HELLENIC REPUBLIC
National and Kapodistrian
University of Athens
EST. 1837

Department of Electronic Physics and Systems

Master Program in Control and Computing

Training an agent to move towards a target interacting with a complex environment

by

Michaila Goula - Dimitriou

Supervisors: Dr. Nikolaos Vlassopoulos
Prof. Dionysios Reisis
Dr. Konstantinos Nakos

Registration Number: 2018503

March, 2022 | Athens

Table of Contents

Table of Contents 2

Περίληψη 4

Abstract 5

Table of Figures 6

Acronyms 7

1 Introduction 8

1.1 Motivation 8

1.2 Structure of the document 9

2 Reinforcement Learning 10

2.1 Machine Learning 10

2.2 The perceptron 11

2.3 Artificial Neural Networks 12

2.4 Gradient Based Training 12

2.4.1 Backpropagation 13

2.4.2 Deep Learning 14

2.4.3 Activation Function 16

3 Deep Reinforcement Learning 18

3.1 Markov Decision processes 18

3.1.1 States 20

3.1.2 Actions 20

3.1.3 The transition function 20

3.1.4 The reward function 21

3.1.5 Policies 21

3.2 Recurrent networks 22

3.2.1 Standard Recurrent Cell 23

3.2.2 Long Short-Term Memory 24
3.2.3 Gated Recurrent Unit 26
3.3 Convolutional Neural Networks 26
3.4 Agents and algorithms 27
3.4.1 Offline and Online Learning Methods 28
3.4.2 Actor-critic 28
3.4.3 Policy Gradient Methods 30
3.4.4 Trust Region Policy Optimization 30
3.4.5 Proximal Policy Optimization 31
3.5 Exploration vs exploitation 32
4 Implementation 33
4.1 Initial Environment created 33
4.1.1 OpenAI Gym 33
4.1.2 CoppeliaSim 34
4.1.3 PyRep 35
4.2 Environment used 36
4.2.1 Box2D 38
4.2.2 PyGame 38
4.2.3 Tensorforce 38
5 Results 40
6 Conclusion 45
References 46

Περίληψη

Στη σημερινή εποχή, το πρόβλημα της αυτόνομης πλοήγησης στα σύγχρονα κινητά ρομπότ αποτελεί σημείο ενδιαφέροντος για την πλειοψηφία της έρευνας που γίνεται γύρω από τη ρομποτική. Αυτό το θέμα γίνεται ακόμα πιο απαιτητικό, καθώς οι απαιτήσεις στα δυναμικά περιβάλλοντα περιλαμβάνουν αυτονομία υψηλού επιπέδου και ευέλικτες δυνατότητες λήψης αποφάσεων για το ρομπότ, ώστε να επιτευχθεί αποφυγής συγκρούσεων. Το Deep learning κατάφερε να λύσει κάποια κοινά ζητήματα στη ρομποτική, όπως η λήψη αποφάσεων, η πλοήγηση και ο έλεγχος, όμως, με εποπτευόμενο τρόπο.

Οι Reinforcement learning τεχνολογίες έχουν συνδυαστεί με το Deep learning, με αποτέλεσμα ένα νέο ερευνητικό θέμα γνωστό ως deep reinforcement learning (DRL). Με τη χρήση του DRL, η διαδικασία μπορεί να αυτοματοποιηθεί με τη μετάφραση δεδομένων αισθητήρων πολλών διαστάσεων σε εντολές κίνησης ρομπότ χωρίς τη χρήση κεντρικοποιημένων πληροφοριών, παρέχοντας έναν μη εποπτευόμενο τρόπο. Αυτό που χρειάζεται, για να ενθαρρυνθεί ο agent μάθησης και μέσω διαδικασίας δοκιμής και σφάλματος με το περιβάλλον, να βρει την καλύτερη δράση για κάθε κατάσταση, είναι μία βαθμωτή συνάρτηση ανταμοιβής. Στην εν λόγω διατριβή, δημιουργήθηκε ένα προσομοιωμένο περιβάλλον με ένα κινητό ρομπότ που αλληλεπιδρά με αυτό. Δύο αλγόριθμοι βασισμένοι στο DRL, οι Actor-Critic και PPO, χρησιμοποιήθηκαν για να εκπαιδεύσουν τον παράγοντα να κινείται με ασφάλεια στο περιβάλλον, αποφεύγοντας τα εμπόδια και στοχεύοντας στην επίτευξη ενός καθορισμένου στόχου. Τα αποτελέσματά τους παρουσιάζονται και συγκρίνονται.

Λέξεις κλειδιά: αυτόνομη πλοήγηση σε κινητά ρομπότ, Reinforcement Learning, Actor-Critic, PPO

Abstract

Nowadays, the problem of autonomous navigation in modern mobile robots is the point of interest for the majority of research in robotics. This topic becomes even more challenging as the requirements in dynamic environments include high-level autonomy and flexible decision-making capabilities for the robot, to achieve collision avoidance. Deep learning has succeeded in solving some common issues in robotics, such as decision making, navigation and control, in a supervised manner though.

Reinforcement learning frameworks have been combined with deep learning, resulting in a new research topic known as deep reinforcement learning (DRL). With the use of DRL the procedure can become automated by mapping high-dimensional sensory data to robot motion commands without using ground-truth information, providing an unsupervised manner. It simply takes a scalar reward function to encourage the learning agent through trial-and-error interactions with the environment, with the goal of finding the best action for each state.

In the project thesis in question, a simulated environment was created with a mobile robot interacting with it. Two DRL-based algorithms, Actor-Critic and PPO were used to train the agent to move safely in the environment, avoiding the obstacles and aiming to reach a specified goal. Their results are presented and compared.

Keywords: mobile robot, Reinforcement Learning, Actor-Critic, PPO

Table of Figures

Figure 1: Representation of the perceptron [4] 12
Figure 2: Example of a neural network with hidden layers [8] 16
Figure 3: Activation functions 17
Figure 4: Typical reinforcement learning cycle 20
Figure 5: Recurrent and Feed-forward neural network representation 23
Figure 6: Original LSTM architecture [14] 25
Figure 7: Actor-critic architecture 29
Figure 8: Visual representation of the initial environment in CoppeliaSim 35
Figure 9: Visual representation of the final environment. 37
Figure 10: Episode length with network lstm(64)/dropout(0.1)/dense(32) and agent PPO. 42
Figure 11: Episode return with network lstm(64)/dropout(0.1)/dense(32) and agent PPO. 42
Figure 12: Episode length with network dense(32)/gru(64)/dense(16) and agent PPO. 43
Figure 13: Episode return with network dense(32)/gru(64)/dense(16) and agent PPO. 43
Figure 14: Episode length dense(128)/gru(64)/dense(64) and agent PPO. 43
Figure 15: Episode return dense(128)/gru(64)/dense(64) and agent PPO. 44

Acronyms

AC	Actor-Critic
AI	Artificial Intelligence
RL	Reinforcement Learning
RNN	Recurrent networks
LSTM	Long Short-Term Memory
ANN	Artificial Neural Network
CNN	Convolutional Neural Networks
GRU	Gated Recurrent Unit
ReLU	Rectified Linear unit
TRPO	Trusted Region Policy Optimization
MM	Minorize Maximization algorithm
PPO	Proximal Policy Optimization
MDP	Markov Decision Process
SDL	Simple DirectMedia Layer

1 Introduction

1.1 Motivation

Artificial Intelligence (AI) is used in a wide range of domains, including robots, computer vision, and gaming, and has sparked widespread interest across all fields, particularly in hazardous applications. Traditional fixed industrial robots have a place in a lot of industries, whereas mobility robots can move around in their working environment. Mobility offers greater flexibility in a wide range of industrial applications, such as warehouse transportation and distribution.

In their working environment, industrial mobile robots should have autonomous navigation capability, which strives to identify a collision-free path from a starting point to a goal point [1], such as when transferring products from one location to another in a warehouse. When a complete and thorough knowledge of the environment is needed, global path planning approaches can be used. However, local path planning strategies that rely on sensory input from mobile robots, such as fuzzy logic control, potential field method, and genetic algorithms, have effectively proven adequate in performing the goal of obstacle avoidance while robots are working in unknown settings.

Classic path planning approaches, on the other hand, have the constraint that the control strategy must be well-designed by programmers. When the environment changes or the robot meets problems not anticipated by the creators, the robot may not react and instead follow the predetermined coping technique. This could result in harmful or even tragic outcomes, such as a collision or a crash.

In this approach, such a robot remains a type of machine that flawlessly does what it is programmed to do but lacks adaptability. However, what someone would anticipate is a truly intelligent autonomous robotic system. As a result, self-learning mobile robots have become a prominent study area. Robot learning is usually accomplished through interaction between the robot and its surroundings. Reinforcement learning is a machine learning technique, based on trial-and-error mechanisms, that improves performance by receiving input from the environment.

Having all these in mind, the idea for this thesis was born. The main goal was to study the concepts of the reinforcement learning field and investigate one of the most interesting navigation problems, the obstacle avoidance, through its methods. In this work, an effort was made to conclude whether a deep reinforcement learning solution will help improve a robot's behavior when it moves around a random environment, trying to reach a certain position. The ability to trigger the obstacle avoidance maneuver within a particular distance, known as the triggering distance, is a challenge for current systems, and this became a motivation for engaging with this project.

1.2 Structure of the document

This work is structured into 6 chapters. Each chapter contains key issues and algorithms used to develop this work, as well as the methods developed in this work. More specifically, in the first chapter an introduction about the motivation and scope of this thesis is presented. In the second chapter, reinforcement learning, machine learning, and Artificial Intelligence methods and algorithms are analyzed.

The third chapter includes all methods used in Deep Reinforcement Learning and concern this implementation, while the fourth chapter presents the implementation details. Chapter five is a summarization of the outcomes resulting from the evaluation of the system. Finally, the last chapter includes a conclusion, presenting a summary of the results presented in this thesis.

2 Reinforcement Learning

Learning by interacting with the environment is a fundamental idea with respect to nearly all theories regarding learning. Reinforcement learning is often perceived as a modern control method but has old underlying ideas frequently found in nature, where difficult control problems are demonstrated with ease. Insects and birds are able to exploit turbulent unsteady aerodynamic flow effects to increase efficiency when flying. A gazelle calf is struggling to its feet minutes after it is born, yet is running at 30 kilometers per hour half an hour later. [2]

Such examples from nature, alongside countless more, have guided the development of reinforcement learning. The agent, analogous to the controller in traditional control theory, is not told which actions to take, but must by trial-and-error discover which actions that generate the best reward. An interesting remark is that the immediate reward received from an action may not result in the best cumulative reward. Reinforcement learning algorithms can therefore be perceived as optimization algorithms. As is the case with most other optimization problems, there is a question of whether the solution found is a local or a global solution. Should the agent find a local solution, it can be difficult to exit from the local solution and continue searching for the optimal solution. Different reinforcement learning methods have different approaches to this issue.

2.1 Machine Learning

Machine learning is the field of computer science where one tries to fit statistical models to a set of data. The goal is then for the computer to be able to make good predictions or actions given a sample set of data, without explicitly programming the computer how to do so. There are three main directions in the field of machine learning; supervised learning, unsupervised learning and reinforcement learning [4]. All of the methods usually use a large amount of gathered data for learning and make predictions and decisions based on them.

