



HELLENIC REPUBLIC
**National and Kapodistrian
University of Athens**
— EST. 1837 —

Department of Physics
Section of Electronic Physics and Systems
Master's Degree on Control and Computing

Master Thesis

**Development of a Tower Defense Game
with Reinforcement Learning Agents**

Maria Manolaki

(RN: 2015511)

Athens

November 2020

Supervisors

Dionysios Reisis, Associate Professor

Dr. Nikolaos Vlassopoulos, Research Associate

Evaluation Committee

Dionysios Reisis, Associate Professor

Ektoras Nistazakis, Associate Professor

Dr. Nikolaos Vlassopoulos, Research Associate

Acknowledgements

I gratefully acknowledge the invaluable assistance of Assoc. Prof. D. Reisis for his support during all my studies. I would like to extend my deepest gratitude to Dr. N. Vlassopoulos for his very helpful suggestions, generous support, encouragement and patience throughout the duration of this Thesis' project.

Contents

1 Abstract.....	1
2 Machine Learning.....	2
2.1 Types of Machine Learning.....	2
2.1.1 Supervised Learning.....	3
2.1.2 Unsupervised Learning.....	3
2.1.3 Reinforcement Learning.....	5
3 Neural Networks.....	7
3.1 Introduction to Neural Networks.....	7
3.1.1 Learning Process of Artificial Neural Network with Backpropagation.....	11
3.1.2 Parameters.....	12
3.1.3 Hyperparameters.....	13
3.2 Architectures of Neural Networks.....	14
3.2.1 Feed Forward.....	14
3.2.2 Recurrent Neural Network.....	15
3.2.3 Long Short-Term Memory (LSTM).....	17
3.2.4 Gated recurrent unit (GRU).....	18
3.2.5 Convolutional Neural Network (CNN).....	19
4 Reinforcement Learning.....	21
4.1 Principles, Basic Terms and Definitions.....	21
4.2 Basic Reinforcement Learning algorithms.....	25
4.2.1 Dynamic Programming (DP).....	26
4.2.2 Temporal Difference Methods (TD Methods).....	27
4.2.3 Sampling, Monte Carlo and TD (λ) methods.....	29
4.2.4 Policy Search methods.....	29
4.2.5 Actor-Critic Methods.....	30
4.3 Deep Reinforcement Learning.....	31
4.3.1 Deep Q-Learning (DQN).....	32

4.3.2 Asynchronous Advantage Actor-Critic (A3C) and Advantage Actor-Critic (A2C).....	33
5 The Game.....	36
5.1 Tower Defense.....	36
5.2 Tools	36
5.3 Game description	37
5.4 Training the enemy to win the player	41
5.4.1 Setting up the actor and the critic	41
5.4.2 Taking an action	42
5.4.3 Reward logic.....	42
5.4.4 Learning logic.....	43
5.4.5 Run Tests.....	44
6 Conclusions	48
References	49
Appendix	52
Tower_defence.py.....	52
Entities.py	60

1 Abstract

The aim of this Master Thesis is to develop a Tower Defense game and to equip it with computational intelligence. A Deep Reinforcement Learning technique, more specifically the Advantage Actor-Critic (A2C), is implemented and tuned in order to provide the enemy with intelligence assisting him to reach the final goal by avoiding the obstacles.

The enemy is trained online. The A2C method, due to its actor-critic part, benefits from the characteristics of both policy search and Q value methods, while the use of a deep neural network permits the handling of large action spaces by reducing the dimension of the problem. The reinforcement learning is a trial-and-error general method letting the agent to learn via the rewards he is receiving from the environment in response to his actions.

By enriching the game with an artificial intelligence component, it is becoming more interesting for the player. In addition, the game served as a means to study and deepen our knowledge further in the field of deep reinforcement learning.

Chapter 2 is discussing aspects and principles of Machine Learning.

Chapter 3 covers the topic of Neural Networks by presenting their basic components, characteristics, learning/training methods and the most important architectures.

The subject of reinforcement learning is discussed in chapter 4. The fundamental terms, definitions and principles are addressed in the beginning of the chapter, then the basic reinforcement learning algorithms are briefly presented. *Deep* reinforcement learning and two relevant algorithms are covered at the end of the chapter.

Chapter 5 deals with the game, more specifically with the design, development, implementation and the reinforcement learning component. The approach followed and the design characteristics, the tools that were employed for the development and the issue of tuning the training parameters are presented in detail. The chapter ends with a presentation and discussion of the results obtained after running the game.

The thesis is finishing with the Conclusions, whereas the full code that has been developed is presented in the Appendix.

2 Machine Learning

Machine Learning (ML) is a subset of artificial intelligence (AI) that provides systems the ability to automatically learn and improve from experience and to become more accurate at predicting outcomes without being explicitly programmed. It relies on underlying hypothesis of creating the model and tries to improve it by fitting more data into the model over time. The primary aim is to allow the computers learn automatically without human intervention or assistance and adjust actions accordingly. ML is applied in many areas, but it is mostly significant in data mining.

Formal Definition given by Mitchel [1]: “A machine is said to learn from experience **E** with respect to some class of tasks **T** and performance measure **P** if its performance at tasks in **T**, as measured by **P**, improves with experience **E**.”

2.1 Types of Machine Learning

Machine learning approaches can be classified into 3 broad categories, depending on the nature of the "signal" or "feedback" available to the learning system:

- Supervised machine learning
- Unsupervised machine learning
- Reinforcement learning

Figure 1 demonstrates machine learning types combined with their main applications.

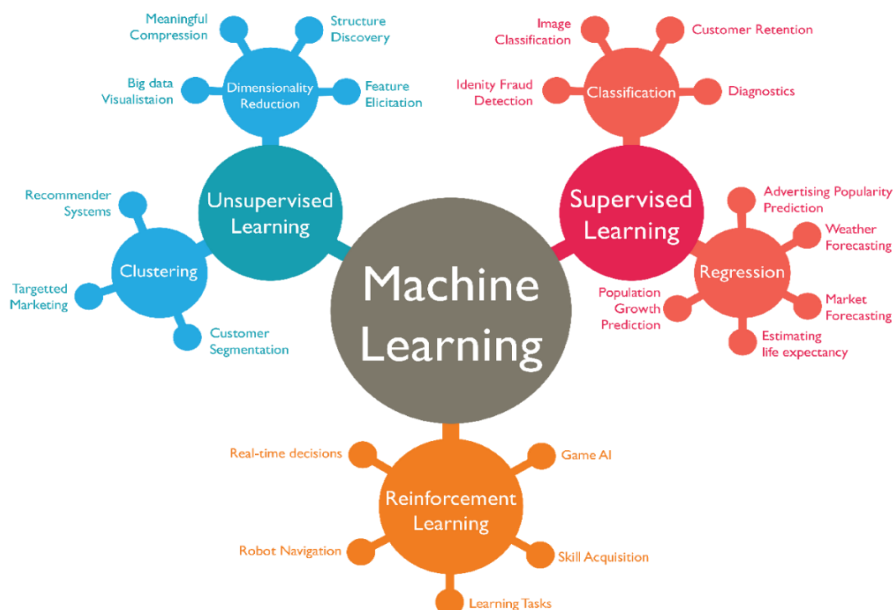


Figure 1. Diagram of Machine Learning subcategories ¹

¹ <https://medium.com/swlh/types-of-machine-learning-algorithms-62608e83d709>

2.1.1 Supervised Learning

Supervised machine learning requires a human supervisor and training data which is a set of input-output pairs. The input consists of data sample, typically a vector, used to make prediction, whereas the output part is the expected outcome, called label or the supervisory signal. The human supervisor is necessary to assign the labels to the pairs. While training a supervised learning algorithm, data is searched for a pattern that correlates with the desired outputs. After the training phase, the supervised learning model is expected to predict the correct label for a newly presented input data.

Supervised learning can be additionally categorized into: Classification and Regression.

Classification is the process of predicting the class (also called target, label, category) of given data points. Classifiers themselves can be divided in two groups based on the number of classes that they work with:

- *Binary classifiers*: have only two classes (e.g. spam mail or not)
- *Multiclass classifiers*: have multiple classes (e.g. whether an image is apple or orange or banana)

Regression analysis is a form of predictive modelling technique which investigates the relationship between a dependent variable, the target, and an independent variable, the predictor. Both dependent and independent variable are real values. This technique can be used in various areas such as: Forecasting or Predictive analysis, Optimization, Error correction, Economics, Finance, etc.

Typically, there are 3 types of Regression: *Linear Regression*, *Non-Linear Regression* and *Logistic Regression*. The objective of the first one is to determine the slope and the constant term (intercept) of the line that fits best the data. Similarly, in the non-linear case the goal is to determine the characteristics of a curve that best fits the data, e.g. Polynomial regression. Finally, the logistic regression uses a logistic function, which will be analyzed later, to model the probabilities and has two modes: the binary and the multinomial. Logistic regression is widely used in classification problems.

2.1.2 Unsupervised Learning

Unsupervised learning is a process where neither class labels or structure of the data are provided for each sample. The training data is of a set of input vectors only i.e. there are no corresponding labels. The goal may be to discover groups of similar examples within the data.

This process is called *clustering*. Another case is to reduce the dimensions of the dataset; for example, to reduce the number of columns of the dataset matrix, or to convert sphere-shaped data to circle. This process is called *dimensionality reduction*.

Clustering 's goal is to discover a structure in a collection of raw data (without any label). It could be described as the process of "organizing objects into groups whose members are similar in some way" [2]. A set of points, with a notion of distance between the points, are grouping into a number of clusters with the following rules:

- Internal distances should be small (members of the same cluster)
- External distances should be large (members of different clusters)

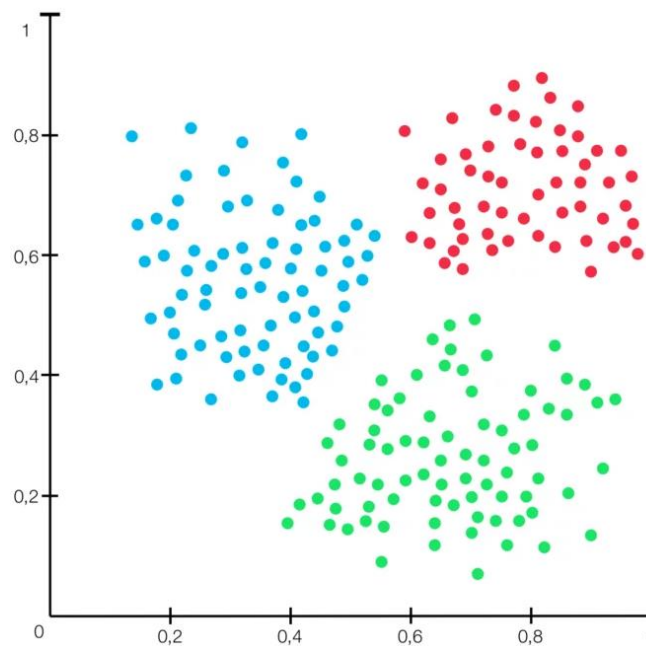


Figure 2. Clustering ²

In **dimensionality reduction**, the goal is to reduce the dimensions of a feature set. If the training of a machine learning model is based on many features, this model becomes dependent on the data used for training (overfitting). In many cases overfitting results in low performance when the model is applied on real data. Dimensionality reduction is an approach towards eliminating overfitting effect, improving model accuracy, minimizing computational burden and storage, and giving possibility to use algorithms which otherwise could not be used for large dimensions.

²: <https://rocketloop.de/en/clustering-with-machine-learning/>

2.1.3 Reinforcement Learning

Reinforcement Learning (RL) deals with learning via acting and getting rewards as feedback. Two basic notions are the *agent(s)* and the *environment*, and the *reward* which is received as feedback from the environment. An agent can perceive its environment, can take actions and interact with it. The goal is to find a suitable action model that would maximize the total cumulative reward of the agent. The action-reward feedback loop of a generic RL model is demonstrated in the following figure.

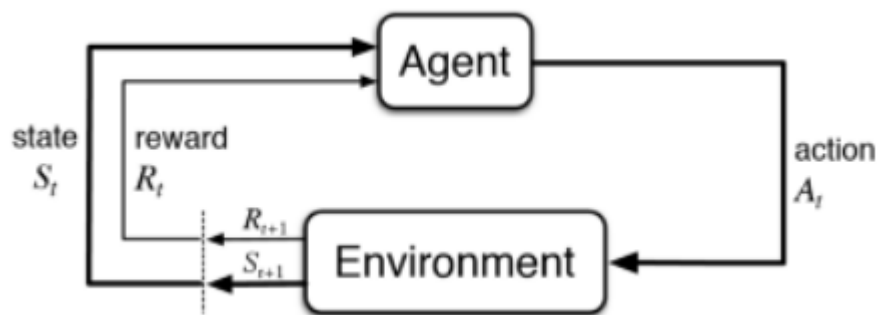


Figure 3. Action-Reward Feedback loop³

The basic elements of a Reinforcement Learning algorithm are:

- Environment: the world where an agent is trained how to make correct decisions for his actions.
- Agent: an entity that learns and makes decisions.
- Action: a status change in the environment caused by the agent.
- Reward: a signal sent from the environment to the agent evaluating an action.

Some other important parts of this technique are:

- Policy: a function that makes the decisions of the agent, which actually maps the agent's state to actions.
- State: a set of variables which describe the internal part of the environment (not always fully observable).
- Value function: a function that returns a real number corresponding to a specific state after following a particular policy. The returned value is used as the long-term reward.
- Model: how the agent perceives the environment, i.e. a map of state-action pairs to probability distributions. Not all RL agents use models of their environment.

³ <https://towardsdatascience.com/reinforcement-learning-101-e24b50e1d292>

Reinforcement algorithms can be applied in multiple areas such as:

- Resources management in computer clusters
- Traffic Light Control
- Robotics
- Web System Configuration
- Personalized Recommendations
- Bidding and Advertising
- Games
- Chemistry

The key difference between reinforcement and supervised learning lies in the feedback provided to the agent. In supervised learning the feedback is a set of actions to perform correctly a task, whereas in RL rewards and punishments are used as feedback to train the agent.

In comparison to unsupervised learning, RL differs in the goals to be achieved. In unsupervised learning the goal is to find differences and similarities in dataset points, while in RL the aim is to search for an action model which would lead to maximum total cumulative reward.

3 Neural Networks

3.1 Introduction to Neural Networks

The idea of Artificial Neural Networks (ANN) or simply Neural Networks (NN) was introduced in 1943 by Warren McCulloch and Walter Pitts [3] who proposed a model, called threshold logic, for neural networks based on mathematics and algorithms. Later in 1958 Rosenblatt developed the perceptron, a pattern recognition supervised learning algorithm, that was using two-layer network [4]. The research on NN slowed down after 1969 when Marvin Minsky and Seymour Papert discovered the limitations due to lack of sufficient computational power at that era demanded by large NN which would lead to very long run time [5]. In the mid of 70's the interest for NN reflatd due to both the development of computers with much higher processing ability and the effectiveness of the proposed backpropagation algorithm [6]. After a decade of low interest, a forceful comeback is taking place the last ten years, mainly because of the highly increased processing power provided by graphics chips.

A neural network is a network of artificial *neurons* which simulates the functionality of human brain. Their ordinary use is in clustering and classification. They present notable ability to cope with complex and raw data and conclude a meaning from them which is very useful for pattern recognition. They can derive trends from this data which cannot be discovered by other computer methods or even sensed by humans. The main advantages are:

- *Use as an expert.* ANN can be perceived as an expert in the area it has been trained for. It can analyze raw data and provide predictions when new situations arise.
- *Adaptive learning.* They present learning ability on how to perform actions based on training data.
- *Self-Organization.* They are capable to develop their own representation of the information provided during the learning period.

