrinsic reward signal is augmented by additional supervised task that encourages the agent explore unseen regions of the environment by using forward model that calculate the loss of between the current input frame and the next frame in the futures. In contrast to reward signal augmentation approachs using additional auxiliary task and reward function engineering, Hindsight experience replay (HER) introduces a very powerful technique yet very simple. HER objective's is to train agents to achieve multiple different goals. HER assumes that every goal $g \in \mathbb{G}$ corresponds to some predicate $f_g : S \rightarrow \{0, 1\}$ and that the agent's goal is to achieve any state s that satisfies

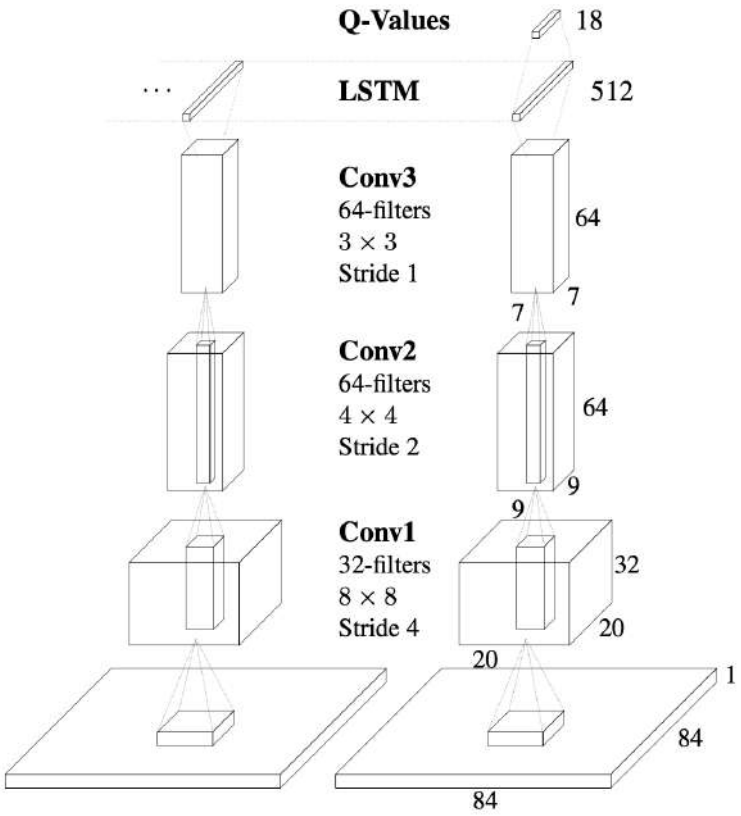

Figure 2.13.: Architecture of Deep Recurrent Q-Network shows the replacement of first dense layer in the original DQN architecture in figure 2.12, with LSTM layer of the same size as the dense layer in DQN architecture [12]

.

$f_g(s) = 1$. Based on the previous assumption the goals can also specify only parts of state properties. Moreover given a state s we can find a goal g that is satisfied in this state, in other words there is a given mapping $m : S \to \mathbb{G}$ such that. $\forall_{s \in S} f_{m(s)}(s) = 1$. [2] The idea behind HER is to store not only every transition with the original goal but also with additional subset of other goals in replay memory . The agent failed trajectory $s_0, s_1, ...s_T$ such that $r(s_T, a, g) < 0$ is then replayed with one of the sampled additional goals $g' \in \mathbb{G}$. The reward function of the replayed trajectory is recalculated for the additional sampled goal g' that substitute the original goal g such that $r' = r(s_t, a_t, g')$. HER also replays each trajectory also with the original goal of the episode. [2]

Given: an off policy RL algorithm π e.g. DQN ;
Given: a strategy Λ for sampling goals for replay e.g. $\Lambda(s_0, ..., s_T) = m(s_T)$;
Given: a reward function $r : S \times A \times \mathbb{G} \to \mathbb{R}$ e.g. $r(s, a, g) = -[f_g(s) = 0]$;
Initialize π;
Initialize replay memory D;
for *episode* $= 1, M$ **do**
  Sample a goal g and a initial state s_0;
  for *t=0,T-1* **do**
    Sample an action a_t using the policy from π $a_t \leftarrow \pi(s_t||g)$;
    Execute action a_t and observe a new state s_{t+1};
  end
  for *t=0,T-1* **do**
    $r_t = r(s_t, a_t, g)$;
    Store the transition $(s_t||g, a_t, r_t, s_{t+1}||g)$ in D;
    sample a set of additional goals for replay $G = \Lambda$(current episode);
    for $g' \in \mathbb{G}$ **do**
      $r' = r(s_t, a_t, g')$;
      Store the transition $(s_t||g', a_t, r', s_{t+1}||g')$ in D;
    end
  end
  for *t=1, N* **do**
    Sample a mini-batch B from replay memory D;
    perform one step of optimization using π and mini-batch B
  end
end

Algorithm 4: Pseudo algorithm of HER, as shown in the algorithm not only the current state s_t is feed as input to the agent to predict the next action a_t but additionally the original goal g concatenated with the state $(s_t||g)$.[23]

In algorithm 4 the agent predict action a_t given the concatenated current state s_t and the original goal g according to off-policy π. The original goals are replaced by the sampled additional goal g'. The reward function is recalculated for the sampled goals, the new modified transitions are stored in the memory buffer in addition to the original transitions. The additional modified transitions provides a reward signal

to the agent that is bigger than zero, in turn the agent does not fail as there is some augmented positive reward signal. Lastly the agent off-policy is update on randomly sampled mini-batches.[2]

In HER they have used different sampling strategy for the additional goals. Future strategy replays the failed trajectory with k random states which comes from the same episode as the transition being replayed and were observed *after* it. Episode strategy replays the failed trajectory with k random states coming from the same episode as the transition being replay. Random strategy replays with k random states encountered so far in the whole training. Lastly the final strategy which replays with the final state s_T which comes from the same episode. Hyper-parameter k is used to control the ratio of HER modified transitions to the transitions coming from normal experience replay in the replay buffer. [2]

3. Implementation

This chapter introduces at first the simulation environment that is used for agent training, the simulation environment action, and its state spaces. Second the agent task and it setup are presented in 3.1. Third the implemented pipeline is introduced 3.2 in two section. In the first section the implemented convolutional auto-encoder that is used provide our agent with dense feature vector of the consecutive RGB-frame of the agent's camera. In the second section we illustrate joining the convolutional auto-encoder with the memory module of our agent and its component, in particular RNN. Lastly the implemented modified version of HER algorithm is presented.

3.1. Simulation environment

AI2-THOR 3D simulation environment, has been used in many other RL researches for RL-agent navigation task. For example in [19] an RL agent based on neural network structure is proposed, which is capable of learning vision-based navigation tasks in continuous state spaces for multiple targets in multiple environments. Also in [36] they used AI2-THOR environment to address the issue of data inefficiency i.e. the model requires several episodes of trial and error to converge, which makes it impractical to be applied to real-world scenarios. The impracticality issue is also related to our RL-agent navigation task.

AI2-THOR provide photo-realistic interactive framework with high-quality indoor images. A scene within AI2-THOR represents a virtual room that an agent can navigate in and interact with different objects in the room. There are 4 scene categories, each with 30 unique scenes within them: *Kitchen, Living Room, Bedroom, Bathroom* shown in figure 3.1. Most of the environments are a single room. The agent moves on a grid, the size and numbers of grid blocks are determined using *grid size* parameter in meters. The agent execute action to move to the neighbouring grid cell. AI-2THOR provides a discrete action space consists of translation action set {*move forward, move backward, move left, move right*} that move the agent with a step equals to the defined *grid size*, rotation action set {*rotate left, rotate right*} and interaction action set to enable the agent interaction with different object in the environment. The discrete action command to the agent would fail if the action precondition is not satisfied. For example any translation action will fail if the agent collided with any of the object in the training scene. AI2-THOR provides a number visual sensors to capture the agent state at each time step. For example the main sensor is a monocular camera attached to the agent that provides 300×300 RGB images to the agent. Additional visual based sensor can also be used e.g. depth camera. [18]

Figure 3.1.: Examples of the AI2-THOR environment different scenes.[18]

Naviagtion task setup

The task of the agent is to navigate successfully from an initial random position and reach random goal position in the environment coordinate system. The goal position given and the agent random initial position are relative to the environment coordinate system. The environment coordinate system follows the right handed coordinate system. The x-axes, y-axes points right and up respectively while the z-axes points forward as shown in figure 3.2.

The agent should navigate to goal position given the available possible actions and the agent current state s_t at time step t. The state s_t of the agent is composed of a single RGB image frame with a resolution of 256×256 and the current agent position po_t at time t. The agent is trained using two different action sets to evaluate the influence of number of possible actions on the learned policy. The core action set consists of the following actions *move forward, rotate left* and *rotate right*, the extended action set has additional three actions namely *move backward, move right, move left*. The rotation actions *rotate left, rotate right* rotate the agent by 90° around the y-axes. while any of the translation actions translate the agent to the neighbouring grid block by the step size equals to 0.18 meter as shown in figure 3.3. Different grid size values are tested but we found the best value equals to 0.18 meter that gives an 781 reachable position that the agent can navigate to. The evaluation results of the learned navigation policy by the agent using the different action set is discussed in the next chapter.

An episode is successful if the agent is in a specific distance to the goal position controller by threshold $d_{threshold}$ equals to 0.36 meter from the goal position po_g that

Figure 3.2.: The coordinate system of the AI2-THOR environment according to the right hand coordinate system.

is 2 × step size or two grid block away from the actual goal position. If the agent collided with any of the object in environment, the episode is failed. The distance between the agent position at time t and episode goal position is calculated each step according to 3.1

Shaped and *spare* reward functions are engineered to evaluate the effect of the modified HER algorithm. Reward function is calculated based on L_2 norm of *Euclidean distance* d_t between agent position po_t at time t and goal position po_g of the current episode according to equation 3.1.

$$||d_t(po_t, po_g)|| = (\sum_{k=1}^{n} |po_t^k - po_g^k|^2)^{1/2} \tag{3.1}$$

The shaped and sparse reward function are conditioned on the distance d_t to the goal position and is computed according to equations 3.2 and 3.3 respectively.

$$r_{shaped,t}(s_t, a_t) = \begin{cases} 0, & \text{if } d_t \leq d_{threshold} \\ -0.1, & \text{if } d_{t+1} < d_t \\ -0.5, & \text{if } d_{t+1} = d_t \\ -1 + (d_{t+1} - d_t), & \text{if } d_{t+1} > d_t \\ -1, & \text{if collided} \end{cases} \tag{3.2}$$

$$r_{sparse,t}(s_t, a_t) = \begin{cases} 0, & \text{if } d_t \leq d_{threshold} \\ -1, & \text{otherwise} \end{cases} \tag{3.3}$$

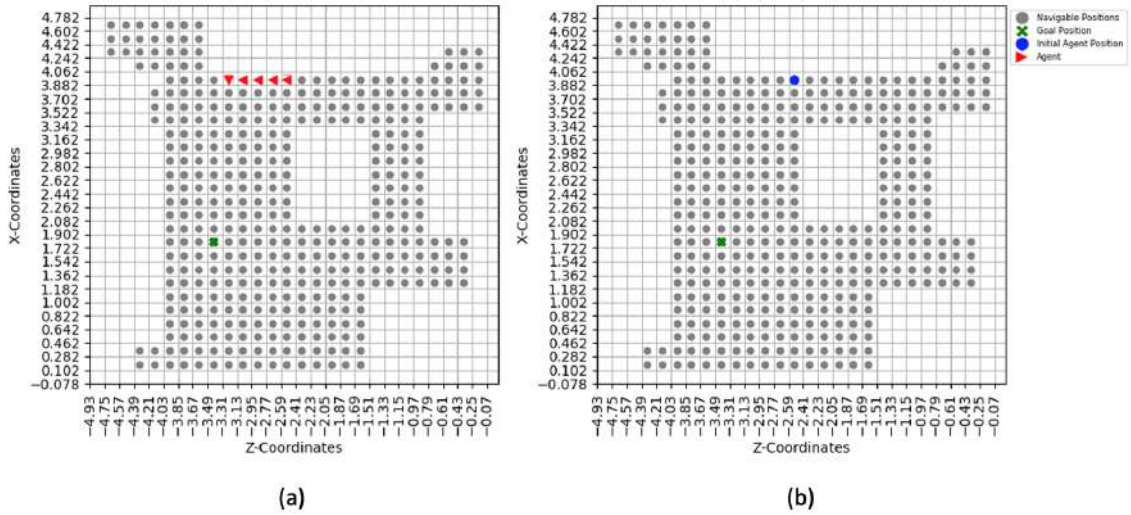

Figure 3.3.: (a) Left figure shows the navigation of the agent toward the goal position. (b) Shows the initial position of the agent and the goal positions and the grid of the scene.