- **Supervised learning**:
  In supervised learning the data that is used for training is labeled, meaning human supervision is necessary. This means that for every data tuple, there exists a label with the true association on it. When training a machine learning algorithm, this boils down

to trying to fit a statistical model to the dataset in such a way that one can do inference on a random sample, and obtain the true answer.

- **Unsupervised learning:**
  In this self-organized subset of machine learning, the goal is to spot trends in the data without explicitly knowing what to look for. In this case, the data is not labeled, in contrast to the supervised setting. Now, the computer is trying to discover unknown patterns and connections in the data, that otherwise could not be seen by the naked eye. This field is dominated by clustering algorithms that find clusters of data with similar characteristics and classifies them accordingly.
- **Reinforcement Learning:**
  In reinforcement learning (RL), there are no known labels in the same way as in supervised learning, but scalar evaluative rewards that guide the agent. The agent then learns by interacting with the environment on its own.

The big difference here is that the data that the statistical models are trained on, have to be generated by the agent itself, and that the training labels are continually updated. This means that both the distribution of training data and the distribution of labels for training are changing simultaneously.

This concludes in a much harder problem to solve in comparison to the supervised learning setting, where the distribution of training data and labels was fixed. Therefore, there is increased difficulty in reaching convergence in a reinforcement learning problem, and much care has to be taken to not end up in local maxima.

2.2 The perceptron

To begin the explanation of artificial neural networks, it is worthwhile to take a look at the perceptron. This is a binary classifier that, given the input data, will either activate an output y or deactivate it. It works by doing a weighted sum of the input data and adding a constant bias and then inputting everything into an activation function φ as shown in Figure 1. The output y can thus be expressed as:

$$y = \varphi(x_1\omega_1 + x_2\omega_2 + ... + x_m\omega_m + b) \tag{1}$$

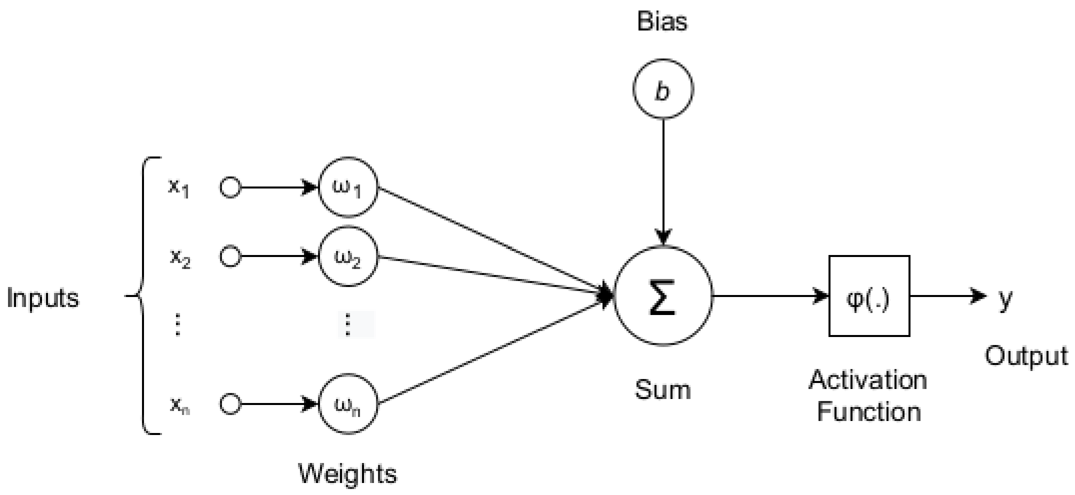


Figure 1: Representation of the perceptron [4]

In the case of the perceptron, this activation function is just a step function that is either true or false.[4]

2.3 Artificial Neural Networks

If several perceptrons are chained together into a fully-connected network and change its activation function, the result is an Artificial Neural Network (ANN) [5]. Each node in this network is now called a neuron. The activation function itself can be changed according to the task at hand (examples are ReLU, leaky ReLU, Sigmoid, Tanh, etc.). This type of classifier is able to learn more complicated classification problems than binary classification, as it easily can handle non-linearities in the input data. Depending on its size, it is also very suitable for finding features in the data without explicitly being told what they are, and thus can do robust inference based on these features.

2.4 Gradient Based Training

The training of an ANN is all about adjusting its weights and biases. For each neuron we have a set of weights for every input, and a bias. The goal now is to be able to accurately predict an output, given an input. One of the major training algorithms used for ANNs is called Stochastic Gradient Descent. Stochastic because there is a random sampling of a batch of data to find a gradient to a loss function. The loss function can be defined as following [6]:

$$L(\vec{\omega},\vec{b}) = \sum_{i=1}^{n}(\hat{y}_i - y_i)^2 \tag{2}$$

where $\hat{y}_i$ is the estimated output given by the network, and y_i is the true label of the input data. Notice that the number of outputs on this network is decided by the designer, and may as well be just one. In such a case, n = 1 and the summation in the above equation disappears. These outputs may be doing either some form of regression, for example predicting the price of a certain item, or classification, for example predicting what type of flower the input data suggests. Either way, the network learns in the same manner. It can be easily seen that this loss function is small whenever the predicted $\hat{y}_i$ is close or equal to the real y_i. Therefore the intention is to do a gradient descent on this loss function, meaning to find the gradient of the function at each point, and do a step in the opposite direction to minimize the loss function.

An important concept in the training of neural networks is overfitting. This is when a neural network is so perfectly optimized for a training dataset that it has basically begun to learn it by heart, instead of actually generalizing and seeing the bigger trends. The symptom of overfitting is when the validation loss starts to climb, instead of decreasing like it is wanted to. The validation loss is a measure of the performance of a neural network on a piece of the data set that it has not been trained for, and it is a better metric of its true performance than the training loss which indicates the performance on its training dataset.

2.4.1 Backpropagation

In order for the loss function to be minimized there is the concept of backpropagation [7]. Focusing on a single output $\hat{y}_1$ and its corresponding true value y_1, the magnitude of the gradient of this output is proportional to how far $\hat{y}_1$ is from the true value y_1, and the direction of the gradient indicates if $\hat{y}_1$ needs to be increased or decreased. Now, knowing how much this output should change and the direction of the change, there can be an observation of the previous layer. The weights, activations and bias that cause this activation in $\hat{y}_1$ need to be changed so that $\hat{y}_1$ either increases or decreases depending on the gradient direction. These weights are then changed proportionally to their corresponding activations, and the amount of change of each previous neuron is noted. Now, this process repeats, as there is knowledge of how much and in which direction the activation of the neuron should change. Thus, a recursive pattern that propagates gradients backwards throughout the network's neurons, weights and biases, can be seen.

When all these gradients have been propagated throughout the network, the negative gradient can be written as:

$$-\eta\nabla L(\omega,b)=[\nabla\omega_1,\nabla\omega_2,...\nabla\omega_m,\nabla b_1,\nabla b_2,...,\nabla b_t] \quad (3)$$

where η is a proportionality constant. Now, having found the gradient of the above function, a gradient descent step by changing the weights and biases in the direction of their negative gradient can be done.

A challenge with deep networks is that the more layers the network has, the smaller the magnitude of these gradients gets. This means that the first layers will learn slower than the last layers. This is mentioned as the vanishing of gradients and is the reason why big neural networks require longer training times.

2.4.2 Deep Learning

Classic machine learning algorithms use techniques like decision trees, linear regression, random forests, support vector machines, and artificial neural networks in order to learn a predictive model on some data. The models' purpose is to generalize, to be able to make predictions. From a mathematical point of view, machine learning's goal is to approximate a function from data.

In the past, when computers didn't have the speed they do today, neural networks that consisted of much fewer layers of fully connected neurons were used, and as a result, they did not perform exceptionally well on difficult problems. This situation totally changed with the introduction of deep learning and the evolution of computers. Now there are deep neural networks that consist of many neuron layers and use different types of connections between them.

When the width of each layer and the number of layers of a neural network increase, it is common to use the term "deep learning". This term denotes the network's ability to learn complex features and trends that shallower networks are unable to capture, and has led to many important advances in the field of computer science like computer vision. Deep learning and deep networks have enabled machine learning to be applied to and solve high-dimensional problems, like recognizing objects in high-resolution images in real-time, and at the same time allowing it to achieve greater accuracy on important tasks.

It is easy to assume that the deeper the network, the more observant it is, and thus the better. However, this is not always the case. Apart from the problem of vanishing gradients and the fact that larger networks require much more computation, there is also a trade-off between generalization and overparameterization. The best size of a neural network is the smallest one where it can still accurately detect and learn the level of complexity in the features of the dataset. As deep ANNs are considered to be black boxes, meaning it is not possible to understand, with any confidence, what happens inside it, choosing the size of a network is more an art than a science.

Deep learning is based on a function f: $X \rightarrow Y$ parameterized with $\theta \in R^{n}{}_{\theta}$ ($n_\theta \in N$) [8]:

$$y = f\,(x;\,\theta) \tag{4}$$

A deep neural network is defined by a series of many processing layers that follow one another. Each layer consists of a non-linear transformation, and the order in which these transformations are performed concludes to the learning of various levels of abstraction.

Given the example of a very simple neural network with only one fully-connected hidden layer, the first layer receives the input values x as a column vector of size n_x ($n_x \in N$). The values of the next layer, which is the hidden one, are a transformation of these input features by a non-linear parametric function. This function is a matrix multiplication by W_1 of size $n_h \times n_x$ ($n_h \in N$), plus a bias term b_1 of size n_h, followed by a non-linear transformation, with A being the activation function:

$$h = A(W_1 \bullet x + b_1) \tag{5}$$

This non-linear activation function is the one that provides the expressivity of the neural network by making the transformation non-linear at each level. The hidden layer h, having size n_h, can in turn be transformed into other sets of values until the last transformation which finally gives the output values y. For this example:

$$y = (W_2 \bullet h + b_2) \tag{6}$$

with W_2 being of size $n_y \times n_h$ and b_2 of size n_y ($n_y \in N$)

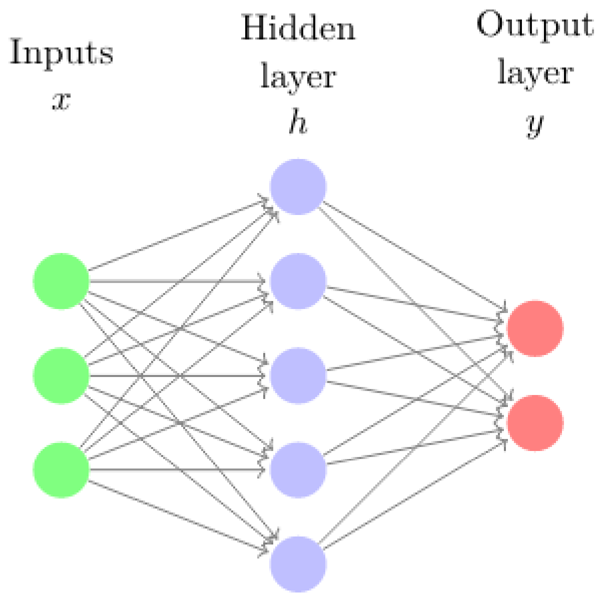


Figure 2: Example of a neural network with hidden layers [8]

Each of these layers has been specifically trained to reduce the empirical error IS[f]. Gradient descent using the backpropagation algorithm is the most frequent approach for optimizing the parameters of a neural network.