A typical ANN architecture includes three layers: *input*, *hidden* and *output* layer.

- *Input Layer:* it receives the raw information.
- *Hidden Layer:* the output of each unit (neuron/node) in this layer is a weighted combination of the inputs.
- *Output Layer:* similarly, the behavior of each unit in the output layer is a weighted combination of the outcomes of the hidden neurons.

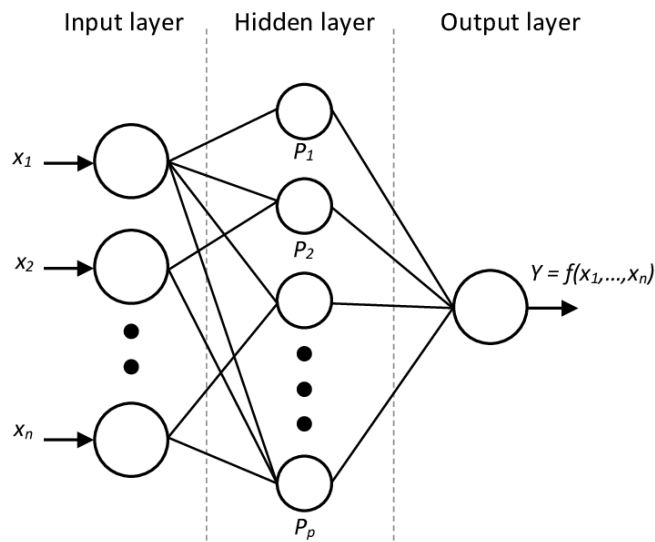


Figure 4. Neural Network with Input layer, Hidden layer, Output layer.⁴

Of course, there are architectures that include none or more than one hidden layer and more than one units in the output layers. Deep is a term to characterize a NN that has at least one hidden layer.

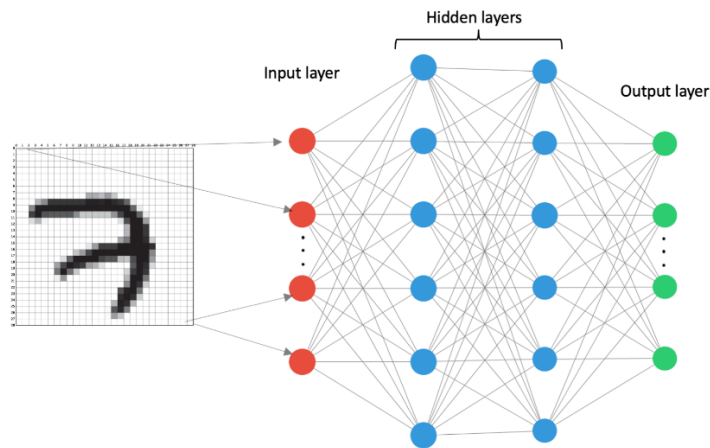


Figure 5. An Architecture with 2 hidden layers and n units in the output layer⁵

The fundamental unit in a neural network is called *neuron* and it is represented as a node in the architecture diagrams. The neuron is fed with inputs by other nodes (in the case of hidden and output layers) or by an external source (in the case of input layer) and derives its output.

⁴ https://www.researchgate.net/figure/Architecture-of-a-multilayer-neural-network-with-one-hidden-layer-The-input-layer_fig3_270274130

⁵ <https://towardsdatascience.com/a-beginners-guide-to-neural-nets-5cf4050117cb>

Each node is connected with the other nodes with some associated weight (w) which represents the relative importance, as it can be seen in Figure 6.

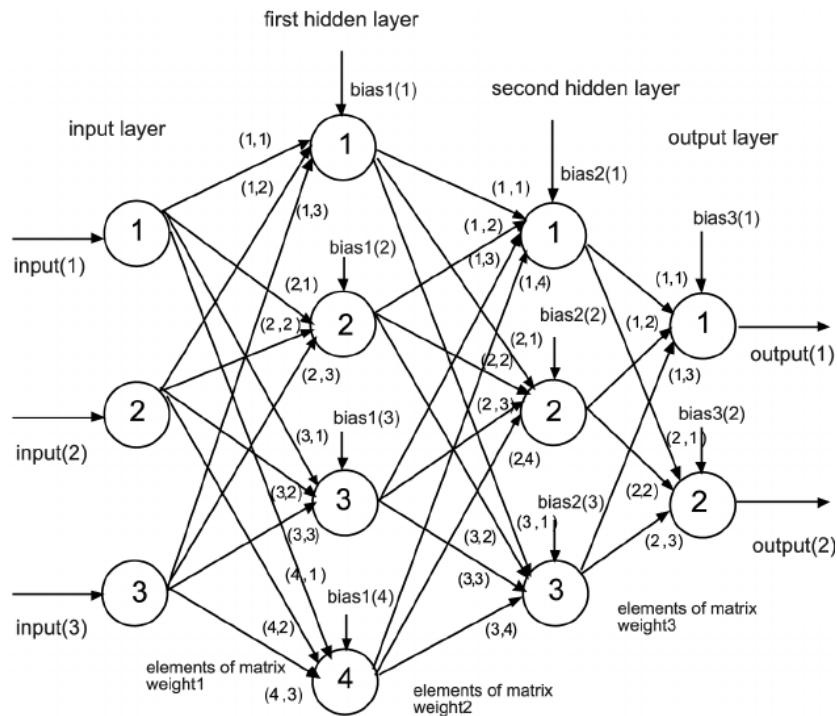


Figure 6. NN with 2 hidden layers and two output nodes ⁶

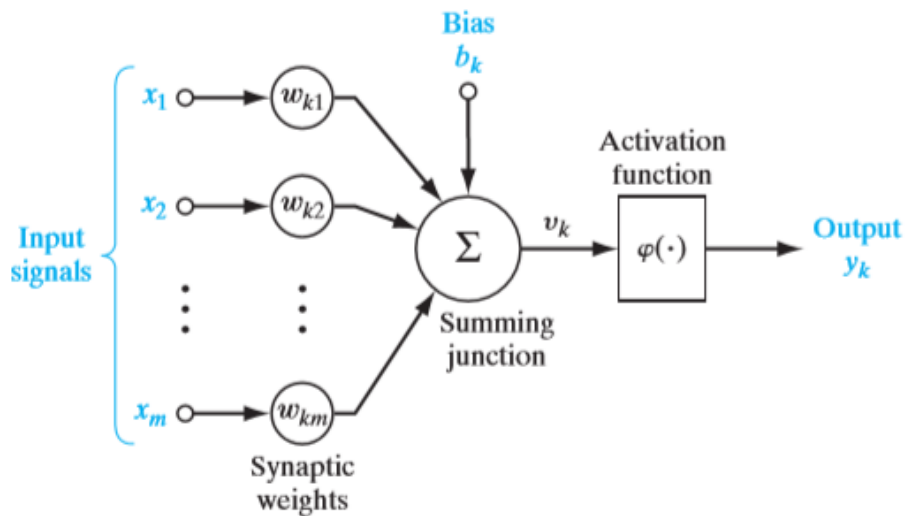


Figure 7. Model of a neuron ⁷

Figure 7 illustrates the model of the k^{th} neuron in a layer. The function of the neuron is described by the following equations:

⁶ https://www.researchgate.net/figure/Diagram-of-a-NN-with-two-hidden-layers_fig4_235308454

⁷ Neural Networks and Learning Machines, Simon Haykin, Prentice Hall

$$u_k = \sum_{j=1}^m w_{kj} x_j$$

$$y_k = \varphi(u_k + b_k)$$

Where

x_i is the j^{th} input

w_{kj} is the *synaptic weight* of input j of neuron k

u_k is the result of the linear weighted *sum* operation

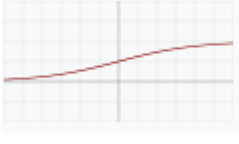


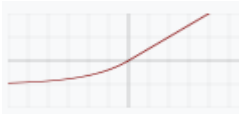
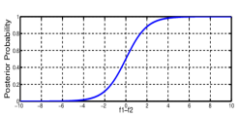
$\varphi()$ is the *activation function* or *squashing function* as it is used to limit the amplitude of the output.

b_k is the bias which performs an affine transformation to the adder output u_k . Its main role is to provide an extra trainable constant value to neuron k .

The activation functions in the most common cases are non-linear. Introducing non-linearity in the neuron output, helps the neuron to learn since most real-world data are not linear.

Some of the main activation functions are shown in the table below:

Table 1. Indicative activation functions

Name	What it does	Plot ^{8,9}	Function
Sigmoid	limits the output to the range [0, 1]		$\sigma(x) = \frac{1}{1 + e^{-x}}$
tanh*	limits the output to the range [-1, 1]		$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
ReLU	Rectified Linear Unit. Permits only positive values, otherwise 0		$\varphi(x) = \begin{cases} 0, & \text{if } x \leq 0 \\ x, & \text{if } x > 0 \end{cases}$
ELU	Exponential Linear Unit. For positive values $y = x$, otherwise a small negative number		$\varphi(x) = \begin{cases} a(e^x - 1), & \text{if } x \leq 0 \\ x, & \text{if } x > 0 \end{cases}$
Softmax*	Limits the output to the range (0,1)		$f_i(\vec{x}) = \frac{e^{x_i}}{\sum_{j=1}^J e^{x_j}}, \quad i = 1, \dots, J$

*used in this master thesis.

⁸ Images taken from https://en.wikipedia.org/wiki/Activation_function

⁹ https://www.researchgate.net/figure/Softmax-activation-function_fig2_319121953

There are several other activation functions like Identity, Threshold, Binary Step, GELU, SELU, SQNL, Gaussian, arctan. Each activation function is designed to face specific application problems.

The selection of an appropriate activation function is part of the architecture design of an NN, and it is considered as a hyperparameter, i.e. it is a parameter relating to the architecture and specified during the design phase of a NN. More details about hyperparameters is going to be discussed in the next pages.

Universal Approximation Theorem proves that any mathematical function $y = f(x)$ can be approximated, to an acceptable error magnitude, by a neural network, that includes one hidden layer containing a finite number of neurons, under the condition that employs the appropriate activation function. George Cybenko proved it only for sigmoid activation functions in 1989 and Kurt Hornik extended the proof for all the activation functions in 1991. The key element to achieve performance is the structure of the neural network, not the type of the activation function. As a conclusion, the Universal Approximation Theorem states there is a type of universality in NNs, i.e. there is a neural network that can achieve an approximation of any given function.

3.1.1 Learning Process of Artificial Neural Network with Backpropagation

Training a neural network means that the values of weights w_{kj} and biases b_k must be adjusted in that way that the neural network obtains the desired behavior. This is an iterative learning process achieved by forward propagating the information of the inputs and by backpropagating the error.

In **forward propagation** all input data pass through all neurons of the network which apply their transformation and finally the output layer will produce estimates for this specific training data. Then, a loss function is used to estimate the magnitude of the error, i.e. the difference from the desired response. Of course, the ideal value of cost function should be close as possible to zero. For this purpose, the synaptic weights and the bias of each neuron must gradually be tuned.

The output error will be propagated backwards (**backpropagation**). Each hidden layer neuron k receives only a part of the error e_j measured in output y_j . Specifically, if neuron k contributed with a weight w_{kj} to the output y_j , then e_j will be weighted with the same weight when it is propagated backwards to node k , that is:

$$e_{kj} = e_j * w_{kj}$$

where e_{kj} is the part of error in neuron k contributed by the subsequent layer neuron j .

The total error of the hidden layer neuron k received backwardly from all output layer neurons is:

$$e_k = \sum_{j=1}^n e_j w_{kj}$$

where n is the number of the output layer neurons.

This information passes in the same way through all neurons starting from the output layer and going backwards layer by layer. So, all neurons will obtain backwardly a signal assessing their contribution to the final error. Then an optimization method can be applied, such as *Gradient Descent*, in order to minimize the output layer error by tuning all the network's synaptic weights. Other optimization methods are SGD, RMSprop, Adagrad, Adadelat, Adam, Adamax and Nadam.

Feeding again the same training data to the network, a better performance should be observed as the weights have been adjusted. Then, the new and smaller error is propagated backwards again and so on. This process is done iteratively in batches of data of all the dataset that is passed to the network.

Training error is the error observed when training data are used, whereas *test error* is the error observed on the new input data. The aim of a machine learning algorithm is to make both the training error, and the difference between training error and testing error small. The term *under-fitting* means that the model cannot attain a low training error, while *over-fitting* means that the difference between training error and test error is large.

One of the approaches to avoid overfitting is to train the network in more examples. Another approach, called *regularization*, tries to constrain/regularize the coefficient estimates in very small values otherwise to shrink (simplify) the model/structure. L1 and L2 are two well-known regularization methods. L1 regularization (Lasso regression) limits the size of the coefficients and yields sparse models as some coefficients can become zero and be eliminated. L2 (Ridge regression) is not yielding sparse models as no coefficient is eliminated but all of them are weakened by the same factor. *Dropout* is a new regularization approach where a simpler structure/model is obtained by randomly "dropping out", i.e. omitting, one or more units during the training phase.

3.1.2 Parameters

Parameter is a variable internal to the model which is used to configure the NN. A value of a parameter could be changed according to the data used during the program iterations. More

specifically, the weights of the connections between the neurons and the biases are such parameters.

Weight initialization: Initially the weights are set to small random values which should be different from neuron to neuron. Otherwise, if two neurons start with the same initial weight values, their values will be identical during the following iterations, which means canceling their ability to learn different characteristics. Setting random values to initialize synaptic weights is not enough to obtain an efficient NN and generally the type of activation function should be taken into account in order to define the initial values heuristically.

3.1.3 Hyperparameters

The term hyperparameter, in contrast to parameters, refers to variables external to the model and they are specified during the design phase by the programmer. A number of such variables relate to the structure and topology of the NN (type of activation functions, number of layers, number of neurons, etc.) whereas other of them relate to the learning algorithm (learning rate, momentum, batch size, epochs, optimization method etc.).

Learning Rate: In backpropagation with gradient descent, in order to update each synaptic weight w_{kj} of a neuron in the network, the following formula is used:

$$w_{kj} = w_{kj} - a \frac{de_k}{dw_{kj}}$$

where

$\frac{de_k}{dw_{kj}}$ Expresses the error change with respect to weight

a Is a scalar value denoting the learning rate, otherwise called step size.

For example, if the learning rate magnitude a is 0.01 and the gradient is 2, then the new weight w_{kj} will be reduced by 0.02.

Epochs: It is the number of times (iterations) the training data pass through the NN during training phase.

Batch size: Usually the training data are split in batches to feed the NN. Batch size is the size of one batche

Momentum: It is based on the weighted average of the gradient of the previous steps. It is a method that accelerates the learning rate when its vector has the same direction with

the current gradient and in the opposite case it helps to avoid local minima of the loss function.

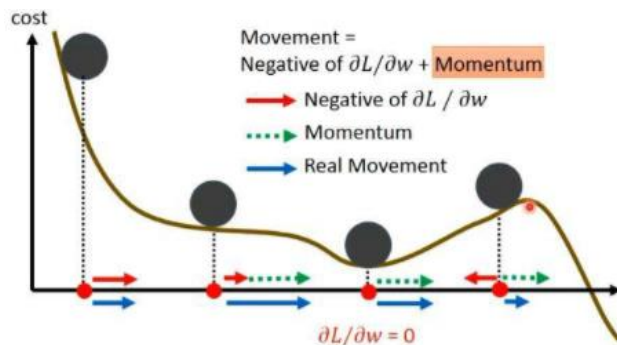


Figure 8. GD with momentum ¹⁰

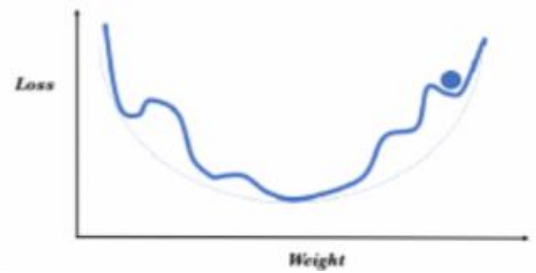


Figure 9. Stuck at a local minimum ¹¹

There are no simple and straight-forward methods to define the optimal values of the hyperparameters, especially the learning rate, momentum, batch size, type of activation function, number of layers and neurons etc. Expertise and usage of extensive trial-and-error are key elements for defining the optimal values of these parameters.