3.2. Deep Neural Network

The deep neural network used in our method is based on the DRQN presented in section 2.4 and is shown in figure 2.13. The shown architecture in figure 3.5 consists of several modules: convolutional autoencoder, LSTM followed by two dense layers. we explain the individual modules one by one.

3.2.1. Convolutional Auto-encoder

Convolutional auto-encoders are nearly same as the dense neural network regards to construction and theory as stated before in section 2.2, but use convolutional layers, discussed in section 2.1.3. The implemented convolutional auto-encoder consists of two modules encoder and decoder each of which consist of six residual blocks [14] with each block consists of 3 layers. The first three layers of each block are convolutional layers(encoder-module) or de-convolutional layers (encoder-module), as show in figures 3.5, and 3.7.

The encoder module input is a resized 256 $\times$256$\times$3 image frame. The resized input frame is then normalized to have mean and variance equals to zero and one respectively.

The preprocessed input frame is forwarded through the encoder consecutive layers. The first layer of each block downsamples its input by stride that is of two. We do not employ any pooling layers e.g. max-pooling [30]. Results showed that using pooling layers for downsampling results in higher reconstruction error leading to poor reconstructed images. On the other hand using only a stride produce acceptable reconstructed images with lower reconstruction error.

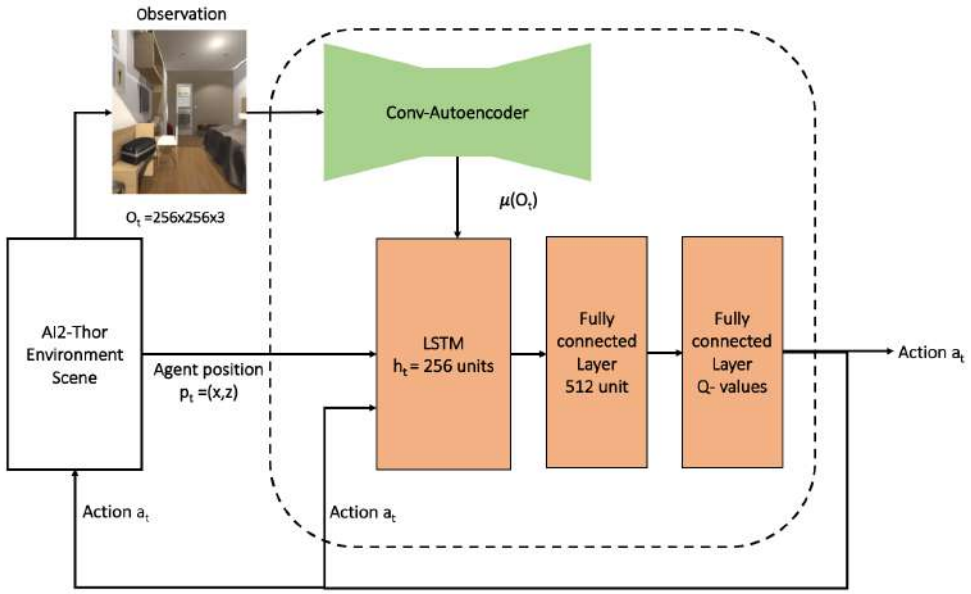

Figure 3.4.: Details of the deep neural network architecture of the agent taking in visual inputs $o_t = 256\times256\times3$, agent position $po_t = (x, z)$ as well as the executed action a_t and producing one hot encoding vector of action Q-values for each action.

The middle layer a dense layer consists of 512 neuron producing a latent feature vector of size equals to 512 as shown in figure 3.5.

The output of the dense layer is then forwarded through the consecutive layer of the decoder module. The decoder consists of de-convolutional layers, in particular 2D-convolutional transpose layers that is one variant of de-convolutional layer that upsamples the latent vector and output an image equals $256\times256\times3$ as shown in figure 3.7. The 2D-convolutional transpose layers not only works exactly like the convolutional layers but also upsamples its input [29], [8], [28]. Figure 3.6 shows the basic concept of 2d-transposed convolution layer. The first layer of each encoder block upsamples its input by a factor of 2 by using stride of 2 as shown in table A.3 at index A. The dense layer of size 1024 in the encoder module is used only to increase the size of the latent vector to be reshaped so that we can upsample the first latent vector to the original input size. See figure 3.7 for detailed architecture of the decoder module.

Each layer of the encoder and decoder blocks is followed by **Leaky-Relu** activation function according to equation 3.4 except the last output layer of each block. The kernel size of all layer is constant equals to 3×3.

$$f(x) = \begin{cases} 0.01x, & \text{if} x < 0, \\ x, & \text{if} x \geqslant 0, \text{where} \quad \text{x} \in \mathbb{R} \end{cases} \tag{3.4}$$

The auto-encoder layers weights and biases are initialized using random samples of truncated distribution centred on zero with standard deviation equals to $\sqrt{2/\beta}$

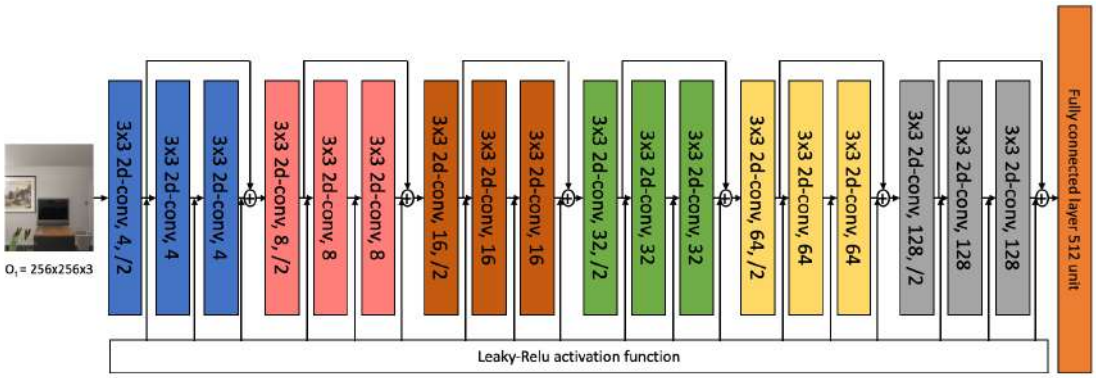

Figure 3.5.: Convolutional auto-encoder encoder module composed of 6 residual blocks, each block consists of 3 convolution layers. Each layer is followed by a **Leaky Relu** activation function.

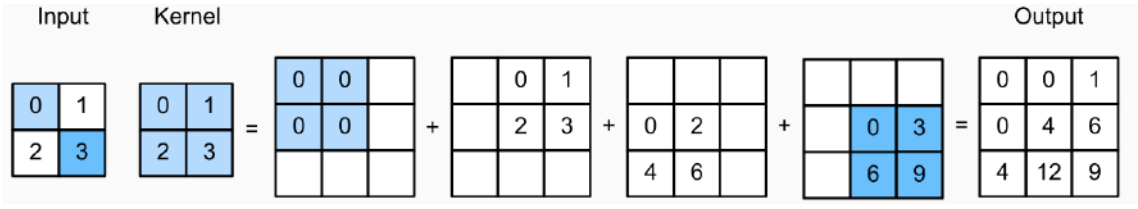

Figure 3.6.: Showing how 2d-convolutional transpose layer upsamples and convolve with the input.

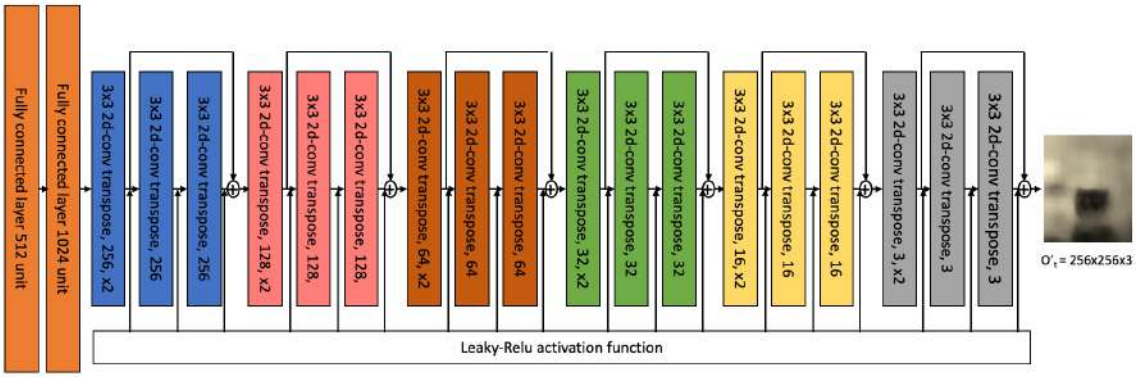

Figure 3.7.: Convolutional auto-encoder decoder module composed of 6 residual blocks each block consists of 3 transposed 2d-convolution layers. Each layer is followed by a **Leaky Relu** activation function, except the last output layer

where β is the number of the inputs in the weight vector. The skip connection or residual block are utilized to overcome the leaky information problem during the downsampling as significant amount is noticed to be lost during training. Moreover working with residual blocks leads to faster convergence. [14]

The reconstruction loss is calculated using mean squared error between the input frame and the output frame according to 2.6. Convolutional auto-encoder concrete components described in table A.3 in appendix A and the training and evaluation details of the convolutional auto-encoder will be presented in chapter 4.

3.2.2. Complete Pipeline

The partial observability of the environment does not allow the agent to uniquely distinguish which state it occupies based on a sole observation. Using previous observations can, however, greatly improve its ability to navigate in the environment. For example, if the agent faces a wall, it can instead look at the previous observation and the action taken.

Overcoming the partial observability problem, In [23] the past observation frames are fed into the network instead of single observation input to deal with partial observation problem. In [12], an LSTM memory was used instead. We have used the latter in our approach.

Past four frames were not enough to capture the complex experience the agent collected when exploring the environment, and more observations lead to an unmanageable increase of the parameter space size and memory requirements. Moreover using last four observations were problematic for our algorithm as we feed also the agent position concatenated with each observation so that we can satisfy HER algorithm requirement. In 2.4 requires to replace the original goal of the episode with additional ones that are sampled from the experience replay, both goal should be in the same space. Using the last four observation for our agent, is inaccurate because we have four position corresponding to each observation for each state that we could sample an additional goal from. Lastly HER algorithm requires to concatenate the goal with the current state which is represented by four observation and their corresponding position.

The Implemented pipeline architecture is based on DRQN architecture described in section 2.4 using a LSTM with 256 node followed by two fully connected layers with 512 rectifier unit and the second one contains number of rectifier units equals to the numbers of action which is 3 in the core agent setting. The last fully connected layer produce a one-hot encoding vector of Q-values for the given action and state.

According to HER assumption that every goal $g \in \mathbb{G}$ corresponds to some predicate $f_g : S \to \{0, 1\}$ and that the agent's goal is to achieve any state s that satisfies $f_g(s) = 1$. In the case when we want to exactly specify the desired state of the system we may use $S = \mathbb{G}$ and $f_g(s) = [s = g]$. [2]

The HER assumption is not satisfied, because in our case the agent state s_t and goal g are defined in different spaces. The proposed solution is to concatenate agent position $po_t \in \mathbb{R}^2$ with the corresponding latent feature vector $\mu(O_t) \in \mathbb{R}^{3\times W\times H}$ from the auto-encoder where W and H are the width and height of the input frame, such that we could apply our modified HER the assumption could be satisfied. Detailed description of the HER in section 3.2.3.

The LSTM input is the partial observation o_t consisted of latent feature vector $\mu(O_t)$, agent current position in the *Euclidean space* $po_t \in \mathbb{R}^2$, and one-hot encoding of agent previous action $a_t \in \mathbb{N}^{N_A}$ concatenated together. Note that o_t is the current partial observation and it is no longer s_t which was considered as the whole current state.

Since our model is based on the DRQN, in our case the network now estimates $Q(o_t, a_t, a_{t-1}, h_{t-1})$ instead of $Q(s_t, a_t, a_{t-1})$ where o_t is the current partial observation, a_{t-1} is the action, and h_{t-1} are the hidden state of the agent at previous step where the hidden state of the LSTM at the current step is defined with $h_t = (o_t, a_{t-1}, h_{t-1})$ and thus estimate $Q(h_t, a_t)$.