In the simplest scenario, the algorithm alters its internal parameters in each iteration to fit the intended function:

$$\theta \leftarrow \theta - a\nabla \theta IS[f] \tag{7}$$

with α being the learning rate.

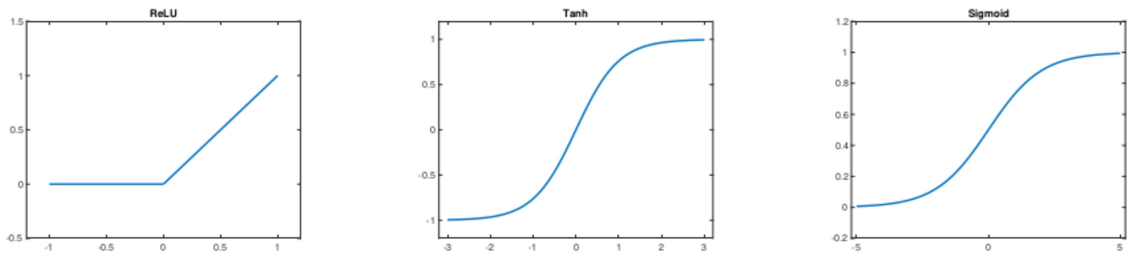
Beyond the simple feedforward networks already mentioned above, several new types of neural network layers have appeared in modern applications. Depending on the application, each type offers distinct benefits.

Furthermore, an arbitrarily high number of layers can be contained within a single neural network, with the intention in recent years being to have an ever-increasing number of layers, with certain supervised learning tasks to requiring more than 100.

2.4.3 Activation Function

As mentioned before, every neuron in a neural network is composed of a weight vector, a bias value and an underlying activation function. The network consists of an input layer, multiple hidden layers and an output layer. The number of neurons, sometimes referred to as nodes, within these layers varies. The input layer usually consists of the observed states, while the output layer often is the action(s) to be done. The hidden layers are incrementally built as the agent interacts with the environment. The information that flows between the layers needs to be converted such that the output in one neuron can be taken as input in another neuron. This is where the activation function is relevant. The most important feature of an activation function is its ability to add non-linearity in a neural network. Each neuron takes the previous neuron's linear output and generates a non-linear output based on this linear input. In this manner, the system is capable of handling nonlinear data.

Many different activation functions exist. The rectified linear unit (ReLU), tanh and sigmoid, which can be seen in Figure 3 are some of the most recognized activation functions. The tanh and sigmoid activation functions are in many ways similar, especially since both increase mostly around x = 0, but the tanh allows negative values also. The sigmoid function is mostly presented for historical purposes and is rarely used in modern applications, as it is computationally expensive, not zero-centered and tends to shift the gradients towards zero. This last property is referred to as the vanishing gradient problem and is due to the network's depth and the activation shifting the value to zero. The tanh activation function solves the zero-centered issue, but is relatively computationally expensive and suffers from the vanishing gradient problem as well. ReLU is easy to compute and does not cause vanishing gradient problems, therefore it is widely used as an activation function. However, since the output is zero for all negative inputs, it can cause neurons to die and not learn anything. Another issue is that it never saturates, sometimes leading to unusable neurons. Several other activation functions have been introduced to solve these issues, such as leaky ReLU [9], parametric exponential linear unit [10] and Hard-Swish [11].



(a) ReLU (b) Tanh (c) Sigmoid

Figure 3: Activation functions

3 Deep Reinforcement Learning

3.1 Markov Decision processes

The representation describing dynamic behavior in a system, meaning the random motion of an object among a discrete set of possible locations, formalized through a transition matrix can be considered as a Markov chain. In each node of the Markov chain, the possibility of transitioning into another node or staying in the same node is denoted by a probability. The probability of the different transitions in each node must sum to one.

There are two types of such models, the discrete-time and the continuous-time model. In the discrete, the state of the Markov chain, namely the position of the object is recorded at every unit of time, as opposed to the continuous, where the state is monitored at all times. The state of the Markov chain changes randomly. Every time, the future motion depends only on the current location, regardless of all the previous.

From this, a Markov Decision Process (MDP) [12] can be defined. MDPs are stochastic, sequential decision processes where the cost and the transition functions depend only on the current state and action of the system. A sequential decision process is a model for dynamic system under the control of a decision maker. Whenever a decision is to be made, the decision maker observes the system state. This observation's derived information guides him to choose one action from a set of available alternatives.

When the dynamic system is in state s at time t there are two consequences of choosing an action. One is that the decision maker receives an immediate reward and the other is that he determines the probability distribution for the state of the system at the next stage. When the reward is positive it can be considered as income and when it is negative as cost.

The decision maker has the objective to choose a sequence of actions, named a policy, that will optimize the system's performance over the horizon of the decision making. The decision maker should take into consideration the future consequences before making his choice for the next action, since the action selected at present has an impact on the future evolution of the system.

The classification of the sequential decision processes is made according to the length of the decision making horizon, the epochs when the decisions are made, the mathematical properties of the state space and the action space and the optimality criteria.

The main goals of the analysis of the sequential decision processes in general and particularly in the MDPs are to provide an optimality equation that describes the supremal value of the objective function, to characterize the form of an optimal policy if one exists and to develop efficient computational procedures to be used to find optimal or close to optimal policies.

The Bellman equation is the fundamental, in MDP theory, entity and is the basis for practically all existence, characterization, and computational results.

The optimality criteria, as well as the nature of the states, the actions, the rewards, and the transition probability functions, determine the form of this equation.

Almost any problem can have a sufficiently general class of policies to yield an optimal or substantially optimal approach. The biggest question, both theoretically and practically, is whether there is an optimal or nearly optimal policy in a specified class, among the class of all other policies, and under which circumstances this happens.

In many applications, if such a policy exists, it is important to focus on whether it has a special form or structure and restrict the search to policies of this form in order to find the optimal ones.

All time points at which the system is observed and decisions can be made are called decision epochs or stages. These epochs can be classified in two ways, being either finite or infinite and either a discrete set or a continuum.

Calculations for finite horizon problems are based on backward induction, which is also known as dynamic programming, whereas calculations for infinite horizon problems are based on value iteration, policy iteration, and their variants.

A decision rule is a function that denotes the action the decision maker will choose with the system being in state s at time t. It is said to be history dependent if it is such a function that summarizes the sequence of previous states and actions of the system. On the other hand, it is referred to as randomized if it specifies a probability distribution on the total of allowable actions, in each state and in that case the action chosen is a random event.

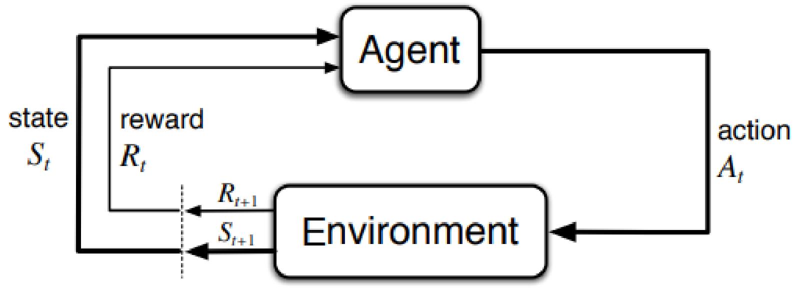


Figure 4: Typical reinforcement learning cycle

A policy denotes the decision rule that the decision maker will use at each time. It is a sequence of decision rules that guides the decision maker to choose a decision under any possible future state that the system might be in.

Having that said, the basic concepts that characterize the Markov decision process are ready to be defined. As almost every Reinforcement Learning problem is theoretically modeled as maximizing the return in a Markov Decision Process these concepts are substantial parts of reinforcement learning, too.

3.1.1 States

The set of states S in the environment is the finite sets $s_1, s_2, ..., s_n$, with the corresponding size of the state space being |S| = N. Each state contains a unique description of everything that matters in a state of the given modeled problem. States can either be legal or illegal, with legal states being the ones the agent can explore, like an empty space, and illegal states being the ones the agent is not able to explore, like the inside of a wall.

3.1.2 Actions

The set of actions A in the environment is the finite set $a_1, a_2, ..., a_k$, where the size of the action space is denoted |A| = K. The set of actions that can be executed in a given state s ∈ S is denoted A(s), where A(s) ⊆ A. An action can be used by the agent to control the system state. An important side note is that not all actions can necessarily be applied in every state.

3.1.3 The transition function

When applying an action a ∈ A in a given state s ∈ S, the system makes a transition from the state s to a new state s_0. This transition is based on a probability distribution over the set of possible transitions from the original state. The transition function T is defined as follows: T:

SxAxS −−> [0, 1]. This symbolizes the probability of ending up in the state s0 after performing an action a in state s, and is denoted T (s, a, s0).

This transition function must fulfill the following two conditions.

1. The probability of a given transition T (s, a, s_0) must be in the set [0, 1]
2. The sum of transition probabilities in each state must sum to one

A very important property of markovian dynamics is that the current state gives sufficient information about the past to make an optimal decision in the current state. In essence, the information of the past history is lumped into the previous state [2]. This is called the Markov Property, and this assumption is a fundamental building block in all methods discussed in this project thesis.

3.1.4 The reward function

The state reward function is defined as: R : S × A × A → R , and often denoted R(s, a, s_0). This reward function is what the agent will use in the learning process to determine what actions to take, and thus it implicitly specifies the goal of the learning. It is up to the designer of the reward function to determine in which way the system, or rather the MDP, should be controlled.

This all wraps up into the definition of a Markov Decision Process. It is defined as a tuple <S, A, T, R> where S is a finite set of states, A is a finite set of actions, T a transition function and reward function R as specified earlier. The transition matrix T and the reward function R constitute the model of the MDP.

3.1.5 Policies

Given a Markov Decision Process <S, A, T, R>, a policy π is a function that maps states into actions, π : S → A. The policy interacts with the MDP in the following way:

1. First, an initial state s0 is generated from the initial state distribution
2. An action $a_0 = \pi(s_0)$ is then performed as decided by the policy π
3. Based on the models for T and R, a transition is made to the next state, s_1 with probability T(s_0, a_0, s_1) and reward R(s_0, a_0, s_1)
4. The process above continues and produces the tuples <s_0, a_0, r_0>, <s_1, a_1, r_1> and so on
5. If the case is an episodic setting, the process ends when the system reaches a predetermined state s goal.

3.2 Recurrent networks

There are numerous different types of deep learning architectures, depending on input data of different types and characteristics. Such architectures are convolutional neural networks (CNN) and recurrent neural networks (RNN). In most cases, CNNs and DNNs are unable to deal with temporal information of input data.

RNNs are easily expandable and can recall critical details about the input they receive thanks to their internal memory, allowing them to precisely predict what will happen next. They can, for example, use data from previously occurred events to provide a wide range of sequence-to-sequence mappings and incorporate numerous types of information, including temporal order. As a result, RNNs are popular in study domains containing sequential data, such as time series, voice, text, financial data, audio, video, and weather. In general, RNNs are a type of neural network that is particularly well adapted to sequential input and, when compared to other algorithms, can create a far deeper grasp of a sequence and its context [13].