3.2 Architectures of Neural Networks

3.2.1 Feed Forward

The Feed Forward (FF) constitutes the first type of NN structure invented. In this network type, the information proceeds only forward as there are no circular connections.

The simplest form is called *Single-Layer Perceptron* (SLP) and consists only of an input and an output layer. Figure 10 demonstrates an SLP where the output is derived after applying an activation-squashing function (usually a sigmoid) to the weighted sum of the inputs. An SLP is a linear classifier.

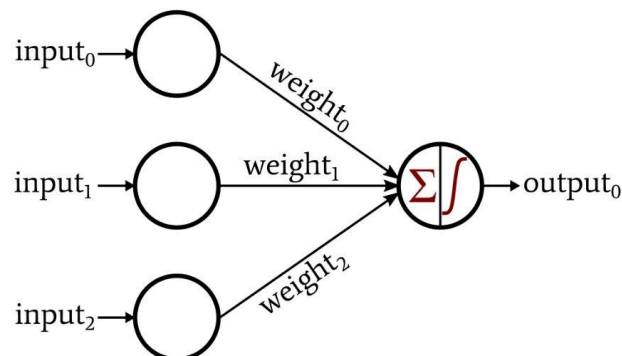


Figure 10. Single-Layer Perceptron ¹²

¹⁰ <https://medium.com/ai%C2%B3-theory-practice-business/hyper-parameter-momentum-dc7a7336166e>

¹¹ <https://towardsdatascience.com/learning-process-of-a-deep-neural-network-5a9768d7a651>

¹² <https://www.allaboutcircuits.com/technical-articles/how-to-perform-classification-using-a-neural-network-a-simple-perceptron-example/>

Another and more complete form of FF network is the Multi-layer Perceptron (MLP) which adds at least one hidden layer between input and output layer. MLP structure is depicted in the next figure.

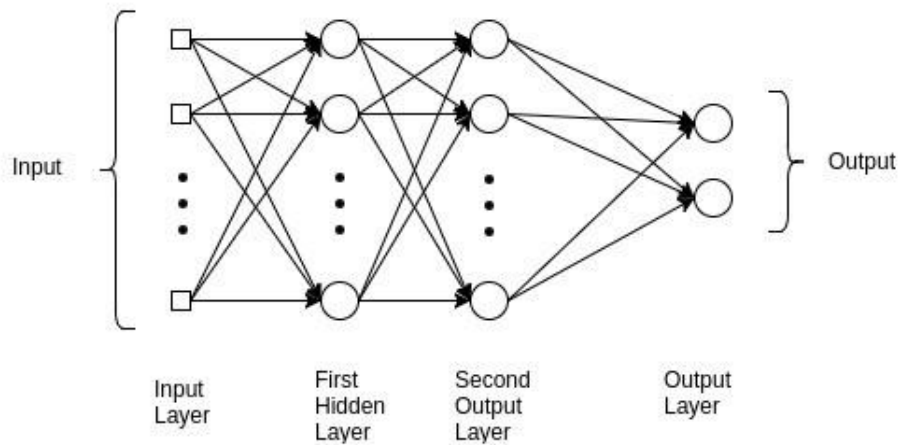


Figure 11. Multi-Layer Perceptron¹³

Since there are no circles in the connections of an FF, the output is determined by the current input dataset. Past inputs do not influence the output of current data.

3.2.2 Recurrent Neural Network

Figure 12 shows the typical chain-like recurrent neural network (RNN) architecture:

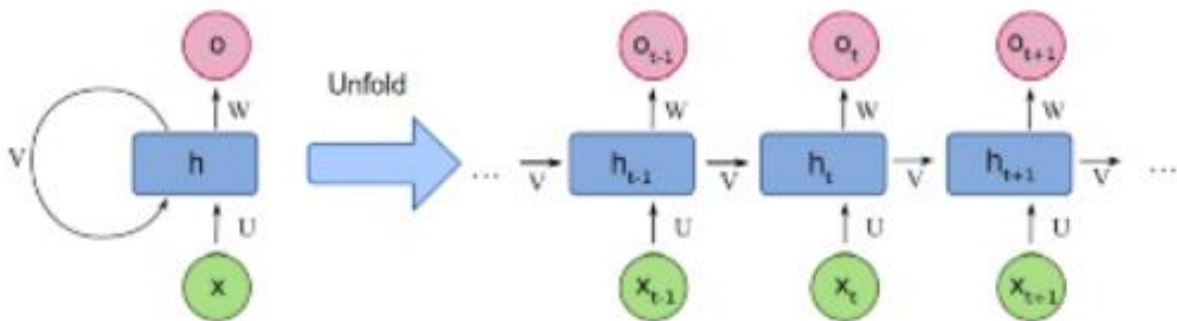


Figure 12. Folded and unfolded representations of an RNN¹⁴

The difference from the FF is that RNN include a loop in the hidden layer. The consequence of this is that the current output depends on both the current input vector and on the previous output. So, the RNN has *memory*. In contrast in an FF network, the output is not affected by the previous input data it handled and its memory is restricted only in things learnt during the training period.

¹³ https://medium.com/@AI_with_Kain/understanding-of-multilayer-perceptron-mlp-8f179c4a135f

¹⁴ <https://towardsdatascience.com/understanding-rnn-and-lstm-f7cdf6dfc14e>

In RNN the output is affected by both the weighted sum of the input and by a hidden “state vector” which is determined by the prior inputs. This can be described in a mathematical context as:

$$a_k = b_1 + Vh_{k-1} + Ux_k$$

$$h_k = \varphi(a_k)$$

$$o_k = b_2 + Wh_k$$

Where,

- x_k denotes the input layer vector at time k
- h_k denotes the hidden layer vector at time k
- o_k denotes the output layer vector at time k
- a_k is an assisting vector
- $\varphi(\)$ indicates the activation function (usually a sigmoid function $\sigma(\)$)
- b_1, b_2 are the bias vectors
- U, W, V are the weighting matrices of the *input-to-hidden* connection, *hidden-to-output* connection and *hidden-to-hidden* (loop) connections respectively

There are applications that their output depends on the total input sequence (preceding and succeeding data vectors as it refers to a specific datapoint). However, RNN's output depends only on the past sequence so the standard RNN cannot give optimal results. A solution to this problem was proposed by M. Schuster [7] who introduced the Bidirectional Recurrent Neural Network (BRNN). BRNN employs two RNNs, one for each time direction. More specifically, the first one evolves forward from the beginning of the data sequence, while the second starts from the end and moves backwards, see Figure 13.

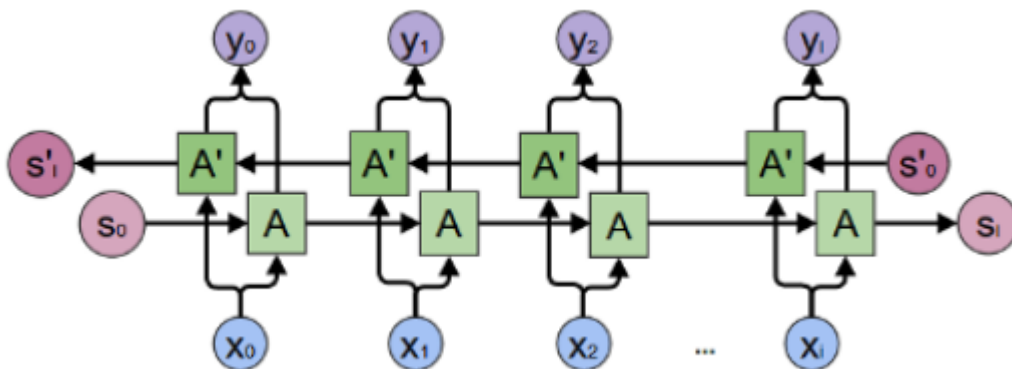


Figure 13. The BRNN structure ¹⁵

¹⁵ <https://towardsdatascience.com/understanding-bidirectional-rnn-in-pytorch-5bd25a5dd66>

Speech, handwriting and image recognition are examples of applications that take advantage of the BRNN architecture.

3.2.3 Long Short-Term Memory (LSTM)

Instability problems have been observed during RNN training because long term dependencies are covered by short term dependences. More specifically, for long term the gradient magnitudes frequently become smaller and smaller or sometimes explode which leads to instability [8]. A first proposal towards eliminating or moderating this problem came from Hochreiter and Schmidhuber [9] and it is an evolution of the classic RNN called long short-term memory (LSTM). Later a new variant of the recurrent structure was proposed by Cho et. Al. [10] called Gate Recurrent Unit (GRU), which will be discussed in the subsequent subsection.

The innovation of LSTM is the introduction of a memory component called *cell*. There are three other components: the input gate, the forget gate and the output gate, see Figure 14.

The function of each component is:

- **Cell:** Monitors the dependencies between the elements in the input data sequence.
- **Input gate:** Defines the part of the new value which will flow into the cell.
- **Forget gate:** Defines the part of the new value which will remain into the cell.
- **Output gate:** Determines the part of the new value in the cell which will be used to calculate the output.

The term “new value” refers to the sum of the input and the previous output. The commonly activation function used in the LSTM gates is the logistic sigmoid function.

In this way, the LSTM earns the capability to keep learning long-term dependencies and to remember information for long periods.

Although the LSTM variant of RNN solves the problem of long-term memory dependencies, it does not solve sufficiently the exploding gradient problem.

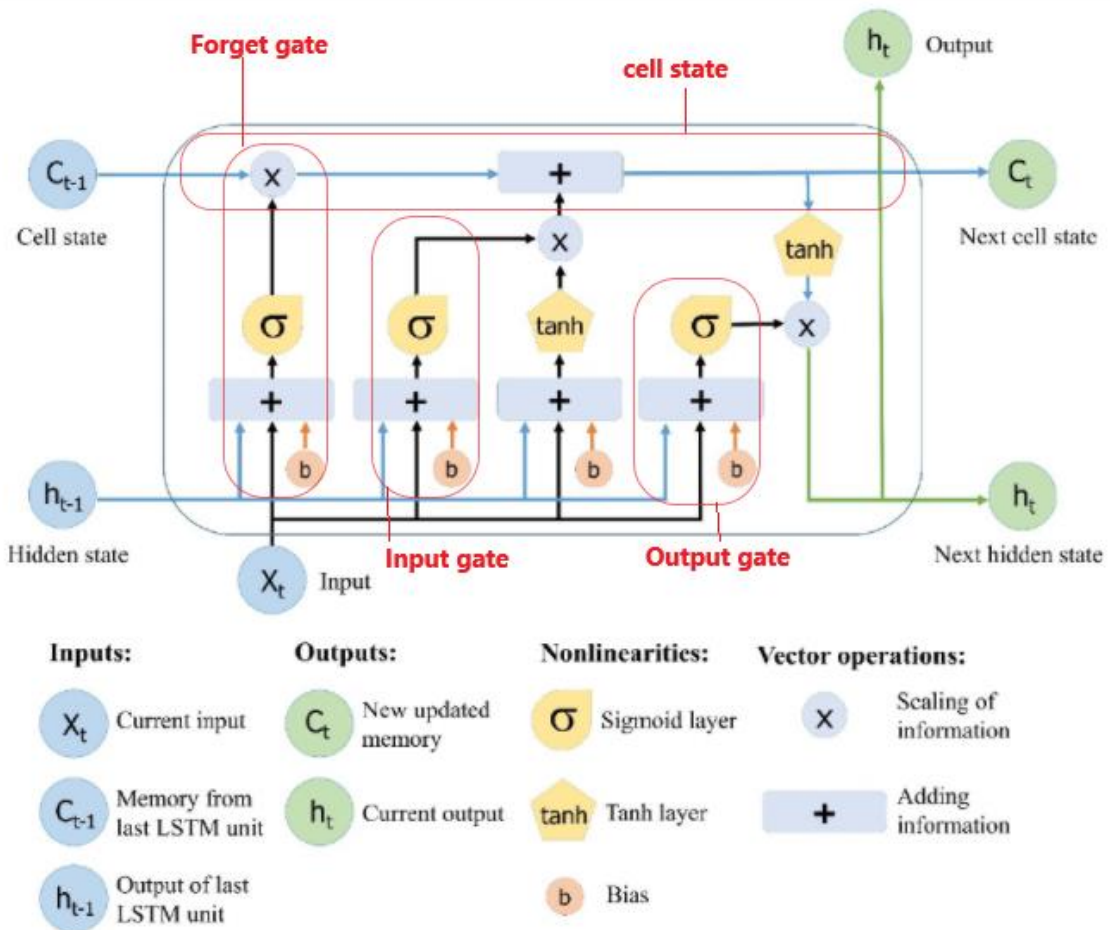


Figure 14. The Long Short-Term Memory structure ¹⁶

3.2.4 Gated recurrent unit (GRU)

Gated recurrent unit (GRU) is a simpler variation of the RNN. It is very similar to LSTM but without an output gate, so it has fewer parameters. It tries also to solve the vanishing gradient problem. The main components of its structure are the update gate and the reset gate, Figure 15.

The role of these gates (vectors) is to determine what information will be transferred to the output. More specifically:

- **Update gate:** Determines the amount of information coming from the past that needs be transferred to the future
- **Reset gate:** Defines the amount of information coming from the past that needs to be forgotten.

¹⁶ Main part of the figure based on Yan, S. Understanding LSTM and Its Diagrams. Available online: <https://medium.com/mlreview/understanding-lstm-and-its-diagrams-37e2f46f1714> (accessed on 26 June 2018)

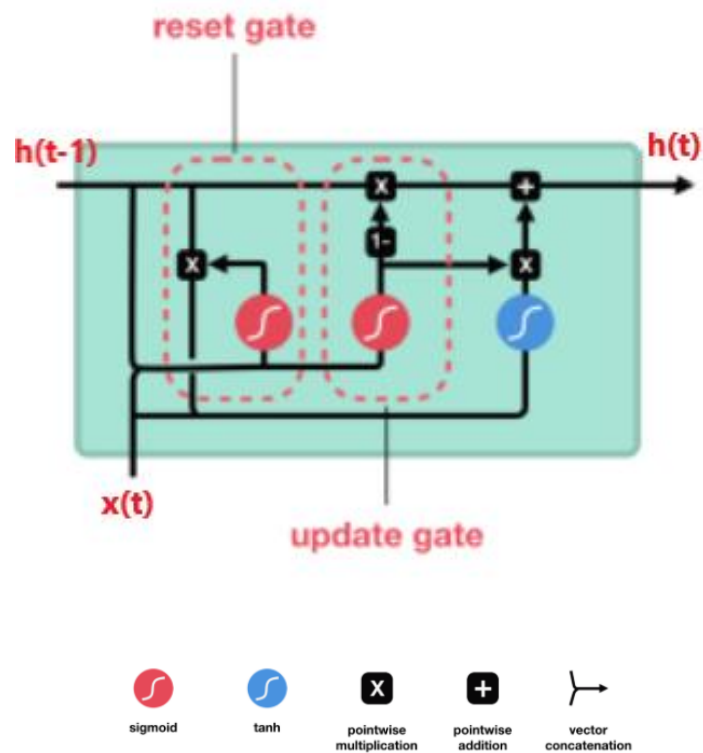


Figure 15. GRU structure ¹⁷

GRU and LSTM have similar performance on tasks like polyphonic music modeling, speech signal modeling and natural language processing, whereas GRU outperforms the latter for some smaller and rare datasets. However, GRU is not efficient in learning simple languages while LSTM is capable to learn them [11]. Moreover, LSTM units persistently exhibit better performance than GRU cells in "the first large-scale analysis of architecture variations for Neural Machine Translation" [12].