DRQN is trained using a random sequence of transitions of length $\tau \subset \{(o_0, a_0, r_0, o'_0),$ $...(o_T, a_T, r_T, o'_T)\}$ sampled randomly from the replay memory, then the temporal difference error in equation 2.24 is optimized, such that DRQN TD-error is defined according to equation 3.5, where $o'_{t+\tau}$ is the estimated next state at the current step t for $T = t$ given the sequence $\tau \subset]t = 0, T[$.

$$L = (r_{t+\tau} + \gamma \arg\max_{a'} Q(o'_{t+\tau}, a', h_{t+\tau}) - Q(o_{t+\tau}, a_{t+\tau}, h_{t+\tau-1}))^2 \tag{3.5}$$

In order to ensure stable recurrent updates we followed the DQN approach of using two separate but identical DRQ-networks namely the *frozen target network* parameterized with $\hat{\theta}$ while the main network DRQ-Network is parameterized with θ. Regularly updating the main DRQ-Network according to the loss function from equation 3.5, while the target Q-Network is updated by copying the parameters of the main DRQ-Network every C time step. Moreover, we also followed the approach of DRQN training using a random sequences sampled from the memory replay in particular *Bootstrapped random updates*, where a random episode is selected from the replay memory and updates begins at random point in the episode and proceed only for the *unrolled iteration* time steps. Using pervious update approach support the DQN random sampling concept but the LSTM hidden state must be zeroed at the start of each update which makes it harder for the LSTM to learn function that lasts longer time scales than the number of the time steps reached by the back propagation through time. Algorithm 5 shows the complete algorithm of the implemented pipeline.

3.2.3. Modified HER

In contrast to the HER algorithm 4, where the policy is trained using trajectories that are revisited with a subset of additional goals substituting the original goal of k individual transitions in replay buffer where $k \in \mathbb{N}$. HER is modified following the same idea of goal replacement for revisited trajectories but for sequence of consecutive transitions belongs to the trajectory not individual transitions. The reward value is recalculated for each transition in the modified sequences given the new goal g' according to reward function $r : S \times A \times \mathbb{G} \rightarrow \mathbb{R}$. Because the off-policy DRQN algorithm is trained on random batches of sequence not on random transition sampled from replay buffer.

In all cases we also train the DRQN policy with the original trajectory where the original goal pursued in the episode. Thus we have k additional trajectories to the original with modified sequences. The modified trajectories have non-zero reward values. The number of trajectories with modified sequences is controlled using k parameter, such that for each trajectory, there is k modified trajectories with sequence of transition.

A random goal sampling strategy is purposed. For a trajectory $o_0, o_1, o_2, ..., o_t, ..., o_T$, Following the HER predicate that the agent's goal is to achieved in any observation that satisfies the goal properties $fg(o) = 1$, the sampled additional goal equals to a random sampled partial observation at random time step t, $g' = o_t$ where $0 < t \leq T$. Note here by trajectory we mean the agent consecutive states and actions from $t = 0, ..., T$ where T is the terminal state of the episode. The original goal g that belong to each transition in sequence $o \in [o_0....o_t]$, where $t \in \mathbb{N}$ is replaced with g'. The reward value of each transition in the additional modified sequences for the given agent trajectory, is calculated according to extrinsic shaped and sparse reward function in equations 3.2 and 3.3 respectively. See Algorithm 6 for a more formal description of the algorithm.

Given reward function $r : O \times A \times \mathbb{G} \to \mathbb{R}$, See function 3.2 or 3.3;
Initialize replay memory D to capacity N;
Initialize action-value function parameters θ_0 with random weights;
Initialize target action-value function parameters $\hat{\theta} \leftarrow \theta_0$;
for *episode* $= 1, M$ **do**
  Init h_0 with zeros Sample random action $a_{t-1} \in A$ as previous action;
  $o_t \leftarrow \mathcal{O}(s)$ partial observation;
  $h_0 \leftarrow 0$;
  while $o \neq$ *terminal* **do**
    With probability ε select random action a_t;
    Otherwise $a_t, h_t = \arg\max_{a'} Q(o_t, h_{t-1}, a'; \theta)$;
    Take action a_t;
    Retrieve next observation and reward (o_t, r_t);
    Store transitions (o_t, a_t, r_t, o_{t+1}) in D;
  end
  Apply modified HER for the current episode see algorithm 6;
  Store additional modified HER sequences with randomly sampled goals in D;
  for *t=1, N* **do**
    Init $h_{t-1} with zeros 0$;
    Sample random mini-batches B of sequences, sequences with length $\tau = [0, t]$;
    $B = \{(o_t, a_t, r_t, o'_t), ..., (o_{t+\tau}, a_{t+\tau}, r_{t+\tau}, o'_{t+\tau})\}_{i=1}^{batchsize} \subseteq D$;
    perform a gradient step descent on $L_j(\theta) = y_j - Q(o_{t+\tau}, a_{t+\tau}, h_{t+\tau-1}; \theta))^2$;
    Where $y_j = r_{t+\tau} + \gamma \arg\max_{a'} Q(o'_{t+\tau}, a', h_{t+\tau}; \hat{\theta})$
  end
end

Algorithm 5: Double Deep Recurrent Neural Network with modified HER and recurrency

Given:

- an off-policy algorithm $\mathbb{A}$;
- reward function $r : O \times A \times \mathbb{G} \to \mathbb{R}$ See functions 3.2 or 3.3;
- random sampling strategy $\mathbb{S}$ for sampling goals for replay, $\mathbb{S}(o_0, o_1, ..., o_t, .., o_T) = m(o_t)$;

Initialize $\mathbb{A}$ with DRQN, see algorithm 5;
Initialize replay memory D to capacity N;
Initialize additional buffer D' to capacity N;
for *episode* $= 1, M$ **do**
  Sample a goal and an initial partial observation o_0;
  while $o \neq o_T$ **do**
    Sample an action a_t using the behavioural policy from A:
    $a_t \leftarrow \pi(o_t, ||g)$, $||$ ▷ donates concatenation;
    Execute action a_t and observe a new partial observation o_{t+1};
    $r_t := (o_t, a_t, g)$;
    Store transition $(o_t||g, a_t, r_t, o_{t+1}||g)$ in D ▷ standard experience replay;
    Sample set of additional goals for replay $G := \mathbb{S}$(**current episode**);
  end
  for $g' \in G$ **do**
    for *t=0, t=*τ ▷ τ *donates sequence length* **do**
      $r' := (o_t, a_t, g')$;
      Store transitions $(o_t||g', a_t, r', o_{t+1}||g')$ in D' ▷ modified HER;
    end
    Store the sequences D' in D ▷ modified HER;
  end
  for *t=1, N* **do**
    Sample a mini-batch B of sequences from the replay memory D
    Perform one step of optimization using A and mini-batch B
  end
end

Algorithm 6: Modified HER algorithm for training with DRQN algorithm

4. Policy Training and Evaluation

The training and evaluation for the convolutional auto-encoder is done separately, then the off-policy neural network is trained. Firstly the training and evaluation dataset of the auto encoder is presented. The training dataset of the off-policy algorithm is composed of one environment scene. The policy generalization is evaluated using another environmental scene. Second the training setup of the auto-encoder and the policy is presented. Lastly, the evaluation results of the auto-encoder and the different trained agents with different input sizes to the LSTM cells and two different action sets.

4.1. Convolutional Auto-encoder Dataset

To train our auto-encoder, we first collect a dataset of a number random rollouts of the environment. We have first an agent acting according random policy to explore the environment multiple times, and record resulting observations from the environment. The gathered dataset consists of training dataset and evaluation dataset for each environment scene category. Each environment scene category has 30 unique scenes splitted into 20 scene used for training and the remaining 10 are used for the auto-encoder evaluation. The auto-encoder is tested using the evaluation dataset but shuffled. Figure 4.1 shows samples of the recorded frames during random rollouts. Table 4.1 summarizes the dataset for the convolutional auto-encoder training and evaluation.

4.2. Convolutional Auto-encoder Training and Evaluation Setup

The auto-encoder is trained and evaluated on mini-batches of sizes 32 and 16 respectively, The auto-encoder is trained for 40×10^3 iterations and evaluated for 5×10^3 iterations. The agent is evaluated 3 times every 10×10^3 training iteration by freezing the network weights and feed the auto-encoder with validation mini-batches. Every iteration the reconstruction loss between the input frames and the generated frames is optimized using Adam optimizer with a decaying rate equals to 1×10^{-3} . [17]

Training dataset	4 Scenes categories×20 scene/category×160 frame
Evalaution dataset	4 Scenes categories× 10 scene/category×160 frame

Table 4.1.: Convoltional auto-encoder training and evaluation dataset summary.

Figure 4.1.: Examples of the training datasets, marked with orange and validation marked with blue. Each recorded frame is a 3D-Tensor of dimension $300 \times 300 \times 3$.

Auto-encoder gradient updates are regularized by clipping the gradient updates to be between -1 and 1. The regularization of the gradient updates was necessary, to mitigate the exploding gradient problem. Lastly the network weights is stored every 50 iteration to be used later to provide the off-policy LSTM layer with a latent feature vector of the agent observation during the policy training. See table A.1 in appendix A for a detailed training hyper-parameter values of the convolutional auto-encoder.

4.3. Off-policy Training and Evaluation Setup

For the off-policy training and evaluation we have a different agent settings.The different agent settings are trained for 15×10^3 episode and evaluated for 100 episodes every 4×10^3. The agent action are sampled according to ε -greedy policy intialized equals to one and decremented in each iteration according to decaying rate (See equation 4.1). Each iteration the probability to execute randomly sampled action or action sampled according to the policy to $\max\{\varepsilon \times \varepsilon_{decay}, \varepsilon_{min}\}$, the minimum value is set to equal 1×10^{-2}. The reward discount hyper-parameter γ is setted to 0.98. The DRQN loss is optimized using a Adam optimizer [17] with a learning rate of 1×10^{-3}. The Neural network loss is optimized each episode for 40 iteration on 32 randomly sampled mini-batches of sequences with length 8, where each sequence is represented by agent consecutive steps of each iteration. The maximum number of step of each episode equals 200 step, if the agent does not reach the goal before the maximum number of steps, the episode is a failed trajectory. Every 4×10^3 training episode the weights and biases of the main network is updated once per main network update by *polyak averaging* (See equation 4.1) following the DQN. In equation 4.1 ρ is a hyper-parameter between 0 and 1. Tables A.4 and A.2 in appendix A summarizes the hyper-parameters of the training and neural network respectively.

The agent is trained using two different action set. The core action set is composed of three actions *Move forward, Rotate left* and *Rotate right*, meanwhile the extend action set has 3 more additional actions *Move back, Move left* and *Move right*. The agent is trained and evaluated be feeding either only consecutive frames encoded in dense feature vector of size concatenated with goal position vector of each episode or additionally the agent position vector is concatenated with the corresponding encoded frames and goal position to produce a vector of size 518 to the agent LSTM. The agent

with the core action set is trained by feeding the LSTM using both inputs setting. Each agent input setting is trained and evaluated two times, firstly where the modified HER algorithm and sparse reward function are employed, while the second time HER and shaped reward are used to see how the different reward functions influence the HER. Additional agent is trained one more time using the shaped reward only to evaluate agent performance without the HER. The agent with the extended action set is trained and evaluated one time by feeding the LSTM with the encoded frames and their corresponding agent position while employing HER with the shaped reward function. Table A.2 in appendix A summerizes the agent training inputs vector and the whether is trained and evaluated using the HER only or HER with the shaped reward function. Notice when only HER is applied during training the sparse reward is utilized (see equation 3.3).

$$\varepsilon_{decay} = 1 - (1/n), \text{where } n \in \mathbb{N} \text{ is a hyper-parameter} \tag{4.1}$$

4.4. Convolutional auto-encoder Evaluation

The thesis main focus is to implement and RL off-policy combined with HER algorithm to perform multi-goal navigation task, so that the quality of the reconstructed images by the convolutional auto-encoder does not need to be high at all.The objects in the reconstructed images need only to some extent distinguishable from each other compared to the original input frames. In other words the auto-encoder reconstruction loss does not need to be very small as long as the reconstructed images are of acceptable quality. The reconstructed frames quality gives an indication on the encoded latent feature vector quality whether it provide a good encoding of the image frame or not.