RNNs, which are derived from feedforward networks, behave similarly to human brains. They are divided into two categories: discrete-time RNNs and continuous-time RNNs. A cyclic connection is a typical feature of the RNN architecture, which allows the RNN to update its current state based on previous states and current input data. Two important aspects in the understanding of RNNs are the feed-forward neural networks and the sequential data (Figure 5).

- Sequential data is simply ordered data in which related items appear one after the other. Financial data or the DNA sequence are two examples, but probably the most typical type of sequential data is time series data, which is just a chronologically ordered list of data points.
  RNN's and feed-forward neural networks' names imply the way that they channel information. The information, in a feed-forward neural network, only flows in one direction, from the input layer to the output layer, passing through the hidden layers. The data travels in a straight way through the network, never passing from the same node twice.
- Feed-forward neural networks have no recollection of the information they receive and are poor predictors of what will happen next. A feed-forward network has no concept of time order because it only analyzes the current input. Except for its training, it has no memory of what previously occurred. More specifically, in

feedforward networks, history is limited as in N-gram backoff models and represented by the context of N-1 words, as opposed to RNNs that have unlimited length history, represented by neurons recurrently connected with each other.

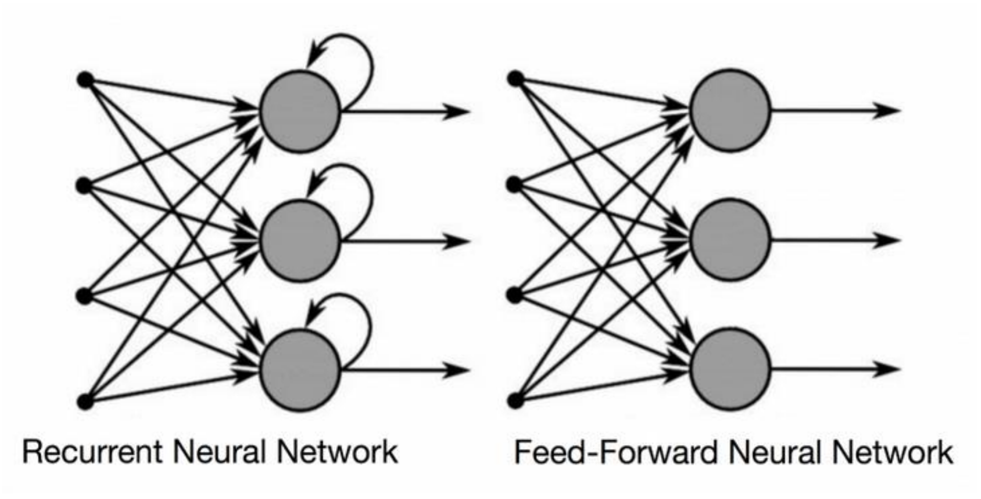


Figure 5: Recurrent and Feed-forward neural network representation

In addition, recurrent networks can learn to compress entire histories in low-dimensional space, whereas feedforward networks can only compress a single word.

RNNs usually have a short-term memory. However, variations with long-term memory also exist. The information in an RNN travels through a loop. It takes into consideration the current input as well as what it has learnt from prior inputs, before it makes a decision. As a result, there are two inputs to an RNN, the present and the immediate past. This is significant because the data sequence provides critical information about what will happen next and is the reason why an RNN can perform tasks that other algorithms cannot.

Like all other deep learning algorithms, a feed-forward neural network applies a weight matrix to its inputs before producing the output. RNNs apply weights to both the current and prior inputs.

In addition, a recurrent neural network will adjust the weights overtime via gradient descent and backpropagation (BPTT).

Finally, RNNs may map one to many, many to many, for translation, and many to one, when classifying a voice, whereas feed-forward neural networks map one input to one output, only.

3.2.1 Standard Recurrent Cell

RNNs typically consist of standard recurrent cells like sigma and tanh cells.

In some problems, standard recurrent cells have shown some successful results. However, recurrent networks made out of standard recurrent cells are incapable of addressing long-term dependencies: learning connection information becomes difficult as the interval between related inputs widens.

The root cause of long-term dependencies problem is that error signals that travel backward in time tend to explode or vanish.

3.2.2 Long Short-Term Memory

Unfortunately, when there is a significant gap between the relevant input data, the RNNs mentioned above are incapable of connecting the important information. The long short-term memory (LSTM) concept was introduced to deal with the long-term dependencies. Simple variants, however, have a restricted amount of input variables that can be efficiently handled.

Gated architectures are designed to address this constraint by incorporating gating units that are trained to control information flow through the network and, as a result, learn to keep information for a long time. In real-world applications, both Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) networks have shown benefits. [14]

However, gated RNNs were not created to engage with the recommendation domain. Specifically, they are not designed to take into account the interaction between the user and the system.

The recurrent layers, also known as hidden layers in RNNs, are made up of recurrent cells whose states are influenced by both previous states and current input via feedback connections. The recurrent layers can be arranged in a variety of ways to create distinct RNNs. RNNs are distinguished primarily by the network design and the recurrent cells. There can be RNNs of different capacities depending on variations in cells and inner connections.

In order to eliminate the problem of long-term dependencies, the LSTM cells were introduced. They boosted the standard recurrent cell's remembering capability by putting a "gate" into the cell. Many researchers have updated and popularized LSTMs since this groundbreaking study. LSTM without a forget gate, LSTM with a forget gate, and LSTM with a peephole connection are some variations, but when referring to LSTM cells, LSTM cells with a forget gate are implied.

Long short-term memory networks (LSTMs) are an extension of recurrent neural networks that provide extended memory. As a result, it is highly suited to learning from significant experiences separated by lengthy periods of time. The layers of an RNN are built using LSTM as the building blocks. LSTMs assign weights to data, allowing RNNs to either let new information in, forget it, or give it enough relevance to affect the output.

LSTMs provide RNNs the ability to recall inputs over a long period of time, due to the fact that they store information in a memory, similar to that of a computer. The LSTM has the ability to read, write, and delete data from its memory. This memory can be thought of as a gated cell, with the definition of gated indicating that the cell selects whether or not to store or erase information, i.e., whether or not to open the gates, based on how important it considers the data to be. Weights, which are also learned by the algorithm, are used to justify the importance of the data. This basically implies that, over the time, it learns which information is important and which is not.

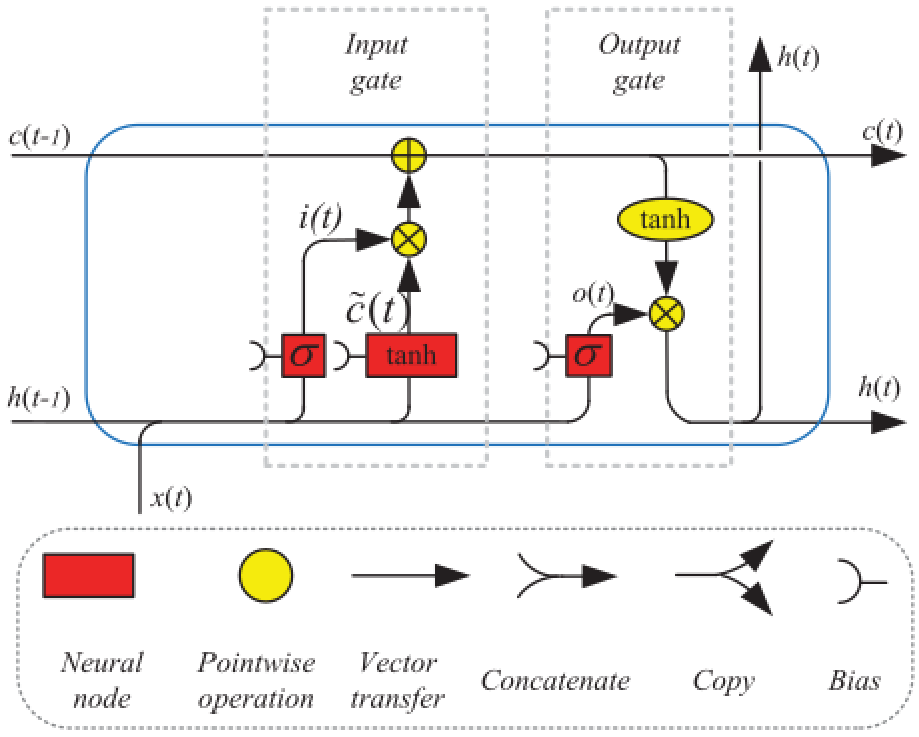


Figure 6: Original LSTM architecture [14]

There are three gates in an LSTM, the input, forget, and output. These gates determine whether fresh input should be allowed (input gate), whether it should be deleted because it is not important (forget gate), or whether it should have an impact on the output at the current timestep (output gate).

An LSTM's gates are analog in the form of sigmoids, which means they range from zero to one, and thus they can perform backpropagation.

The problem of disappearing gradients, where the model stops learning or it takes far too long to learn due to very small gradient values, is overcome by LSTM, as it keeps the gradients steep enough, resulting in a quick training period and good accuracy.

3.2.3 Gated Recurrent Unit

The LSTM cell's learning capacity is superior to that of a conventional recurrent cell. The additional parameters, on the other hand, add to the computing load. This resulted in the introduction of the gated recurrent unit (GRU) [14].

The GRU cell integrates the forget gate and input gate of the LSTM cell as a single, update gate so as to reduce the number of parameters. There are only two gates in the GRU cell, an update gate and a reset gate. Hence, it could save one gating signal and the associated parameters. The GRU is just a forget gated version of a vanilla LSTM.

The single GRU cell lacks power compared to the original LSTM since one gate is missing. After empirical evaluation of the performance of the LSTM network, the GRU network and the traditional tanh-RNN, it was found that the LSTM, as far as the GRU cell outperformed the classic tanh unit when both networks had roughly the same amount of parameters.

3.3 Convolutional Neural Networks

Over the last decade, Convolutional Neural Networks have achieved breakthroughs in many pattern recognition domains, ranging from image processing to speech recognition. Convolutional Neural Networks (CNNs) [15] are similar to classic Artificial Neural Networks (ANNs) in that they are made up of neurons that learn to optimize themselves. Each neuron will still receive an input and conduct an action, which is the foundation of innumerable artificial neural networks.

The most advantageous feature of CNNs is that they reduce the number of parameters in ANN. This accomplishment has motivated both researchers and developers to consider larger models in order to perform complex tasks that were previously impossible to solve with traditional ANNs.

The most significant assumption about issues solved by CNN is that they should not contain spatially dependent properties. To put it another way, in a face detection problem, the position of the faces in the image should not be a concern. The only thing that matters is that they be detected, regardless of where they are placed in the photos.

Another essential property of CNN is the ability to extract abstract features as input propagates through deeper levels. The entire network will still express a single perceptual scoring function, the weight, starting from the input raw picture vectors and ending to the final output of the class score. The last layer will contain the loss functions related to the classes, and all of the standard ANN features will still apply.

One of the most significant changes is that the layers within the CNN are made up of neurons that are organized in three dimensions, the input's spatial dimensionality (height and width) and depth. The depth is referred to the third dimension of an activation volume and not the total number of layers within the ANN.

Unlike traditional ANNs, the neurons in each layer only connect to a small portion of the layer before it.