3.2.5 Convolutional Neural Network (CNN)

CNN is a feedforward NN with convolutional layers, pooling layers and fully connected layers, mostly applied to visual processing. They were inspired by the operation of visual cortex in human and animal brains. Their architecture follows the neurons' connectivity pattern of this area.

¹⁷ <https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21>

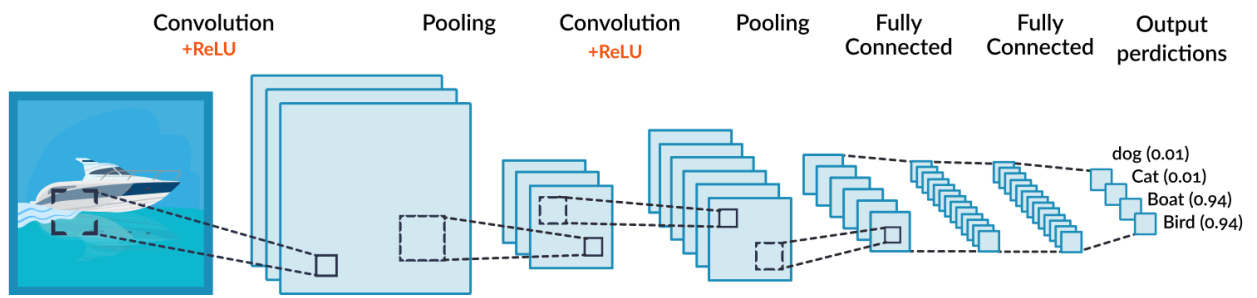


Figure 16. CNN architecture ¹⁸

- **Convolutional layer:** Firstly, a filter is used to view a small part of the broader image each time, e.g. a frame of 4x4 pixels. It starts from the beginning and moves until it parses all the width. The convolution is performed by a dot product of the input pixel values with the filter's weight matrix. The result is a scalar value representing all the pixels observed by the filter. In this way, the convolution layer produces a matrix which is much smaller than the original. The activation function applied is usually a ReLU function.
- **Pooling layer:** The purpose of this layer is to further reduce the size of the matrix. This is achieved by combining the outputs of the previous layer into one single node in the current layer. Pooling could be applied locally (where small clusters, e.g. 2x2, are processed) and globally (all neurons' outputs of convolutional layer are processed at once). Furthermore, there are two types of pooling that are used: *max pooling*, that uses the maximum value in each cluster, and *average pooling*, where the average value of each cluster is selected.
- **Fully connected layer:** Like in the typical multi-layer perceptron, all neurons in one layer are connected to all neurons in another layer. The goal of this layer is to classify the flattened matrix derived from the above-mentioned layers.

CNNs are very effective in applications such: image classification, image and video recognition, medical image analysis, natural language processing.

¹⁸ <https://missinglink.ai/guides/convolutional-neural-networks/convolutional-neural-network-tutorial-basic-advanced/>

4 Reinforcement Learning

It is a common perception that we learn by interacting with our environment. When an infant play, there is no explicit teacher, but it is directly observing the environment. If for some action it receives positive feedback (reward) from the environment, it repeats this action, otherwise it stops. In this way, an abundance of information is produced about the consequences of actions and what to do in order to achieve goals. All theories of learning and intelligence consider ‘learning from interaction’ as the fundamental principle.

An aim in artificial intelligence (AI) is to develop computational methods so that machines will become capable of learning from interaction with their environment, improving continuously through trial and error. Reinforcement learning (RL) is a mathematic framework for experience-driven autonomous learning.

4.1 Principles, Basic Terms and Definitions

The core principle of RL is learning *through interaction*. An *agent* acts, then observes the consequence of its action and learns to adjust its own behavior based on the reward/punishment it receives from the environment. The root of this trial-and-error learning approach comes from the behavioral psychology – behaviorism and constitutes one of the main foundations of RL [13]. *Optimal control* is the second key influencer on RL, which borrowed its mathematical formalism in this field.

An agent observes a state s_t of its environment at time t . Then the agent takes an action \mathbf{a}_t that is affecting the environment which transitions to a new state \mathbf{s}_{t+1} , which could be expressed as:

$$\mathbf{s}_{t+1} = f(\mathbf{s}_t, \mathbf{a}_t)$$

i.e. the new state is a function of the current state and the action chosen by the agent.

The state \mathbf{s}_t condenses all the sufficient information of the environment up to time t , so the agent, being provided with this, can choose an optimal action. It should be noticed here that in optimal control literature, state is denoted by \mathbf{x}_t and actions by \mathbf{u}_t .

Each time the environment progresses to a new state \mathbf{s}_{t+1} it also sends a reward r_{t+1} to the agent as a feedback. The reward is a scalar value. The aim of the algorithm controlling the behavior of the agent is to learn a *policy* (control strategy) π that will lead to a maximum value of the expected *return* (cumulative, discounted reward). Under this view, the problem of finding the optimal policy in RL is similar to optimal control problems. However, in optimal control there exists a model of the state transition dynamics which is not holding for the general RL case, where the agent does not have such a model for the environment dynamics so it can only apply

trial-and-error in order to learn through the consequence of its actions. Figure 12 presents this perception-action learning loop.

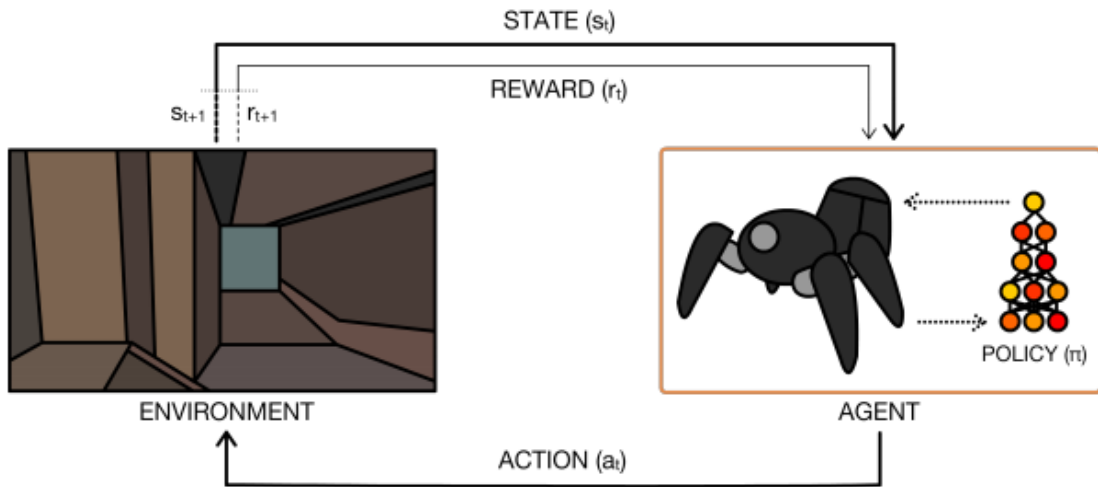


Figure 17. The perception-action-learning loop¹⁹

Markov Decision Processes (MDP)

The following notation is used:

s_t	State of the environment at time t
\mathcal{S}	A set of states
a_t	Action taken by the agent at time t
\mathcal{A}	A set of actions
$\mathcal{T}(s_{t+1} s_t, a_t)$	Transition dynamics. The new state and its distribution are derived on the basis of the current state and action.
π	Policy. Generally, it is a mapping from states, \mathcal{S} , to a probability distribution of the actions, that is $\pi : \mathcal{S} \rightarrow p(\mathcal{A} = a S)$.
π^*	Optimal Policy.
r_t	Reward. A scalar value returned to the agent at time t , indicating how well or bad is doing at step t .
γ	Discount factor. As lower is the value of γ as much emphasis is given on immediate rather on future rewards, $\gamma \in [0, 1]$
R	Cumulative Reward

¹⁹ Reference [15]

Cumulative Reward in Episodic systems

Episodic system means that the state is reset after each episode of length T steps. In an episode, the sequence of actions, states and rewards composes a policy *trajectory*, otherwise *rollout* or *horizon*. In every policy rollout, the environmental rewards are accumulated in the return value, R , that is:

$$R = \sum_{k=0}^{T-1} \gamma^k r_{t+k+1} = r_{t+1} + \gamma^1 r_{t+2} + \gamma^2 r_{t+3} + \dots.$$

Cumulative Reward R in non-Episodic systems (continuous)

Non-episodic system means that there is no reset of the state at specific time intervals, so $T = \infty$, in other words the trajectory is a complete one. In this case, having discount $\gamma < 1$ does not let an infinite sum of rewards to accumulate. However, non-episodic methods are no longer applicable.

$$R = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

Aim of the RL algorithm is to find an optimal policy, π^* , otherwise a series of state-action pairs $\{(\mathbf{s}_1, \mathbf{a}_1), (\mathbf{s}_2, \mathbf{a}_2), \dots, (\mathbf{s}_t, \mathbf{a}_t), \dots\}$, so that the expected cumulative return from all states will be maximum, i.e.

$$\pi^* = \operatorname{argmax}_{\pi} \mathbb{E}[R|\pi]$$

Markov Decision process (MDP)

Reinforcement Learning is described formally as a Markov Decision Process, with transition dynamics $\mathcal{T}(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$ that maps state-action pair $(\mathbf{s}_t, \mathbf{a}_t)$ to new state \mathbf{s}_{t+1}

Markov property

A state \mathbf{s}_{t+1} has the Markov property if and only if

$$p[\mathbf{s}_{t+1}|\mathbf{s}_t] = p[\mathbf{s}_{t+1}|\mathbf{s}_0, \mathbf{s}_1, \dots, \mathbf{s}_t],$$

i.e. that the probability distribution characteristics of the current state \mathbf{s}_t is enough to predict the state of the next step \mathbf{s}_{t+1} . All the past sequence $\mathbf{s}_0, \mathbf{s}_1, \dots, \mathbf{s}_{t-1}$ is not necessary any more. It can also be expressed as: The state of a system at time $t+1$ depends only the state at time t . So, the decision for the action at time $t+1$ can be based only on state \mathbf{s}_t . For the majority of RL algorithms this assumption is valid but it requires *fully observability* of the states.

Partially Observable MDP (POMDP)

A more generalized form of the MDP is the partially observable MDP (POMDP), where the agent receives an observation vector \mathbf{o}_t , whose probability distribution depends on the current state and the previous action, i.e.:

$$p_t(\mathbf{o}_t | \mathbf{s}_t, \mathbf{a}_{t-1}),$$

Usually, POMDP algorithms update a belief of the current state, given the current observation, the action taken and the belief state of the previous step.

However, in deep learning an approach followed more frequently, is to employ Recurrent Neural Networks (RNN) which, as discussed before, are dynamical systems i.e. they have an internal memory and are suitable to handle sequences of data in contrast to feedforward neural networks who are restricted to use independent data only.

Reinforcement learning main characteristics and challenges

- There is no supervisor. A reward signal is used for training. The optimal policy has to be concluded by trial and error.
- Time matters because data are processed in sequential form.
- An action of the agent influences the subsequent data it receives, in other words the observations it receives depend on its previous actions and can contain temporal correlations. Consequently:
 - Feedback is not instantaneous, but delayed.
 - Agents must cope with long term dependencies: Frequently, the consequence of an action appears only after many transition steps of the environment. This is the credit assignment problem [13].

Some Basic RL terms

Prediction problem or problem evaluation: refers to the computation of state value function or action value (quality) function for a policy.

Control problem: is to find the optimal policy.

Planning: is to construct a value function for policy with a model.

On-policy methods: they assess or improve the behavior policy. An example is State-Action-Reward-State-Action (SARSA) algorithm which tries to improve the estimate of action value function using samples derived by the same policy.

Off-policy methods: In these methods the agent learns an optimal value function unrelated to the followed behavior policy. Such a method is Q-learning which tries to find

state-action values for the optimal policy without trying to fit to the policy generated the data.

Exploration vs exploitation: It refers to the dilemma faced, i.e. what is better to do in short time?

Seek an immediate maximum reward by using the currently -not optimal yet- best action or to continue exploring the environment looking for an optimal action so the long-term cumulative reward will maximum.

Model free methods: The transition model is not known and the agent learns with trial-and -error.

Model based methods: They rely on a model of the environment. The model could be known or learned.

Bootstrapping: An estimate (state or action value) is updated in the next step from subsequent estimates.

4.2 Basic Reinforcement Learning algorithms

Two main approaches are used: methods based on Value Functions (such as Dynamic Programming, SARSA and Q-Learning) and on *Policy Search*. A third approach called *Actor-Critic* is hybrid as it combines both value functions and policy search.

Value Function methods

They use a function to estimate the expected cumulative return of being in a given state.

The *state-value function* starting from state \mathbf{s} , and when policy π is followed, is

$$V^\pi(\mathbf{s}) = \mathbb{E} [R|\mathbf{s}, \pi]$$

The optimal state-value function V^* , when optimal policy π^* is followed, is

$$V^*(\mathbf{s}) = \max_{\pi} V^\pi(\mathbf{s}), \quad \forall \mathbf{s} \in \mathcal{S}$$

If $V^*(\mathbf{s})$ was available, the optimal policy could be retrieved by picking up among all available actions at \mathbf{s}_t .

However, in RL environments, the transition dynamics \mathcal{T} are not known, so another function value is used, that is the *state-action* or *quality* function $Q(\cdot)$. It is similar to $V(\mathbf{s})$ but it differs in that now the initial action is provided and the policy is followed only from the succeeding state forward:

$$Q^\pi(\mathbf{s}, \mathbf{a}) = \mathbb{E}[R|\mathbf{s}, \mathbf{a}, \pi]$$

which is defined by the expected return for selecting action \mathbf{a} in state \mathbf{s} and following policy π . The best policy π^* , given $Q^\pi(\mathbf{s}, \mathbf{a})$ can be found by choosing action \mathbf{a} greedily at every state: $\operatorname{argmax}_{\mathbf{a}} Q^\pi(\mathbf{s}, \mathbf{a})$. Then, $V^\pi(\mathbf{s})$ could be defined by maximizing $Q^\pi(\mathbf{s}, \mathbf{a})$: $V^\pi(\mathbf{s}) = \max_{\mathbf{a}} Q^\pi(\mathbf{s}, \mathbf{a})$.

4.2.1 Dynamic Programming (DP)

Dynamic Programming is a general method to solve problems that present optimal substructure and overlapping subproblems. Of course, perfect knowledge of the transition model, as an MDP, is required. Due to their demand for high computational power and the fact that most of the environments fail to meet the requirement of a perfect model, in practice, they are of limited use in RL. However, the concepts introduced create the foundation for understanding other RL algorithms, actually most RL algorithms could be seen as approximations of DP.

For MDPs that meet these properties, Bellman equation [14] can be used to derive a recursive decomposition.

$$V^\pi(\mathbf{s}_t, \mathbf{a}_t) = \mathbb{E}_\pi [r_{t+1} + \gamma V^\pi(\mathbf{s}_{t+1} | \mathbf{s}_t)]$$

Policy Iteration (PI) is used to obtain an optimal solution. In every step it has two phases that are used alternatively: policy evaluation which evaluates a given policy π (that is the prediction problem) and policy improvement which is aiming to find an optimal policy (that is the control problem). The state value function converges to v^* and the policy value function to π^* . In Generalized Policy Iteration (PGI) any policy evaluation and any policy improvement can be used. Figure 18 and Figure 19 clarify PI and PGI.