The auto-encoder reaches a minimum reconstruction loss value achieved during training of 8.281×10^3. Figure 4.2 shows the reconstruction loss during training. The auto-encoder loss curve starts to saturate after 20 hours of training and the construction loss does not goes below the minimum reconstruction loss that was achieved after 80 hours of training.

The auto-encoder evaluation result was almost identical to the training result, meaning that the auto-encoder does not suffer from over or under fitting problem. Neural network model that is underfit will have higher training and high testing errors while an overfit model will have extremely low training error but a high testing error. In other words an underfit model will not be able to recognize seen or unseen data but an overfit model would be able to recoganize only seen data during training but will not generalize to unseen dataset.[7]

Figure 4.3 shows the auto-encoder loss validation. The resulted loss value during auto-encoder validation are approximately between 15×10^3 and 60×10^3. The minimum reconstruction loss value achieved during auto-encoder validation is 13.312×10^3.

Lastly figure 4.4 shows both the auto-encoder reconstruction loss during training and model validation combined together for comparison, proving that the implemented auto-encoder does not suffer from overfitting or underfitting problems given

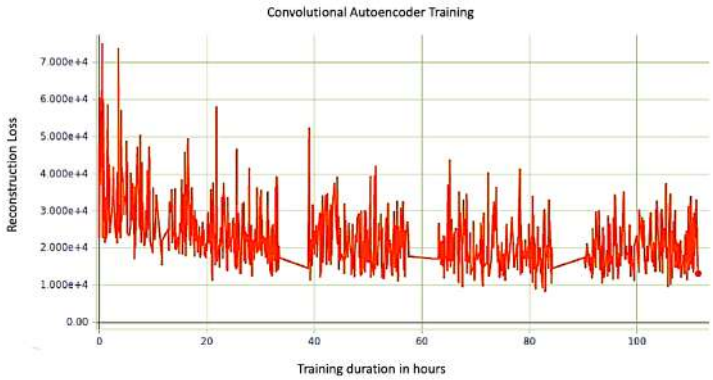

Figure 4.2.: Convolutional auto-encoder training Reconstruction Loss.

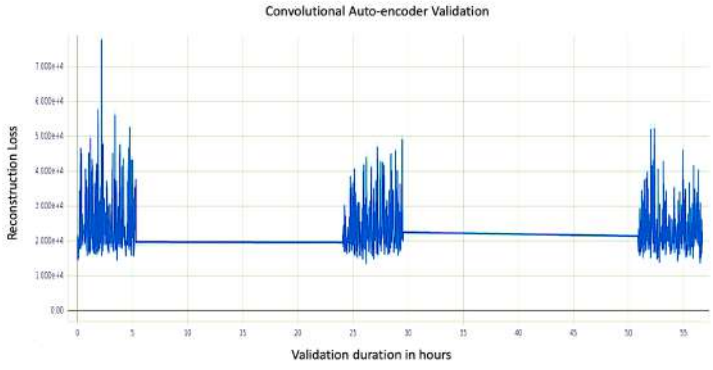

Figure 4.3.: Convolutional auto-encoder validation Reconstruction Loss.

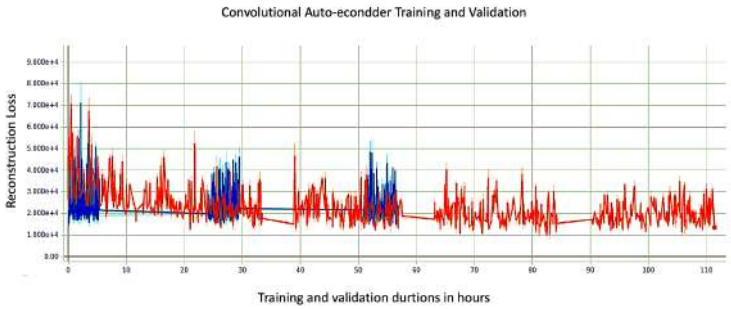

Figure 4.4.: Convolutional auto-encoder reconstruction loss graph during training and validation.

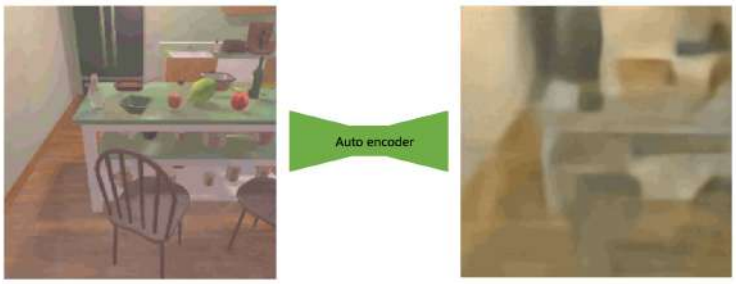

Figure 4.5.: Convolutional auto-encoder image frame input and the corresponding reconstructed frame of training dataset.

the training dataset. Figures 4.5 and 4.6 shows samples of the training and validation dataset and their corresponding reconstructed images by the auto-encoder.

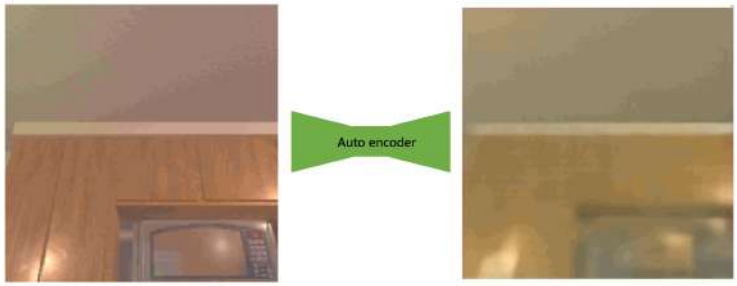

Figure 4.6.: Convolutional auto-encoder image frame input and the corresponding reconstructed frame of validation dataset.

4.5. DRQN Off-policy Evaluation

4.5.1. Quantitative Evaluation

Agent with the First Input Setting Evaluation

The success and failure rate averaged by the number of training or validation episodes, is the metric used for evaluating the performance of the agent. Moreover F1 metric is also used for evaluating the agent learned policy.

Figures 4.7 and 4.8 show the average scores of successful and failed episodes of an agent with first input setting to the LSTM (See table A.2 in appendix A) trained with HER combined with the shaped reward. Both figures show good learning progress and scores achieved during the agent training. After 15×10^3 training iterations, the agent was able to achieve an average success rate of 0.606 and the failure rate decrease to a minimum value of 0.45, while the ratio between successful and failed episodes reached a maximum value of 1.78 at the end of the training.

During training the agent is evaluated by rolling out 100 episodes every couple training episodes to check the agent learning progress. The agent evaluation results in figure 4.9 shows also a very close result that matches the agent training scores in figures 4.7 and in figures 4.8.

In Figures 4.10 and 4.11 show scores of successful and failed episodes of an agent with first input setting to the LSTM (See table A.2 in appendix A) trained with HER while sparse reward function is employed during training. Unfortunately The agent was barley able to reach reach an average success rate of 0.16 while the failure rate stays around 0.9 by the end of agent training. The F1 score of this agent give a clear indication that the agent was not able to learn almost anything, although HER is employed.

The evaluation of the agent training with sparse reward function, shows similar result to the agent training with a maximum success rate around 0.2. The F1 evaluation score is the same as for the training as shown in figure 4.12. Clearly the agent

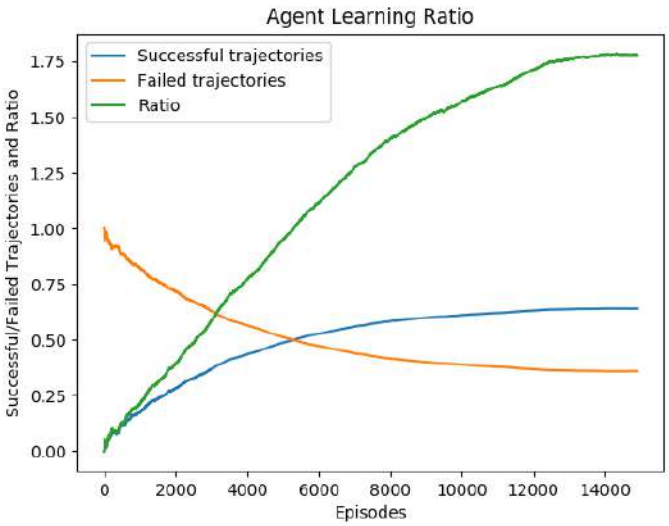

Figure 4.7.: Learning progress and average scores of an agent with first setting to the LSTM trained with HER combined with shaped reward.

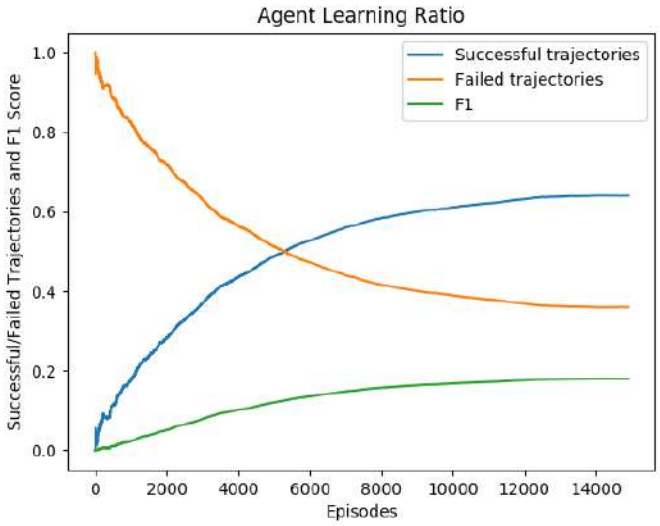

Figure 4.8.: Average successful, failure rates and F1 score of an agent with first setting to the LSTM trained with HER combined with shaped reward.

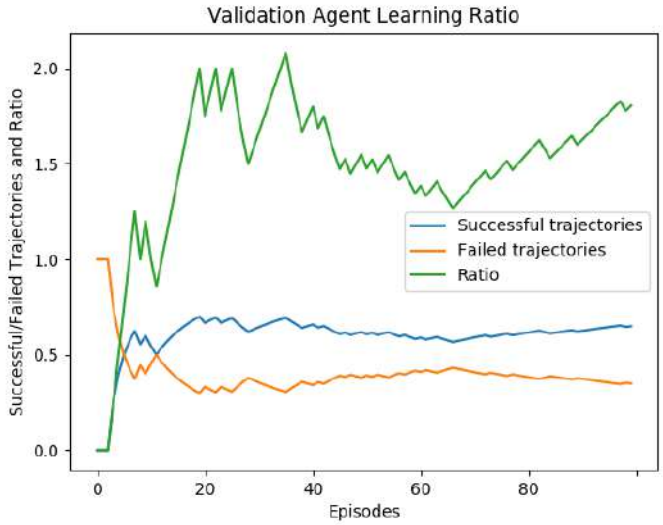

Figure 4.9.: Evaluation scores of failed and successful episodes of agent with first input setting after trained with HER and shaped reward.

policy was barley able to learn anything other than random success when the used reward signal is sparse and binary.

Agent with the Second Input Setting Evaluation

As stated before the second agent input setting to the LSTM is only composed of the encoded latent vector of the consecutive frames during agent training. Training average scores of this agent are almost identical to agent trained on encoded frames vector concatenated with the agent position vector at each step.

Figures 4.13 and 4.14 shows average success and failed episode. The learning progress of the agent yields acceptable performance and average success score of 0.61, while the number of failed episodes reach a minimum value of 0.41 at the end of the training.

The evaluation of the trained agent on encoded image vector while shaped reward is employed, is almost identical to the training results of this agent as shown in figure 4.15. The agent was able to reach a success rate of 0.54 and ratio value of 2 between the successful and failure episode after rolling out 100 episodes.

While trained agent on encoded images using shaped reward, yields good scores, the same agent trained using sparse reward signal instead of shaped one shows poor performance during training and evaluation. As shown in figures 4.19 and 4.20 the agent reach a maximum score around 0.080 of successful episodes, while the failure rate was almost stable and barley decreases below 0.98.