A convolutional neural network would have been suitable if the approach of the complex environment was followed, as the image input would have played a significant role. The CNN would have processed the image gathered from the state to help train the agent to move safely in the room and also recognize the target.

3.4 Agents and algorithms

The best possible future reward can be computed in several ways. Simulation is often favored because it allows not having a predefined mathematical model. Learning without a predefined mathematical model is a major idea that has made reinforcement learning stretch beyond its classical borders. One of several methods to obtain the optimal future reward is by parametric cost approximation. In this method, the optimal future reward is chosen to be a member of a parametric class of functions where the parameters are optimized by a given algorithm or even by a neural network. The reward can, for instance, be given based on an incremental least square algorithm. Actor-critic methods, or actor-critic inspired methods, such as TRPO and PPO, fall under this category.

3.4.1 Offline and Online Learning Methods

It is common to classify reinforcement learning algorithms as offline or online methods. In the approximation of value space, one could compute the cost-to-go and the suboptimal control policy offline, meaning before the control process begins, or online, meaning after the control process begins. This design choice can be significant for the system. With online learning methods, the system will learn "on the go" and calculate the cost-to-go just after the current state is known. If the system encounters a never-before-experienced situation, the network will improve its knowledge by learning how to handle the new situation online. Online learning is also popularly called incremental learning and has a typical depth-first search structure. PPO and TRPO are online algorithms. In an offline scheme, the model is trained with all possible situations of the environment. The entirety of the cost-to-go function is calculated at every time step before the control process begins in a breadth-first search structure. The weights depend on the whole set of data. In this method, Actor-critic algorithms are widely used.

3.4.2 Actor-critic

There are two main types of RL methods, the value based and the policy based method. In value based method an attempt is made to discover or approximate the optimal value function, being a mapping between the action and the value. In this category, the most well-known algorithm is the Q learning and all the algorithms created to improve it. On the other hand, policy based algorithms, such as Policy Gradients and REINFORCE, attempt to determine the optimal policy without Q-value interfering. Policy based algorithms have faster convergence and are more suitable for continuous and stochastic environments, whereas value based are steady and more sample efficient. By combining these two types of methods and trying to keep all their advantages and, at the same time, eliminate any drawbacks, the Actor-critic methods were established. [16]

The algorithm that receives the state as input and produces the best action is the actor. It effectively directs the agent's behavior by learning the best policy and it represents the policy-based part. On the other hand, the critic calculates the value function to evaluate the action taken and as a result, it is the value-based part. The two models are in a process where they both improve in their respective roles as time passes. Consequently, the combination of the two methods resulted in better training of the whole architecture and proved more efficient than any of the two methods individually.

The actor can be an approximator function, such as a neural network, a fully connected one or a convolutional one, having the task of producing the best action for a given state. The critic is also an approximator function with two inputs, the environment's state and the actor's chosen action. It concatenates them and produces the action value (Q-value) for the pair created.

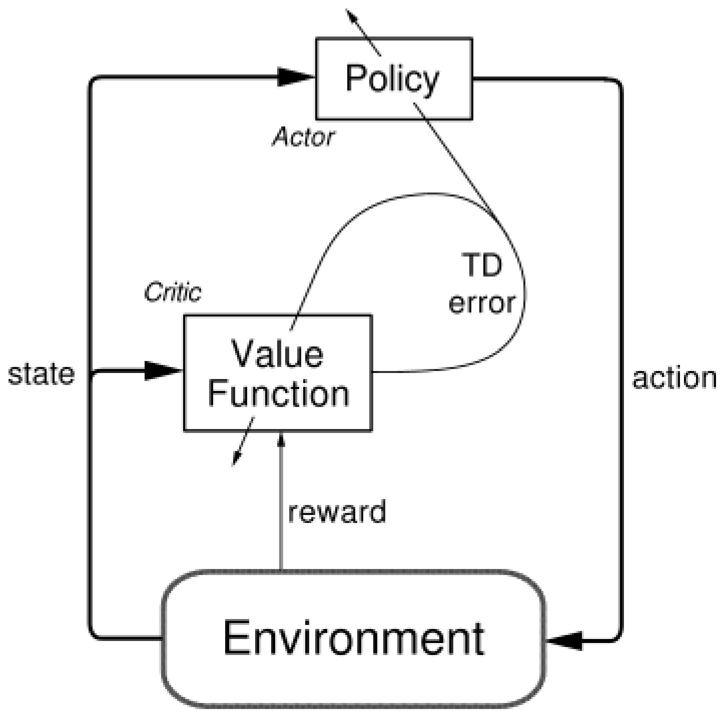


Figure 7: Actor-critic architecture

A policy is a mapping between states and actions and can be thought of as the agent's "strategy" to reach the goal. A policy iteration network generates a sequence of stationary policies and a corresponding sequence of approximate cost functions that are evaluated during the simulation of the system and is possible to determine the next improved policy. Where the critic's task is to evaluate the current policy, the actor's task is to actually improve the policy.

The two networks are trained individually, and their weights are updated using gradient ascent, to obtain the global maximum.

The actor-critic scheme, when using neural networks, will gradually learn better policies by observing behavior. However, an important note to make is that even though the system learns by itself, it is dependent on external guidance - the predefined reward function. The system learns to behave better by improving the policy through experiences gained when interacting with the environment. It's also worth noting that, unlike policy gradients, the weights are updated at each step rather than at the end of the episode.

3.4.3 Policy Gradient Methods

Policy gradient learning methods [17] are often advantageous in robotics and other continuous space problems. They optimize a parametrized control policy by maximizing the expected reward, usually by using direct gradient descent or some other optimization algorithm. The policy gradient method differs from the value function approximation method, which derives policies indirectly from an estimated value function. Policy gradient methods prevail because they enable the incorporation of domain knowledge. They can also naturally handle continuous states and actions, as well as imperfect state information.

Many policy gradient methods guarantee convergence to a locally optimal point and often have a more compact optimal policy than the corresponding value function. However, there are certain disadvantages, particularly in terms of the timestep. Too small timestep and the method will be slow, too large and it will be affected by noise and diverge or not reach a stable solution.

However, the major problem is that the policy gradient, while often guaranteeing a local optimum, will struggle to identify local optima that are not globally optimal. PPO and TRPO are all methods that use ideas inspired by policy gradient methods.

3.4.4 Trust Region Policy Optimization

The policy gradient methods have been fundamental to the development and breakthrough of deep reinforcement learning for control. However, getting good and reliable results has been difficult due to learning sensitivity caused by the stepsize. If the step size is too small, the system will be too slow. If it is too large, the system will be significantly affected by noise and may yield unwanted results. There are two major optimization methods, line search (similar to gradient ascent) and trust region. In line-search, the algorithm makes the directional decision first based on the steepest ascent, then takes the step in that direction. When using trust region, the maximum allowed step size, referred to as the trust region, is first determined. When the trust region is chosen, the optimal point to move within the trust region is chosen.

Gradient-based optimization algorithms enjoy much better sample complexity guarantees than gradient-free methods. Continuous gradient-based methods have been successful at learning function approximators with huge numbers of parameters. However, their success has often been associated with supervised learning. With the trusted region policy optimization (TRPO) algorithm [18], this was extending the field into the reinforcement learning area. This allowed for more efficient training of more powerful and complex policies. TRPO guarantees policy improvement with non-trivial step sizes by minimizing a surrogate objective function. To understand the TRPO algorithm, there are three core concepts to be aware of:

1. The Minorize Maximization algorithm (MM): An algorithm that guarantees that any policy updates always improve the expected rewards by iteratively maximizing a lower bound function. This will eventually lead to a local or a global optimal policy.
2. Trust Region: The maximum step size to explore the environment is first chosen, then the optimal location to move, within the radius of the trust region, is found.
3. Importance sampling: Calculates the expected value of a function given a data distribution. This is important because it makes it possible to rewrite the objective and use samples from an old policy to calculate a new policy.

TRPO can guarantee monotonic improvement for policy optimization. In practice, this means that every step in the algorithm will have a better policy than the previous step. However, this guarantee is only valid within the trust region.

3.4.5 Proximal Policy Optimization

Proximal Policy Optimization (PPO) [19], like the TRPO, is an online policy gradient method. However, it is significantly easier to implement, it has better sample complexity, and is easier to tune. PPO was motivated by the desire to design an algorithm that achieves TRPO's data efficiency and reliability, however, it provides only first-order optimization. Instead of a KL-constraint, PPO employs an adaptable penalty (adaptive KL-penalty). The algorithm calculates the update that optimizes the reward while keeping the deviation from the previous policy to a minimum at each step.

PPO alternates between applying a stochastic gradient ascent to optimize a surrogate objective function and sampling data from the environment. The policies are optimized somewhere between sampling data from the policy and running various optimization epochs on the sampled data. A unique objective with clipped probability ratios is used to find the lower bound. The Hessian calculation and its derivative are the major objectives that PPO addresses differently than TRPO. Because this is a costly operation, neither algorithm directly solves it. TRPO reduces the computing complexity by approximating the hessian. PPO, on the other hand, brings the first order derivative closer to the Hessian by using soft constraints. This is done at the expense of ensuring policy improvement at every step; however, because of the soft constraints, the chances of making a faulty decision are reduced.

3.5 Exploration vs exploitation

At any point in time, an agent has always partial knowledge about the state, actions, rewards and upcoming states in a given environment. However, its actions serve two purposes at the same time, the purpose of exploring (learn) and a means to exploit (optimize).

Because exploration is intrinsically expensive in terms of resources, time, and opportunity, the reconciliation of the conflict between uncharted territory exploration and existing knowledge exploitation is a natural and important challenge in RL problems. This dilemma arises in both stateless RL contexts and more general multi-state RL situations.

In particular, the agent must strike a balance between greedy exploitation of the knowledge he has already gained, to choose actions that offer bigger rewards in the short term, and continuous exploration of the environment, to get additional information in order to potentially attain long-term gains. Thus, the two basic terms, exploration and exploitation can be defined as follows:

- **Exploration:**
  Exploration is more of a long-term benefit idea in which the agent improves its knowledge of each action, potentially leading to long-term benefit.
- **Exploitation:**
  Exploitation is the concept where the agent exploits the current estimated value and in order to maximize the reward, it chooses the greedy approach. However, in this case, the agent is being greedy with the estimated value rather than the actual value, resulting in the possibility of not getting the highest reward.

An example of a concept that intends to achieve a balance between exploration and exploitation is the e-greedy method. It chooses the currently best action a in a state s with probability 1-e and a completely random action with probability e. The value for e can, thus, be tuned to lead to this balance.

4 Implementation

4.1 Initial Environment created

In the initial implementation, a more complex yet realistic idea was in mind for the design of the environment. In this idea, a three-dimensional environment was designed for the robot to move in. It simulated a room, with walls around, some cuboid obstacles and a human body, each placed in a random position for every episode. The robot's starting position was also random and its goal was to approach the human in the room, avoiding the obstacles it might come across on its way. The robot was enhanced with five proximity sensors able to detect objects within a detection volume and a vision sensor, being a camera-type sensor, reacting to light, colors and images.

In this version, the state of the environment each time consisted of two parts. The one part contained the sensors' readings, the polar coordinates and the linear and angular velocities and the other the image input of the robot's camera.

The environment in question was designed using three significant, in the robotics field, tools, OpenAI gym, CoppeliaSim and PyRep.