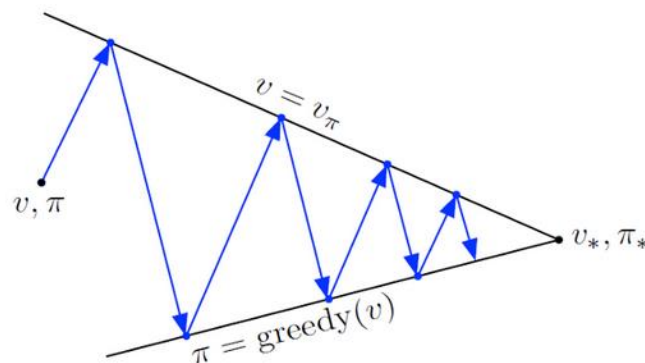


Figure 18. The two alternating phases of Policy Iteration to achieve optimal state values and policies²⁰

²⁰ <https://medium.com/gradientcrescent/fundamentals-of-reinforcement-learning-navigating-gridworld-with-dynamic-programming-9b98a6f20310>

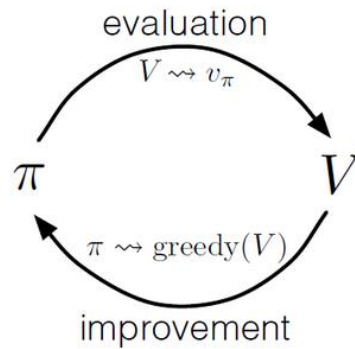


Figure 19. Generalized policy iteration²¹

4.2.2 Temporal Difference Methods (TD Methods)

Temporal Difference plays a central role in RL with value function evaluation [15]. Both SARSA and Q-learning are TD control methods. TD learning is a prediction problem. The action value function is:

$$Q^\pi(\mathbf{s}_t, \mathbf{a}_t) = \mathbb{E}_{\mathbf{s}_{t+1}} [r_{t+1} + \gamma Q^\pi(\mathbf{s}_{t+1}, \pi(\mathbf{s}_{t+1}))]$$

The above equation can be written in a recursive form as

$$Q^\pi(\mathbf{s}_t, \mathbf{a}_t) \leftarrow Q^\pi(\mathbf{s}_t, \mathbf{a}_t) + \alpha \delta = Q^\pi(\mathbf{s}_t, \mathbf{a}_t) + \alpha (Y - Q^\pi(\mathbf{s}_t, \mathbf{a}_t))$$

Where

α is the learning rate

$\delta = Y - Q^\pi(\mathbf{s}_t, \mathbf{a}_t)$ is the *Temporal Difference* (TD) error.

Y is the *TD target* as it represents an estimate for the true value of $Q^\pi(\mathbf{s}_t, \mathbf{a}_t)$ at the new iteration

So, a *bootstrapping* method as mentioned above can be used to improve iteratively the estimate by using the current value of the estimate of Q^π . Bootstrapping methods advantages are that they are fast in learning and the learning process is performed on line and continual.

This recursive form is the fundamental of Q-learning [16] and the state-action-reward-state-action (SARSA) algorithm [17].

²¹ <https://medium.com/gradientcrescent/fundamentals-of-reinforcement-learning-navigating-gridworld-with-dynamic-programming-9b98a6f20310> (taken from Sutton 2018)

SARSA is an *on-policy* learning algorithm developed to give improved estimates of Q^π by using transitions derived by the policy generated from Q^π (same policy), where

$$Y = r_{t+1} + \gamma Q^\pi(\mathbf{s}_{t+1}, \mathbf{a}_{t+1})$$

$$\delta = r_{t+1} + \gamma Q^\pi(\mathbf{s}_{t+1}, \mathbf{a}_{t+1}) - Q^\pi(\mathbf{s}_t, \mathbf{a}_t)$$

$$Q^\pi(\mathbf{s}_t, \mathbf{a}_t) \leftarrow Q^\pi(\mathbf{s}_t, \mathbf{a}_t) + \alpha (r_{t+1} + \gamma Q^\pi(\mathbf{s}_{t+1}, \mathbf{a}_{t+1}) - Q^\pi(\mathbf{s}_t, \mathbf{a}_t))$$

The last equation could be described as: the new estimate of Q^π derives from the previous Q^π plus a correction/difference defined by the reward and the value of Q for the new state, both of them being scaled appropriately by learning rate α and discount γ .

Q-learning is an *off-policy* learning algorithm developed to give improved estimates of Q^π by using transitions generated not necessarily by the derived policy. Now the TD error is

$$\delta = r_t + \gamma \max_{\mathbf{a}} Q^\pi(\mathbf{s}_{t+1}, \mathbf{a}_{t+1}) - Q^\pi(\mathbf{s}_t, \mathbf{a}_t)$$

$$Q^\pi(\mathbf{s}_t, \mathbf{a}_t) \leftarrow Q^\pi(\mathbf{s}_t, \mathbf{a}_t) + \alpha (r_{t+1} + \gamma \max_{\mathbf{a}} Q^\pi(\mathbf{s}_{t+1}, \mathbf{a}_{t+1}) - Q^\pi(\mathbf{s}_t, \mathbf{a}_t))$$

which directly approximates Q^* .

The main difference from SARSA is that Q-learning is an off-policy method, so it does not follow the current policy to choose the next action \mathbf{a}_{t+1} , while in SARSA an action \mathbf{a} is chosen by following a certain policy. It directly estimates Q^* out of the best Q values and \mathbf{a}^* is chosen by simply taking the max of Q over it. In the next step, Q-learning may not follow \mathbf{a}^* .

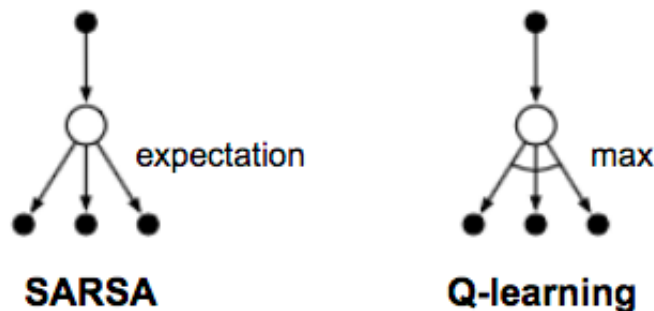


Figure 20. SARSA and Q-Learning approaches²²

²² <https://lilianweng.github.io/lil-log/2018/02/19/a-long-peek-into-reinforcement-learning.html> (replotted based on Fig 6.5 in [15])

4.2.3 Sampling, Monte Carlo and TD (λ) methods

An alternative to the bootstrapping dynamic programming and TD value function methods is to use Monte Carlo methods. These methods estimate the $\mathbb{E}[R|\mathbf{s}, \mathbf{a}, \pi]$ by averaging the return after running many rollouts of a policy adding random errors in the sequence. Due to this, Monte Carlo methods can be applied in non-Markovian environments. One drawback is that they can be applied only in episodic MDPs because the rollout must have finite number of steps in order to be possible to calculate the expected return after its end.

It is possible to combine both methods, i.e Temporal Difference methods and Monte Carlo methods in order to get the best of them, as it is done in the TD(λ) algorithm [13]. The λ is used to interpolate between bootstrapping and Monte Carlo evaluation.

Monte Carlo and TD methods (SARSA and Q-Learning) and are model-free methods while Dynamic Programming requires the transition model.

TD methods and Dynamic Programming use bootstrapping while Monte Carlo does not use it.

4.2.4 Policy Search methods

Policy search methods search directly for optimal π^* (with function approximation) instead of maintaining a value function. A parameterized policy π_θ is chosen and then either gradient-based or gradient-free optimization methods are applied in order to update the parameters and maximize the $\mathbb{E}[R|\theta]$. Policy-based methods in comparison to value-based methods, usually present better convergence, are effective in high-dimensional spaces and are suitable to learn stochastic processes. However, usually they suffer from converging to local optimum, present inefficiency to evaluate and encounter high variance [18].

Policy Gradients: The gradients provide a strong learning signal on how to improve a parameterized policy. The most known estimator of the gradient is the REINFORCE [19]. Neural Networks designed to encode policies have been successfully trained for policy search. [20]

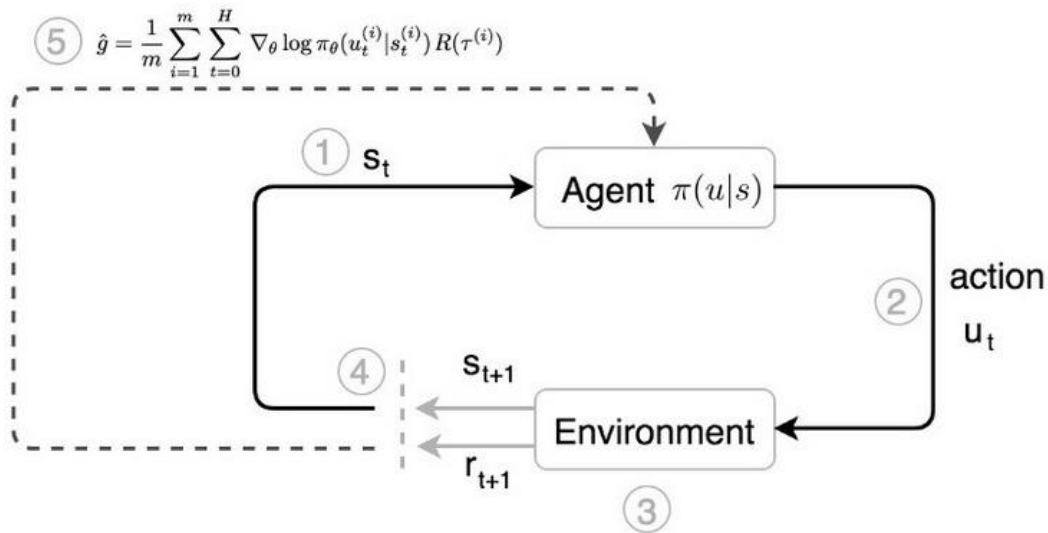


Figure 21 The Policy Gradient Method for Optimal Policy Search²³

4.2.5 Actor-Critic Methods

These methods are hybrid as they combine both the value function approach and an explicit representation of the policy as it is shown in the next figure. The critic realizes the value function component while the policy component is implemented in the actor. The actor has as learning inputs the state, received as feedback from the environment, and the Temporal Difference error from the critic. The actor is not receiving any reward directly but instead the reward is fed to the critic. The other input to the critic is the state. The actor-critic methods utilize a *learned* value function and this is the main difference in comparison with the other baseline methods used in classic policy search methods. The role of the critic is to update action-value function parameters while the actor updates the policy parameters [21].

²³ <https://jonathan-hui.medium.com/rl-policy-gradients-explained-9b13b688b146>

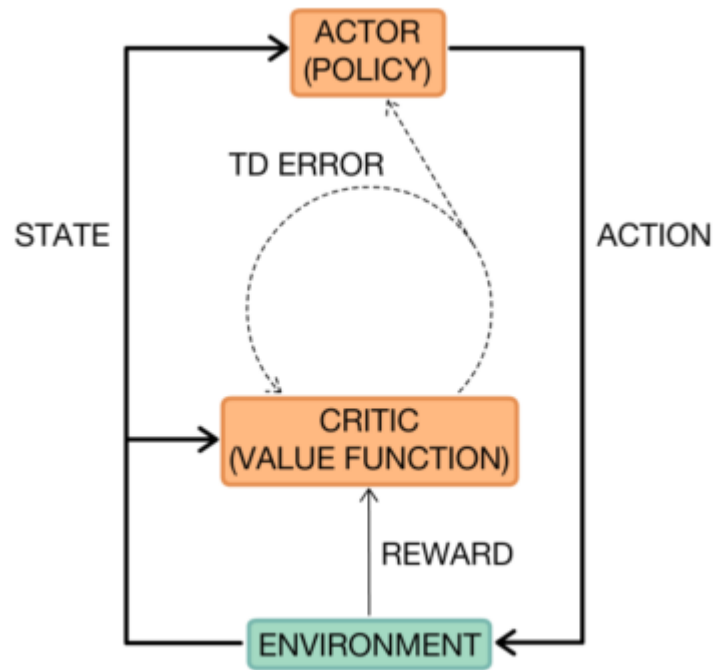


Figure 22. Actor -Critic Structure²⁴

4.3 Deep Reinforcement Learning

Although RL had already shown some successful applications in the past, the first approaches had inherent limited applicability to low-dimensional problems and were suffering from lack of scalability due to memory, computational and sampling complexity [22]. The development of deep learning, witnessed in recent years due to the *powerful function* approximation and *representation learning* properties of *deep neural networks*, supplied new tools towards overcoming these problems.

The most important property of deep learning is the ability of deep neural networks to significantly reduce the dimensions of the problem by finding low-dimensional representations (features) of high-dimensional data (e.g., audio, text and images, text). The use of deep learning algorithms within RL defined the field of deep reinforcement learning (DRL) and a significant accelerated progress occurred in the area of RL [23]. Deep learning provides tools enabling to cope with decision-making problems that were previously unmanageable, i.e., problems with high-dimensional state and action spaces. For example, Convolutional Neural Networks (CNN) can be part of RL agents permitting them to learn directly from high-dimensional visual raw inputs. In general, basic element of DRL is the

²⁴ in [31] recreated from [13]

utilization of deep neural networks which are trained to approximate the optimal policy π^* , and/or the optimal value functions such as V^* , Q^* .

4.3.1 Deep Q-Learning (DQN)

The manifesting difference between deep and “shallow” RL derives from the type of function approximator that is used. Neural networks are used for function approximation in deep RL, while linear and non-linear functions, decision trees (that may be non-linear), tile coding etc., are used in “shallow” RL. There have been reported *divergence* and *instability* problems when bootstrapping and function approximation (either linear or non-linear) have been combined [24]. However, subsequent works like deep Q-network [25] stabilized the learning and achieve outstanding results.

Mnih et. Al [25] introduced DQN and started the field of Deep RL. DQN uses a CNN with fully connected layers and ReLU activation function to approximate the function of optimal action value $Q^*(\mathbf{s}_t, \mathbf{a}_t)$.

$$Q^\pi(\mathbf{s}_t, \mathbf{a}_t) \leftarrow Q^\pi(\mathbf{s}_t, \mathbf{a}_t) + \alpha (r_{t+1} + \gamma \max_{\mathbf{a}} Q^\pi(\mathbf{s}_{t+1}, \mathbf{a}_{t+1}) - Q^\pi(\mathbf{s}_t, \mathbf{a}_t))$$

The DQN network uses the reward to update its estimate of Q, and backpropagates the error between the previous estimate and the new estimate.