The validation results of this agent was as expected and the evaluation scores were almost equal to the training using sparse reward function results as shown in figure

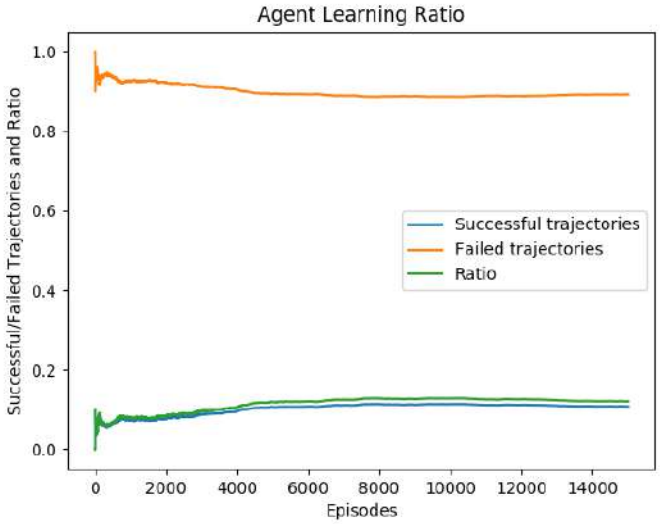

Figure 4.10.: Learning progress and average scores of an agent with first setting to the LSTM trained with HER employing sparse reward function.

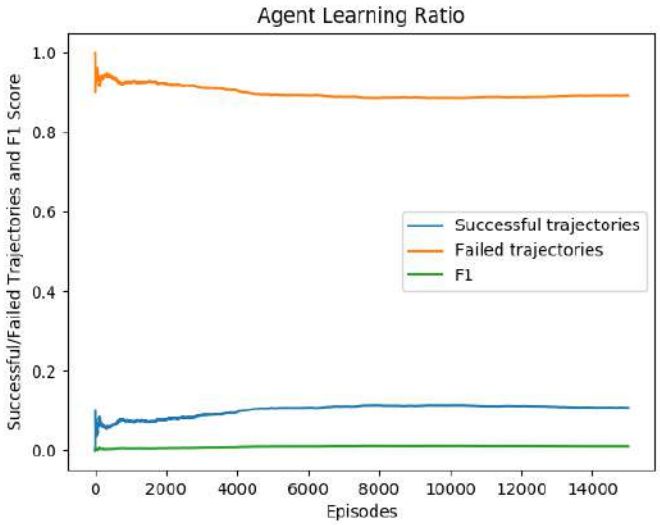

Figure 4.11.: Average successful, failure rates and F1 score of an agent with first setting to the LSTM trained with HER, with sparse reward function.

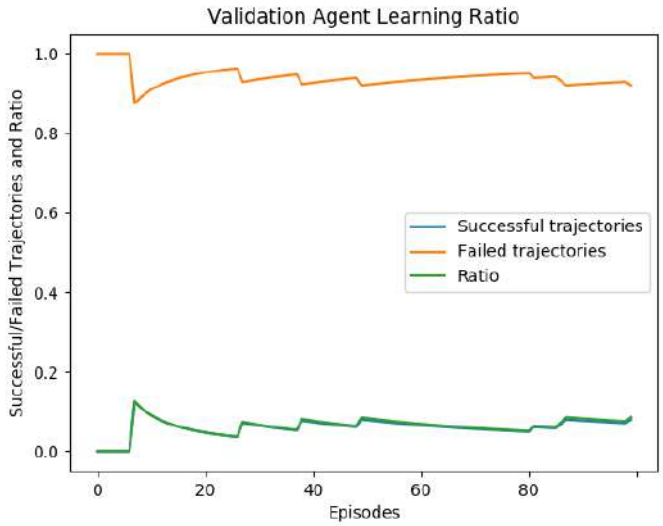

Figure 4.12.: Evaluation scores of the trained agent with first input setting employing sparse reward function.

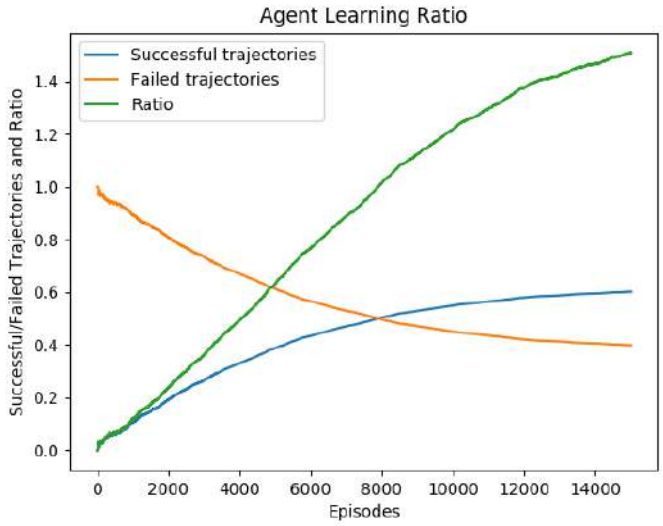

Figure 4.13.: Training result of an agent trained on encoding vectors of the frames only and using shaped reward.

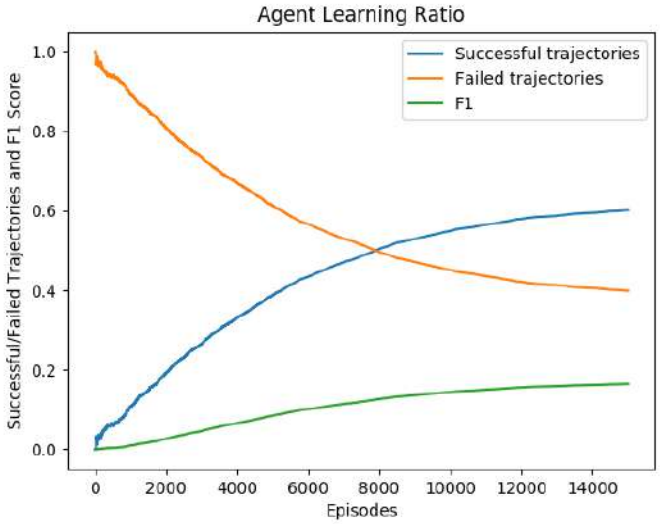

Figure 4.14.: Agent F1 score and the success and failures average scores.

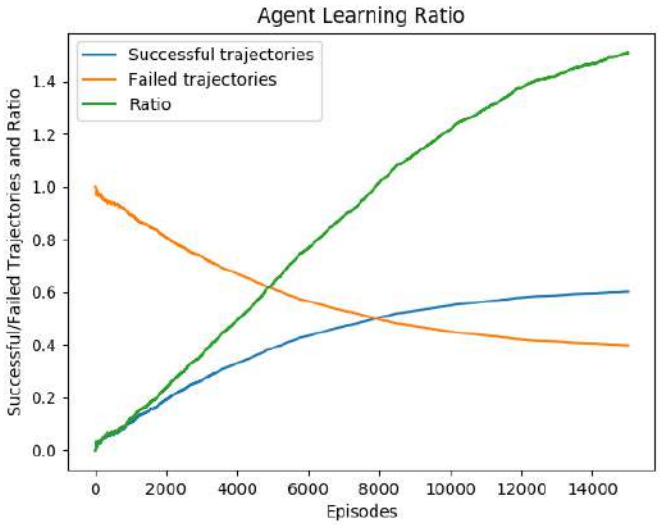

Figure 4.15.: Evaluation results of the agent trained on second input setting and shaped reward.

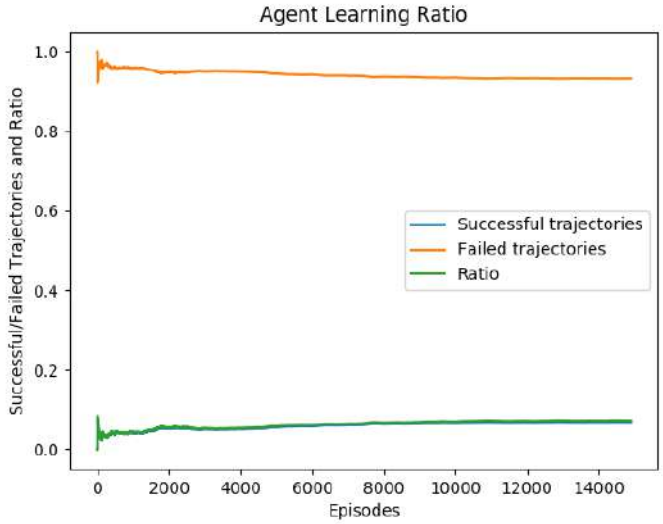

Figure 4.16.: Training result of an agent trained on encoding vectors of the frames only and using shaped reward.

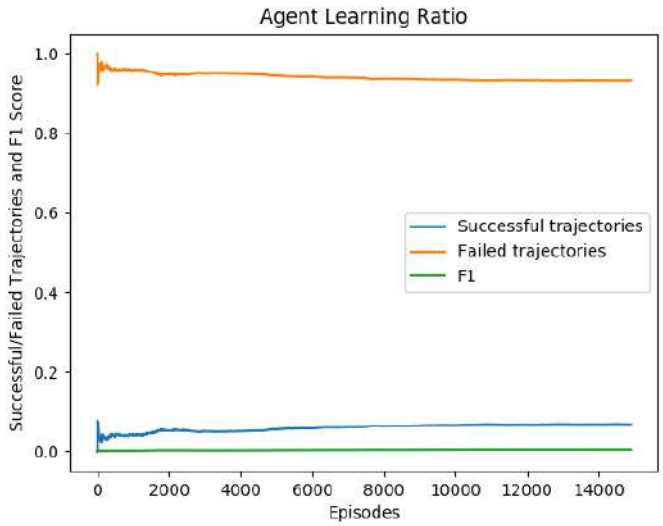

Figure 4.17.: Agent F1 score and the success and failures average scores.

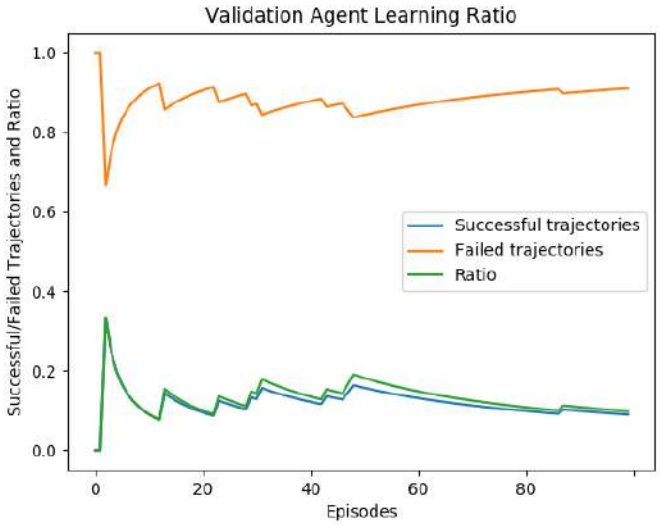

Figure 4.18.: Agent F1 score and the success and failures average scores.

4.21. The maximum success score of the agent during validation is 0.05 while the failure rate is almost stable and equals to 0.95.

Evaluation of agent trained using Shaped Reward

The last agent was trained only on shaped reward without HER, while the LSTM input was composed of the encoded feature vector and the corresponding agent position vector. This agent was trained to evaluate the agent performance without HER. The training results shown in figures 4.19 and 4.20 shows a comparable scores to other trained agent using shaped reward. At the end of the training the agent was able to achieve an average success score of 0.516, failure score of 0.484 and ratio of 1.065. Figure 4.21 show the agent validation scores. After validating the agent for 100 episodes, the agent scores were 0.85 and 0.15 for successes and failures respectively.

By reviewing the different agent training scores and validation score, it is clear that the implemented HER influence on the agent learned policy is very small. As stated before HER was purposed to overcome the sparse reward problem but the modified HER does not influence the learned policy by the agent when trained using sparse reward function. In contrast to the HER and sparse reward function influence, training the agent using shaped reward function affects the agent policy to adjust its behaviour appropriately to reach the given goal position and perform its given task. Figures 4.22 and 4.23 shows the effect of the used reward functions during agents training on the policy convergence. When the agents were trained using the shaped reward, the policy loss curve shows stable descent and reach a minimum value of zero as shown in figure 4.23. Furthermore training the agents using sparse reward results in less stable loss updates during training as shown in figure 4.22

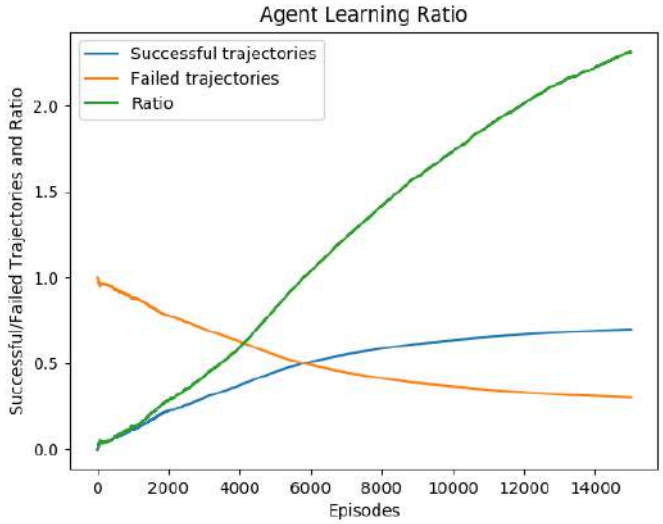

Figure 4.19.: Training result of an agent trained on encoding vectors of the frames only and using shaped reward.