4.1.1 OpenAI Gym

OpenAI Gym [20] is a toolbox that enables development and comparison of reinforcement learning algorithms. It provides simulated environments to be used for RL algorithms training and as benchmarks to demonstrate the usefulness of any new research methodology. This variety of environments ranges from simple games to large physics-based engines.

A wide range of environments that are used as benchmarks for proving the efficacy of any new research methodology is implemented in OpenAI Gym, out-of-the-box. Except for the predefined environments, OpenAI gym provides an easy way for custom environment creation to fulfill all the needs.

In action, OpenAI gym gives access to an “agent” that can perform actions on an “environment”, building an environment-agent arrangement. The return, as in the consequence of a particular action performed in the environment, is an observation and a reward.

For each "step" taken by the agent, the environment returns four different values.

1. **Observation (object):** an environment-related object that represents the observation of the environment, in this case, the two-part state of the environment.
2. **Reward (float):** the amount of reward/score earned as a result of the preceding action. Although the scale varies depending on the environment, the intention is always to improve the total reward/score.
3. **Done (boolean):** indicates that it is time for the environment to be reset. In this case, done is true if the robot hits an obstacle or reaches the goal.
4. **Info (dict):** diagnostic data that can make debugging easier.

4.1.2 CoppeliaSim

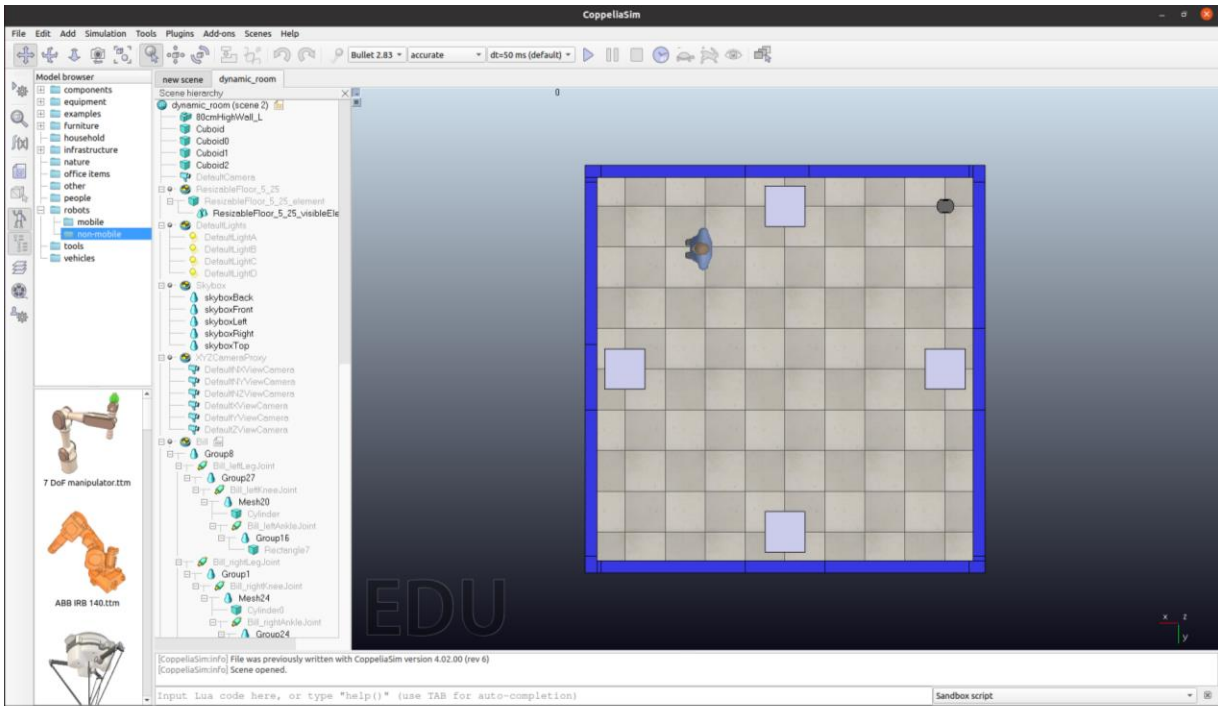
CoppeliaSim [21] is a robotic simulator with a built-in development environment that has been utilized in industry, education, and research. It uses a distributed control architecture, which means that each object/model can be controlled separately via an embedded script, a plugin, a ROS node, a remote API client, or a custom solution. As a result, CoppeliaSim is extremely adaptable and well-suited to multi-robot applications.

CoppeliaSim provides rigid body simulation by using a kinematics engine for forward and inverse kinematics calculations, as well as many physics simulation libraries, like Bullet, ODE, Vortex and Newton Game Dynamics. Models and scenarios are created by putting together a hierarchical structure of numerous items (meshes, joints, various sensors, Point clouds, OC trees, and so on). Motion planning (through OMPL), synthetic vision and imaging processing (e.g. via OpenCV), collision detection, minimum distance calculation, specialized graphical user interfaces, and data visualization are all features available by plug-ins. Among other things, CoppeliaSim is used for fast algorithm development, factory automation simulations, rapid prototyping and verification, robotics teaching, remote monitoring, safety double-checking, and as a digital twin.

CoppeliaSim can be used as a standalone application or easily integrated into a major client program. It is incredibly versatile thanks to the Lua or Python script interpreter, which allows the user to combine low/high-level functionalities to create new high-level functionalities.

CoppeliaSim was used to create the scene of the environment. As shown in the image below, the scene composition is visualized in a scene hierarchy view, indicating object aliases, types, selection and visibility states, associated control scripts, loop closures, selection and visibility states etc. This structure made it easy to drag and drop the composing elements of the scene, place them on the floor available on scene creation, select them, change their position in all dimensions, affect their visibility and make them detectable and collidable. In addition, the robot was also designed via this tool, by adding easily its visual and ultrasonic sensors, adjusting its camera angle etc.

The tool was also used for watching the simulation with the robot moving in the environment created, among the training episodes.



```
[CoppeliaSim:info] File was previously written with CoppeliaSim version 4.02.00 (rev 6)
[CoppeliaSim:info] Scene opened.
Input Lua code here, or type "help()" (use TAB for auto-completion)
```

Figure 8: Visual representation of the initial environment in CoppeliaSim

4.1.3 PyRep

PyRep [22] is a robot learning research toolbox built on top of CoppeliaSim/V-REP. It enables Python to launch CoppeliaSim's simulation thread. As a result, PyRep python code can be executed synchronously with the simulation loop. This considerably boosts speed as compared to a Python client interacting with CoppeliaSim via remote procedure calls. These characteristics make PyRep especially relevant to learning algorithms in the fields of reinforcement learning, imitation learning, state estimation, and so on.

The PyRep API keeps track of simulation handles and provides an object-oriented interface to the simulation environment. Furthermore, it provides simple addition of new robots with motion planning capabilities, by writing only a few lines of Python code.

The use of the PyRep library provided a simple way to control the robot and manipulate the scene. It built the connection between the visual representation of the elements and the actual access in their attributes through the Python code, so as the distances, the velocities and the positions could be available and used to format the state and compute the reward each time.

4.2 Environment used

The above environment added a lot of complexity and computational load to the project and due to lack of resources, it could not be used to complete the training of an agent and yet to fulfill the purposes of the implementation.

The final environment created, the one where the two types of agents, Actor-critic and PPO were configured and tested, is much simplified compared to the initial one. Two libraries were used, Box2D, to create the elements of the environment and implement the physics and PyGame to visualize the simulation.

It is, too, represented as a room with size 16x16 meters with walls around, of 30cm thickness. In each episode, eight obstacles are generated with random sizes and positions, along with a target of size 10x10cm, also being placed in random position.

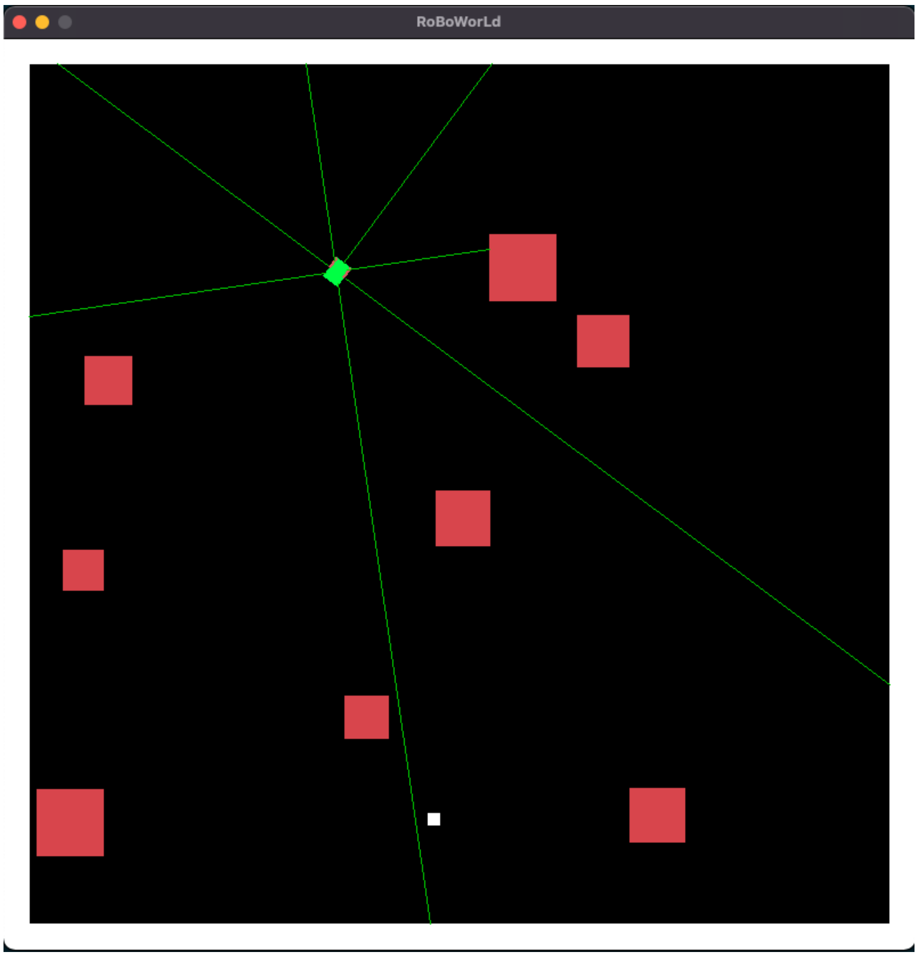


Figure 9: Visual representation of the final environment.

The robot that moves around the room, trying to reach the target, is 30x40cm and has two wheels of 10cm length and 2mm thickness. To make the robot move, it was necessary to use physics, in order to create a realistic result. To start its movement, a boost is given to the wheels in such a way that it can simulate an engine's operation. Due to lack of gravity, a force of the robot's linear velocity multiplied by -5 had to be applied, so that it could brake quickly. Furthermore, in order to avoid spinning when braking, an impulse of the robot's inertia multiplied by its angular velocity, multiplied by -1 had to be also applied.

In order to detect the collision of the robot with an obstacle, the ray cast concept of Box2D library was used. Ray casting is frequently used to determine what items are present in a given location. A ray is simply a straight line created from a specified start and end point. Box2D traces the line from start to finish and returns all its fixtures that it collides with. In this case, the collision threshold is set to be 2cm and the target threshold 1cm.