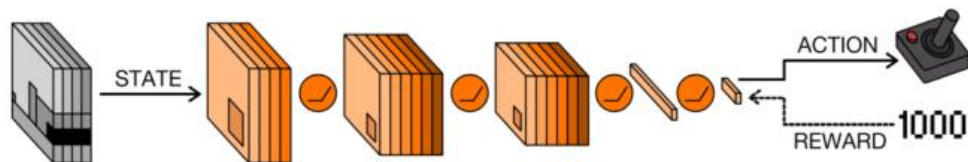


Figure 23 The structure of DQN with a CNN. (from [25])

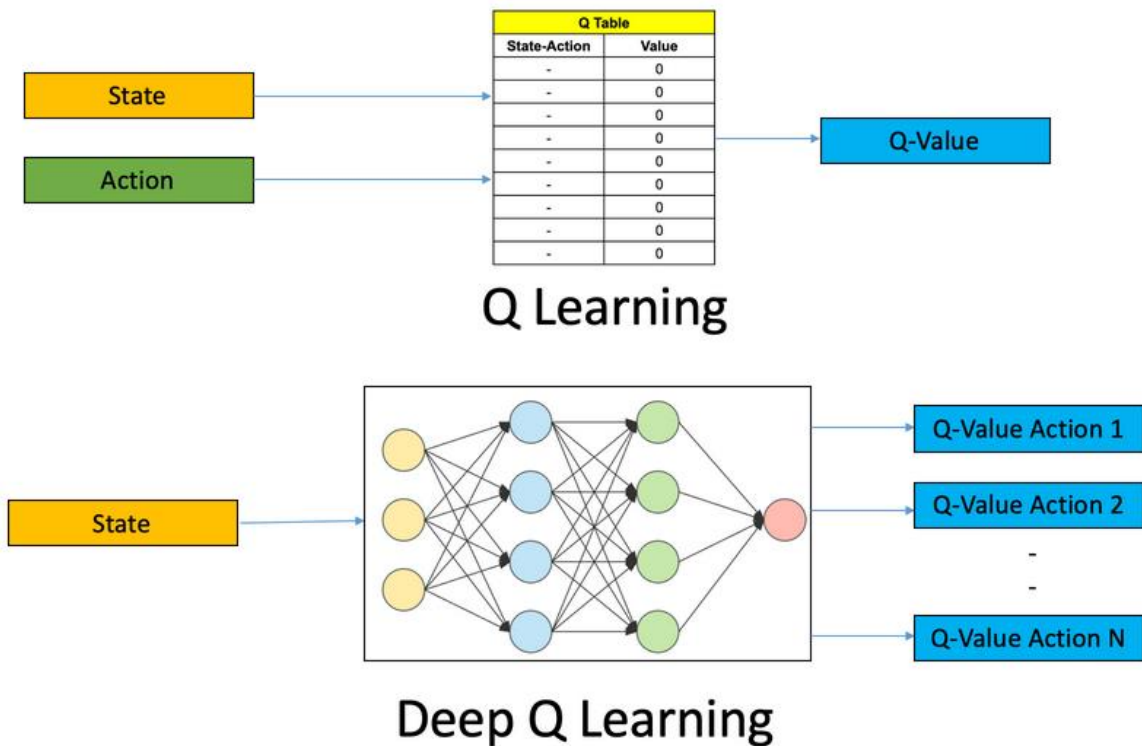


Figure 24. Q-Learning and Deep Q Learning (DQN)²⁵

DQN solved the instability and divergence problems mentioned above by using experience replay and target networks.

4.3.2 Asynchronous Advantage Actor-Critic (A3C) and Advantage Actor-Critic (A2C)

Asynchronous Advantage Actor-Critic (A3C) was proposed by Mnih et al. [26] as a simple architecture to exploit parallel processing and it is one of the most popular DRL techniques in recent years. A3C perform much more efficiently in comparison to the other asynchronous methods presented in [26], specifically the one-step SARSA, one-step Q-learning and n-step Q-learning.

Actor-Critic Part: In a typical Deep Convolution Q-Learning model, the output is a Q-value to be used for the candidate actions that the agent could pick for a given state. However, in A3C, the NN involved produces two outputs: one is the Q-values for the different actions and the second is the state value $V(s)$ which shows how good the action taken is. As mentioned previously, actor critic methods are combining characteristics of both policy methods and value function methods. The value function part is used for bootstrapping, i.e. to reduce

²⁵ <https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/>

variance and accelerate learning via updating the state according to subsequent estimates [15].

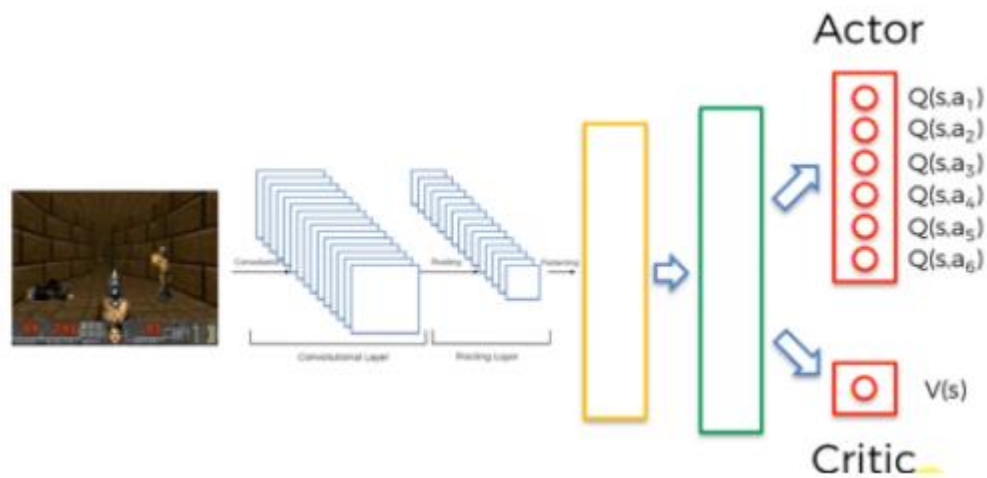


Figure 25. A3C schematic diagram²⁶

Asynchronous Part: In typical DQN there is a single agent represented by a single NN to interact with the environment. However, A3C employs more than one agents in parallel, being initialized differently, employing different exploration policies and sharing their experiences between them during the execution. The result of this asynchronous parallelism is a faster stabilized training with more accuracy-efficiency. Asynchronous methods are designed to run in distributed processing structures, however A3C has been developed to be implemented in both distributed and single multi-core CPU settings.

²⁶ <https://medium.com/analytics-vidhya/reinforcement-learning-with-a3c-20837aafe0ca>

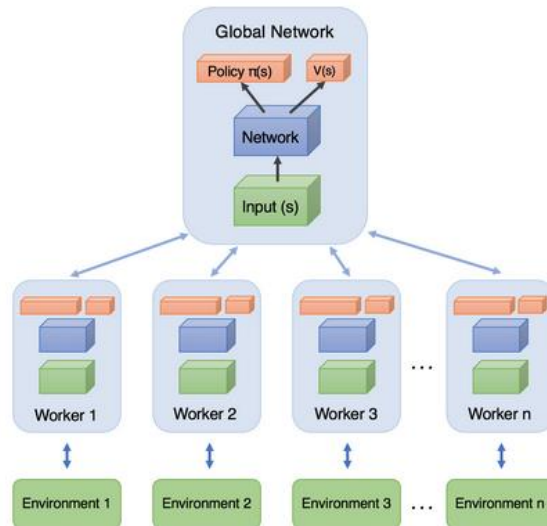


Figure 26 . A3C asynchronous parallel structure²⁷

Advantage Part: The role of this component of A3C is to calculate the advantage equation, i.e.

$$A(\mathbf{s}_t, \mathbf{a}_t) = Q(\mathbf{s}_t, \mathbf{a}_t) - V(\mathbf{s}_t)$$

The advantage function value expresses the amount of improvement that there is in a certain action $Q(\cdot)$ in comparison to the expected value $V(\cdot)$ of the state that was based on. The aim of the model is to maximize $A(\cdot)$.

The algorithm of A3C may be implemented without the asynchronous part, i.e. with *just one agent*, and in that case, it is called **advantage actor-critic (A2C)** [27].

²⁷ <https://medium.com/@shagunm1210/implementing-the-a3c-algorithm-to-train-an-agent-to-play-breakout-c0b5ce3b3405>

5 The Game

The game developed in this thesis belongs to the tower defense genre of games combined with RL.

5.1 Tower Defense

Tower defense games is a subcategory of real-time strategy games as the player's goal is the protection of his base against enemies by strategically placing defensive obstructions in the area that they are moving.

Usually, the enemies move in the area through a path, trying to reach a destination (base) which is significant for the player, e.g. house, possessions, loved ones etc. They can also follow multiple paths, can be damaged and killed, and can appear in waves, each of which usually has a certain number and type of enemies.

On the other hand, the player aims to prevent the enemies from achieving their goal by stopping, attacking or destroying the enemies. For that case, the defensive obstructions, e.g. a tower, can be placed everywhere in the game area, except from the enemies' path, and could damage the enemy. In modern tower defense games, some features are introduced in order to make the game more intriguing and engaging for the player. That could be the player's ability to upgrade or repair the obstruction and to collect virtual money (game-currency) or points in order to purchase advanced features for defeating the enemy.

Notable examples of tower defense games are: Flash Element TD, Iron Grip: Warlord, Facebook platform's Bloons TD.

5.2 Tools

Before the game development, many decisions had to be determined regarding the programming language as well as the libraries to both be convenient in developing the game as well as connecting the machine learning. The following choices have been made:

- *Python*: was found to be a great choice as is the major programming language for AI and ML. It has a great library ecosystem as well as numerous documentation and examples for RL algorithms.
- ML Libraries:
 - *Tensorflow*: is an open source machine learning and deep learning platform which can set up and train ANNs. Tensorflow was developed by Google.

- *Keras*: is a library that focuses on modern machine learning and allows calculations at high speed because it combines both CPU and GPU for processing.
- *Pygame*: is a cross-platform set of Python modules, designed for video games development.

Operation System	Windows 10, GPU: INTEL
Programming Language	Python 3.8.4
Libraries	Pygame 1.9.6 Keras 2.4.3 Tensorflow 2.3.1 NumPy 1.18.5

5.3 Game description

The game of this thesis was designed to have two perspectives of view. The first one is the perspective of the user, where, as in any tower defense game, the goal is to add defensive obstacles in order to prevent the enemy to reach the castle. The second perspective, on the other hand, is that of the enemy who wants to reach the castle. In order to achieve this goal, the enemy has to stay alive through the towers' attacks and to reach the castle. For that case, the enemy is trained in *real-time* with the *A2C* deep learning algorithm.

The basic elements are the enemy, the defensive towers and the castle. Figure 27 illustrates a screen of the game with all the elements.

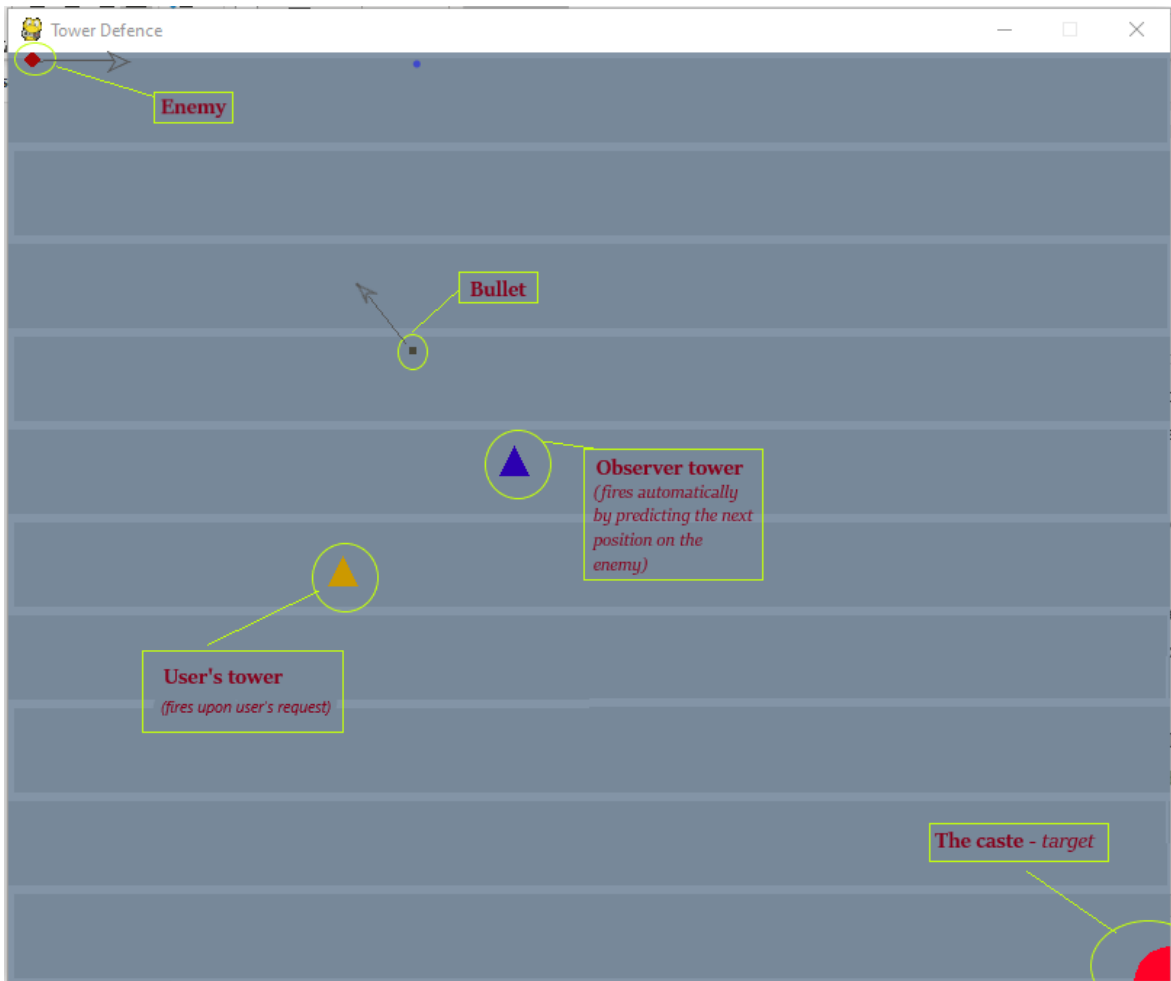


Figure 27. Game Elements

Before diving into the details of these elements shown in Figure 27, it is essential to state that the screen of the game is 800 x 640 pixels. In order to have a better understanding of the game's physics, i.e. the movement of the objects, all distances are calculated in meters by the following analogy: 800 x 640 pixels \rightarrow 50 x 40 meters.

In more detail, the elements and their functionalities in the game are:

- **path:** the only route that the enemy follows.
- In Figure 27 the path is the zig-zag light-greyed line.
- **castle:** the object the user must protect by the user
- **bullets:** moving element thrown by the towers to eliminate the enemy.
- It has constant velocity and heading.
- **enemy:** is being trained during the game (in real-time) to reach the user's castle and to avoid the bullets.
- The training is based on a Reinforcement Learning algorithm, and most specifically the A2C.

- He has a constant velocity magnitude, but its direction depends on the segment of the zig-zagged path.
- He has a health indication which is decreased every time a bullet hits him. As the enemy's health reduces, his color changes to a more lighter color. For example, if it turns to white color it needs just one more bullet hit to die.
- The enemy can know always the 2 closest bullets to him, i.e. bullets that are in a distance less than 5 meters.

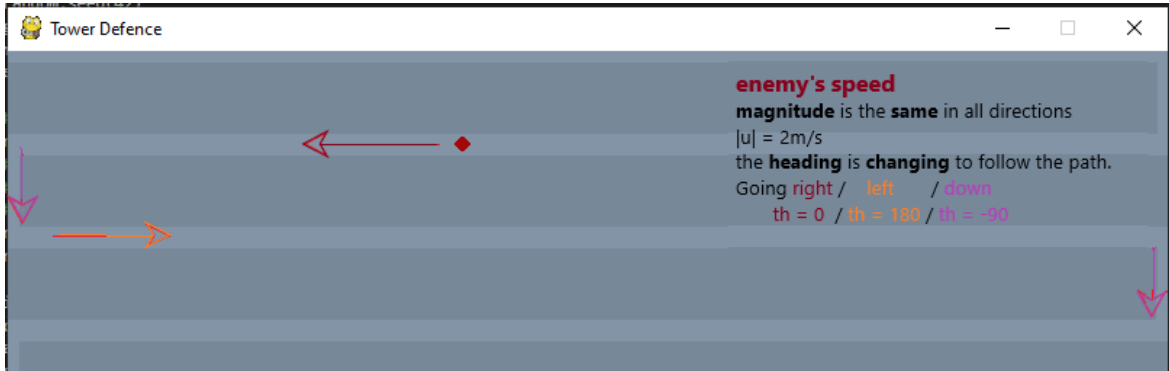


Figure 28. The enemy moving in the path.