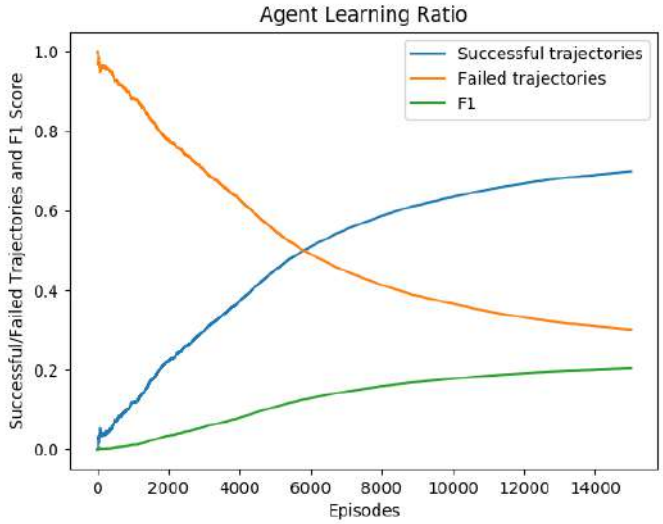

Figure 4.20.: Agent F1 score and the success and failures average scores.

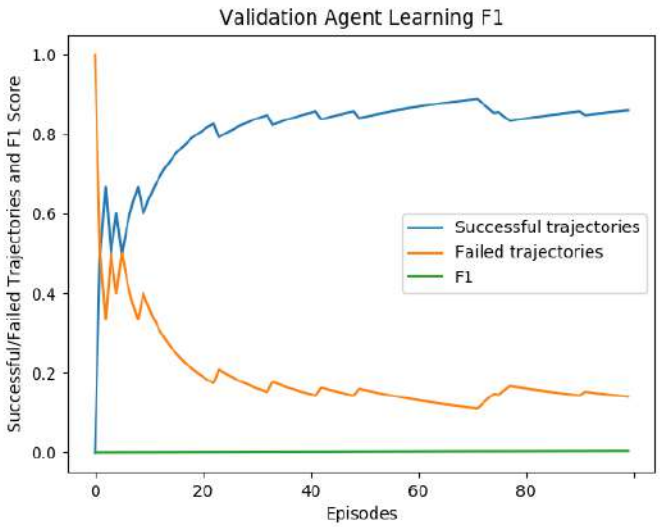

Figure 4.21.: Validation of trained Agent using shaped reward.

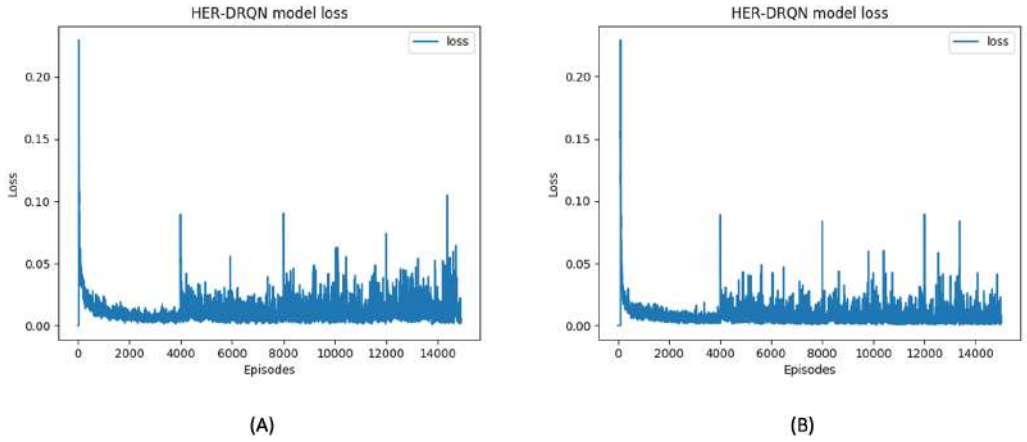

Figure 4.22.: Loss updates of agents with sparse reward function, trained on encoded images (Diagram A) and encoded images and position vector (Diagram B) as observation.

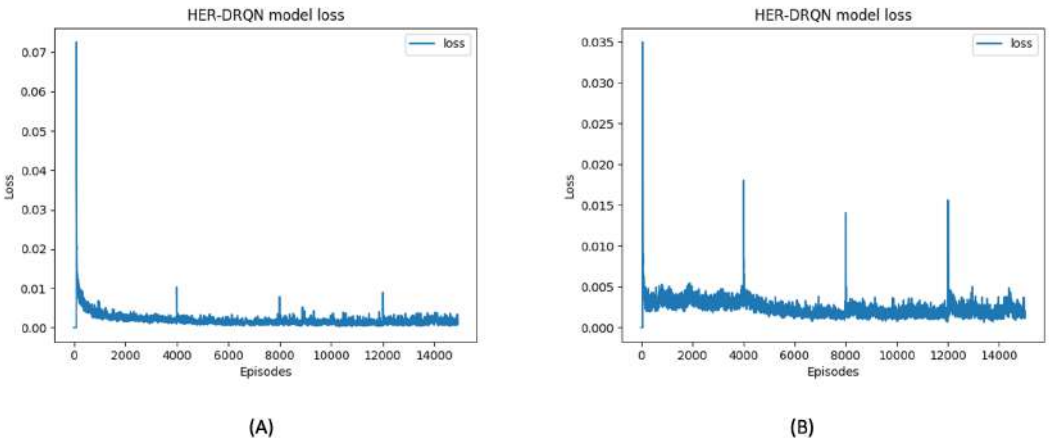

Figure 4.23.: Loss updates of agents with shaped reward function, trained on encoded images (Diagram A) and encoded images and position vector (Diagram B) as observation.

Lastly we train only one agent using the extended discrete action set. The training has been done using only shaped reward since the scores of sparse reward trained agent were very low compared to the shaped ones. At the end of the training the agent a maximum average success score of 0.603, average failure score decreased to 0.397 and ratio of 1.5 between successful and failed episodes as shown in figures 4.24 and 4.25. Also as shown in figure 4.26 the agent is validated for 100 episodes where the agent achieves 63% success rate, 37% average failure score. In A table A.4 summarizes all trained agent average successful, average failed episodes at the end of trained and the ratio between successes and failures.

4.5.2. Qualitative Evaluation

To fully evaluate the results of the agents, it is not sufficient to only consider the quantitative results. It is also interesting and necessary to look into the learned policy. In the course of work, identifying which trained agent is better than other given different agent input settings and evaluating the trajectory of each agent to the goal.

During the trained agents evaluation, it is noticed that the agents that were trained using only encoded image vectors as observation tends to take longer trajectory to the different goals, meanwhile the trained agent using not only encoded image vector but also position vector take shorter trajectory. In figure 4.27 the showed agent trajectory to the goal position is 3 steps longer than the agent trajectory in figure 4.28. The

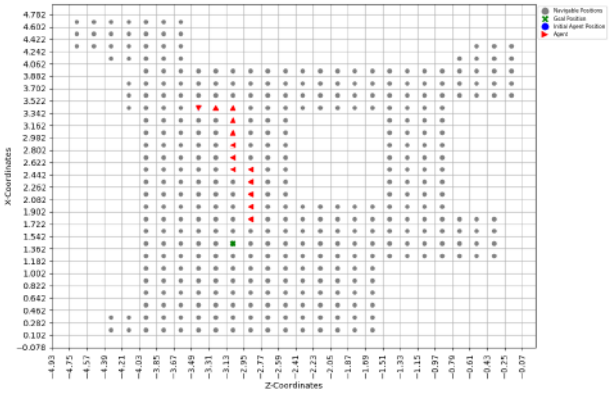

Figure 4.24.: Scores of an agent trained using extend discrete action set.

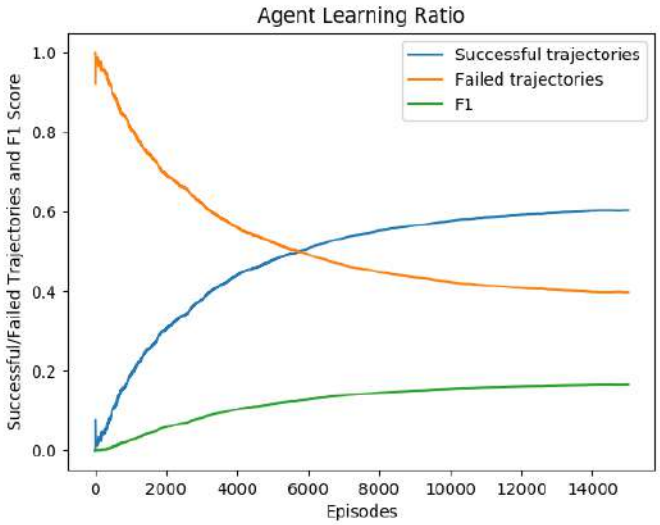

Figure 4.25.: F1, success and failure scores of an agent trained using extend discrete action set.

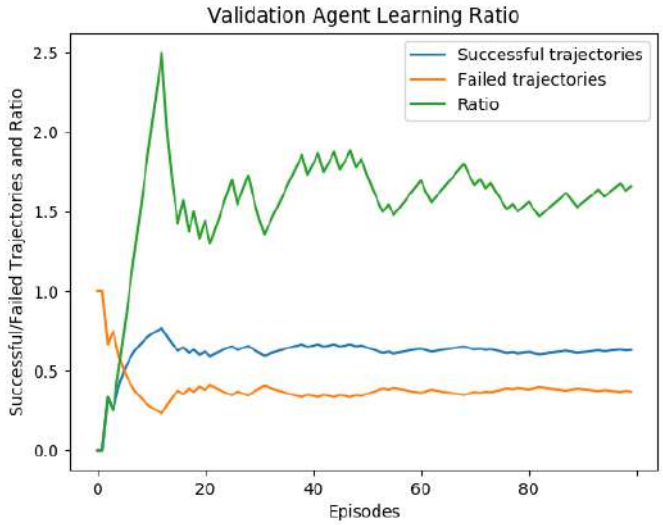

Figure 4.26.: Validation scores of an agent trained using extend discrete action set.

first agent trained only on encoded image vector while the later trained on encoded image and position vectors.

Notice in figures 4.27 and 4.28 not all steps of the agent could be shown, there are some steps that are already written over by later steps. In order to have a clear distinction on the behaviour of both agent, we evaluated the agents trajectories much further. Both agents trajectories are evaluated by playing 50 episode for each agent, and then count the number of steps of each episode for each agent. Figure 4.29 shows the number of step per episode for the agent with input of encoded images and position vectors, the agent reached the maximum number of 200 step per episode only 4 times. The blue bars are the successful episodes, where the agent was able to reach the goal in number of steps ranges between 5 and 37 steps. Comparing figure 4.29 with figure 4.30 of an agent trained only on latent feature vector, as shown the agent reached maximum number of steps in 5 episode, and the number of steps for the successful episodes approximately 2 steps higher than the other agent. On contrast to the previous agent which was able success only 26 time, this agent was able to success 30 times in a number of step comparable to the previous agent.

Figure 4.31 show the trajectory of an agent trained using extended discrete action set, that has more three action as stated before. The agent trajectory shows that agent has a favoured the extended action namely move left, move right and move back over other rotation actions. This behaviour was expected because of the used shaped reward, which punishes the agent with a negative reward signal of -0.5 if the agent stand still and does not step closer to the goal. Although the agent was not able to see where it is going in the next step, the agent still able to memorize and reach the goal position.