4.2.1 Box2D

Box2D [23] is a game-specific 2D rigid body simulation package. It can be used by programmers to make objects move in realistic ways and make the game world more interactive in their games. A physics engine is just a system for generative procedural animation, as seen from the perspective of the game engine. Box2D is written in the portable C++ programming language.

It can imitate convex polygons, circles, and edge forms as bodies. Bodies are connected by joints and influenced by forces. Gravity, friction, and restitution are also applied by the engine.

An incremental sweep and prune broad phase, a robust linear-time contact solver and a continuous collision detection unit make up Box2D's collision detection and resolution system. These techniques make it possible to simulate rapid bodies and big stacks efficiently, without missing collisions or triggering instabilities.

4.2.2 PyGame

Pygame [24] is a set of cross-platform Python tools for video games creation. It provides sound and graphics libraries that can be utilized with the Python programming language. It makes use of the Simple DirectMedia Layer (SDL) library, with the goal of facilitating real-time computer game production without the low-level mechanics of C and its variants. This is founded on the idea that the most expensive functions in games may be abstracted from the game logic, allowing the game to be structured using a high-level programming language like Python. Vector math, collision detection, 2D sprite scene graph management, MIDI support, camera, pixel-array manipulation, transformations, filtering, advanced freetype font support, and drawing are among SDL's other features.

The two types of agents used for the training process, Actor-Critic and PPO, along with the RNN network architectures, were provided from the Tensorforce library.

4.2.3 Tensorforce

TensorForce [25] is an open-source deep reinforcement learning framework that aims to provide flexible modularized library design and facilitate application both in research and in practice. It is built on top of the TensorFlow library and requires Python 3 for utilizing this deep RL framework.

Tensorforce attempts to keep its feature implementations as flexible and feasible as possible. An important feature is that it allows parallel execution of multiple RL environments. It also provides separation between the application and the RL algorithm, meaning that the algorithms that use the library are agnostic both to the states, the actions and the interaction with the application's environment. It supports a wide variety of neural network layers like fully connected, 1 or 2D convolutions, pooling etc, optimization algorithms, L2 and entropy regularization techniques, random replay memory and batch buffer memory. Finally, the whole reinforcement learning logic, including control flow, is implemented in TensorFlow, allowing for portable computation graphs independent of the application programming language and facilitating model deployment, also by supporting TensorBoard.

5 Results

In this implementation, several tests were performed with different reward approaches, different network architectures and different number of episodes among the runs, for both agent types, Actor-Critic and PPO. The basic experiments conducted to train the agent contained the following reward computation functions:

The agent gets:

1. *fixed penalty*: when he hits on an obstacle or wall, when he moves close to obstacles or walls and when he gets further away from the target

   reward: when he reaches the target and when he moves close to the target

2. *fixed penalty*: when he hits on an obstacle or wall, when he moves close to obstacles or walls and for each time step he is "alive" and moving, to pressure him and give him motivation to move faster towards the target

   variable penalty: proportional to the distance he has from the target, meaning less when he is close to the target and more when he moves away from it

   reward: when he reaches the target and when he moves close to the target

3. *fixed penalty*: when he hits on an obstacle or wall

   reward: when he reaches the target

4. same as case 3, with a huge negative reward on hitting an obstacle

The above reward function cases were used with several network architectures that combined both dense and LSTM layers and the two agents, Actor-Critic and PPO mentioned in an earlier chapter.

The first reward case was tested with two different network architectures with different type of agent each, a dense(32)/lstm(16)/dense(32) with AC and a dense(32)/lstm(128)/dense(32) with AC and dropout layers p=10%. The first one needed to stop early because it was not giving any good results. The second one, with the large LSTM, has satisfying results, however it needed to run with many more episodes for the LSTM to be fully exploited.

Regarding the third case of reward, its calculation is very simple, denoting positive on success and negative on failure. It was tested with a PPO agent and an lstm(4)/dense(64) network. This case is all about exploration, meaning that the agent has no other way to understand how he could maximize his reward, but moving around the environment and eventually crash on an obstacle or reach the goal, and finally realize that his action could have some impact. In such a scenario, many episodes are required for the agent to pass the exploration barrier, discover the target and start moving towards it. However, despite the fact that limited episodes were run, their results were satisfactory.

Approximately, the same situation occurred also with the use of the second reward approach. This approach was combined with two different network architectures, an lstm(128)/dense(64)/dense(32) with PPO agent and a dense(64)/lstm(64)/dense(32) with AC. The most promising between them was the one with the PPO agent, which found a good solution, but tended to “quit early”, meaning that the robot kept leading to obstacles in order to quit the episode and in that way minimize the negative reward. The one with the AC oscillates between good behavior, with the robot searching the environment for the target and bad behavior, with it crashing really quickly. Both approaches need to be run for more episodes for the exploration to kick in.

Some early tests, without the use of LSTM, performed poorly, with the agent unable to learn. In these tests the most common results were for the agent to keep spinning around himself eternally or keep crashing with ease on the closest obstacle without taking time to move around the environment. On the other hand, the tests that were performed with LSTM networks achieved to train the agent to move and explore the environment, and successfully avoid the obstacles.

To sum up, both of the actor types used, had similar results, with the network architectures and the reward selections being the aspects that brought the differences in the training efficiency. Architectures with LSTM appear to have small variations. Besides the runs with a small number of steps per episode, that did not have the required time to infer on the reward, the differences appeared mostly on how fast they will reach ascending slopes.

By using the Tensorforce library there was a limitation on the control over the LSTM. In a more custom implementation, where there could be parametrization of the LSTM, the differences in the results and the overall training could have been substantial.

Indicatively, there are some graphic representations cited, showing the progress of the training of the agent. When the episode length increases, it indicates that the agent manages to stay “alive” longer, without hitting an obstacle. In addition, the episode return shows that the agent learns, either going through several stages, where it “remembers” and “forgets” (Figure 11), or having a permanent upward course (Figure 15), where it improves in each episode.

Since there was no direct control over the structure of RNN via Tensorforce, a manual increase of the neural network’s input was made, to include prior states, thus increasing both agent return and episode length, which means that the agent learns better. The result is presented in Figure 15.

In all cases, the simulation needed more episodes to run in order to have better results and a more clear learning state of the agent.

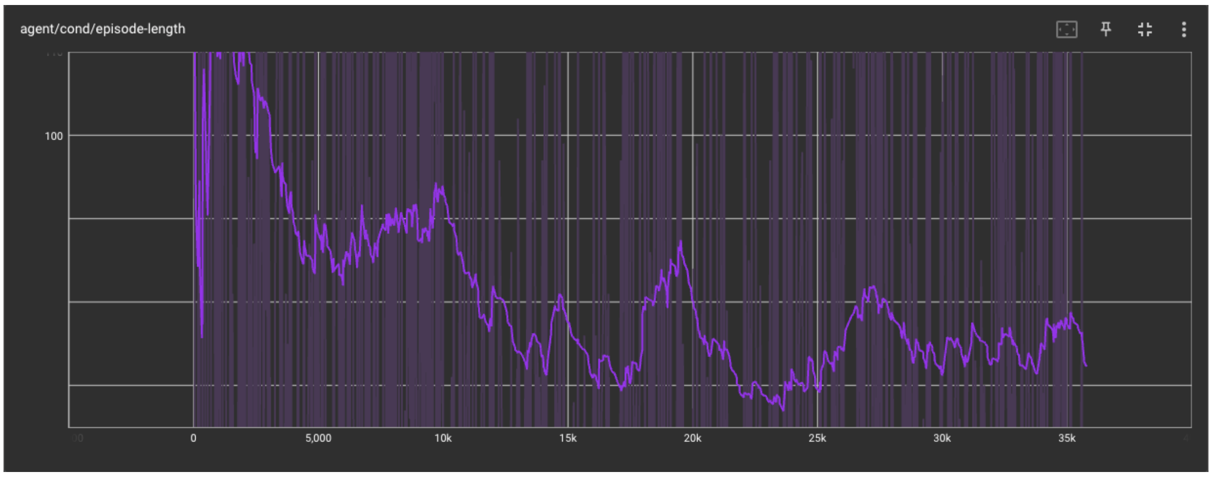


Figure 10: Episode length with network lstm(64)/dropout(0.1)/dense(32) and agent PPO.

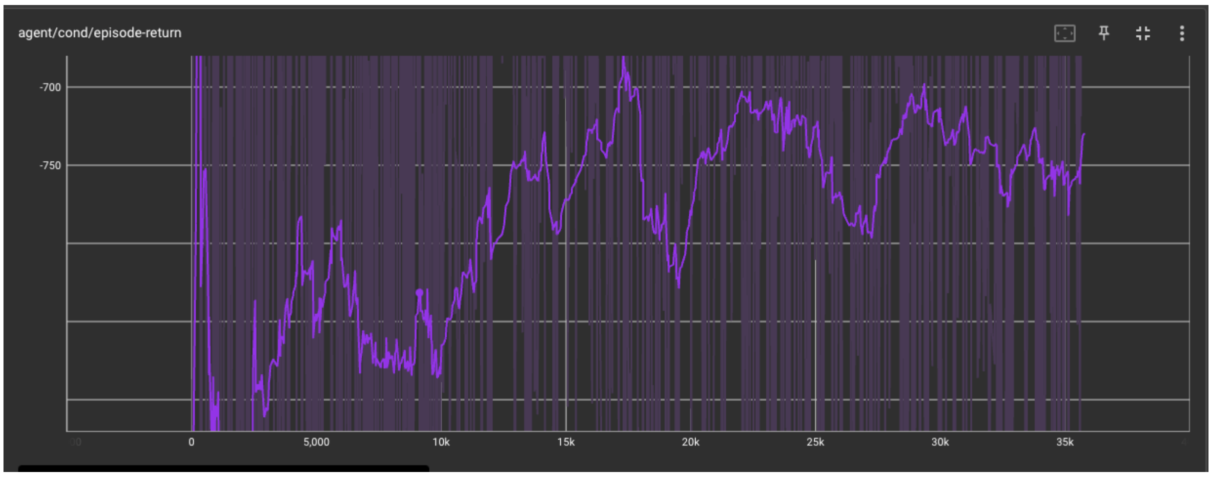


Figure 11: Episode return with network lstm(64)/dropout(0.1)/dense(32) and agent PPO.