- **observer tower**: is an automatically shooting tower. It always succeeds on dummy target, i.e. an object following a linear constant speed motion.
 - The user strategically places the observer tower in a position, which remains the same during the game session.
 - It can be created by right clicking the desired position.
 - It shoots one bullet per 1.5 seconds against the enemy.
 - The tower knows the exact position, the direction and magnitude of the speed of the enemy. It assumes that the enemy is moving in a linear motion with no acceleration and estimates in which direction to throw the bullet in order to collide with the enemy and consequently to reduce his health.

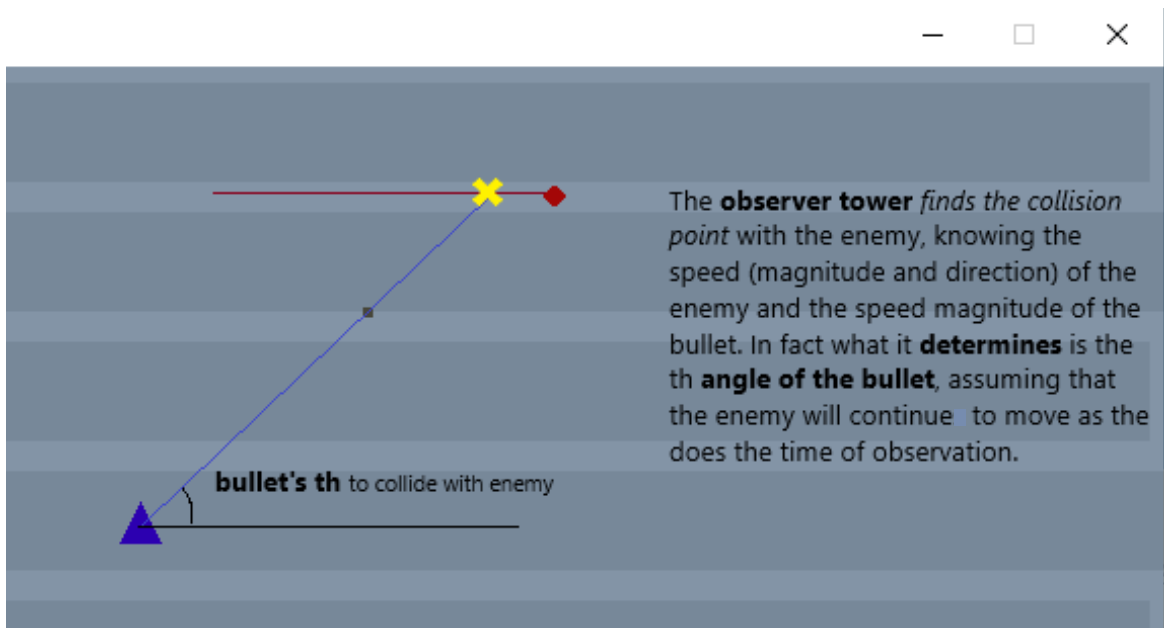


Figure 29. Observer Tower predicting the collision point and fires a bullet

- **user's tower**: is a tower that fires upon user's request.
 - The user strategically places "his" tower in a position, which remains the same during the game session.
 - It can be created by right clicking the desired position.
 - It fires the bullets when the user clicks on the screen. The bullet is heading towards the position that the user clicked.

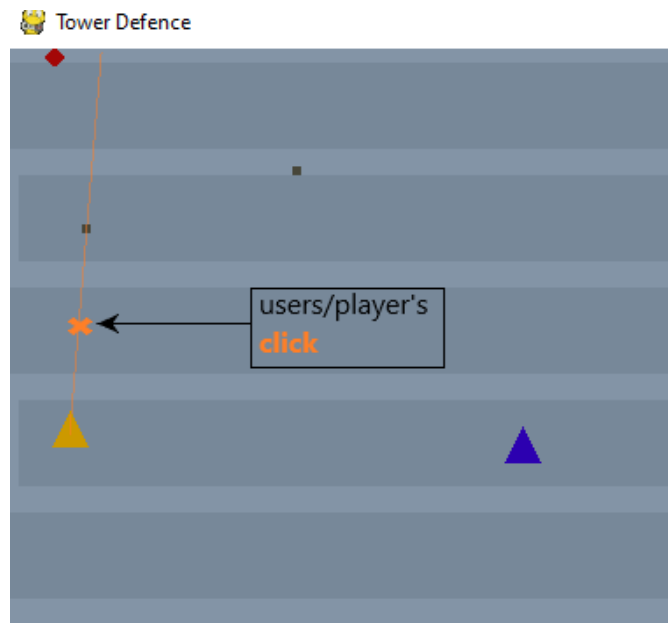


Figure 30. User tower fires upon click

- Game **coins**: are collected by the user and can be used to buy a new observer castle.
 - The coins are collected when the enemy is killed. (1 point per kill)

- When the game starts the user is able to buy one observer and one user's tower.
- One observer tower cost 100 coins.

Another point is that the enemy can be hit by the bullet, but a collision cannot exist between other objects, i.e. towers with bullet, bullets with bullets and tower with tower. The collision check is done by a continuous loop for all bullets and the enemy. The time complexity of this operation is $O(\#enemy \times \#bullets)$. Since there is only one enemy, the time complexity depends only on the number of bullets.

Moreover, it should be noted that a tower is prohibited to be placed in the path of the enemy. So, the user cannot place any of them in such a position.

To sum up, when the game starts, the enemy starts learning to move forward. When the player adds a tower and bullets are fired at the enemy, the latter learns to avoid them. The player gets a coin for every killed enemy, so that every time that he collects 100 coins he can buy an observer tower.

5.4 Training the enemy to win the player

The enemy is trained online in order to stay alive by avoiding the bullets and to reach the castle. Despite the fact that A3C could be the best RL algorithm for this case, it could add delays and make the game heavier and more unresponsive during the real-time training. For that reason, the A2C algorithm was chosen for this game.

5.4.1 Setting up the actor and the critic

The model used is that the actor and critic components utilize two feedforward NNs that share a hidden network. After multiple tests, it was observed that the optimal size of layers for the hidden network is *16 units*. Moreover, it was found that the best combination of the activation functions for the actor and common hidden layer is the *softmax* and the *tanh* respectively.

```
inputs = layers.Input(shape=(num_inputs,))
common = layers.Dense(num_hidden, activation="tanh")(inputs)
action = layers.Dense(num_actions, activation="softmax")(common)
critic = layers.Dense(1)(common)
```

Code segment 1. Initializing the NNs

5.4.2 Taking an action

The enemy has two options of acting: going forward (0) or going backwards (1).

```
#going forward or backward in the path.  
# 0 : going forward  
# 1 : going backwards  
num_actions = 2
```

Code segment 2. Definition of actions.

The enemy moves along the path and his position is defined by the distance he covered from the starting point. This path distance is affected by the actions decided by the actor. In the program, it is indicated as property “p” in the enemy class.

In more detail, in order the actor (enemy) to take a decision, he requires to know the state s_t of the environment at that moment which in our case consists of the following:

- The position and speed of the two closest bullets. A bullet is considered to be a threat if it is closer to him than 5 meters.
- His own path distance, his coordinates and his speed in the game window.

```
tc_bullets = e.find_two_closest_bullets(all_bullets)  
action = eb.take_an_action(tc_bullets)  
# the distance path is affected by the action  
e.p += ((-1)** action) * e.r_and_u.u.magnitude * dt
```

Code segment 3. Action effect in distance path

Because the state of the environment, that the enemy needs to know, is not requiring to “seeing” the whole picture of the game, a *CNN is not needed* in this implementation.

To sum up, the state is an array of 13 variables [enemy_p, enemy_x, enemy_y, enemy_speed_x, enemy_speed_y, bullet1_x, bullet1_y, bullet1_speed_x, bullet1_speed_y, bullet2_x, bullet2_y, bullet2_speed_x, bullet2_speed_y].

5.4.3 Reward logic

The action is *rewarded*:

- Highly, if the enemy reached the castle.
- Lightly, if he moves forwards, as the castle is forwards.

The action, though, can receive a *penalty*:

- Very small, if he is going backwards.
- Medium, when he is being hit.

- Medium, when he dies.

Going backwards is a move absolutely acceptable and necessary to avoid bullets and initially it was intended to have a zero or a very small positive reward. After an amount of trials, the phenomenon of exploitation vs. exploration was observed, especially when the enemy had been hit many times or/and the hit penalty was slightly big. What happened was that, in order to stay alive, he made small steps back and forth in the same spot, and sometimes he even moved backwards all the way back to the starting point. After this oscillating behavior, a decision was taken to have very small negative reward when moving backwards.

In addition, many other tests were performed to determine the penalty values in case of hit and death of the enemy. Big negative values in death and hit lead to exploitation behavior. Also, please note, that when he dies, he is first hit so in that case the death penalty is additive to hit penalty.

After multiple value combinations, the set of the reward and penalty values that resulted in giving the best behavior is:

```
reward_reach_the_castle = 100
reward_moving_forward = 2
reward_moving_backwards = -0.1
penalty_hit = -10
penalty_death = -10
```

Code segment 4. Reward and penalty values

5.4.4 Learning logic

The enemy learns, otherwise updates his mind, when:

- The episode ends
- The enemy is killed

When something of the above occurs, the critic is evaluating the actions that were taken during that period and updates the probability of the actions. For this purpose, firstly the critic calculates the expected values from the rewards, then the loss values to update the network with the Mean Squared Error metric and, subsequently, backpropagates the results with the Adam optimizer, which is an extension of stochastic gradient descent. Additionally, the discount factor of future rewards is $\gamma = 0.99$ and the learning rate is $a = 0.05$.

An episode is defined by a specific number of steps which, in the case of this game, is defined by each time (dt) the enemy *must* take an action to move. The number of steps resulting to optimal behavior is 200.

5.4.5 Run Tests

The game was executed 20 times in order to visualize the reward evolution per episode and per step for the total number of 50 episodes, which need totally 10000 steps. In order to have more consistent results, the observer tower is placed in the same position in every run:



Figure 31. Observer tower's constant position in tests

The following figures [Figure 32, Figure 33] illustrate the average accumulated reward and average cumulative accumulated reward per episode, while [Figure 34, Figure 35] show the average rewards and average cumulative reward per step.

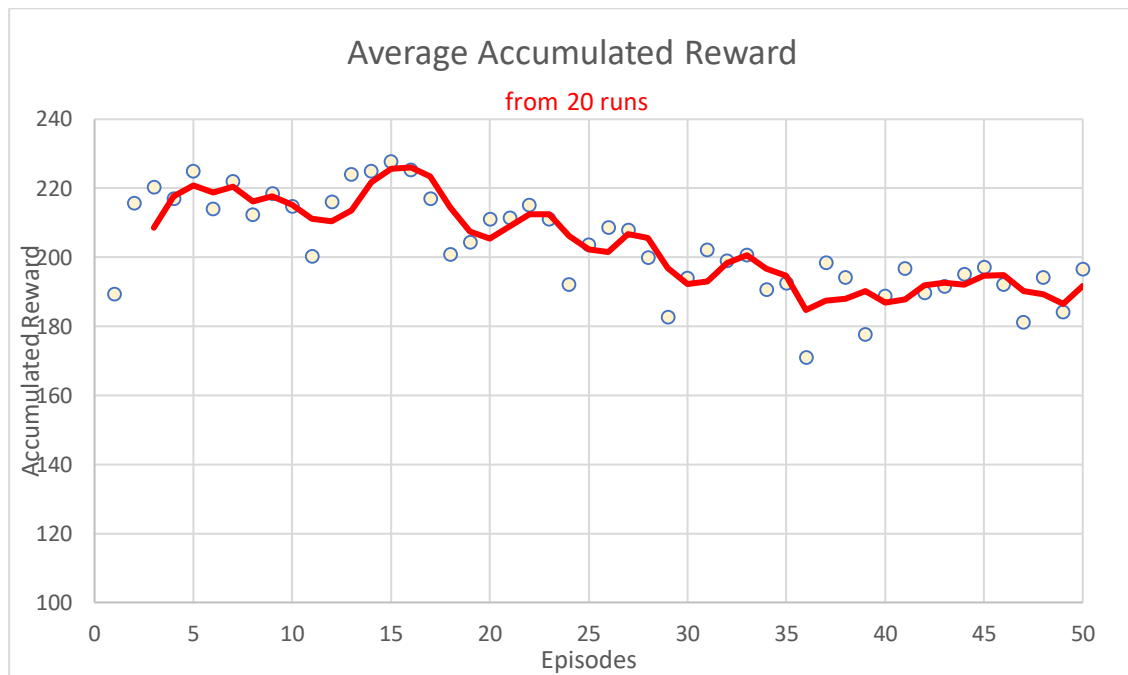


Figure 32. Average reward per episode

Figure 32 shows that in first episode the rewards are better than the next ones. The reason for this behavior is that the enemy has not yet found out the best direction to go, as the observer assumes that the enemy will continue to move in the same direction and speed. So, the average reward for almost the first 17 episodes demonstrated exactly what the enemy is trying to learn; going forward is a profitable action. On average around episode 17 the reward starts to reduce. This is the first indication that the agent has learnt to go right, as now the observer can make a correct estimation of the collision point. After that the reward is unstable and reaches a low value around episode 36. Though the next episodes indicate a promising slow reward increment, which means that the agent is starting to find ways to avoid the bullets. The low responsive to the bullet hit is expected as the bullets can approach the enemy from many angles and the training needs more time.

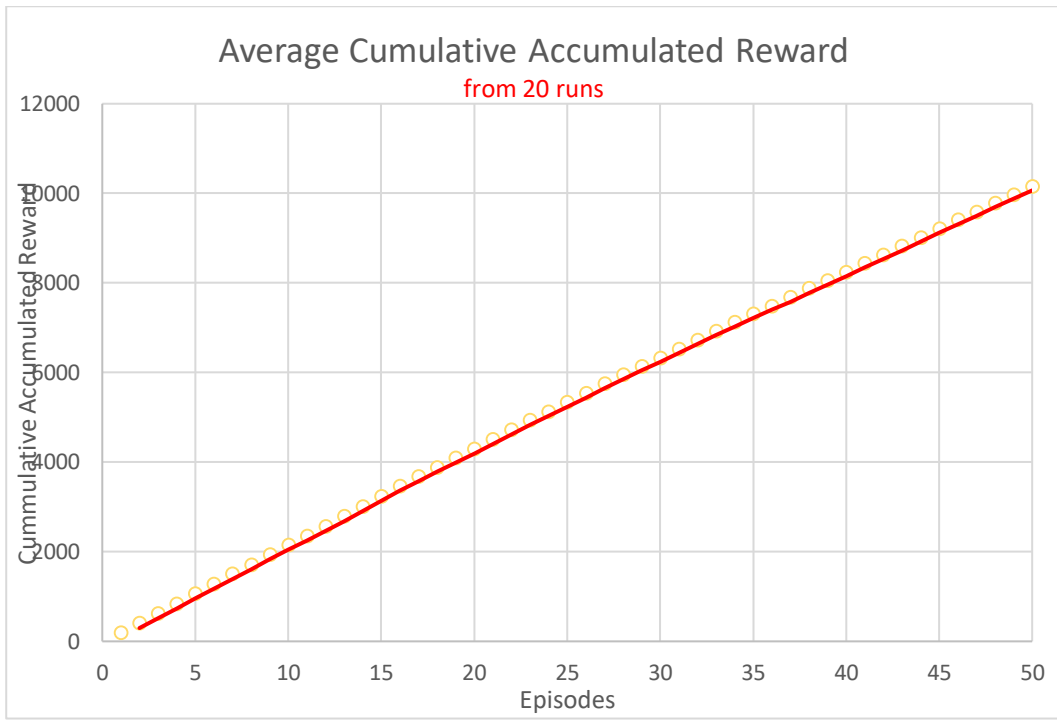


Figure 33. Average Cumulative rewards per episode

The cumulative reward, seen in Figure 33, demonstrates the same result, as from approximately episode 20 the slope of the line is smaller.