We extend the evaluation of this agent, to check for the number of steps agent needed to reach the goal. Figure 4.32 shows that the agent reaches the maximum

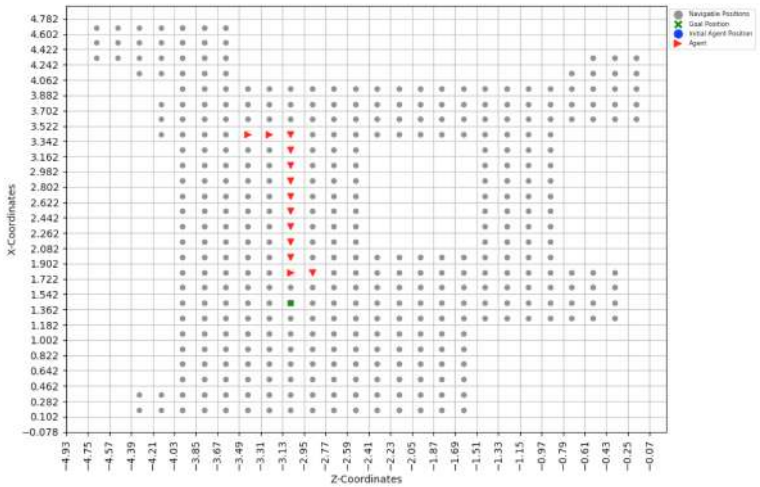

Figure 4.27.: Trajectory to the goal position of an agent trained using encoded image vector as observations.

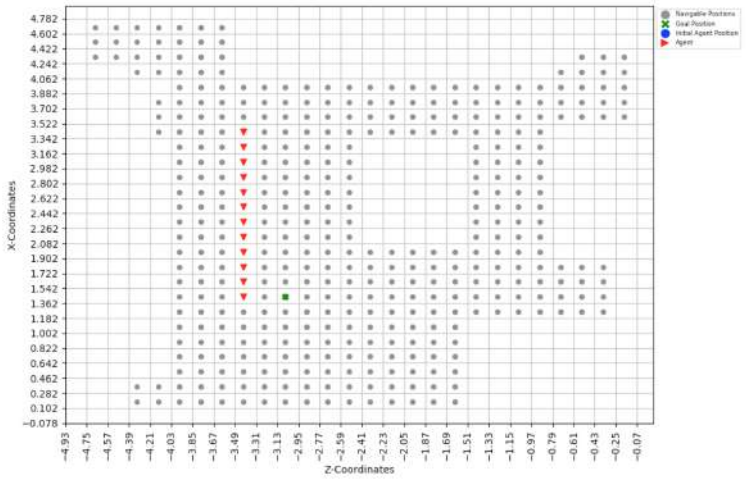

Figure 4.28.: Trajectory to the goal position of an agent trained using encoded image vector and position vectors as observations.

Figure 4.29.: Number of successful vs failed episodes and the number of steps per episode for image encoded agent.

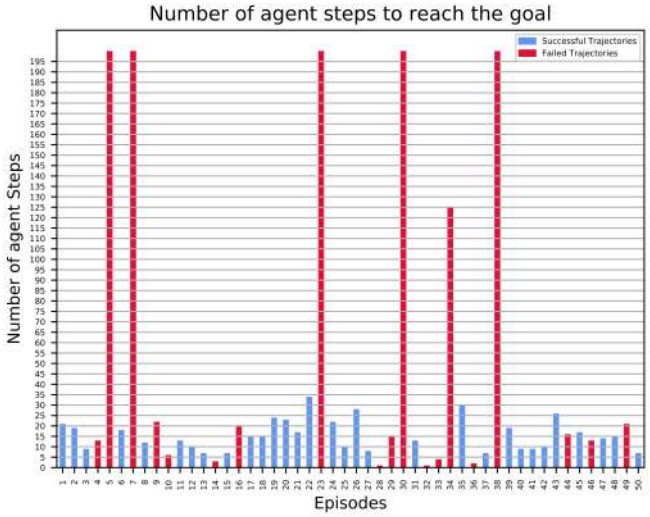

Figure 4.30.: Number of successful vs failed episodes and the number of steps per episode for image encoded and position vectors agent.

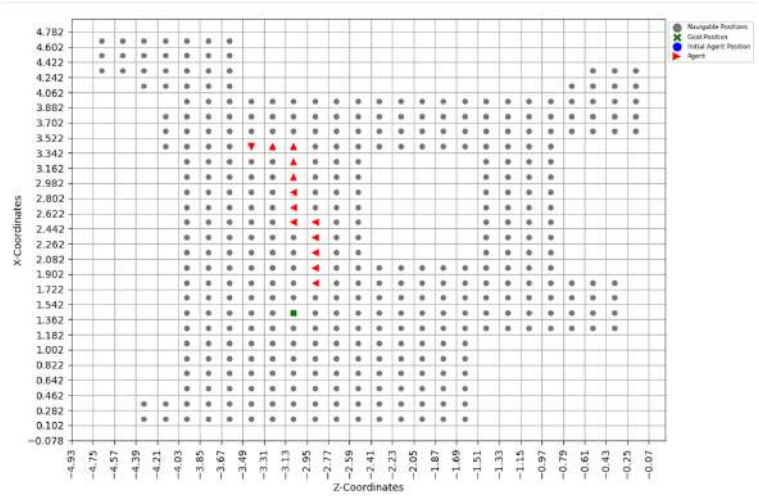

Figure 4.31.: Example trajectory to the goal position of an agent trained using extended action set.

allowed number of steps per episode 3 times, and 34 success when evaluated for 50 episodes. The number of step required to reach goal position ranges between 3 and 32, this range is lower than the number of all other agents by 2 steps.

The policy generalisation to other environment scene. An agent trained on encoded frames and position vector as input played 50 episode, 4.33 clearly show that the agent collide in all episodes, and was able to success in only 1 episode. This behaviour was expected as the agents were trained one environment only. In order for the agent to generalize to other scene, the agent neural network must be trained on multiple environment at the same time. Due to long training times and not having enough time for additional runs.

Figure 4.32.: Number of successful vs failed episodes and the number of steps per episode for an agent trained using extended discrete action set.

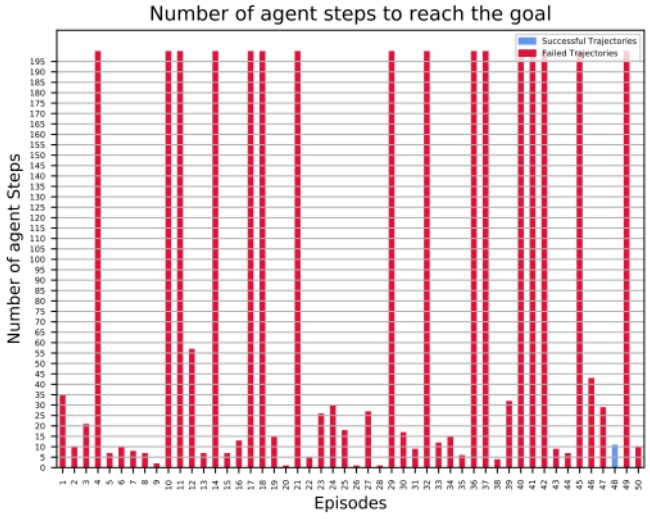

Figure 4.33.: Number of successful vs failed episodes and the number of steps per episode for an agent evaluated in another environmental scene.

5. Conclusion and Future work

5.1. Conclusion

In this thesis, Deep Reinforcement learning (DRL) has been applied in the navigation task where a randomly deployed agent trained to navigate from its initial position to a goal position. Several agent has been trained on different observations vectors, the first observation vector composed of the encoded image frames and agent position vector, while the second agent trained only on encoded image frames as observation vector. Those agents trained by employing two different reward signals, shaped reward signal and sparse reward signal. Moreover a purposed modified version of Hindsight Experience Replay (HER) algorithm is implemented to deal with the sparse reward problem so that agents could be trained using Deep recurrent Q-network off-policy algorithm. Moreover the agents trained on two different discrete action sets, composed of three and 6 actions.

Evaluation of the trained agents shows that all agents performs very well when trained using shaped reward function, and they were able to achieved success rate ranges between 50% and 60% for both agents. Sparse reward trained agents evaluation results shows much less success rate of 2%, although the usage of of HER modified algorithm. In other words our purposed HER influences on trained agent is very low when used with sparse reward compared to the shaped reward signal.

Unexpectedly the agent that trained using latent dense feature vector as observation outperforms the other agents using latent dense feature and position vectors. Indicating that the model learn a rough map of the environment and has the capability of localization with respect to this map.

Training the agents on two different discrete action sets, showed how the reward function affect action executed in each time step. So for the core action set, the agent has to learn a policy that use all of the action, in order to reach the goal position. For the extended action set the learned policy at the end of the training favoured some actions over others. Because of the reward function influence on those action, the agent almost discarded the rotation action, as the agent gets negative reward for not minimizing the distance to the goal position. Although the agent does not looks where it is heading, it was still able to reach the goal showing that the introduced recurrent model memorize not only where the goal is, but also the object positions that the agent can collide with.

Because of the long training period, we do not have enough time to train the agent on multiple environment at the same time. So that the learned policy does not generalizes well for other environment. But we are sure if the agent was trained on multiple environment, the agent would be able to navigate in different environment.

Our result cannot be compared to results of the original implementation of HER, as both HER techniques were used to train agent for different tasks. In the original paper the agent is trained to do pushing, sliding, and pick and place tasks, while our task for the agent is to navigate to goal position. Although our HER implementation we also tried different number for the additional modified sequence, the effect of our HER on the learned policy does not change and produces the same poor performance reaching the goal positions.

5.2. Future work

In conclusion, the obtained results are very convincing and the proof-of-concept for applying Deep Reinforcement Learning to the navigation task robot is confirmed. A high potential for improvement can be expected if more work and further investments are made in this field. Furthermore, hybrid variants like trying the the original HER implementation with DRQN would yield better results regarding the HER with sparse reward function. Introducing a recurrent layer to other off-policy algorithms e.g Deep Deterministic Policy Gradient (DDPG) [21], combine it with our HER would also give a different results than ours. Regarding the modified HER, other goals sampling strategy could be introduced. Instead of training the agent on position vector concatenated with encoded frames vector, other approach could be used like in [3], where a dense embedding feature vector of the agent observations and agent position.

A. Appendix

Hyper-parameter	Value
Training mini-batch size	32 batches
Evaluation mini-batch size	16 batches
Conv auto-encoder kernel parameter initializer	He-Normal initializer [13]
Conv auto-encoder activation function	Leaky-Relu
Loss optimizer	Adam optimizer [17]
Learning rate	1×10^{-3}

Table A.1.: Convoltional auto-encoder training and evaluation hyper-parameters.

	LSTM input components	
Trained using	latent feature vector with position vector	latent feature vector
Shaped reward only	✓	-
HER with sparse reward	✓	✓
HER with shaped reward	✓	✓

Table A.2.: Summary of DRQN off-policy different input setting and which reward function is employed with HER during training.

Layer	Layer type	Input size	Output size	Stride	Number of Filter
1	2D-Conv	256×256×3	128×128×4	2	4
2	2D-Conv	128×128×4	128×128×4	1	4
3	2D-Conv	128×128×4	128×128×4	1	4
4	2D-Conv	128×128×4	64×64×8	2	8
5	2D-Conv	128×128×4	64×64×8	1	8
6	2D-Conv	128×128×4	64×64×8	1	8
7	2D-Conv	64×64×8	32×32×16	2	16
8	2D-Conv	32×32×16	32×32×16	1	16
9	2D-Conv	32×32×16	32×32×16	1	16
10	2D-Conv	32×32×16	16×16×32	2	32
11	2D-Conv	16×16×32	16×16×32	1	32
12	2D-Conv	16×16×32	16×16×32	1	32
13	2D-Conv	16×16×32	8×8×64	2	64
14	2D-Conv	8×8×64	8×8×64	1	64
15	2D-Conv	8×8×64	8×8×64	1	64
16	2D-Conv	8×8×64	8×8×128	2	128
17	2D-Conv	8×8×128	8×8×128	1	128
18	2D-Conv	8×8×128	8×8×128	1	128
19	Fully Connected Layer	8×8×128	512×1	-	-
20	Fully Connected Layer	512×1	1024×1	-	-
21	2D-Conv Transpose	8×8×16	16×16×256	2	256
22	2D-Conv Transpose	16×16×256	16×16×256	1	256
23	2D-Conv Transpose	16×16×256	16×16×256	1	256
24	2D-Conv Transpose	16×16×256	32×32×128	2	128
25	2D-Conv Transpose	32×32×128	32×32×128	1	128
26	2D-Conv Transpose	32×32×128	32×32×128	1	128
27	2D-Conv Transpose	32×32×128	64×64×64	2	64
28	2D-Conv Transpose	64×64×64	64×64×64	1	64
29	2D-Conv Transpose	64×64×64	64×64×64	1	64
30	2D-Conv Transpose	64×64×64	128×128×32	2	32
31	2D-Conv Transpose	128×128×32	128×128×32	1	32
32	2D-Conv Transpose	128×128×32	128×128×32	1	32
33	2D-Conv Transpose	128×128×32	256×256×16	2	16
34	2D-Conv Transpose	256×256×16	256×256×16	1	16
35	2D-Conv Transpose	256×256×16	256×256×16	1	16
36	2D-Conv Transpose	256×256×16	256×256×3	2	3
37	2D-Conv Transpose	256×256×3	256×256×3	1	3
38	2D-Conv Transpose	256×256×3	256×256×3	1	3

Table A.3.: Detailed description of the convolutional auto-encoder layers, layers input and output sizes, stride, number of filter and operation type whether 2d-convolution or 2d-convolution transpose. Note that each layer is followed by a **Leaky-Relu** activation function and the kernel size used for each layer is constant equals to 3×3.