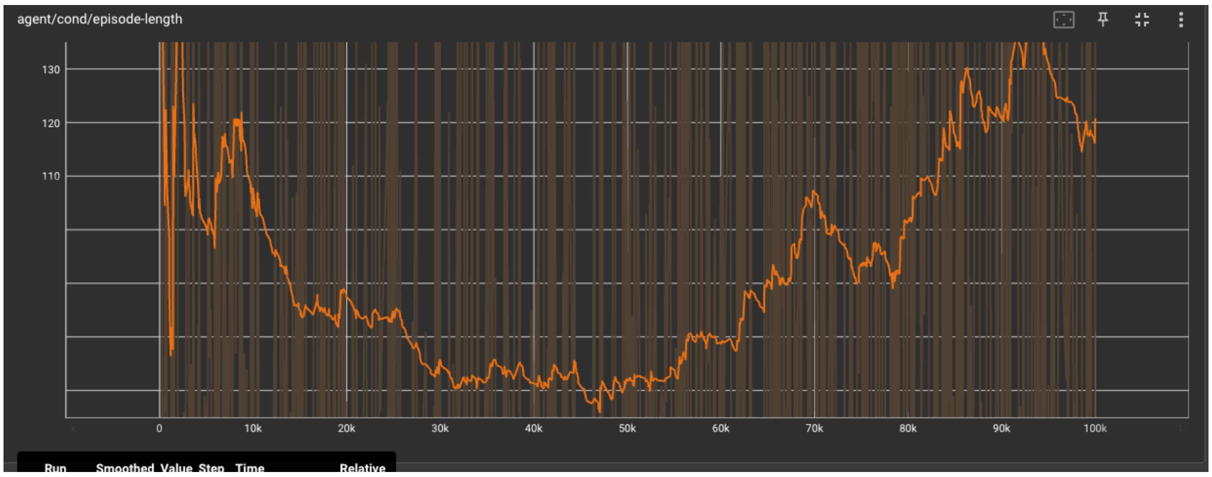


Figure 12: Episode length with network dense(32)/gru(64)/dense(16) and agent PPO.

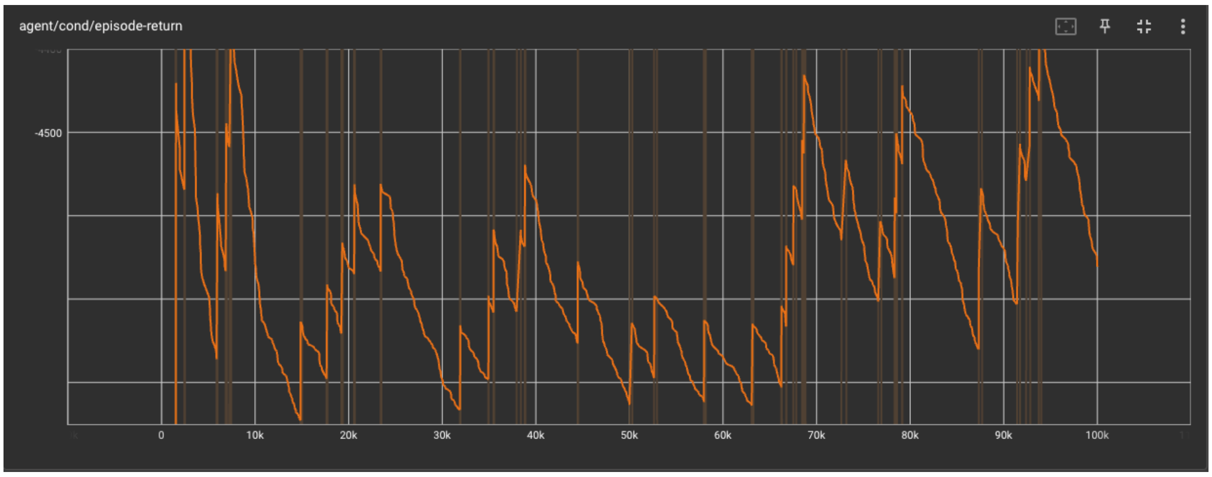


Figure 13: Episode return with network dense(32)/gru(64)/dense(16) and agent PPO.

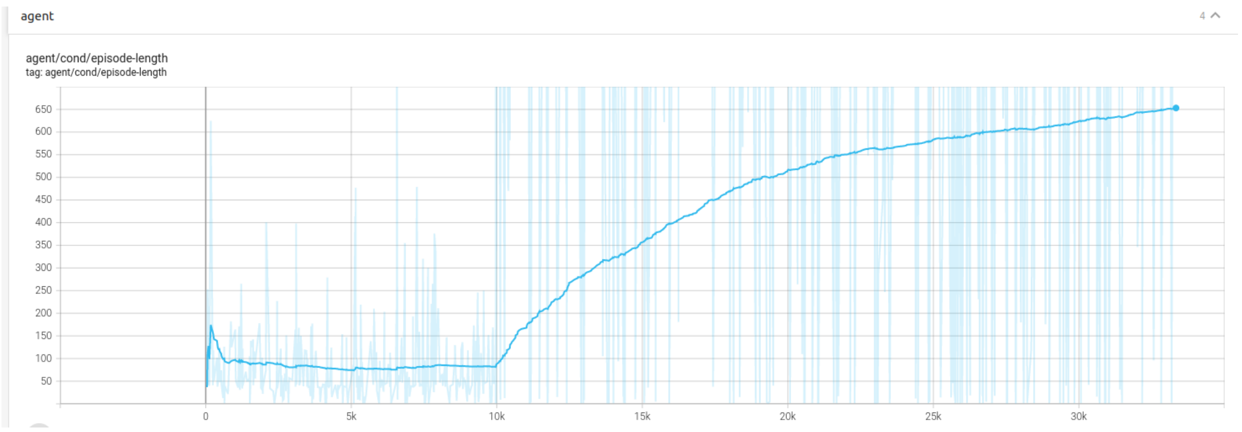


Figure 14: Episode length dense(128)/gru(64)/dense(64) and agent PPO.

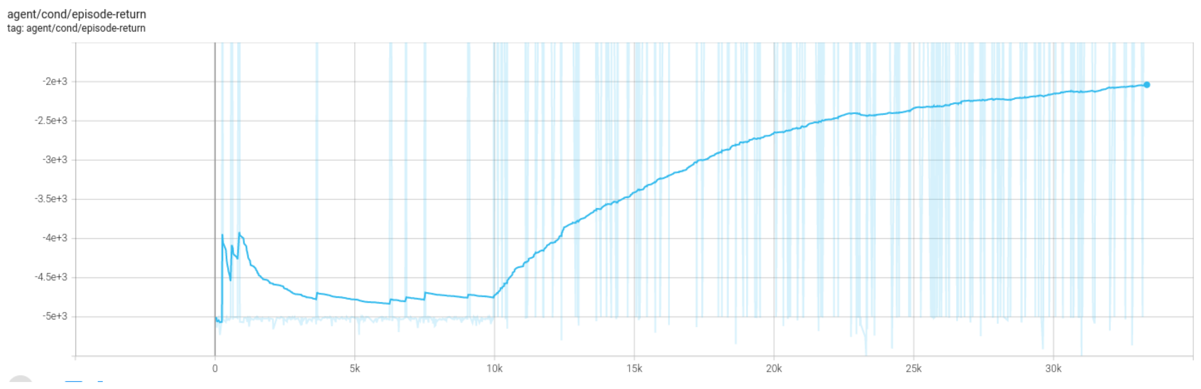


Figure 15: Episode return dense(128)/gru(64)/dense(64) and agent PPO.

6 Conclusion

In this thesis, a study on reinforcement learning took place and an effort was made to find a solution to the problem of navigating a blind robot in an unknown environment. Two types of actors were used with different results each. The results were satisfactory, having in mind the low computational power of the machine used to train the agent.

Ideally, to have more complete and robust results, runs of many episodes should be performed, using both the PPO and Actor-Critic algorithms, the four different reward cases mentioned above and the five distinct network architectures. These runs will provide sufficient outcomes that will, eventually, help adjust the settings of the reward function and the networks in such a way that the agent would, finally, be fully trained and capable of finding the target with a collision-free movement. These runs, unfortunately, could not take place at this time due to the long time and the resources they demand.

Furthermore, it would be very interesting to be able to use the first version of the environment, which achieves a more complete simulation. An agent should be trained to move around a room without colliding, by utilizing the image input from the camera attached to it. In this version, the agent won’t be blind anymore and it would recognize its target and move safely towards it.

The real challenge would be to bring this whole application from simulation to reality, by creating a real robot, possibly using a 3D printer. The robot should be equipped with the appropriate distance and camera sensors, and have a board embedded with the algorithm running on it. In this way, it can start moving in a real environment, such as a house, and learn by trial-and-error to avoid the obstacles and reaching the goal, under real circumstances.

References

[1] D. Filliat and J.-A. Meyer, “Map-based navigation in mobile robots: I. A review of localization strategies,” Cognitive Systems Research, vol. 4, no. 4, pp. 243–282, Dec. 2003.

[2] Sutton, R. S., & Barto, A. G. (2018). Reinforcement learning: An introduction. MIT press.

[3] Aske Plaat. (2022). Deep Reinforcement Learning. https://arxiv.org/pdf/2201.02135.pdf

[4] Frank Rosenblatt. “The perceptron: a probabilistic model for information storage and organization in the brain.” In: Psychological review 65.6 (1958), p. 386.

[5] C Van Der Malsburg. “Frank Rosenblatt: principles of neurodynamics: perceptrons and the theory of brain mechanisms”. In: Brain theory. Springer,

[6] Ian Goodfellow et al. Deep learning. Vol. 1. 2. MIT press Cambridge, 2016.

[7] Seppo Linnainmaa. “The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors”. In: Master’s Thesis (in Finnish), Univ. Helsinki (1970), pp. 6–7

[8] François-Lavet, V., Henderson, P., Islam, R., Bellemare, M. G., & Pineau, J. (2018). An introduction to deep reinforcement learning. arXiv preprint arXiv:1811.12560.

[9] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. Rectifier nonlinearities improve neural network acoustic models. In Proc. icml, volume 30, page 3, 2013.

[10] L. Trottier, P. Giguere, and B. Chaib-draa. Parametric exponential linear unit for deep convolutional neural networks. In 2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA), pages 207–214, 2017. doi: 10.1109/ICMLA.2017.00038.

[11] R. Avenash and P. Viswanath. Semantic segmentation of satellite images using a modified cnn with hard-swish activation function. In VISIGRAPP, 2019.

[12] Puterman, M. L. (1990). Markov decision processes. Handbooks in operations research and management science, 2, 331-434.

[13] Donkers, T., Loepp, B., & Ziegler, J. (2017, August). Sequential user-based recurrent neural network recommendations. In Proceedings of the eleventh ACM conference on recommender systems (pp. 152-160).

[14] Yu, Y., Si, X., Hu, C., & Zhang, J. (2019). A review of recurrent neural networks: LSTM cells and network architectures. Neural computation, 31(7), 1235-1270.

[15] O'Shea, K., & Nash, R. (2015). An introduction to convolutional neural networks. arXiv preprint arXiv:1511.08458.

[16] Konda, V., & Tsitsiklis, J. (1999). Actor-critic algorithms. Advances in neural information processing systems, 12.

[17] J. Peters. Policy gradient methods. Scholarpedia, 5(11):3698, 2010. doi: 10.4249/scholarpedia.3698.

[18] Schulman, J., Levine, S., Abbeel, P., Jordan, M., & Moritz, P. (2015, June). Trust region policy optimization. In International conference on machine learning (pp. 1889-1897). PMLR.

[19] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. 07 2017

[20] OpenAI Gym. Available at https://gym.openai.com/

[21] CoppeliaSim. Available at https://www.coppeliarobotics.com/

[22] James, S., Freese, M., & Davison, A. J. (2019). Pyrep: Bringing v-rep to deep robot learning. arXiv preprint arXiv:1906.11176.

[23] Box2D. Available at https://box2d.org/

[24] PyGame. Available at https://www.pygame.org/

[25] Tensorforce. Available at https://tensorforce.readthedocs.io