Reward fun, LSTM input	Train Success	Train Failures	Ratio	Val. success	Val. failed
r_{shaped}, Images only	0.522	0.478	1.090	50	50
r_{shaped}, Images and positions	0.605	0.395	1.533	72	28
r_{sparse}, Images only	0.078	0.922	0.085	6	94
r_{sparse}, Images and positions	0.121	0.881	0.14.	8	92
r_{shaped}, Images and positions (extended action)	0.603	0.397	1.522	63	37

Table A.4.: Average scores of all trained agents, during training and validation for 100 episode.

List of Figures

List of Tables

Bibliography

Agarwal Agarwal, M. Back propagation in convolutional neural networks-intuition and code. `https://becominghuman.ai/back-propagation-in-convolutional-neural-networks-intuition-and-code-714ef1c38199`. (Accessed on 03/08/2020).

2 Andrychowicz, M., Wolski, F., Ray, A., Schneider, J., Fong, R., Welinder, P., McGrew, B., Tobin, J., Abbeel, P., and Zaremba, W. (2017). Hindsight Experience Replay.

3 Baker, B., Kanitscheider, I., Markov, T., Wu, Y., Powell, G., McGrew, B., and Mordatch, I. (2019). Emergent Tool Use From Multi-Agent Autocurricula.

4 Basheer, I. and Hajmeer, M. (2001). Artificial Neural Networks: Fundamentals, Computing, Design, and Application. *Journal of microbiological methods*, 43:3–31.

5 Bengio, Y., Simard, P., and Frasconi, P. (1994). Learning Long-Term Dependencies with Gradient Descent is Difficult.

6 Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). OpenAI Gym.

7 Caruana, R., Lawrence, S., and Giles, C. L. (2001). Overfitting in Neural Nets: Backpropagation, Conjugate Gradient, and Early Stopping. In Leen, T. K., Dietterich, T. G., and Tresp, V., editors, *Advances in Neural Information Processing Systems 13*, pages 402–408. MIT Press.

8 Dumoulin, V. and Visin, F. (2016). A guide to convolution arithmetic for deep learning.

9 Feng, W., Guan, N., Li, Y., Zhang, X., and Luo, Z. (2017). Audio visual speech recognition with multimodal recurrent neural networks. pages 681–688.

10 Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. `http://www.deeplearningbook.org`.

11 Graves, A. (2008). Supervised Sequence Labelling with Recurrent Neural Networks. In *Studies in Computational Intelligence*.

12 Hausknecht, M. and Stone, P. (2015). Deep Recurrent Q-Learning for Partially Observable MDPs.

13 He, K., Zhang, X., Ren, S., and Sun, J. (2015). Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. In *The IEEE International Conference on Computer Vision (ICCV)*.

14 He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep Residual Learning for Image Recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

15 Hochreiter, S. (1991). Untersuchungen zu dynamischen neuronalen Netzen.

16 Jaderberg, M., Mnih, V., Czarnecki, W. M., Schaul, T., Leibo, J. Z., Silver, D., and Kavukcuoglu, K. (2016). Reinforcement Learning with Unsupervised Auxiliary Tasks.

17 Kingma, D. and Ba, J. (2014). Adam: A Method for Stochastic Optimization. *International Conference on Learning Representations*.

18 Kolve, E., Mottaghi, R., Han, W., VanderBilt, E., Weihs, L., Herrasti, A., Gordon, D., Zhu, Y., Gupta, A., and Farhadi, A. (2017). AI2-THOR: An Interactive 3D Environment for Visual AI. *arXiv*.

19 Kulhanek, J., Derner, E., de Bruin, T., and Babuska, R. (2019). Vision-based Navigation Using Deep Reinforcement Learning. *2019 European Conference on Mobile Robots (ECMR)*.

20 Le, T. P., Vien, N. A., and Chung, T. (2018). A Deep Hierarchical Reinforcement Learning Algorithm in Partially Observable Markov Decision Processes. *IEEE Access*, 6:49089–49102.

21 Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. (2015). Continuous control with deep reinforcement learning.

22 Mirowski, P., Pascanu, R., Viola, F., Soyer, H., Ballard, A. J., Banino, A., Denil, M., Goroshin, R., Sifre, L., Kavukcuoglu, K., Kumaran, D., and Hadsell, R. (2016). Learning to Navigate in Complex Environments.

23 Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533.

24 O'Shea, K. and Nash, R. (2015). An Introduction to Convolutional Neural Networks. *ArXiv e-prints*.

25 Pathak, D., Agrawal, P., Efros, A. A., and Darrell, T. (2017). Curiosity-Driven Exploration by Self-Supervised Prediction. *2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*.

26 Poole, D. and Mackworth, A. (2017). *Artificial Intelligence: Foundations of Computational Agents*. Cambridge University Press, Cambridge, UK, 2 edition.

Rumelhart et al. Rumelhart, D. E., Hinton, G. E., and Williams, R. J. Learning representations by back-propagating errors. *Nature*, (6088):533–536.

28 Shelhamer, E., Long, J., and Darrell, T. (2017). Fully Convolutional Networks for Semantic Segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(4):640–651.

29 Shi, W., Caballero, J., Theis, L., Huszar, F., Aitken, A., Ledig, C., and Wang, Z. (2016). Is the deconvolution layer the same as a convolutional layer?

30 Springenberg, J. T., Dosovitskiy, A., Brox, T., and Riedmiller, M. (2014). Striving for Simplicity: The All Convolutional Net.

31 Sutton, R. S. and Barto, A. G. (2018). *Reinforcement Learning: An Introduction*. The MIT Press, second edition.

32 van Hasselt, H., Guez, A., and Silver, D. (2015). Deep Reinforcement Learning with Double Q-learning.

33 Williams, R. J. and Zipser, D. (1995). Gradient-based learning algorithms for recurrent. *Backpropagation: Theory, architectures, and applications*, 433.

34 Yu, S. and Príncipe, J. C. (2019). Understanding autoencoders with information theoretic concepts. *Neural Networks*, 117:104–123.

35 Zhu, Y., Mottaghi, R., Kolve, E., Lim, J. J., Gupta, A., Fei-Fei, L., and Farhadi, A. (2017a). Target-driven visual navigation in indoor scenes using deep reinforcement learning. *2017 IEEE International Conference on Robotics and Automation (ICRA)*.

36 Zhu, Y., Mottaghi, R., Kolve, E., Lim, J. J., Gupta, A., Fei-Fei, L., and Farhadi, A. (2017b). Target-driven visual navigation in indoor scenes using deep reinforcement learning. *2017 IEEE International Conference on Robotics and Automation (ICRA)*.

List of Symbols

ξ_p	Propagation Function
W_i	Weight Matrix of i hidden layer
b_i	Bias vector of i hidden layer
θ	Neural network parameters, weights and biases
$L(.)$	Loss function
y^i_{true}	True values of the i neural network hidden layer
y^i_{pred}	Predicted values of the i neural network hidden layer
ω	Learning rate
$C(i,j)$	Discrete convolution operation for two dimensional matrix
$K(i,j)$	2-dimensional kernel of convolution layer
I	2-dimensional input matrix to convolutional layer
W_{ij}	Weight Matrix of components i and j
F	Depth dimension of the output feature map
P	Padding size
Str	Stride
O_{conv}	Spatial size of the feature map output volume of convolution layer
χ	Auto-encoder target output
χ'	Auto-encoder predicted output
$\frac{\partial L}{\partial x}$	Partial derivatives of L with respect to x
$\mathbb{E}[X]$	Expectation of a random variable X i.e. $\mathbb{E}[X] \doteq \sum_x p(x)x$
$\doteq$	Equality relationship that is true by definition
$arg\max_a f(a)$	a value of a at which $f(a)$ takes its maximal value
$W^\top_{ij}$	Transposed weight matrix of components i and j

t	Discrete time step
$T, T(t)$	Final time step of and episode, or of the episode including time step t
s, s'	Agent states
r	A reward
a	An action
$Pr\{X = x\}$	Probability that a random variable X takes on the value x
S_t	State at time t, typically due, stochastically, to S_{t-1} and A_{t-1}
R_t	Reward at time t, typically due, stochastically, to S_{t-1} and A_{t-1}
X_i	Input vector of component i
A_t	Action at time t
G_t	Return following time t
γ	Discount rate parameter
π	Policy (decision-making rule)
$v_\pi(s)$	Value of state s under policyπ (expected return)
$v_*(s)$	Value of state s under the optimal policy π
$q_\pi(s, a)$	Value of taking action a in state s under policy π
$q_*(s, a)$	Value of taking action a in state s under policy π
$\leftarrow$	Assignment
b	Bias vector
V, V_t	Array estimate of state-value function v_{pi} or v_*
Q, Q_t	Array estimate of action-value function q_{pi} or q_*
$(a, b]$	The real interval between a and b but not including a
$\in$	Is an element of; e.g., $s \in S, r \in R$
ε	Probability of taking a random action in ε-greedy policy
α	Step-size parameter
$d_{threshold}$	Distance threshold to goal, typically $d_{threshold} = 0.36$ meter
d_t	Distance between the agent current position at time step t and goal at time step

po_t Agent position vector at time step t

$po_t(x, z)$ Agent position vector at time step t of x and z components

f_{act} Differentiable activation function

po_g Goal position of episode

$r_{shaped,t}(s_t, a_t)$ Shaped reward function due to, s_t and a_t at time t

$r_{sparse,t}(s_t, a_t)$ Sparse reward function due to, s_t and a_t at time t

O_t Input frame to the Agent, typically a 2d RGB image

g goal of a given episode

g' Sampled additional goal

$\mathbb{G}$ Set of goals

$\mathbb{N}$ Natural numbers

$\mathbb{R}$ Real numbers

$\mu(O_t)$ latent feature vector of the agent input frame at time t

y_{out} Neural network final output

h Hidden state of the LSTM

$Q(o_{t+\tau}, a_{t+\tau}, h_{t+\tau-1}; \theta)$ Array estimate of action-value function for a sequence $t, ...t+\tau$ parameterized with θ

$\hat{\theta}$ Parameter of the Frozen target network

$\nabla_\theta L(\theta)$ Gradient of loss function of θ parameters

δ_j^t The RNN gradient of loss function L

ξ_i Propagation of i hidden neural network hidden layer

y_{i-1} Activation function output of i hidden neural network hidden layer

Versicherung an Eides Statt

Ich versichere an Eides Statt durch meine Unterschrift, dass ich die vorstehende Arbeit selbständig und ohne fremde Hilfe angefertigt und alle Stellen, die ich wörtlich oder annähernd wörtlich aus Veröffentlichungen übernommen habe, als solche kenntlich gemacht habe, mich auch keiner anderen als der angegebenen Literatur oder sonstiger Hilfsmittel bedient habe. Ich versichere an Eides Statt, dass ich die vorgenannten Angaben nach bestem Wissen und Gewissen gemacht habe und dass die Angaben der Wahrheit entsprechen und ich nichts verschwiegen habe. Die Strafbarkeit einer falschen eidesstattlichen Versicherung ist mir bekannt, namentlich die Strafandrohung gemäß § 156 StGB bis zu drei Jahren Freiheitsstrafe oder Geldstrafe bei vorsätzlicher Begehung der Tat bzw. gemäß § 163 StGB bis zu einem Jahr Freiheitsstrafe oder Geldstrafe bei fahrlässiger Begehung.

Aachen, 16.04.2020	 Mohamed Nagi, 3006823
Ort, Datum	Unterschrift