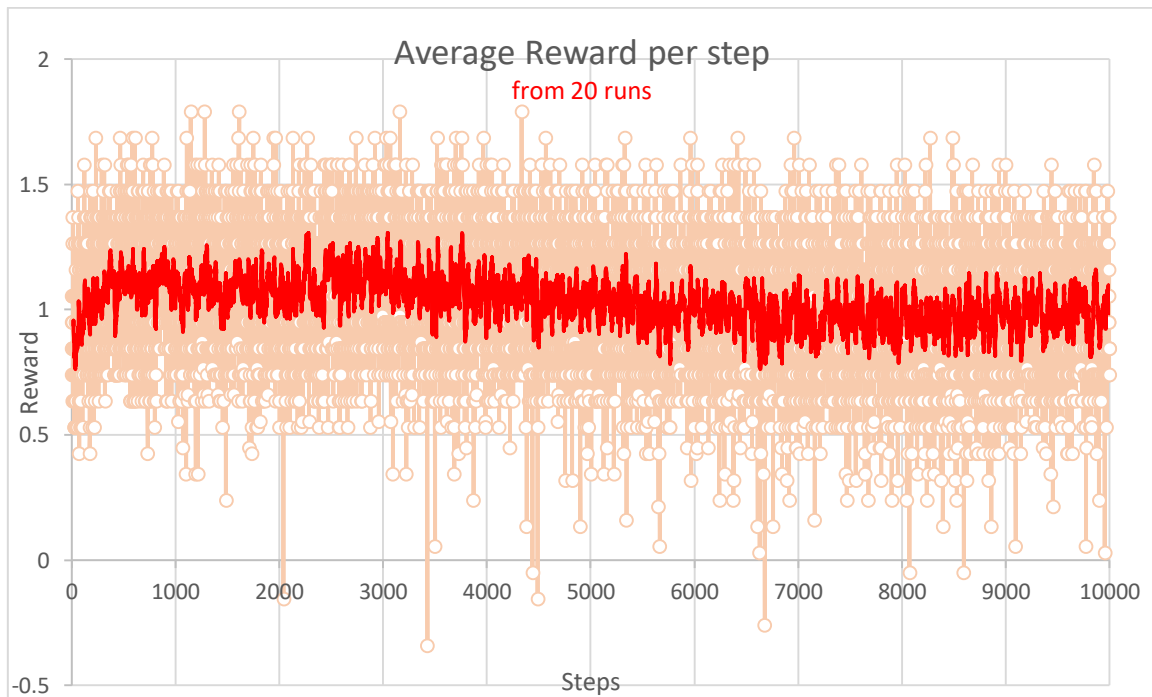


Figure 34. Average reward per step

Figure 34 shows that the gained rewards per step are widely variant. Similarly, to the previous plots, it has the same behavior after the 3500 steps like the reward per episode. Figure 35 demonstrates an almost linear increase of the reward.

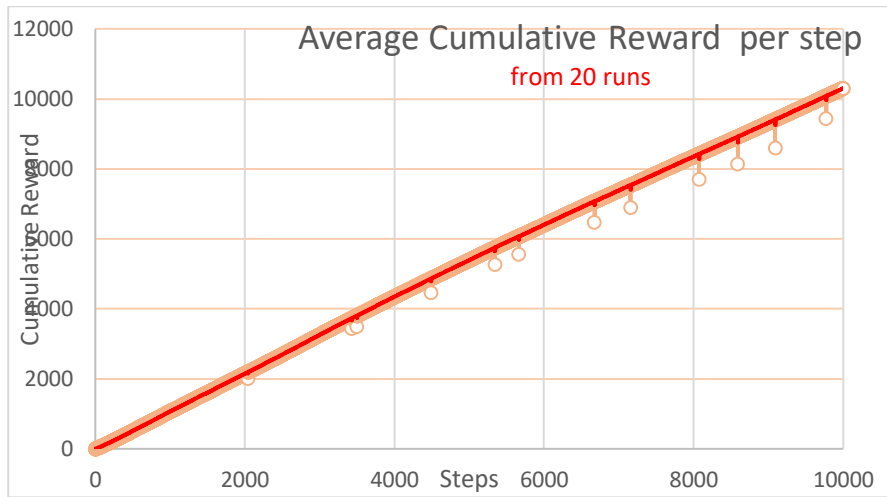


Figure 35. Average cumulative reward per step

An example of a well-trained agent can be seen in Figure 36 where the agent has passed very close to the observer tower and managed to trick him and avoid his bullet. After the predicted collision point 2, though, the enemy is repeatedly hit. The cause of this “failure” is that he had not been trained yet to avoid bullets with small angle, like th2 or smaller.

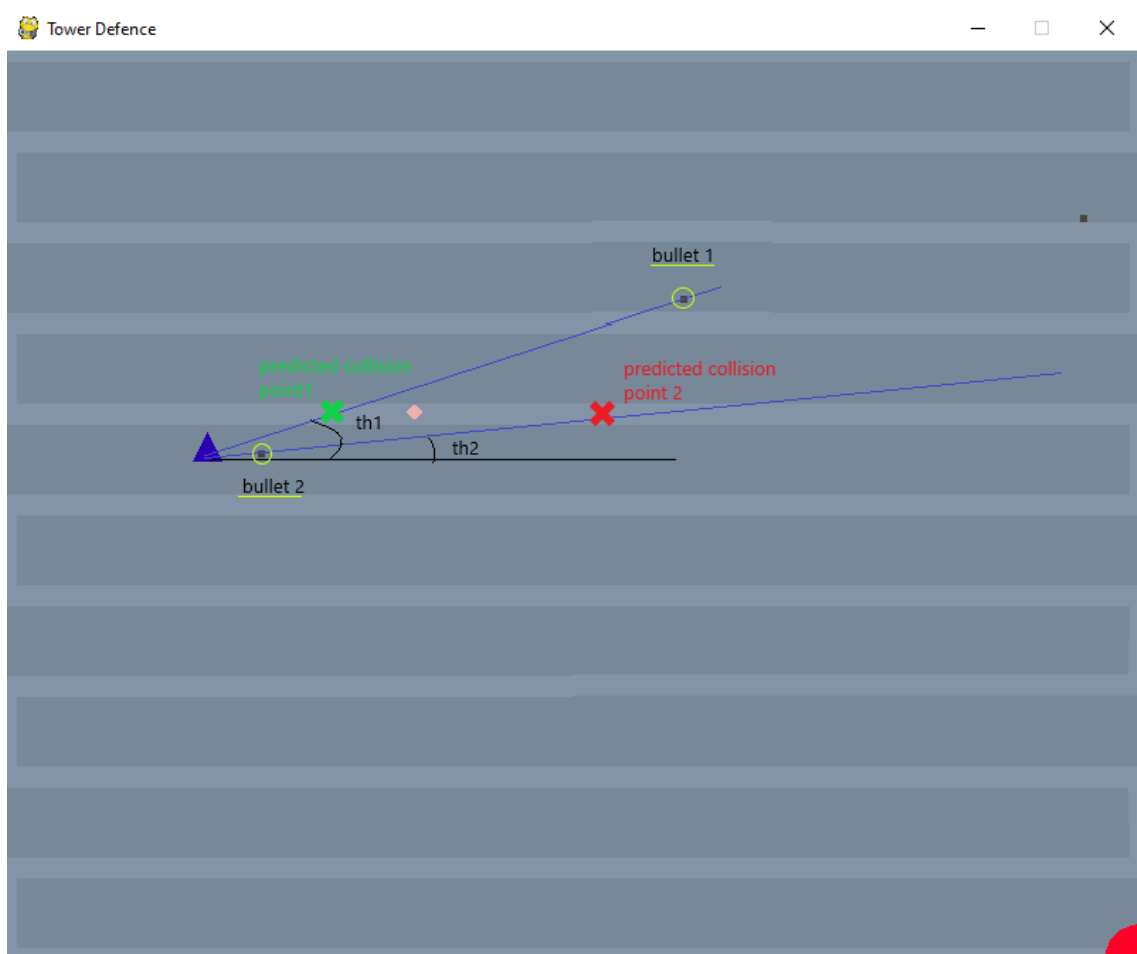


Figure 36. Well-trained enemy passes the observer tower

6 Conclusions

The tower defense game that was developed herein employed the A2C reinforcement learning algorithm to provide the enemy with intelligence to reach the castle by avoiding the bullets. The chosen development environment is composed by Python, TensorFlow, Keras and Pygame. The combination of them proved to be very efficient in developing the code, running and testing the algorithms of the game. The values of the algorithm's parameters were tuned after multiple tests. The algorithm implemented performs sufficiently but there is a lot of space for improvement.

There could be more research to determine the most efficient parameter and network configuration to help enemies learn better. In the spirit of making the enemies more clever, an LSTM NN could also be used to help the agents to make decisions based on their memory.

An additional improvement would be to have more enemies which would be trained asynchronously to reach the castle. This would be achieved with the A3C algorithm. Four of the enemies (agents) would be trained simultaneously and their "brain" would be updated every time an episode ends by exchanging their experience. This algorithm would have quicker training response and would advance the enemies' knowledge and capability, making thus the game more interesting to play. Of course, more than four enemies could try to reach the castle, but not all of them could be trainable, as there are limitations depending on the CPU's number of cores/maximum threads.

The need for another improvement would emerge in the case of having many agents/enemies, that is the enhancement of the collision detecting process. As it was mentioned before, the time complexity of the current implementation is $O(\#enemies \times \#bullets)$. This issue could be tackled with python's sprites.

An interesting idea regards the enemies' path which could be defined by splines in order to be a smooth curve instead of zig-zag.

Another game improvement would be to train the observer tower as well. In that way, the game would have two elements with trainable brain competing each other. It will be very interesting to see what defense mechanisms and tricks each opponent part will develop to gain more rewards. This training would require a CNN as the tower would have to analyze images from the game screen.

References

- [1] T. M. Mitchell, *Machine Learning*, 1997.
- [2] R. Bhardwaj, V. S. Dixit and A. Upadhyay, "A Fuzzy Intra-Clustering Approach for Load Balancing in Peer-to-Peer System," *Journal of Information and Computing Science*, vol. 7, no. 1, pp. 19-24, 2011.
- [3] W. McCulloch and W. Pitts, "A Logical Calculus of Ideas Immanent in Nervous Activity," *Bulletin of Mathematical Biophysics*, vol. 5, pp. 115-133, 1943.
- [4] F. Rosenblatt, "The Perceptron: A Probabilistic Model For Information Storage And Organization In The Brain," *Psychological Review*, p. 386-408, 1958.
- [5] M. Minsky and S. Papert, *An Introduction to Computational Geometry*, MIT Press, 1969.
- [6] P. Werbos, *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*, Thesis (Ph. D.)-Harvard University, 1975.
- [7] M. Schuster and K. K. Paliwal, "Bidirectional recurrent neural networks," *IEEE Transactions on Signal Processing*, vol. 45, no. 11, p. 2673-2681, 1997.
- [8] Y. Bengio, P. Simard and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE Transactions on Neural Networks*, vol. 5, no. 2, p. 157-166, 1994.
- [9] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, p. 1735-1780, 1997.
- [10] K. Cho, B. v. Merriënboer, D. Bahdanau and Y. Bengio, "On the properties of neural machine translation: Encoder-decoder approaches," *In Machine Learning and Knowledge Discovery in Databases*, 2014.
- [11] G. Weiss, Y. Goldberg and E. Yahav, "On the Practical Computational Power of Finite Precision RNNs for Language Recognition," in *ACL*, 2018.
- [12] D. Britz, A. Goldie, M.-T. Luong and Q. Le, "Massive Exploration of Neural Machine Translation Architectures," in *2017 Conference on Empirical Methods in Natural Language Processing*, 2017.
- [13] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, MIT Press, 1998.

- [14] R. Bellman, "On the Theory of Dynamic Programming," *PNAS*, vol. 38, no. 8, pp. 716-719, 1952.
- [15] R. Sutton and A. G. Barto., *Reinforcement Learning: An Introduction (2nd Edition)*, MIT Press, 2018.
- [16] C. J. Watkins and P. Dayan, "Q-Learning," *Machine Learning*, vol. 8, no. 3-4, pp. 279-292, 1992.
- [17] G. A. Rummery and M. Niranjan, *On-Line Q-Learning Using Connectionist Systems*, Cambridge University, 1994.
- [18] D. Silver, "UCL reinforcement learning course," [Online]. Available: <http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html>. [Accessed 10 10 2020].
- [19] R. Williams, "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning," *Machine Learning*, vol. 8, no. 3-4, pp. 229-256, 1992.
- [20] F. Gomez and J. Schmidhuber, "Evolving Modular Fast-Weight Networks for Control," 2005, ICANN.
- [21] J. Schulman, P. Moritz, S. Levine, M. Jordan and P. Abbeel, "High-Dimensional Continuous Control using Generalized Advantage Estimation," in *ICLR*, 2016.
- [22] A. Strehl, L. Li, E. Wiewiora, J. Langford and M. Littman., "PAC Model-Free Reinforcement Learning," in *IICML*, 2006.
- [23] Y. Li, "Deep Reinforcement Learning: An Overview," 2017. [Online]. Available: [arXiv:1701.07274](https://arxiv.org/abs/1701.07274). [Accessed 09 10 2020].
- [24] J. Tsitsiklis and B. V. R. , "An analysis of temporal-difference learning with function approximation," *IEEE Transactions on Automatic Control*, vol. 42, no. 5, pp. 674-690, 1997.
- [25] V. Mnih, K. Kavukcuoglu, D. Silver, A. Rusu, J. Veness, M. Bellemare, A. Graves, M. Riedmiller, A. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie and A.Sadik, "Human-Level Control through Deep Reinforcement Learning," *Nature*, vol. 518, no. 7540, pp. 529-533, 2015.
- [26] V. Mnih, A. P. Badia, M. Mirza, A.Graves, T. Lillicrap, T. Harley, D. Silver and K. Kavukcuoglu., "Asynchronous Methods for Deep Reinforcement Learning," in *ICLR*, 2016.

- [27] J. Wang, Z. Kurth-Nelson, D. Tirumala, H. Soyer, J. Leibo, R. Munos, C. Blundell, D. Kumaran and M. Botvinick, "Learning to Reinforcement Learn," in *CogSci 2017*, 2017.
- [28] S. Mishra, "Unsupervised Learning and Data Clustering," 20 05 2017. [Online]. Available: <https://towardsdatascience.com/unsupervised-learning-and-data-clustering-eeecb78b422a>.
- [29] J. Chung, C. C. K. Gulcehre and Y. Bengio, "Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling,," in *NIPS : Deep Learning and Representation Learning Workshop*, 2014.
- [30] W. Feng, N. Guan, Y. Li, X. Zhang and Z. Luo, "Audio visual speech recognition with multimodal recurrent neural networks," pp. 681-688, 2017.
- [31] K. Arulkumaran, M. P. Deisenroth, M. Brundage and A. A. Bharath, "A Brief Survey of Deep Reinforcement Learning," *IEEE Signal Processing Magazine*, vol. 34, no. 6, pp. 26-38, 2017.

Appendix

The code of this thesis was developed in two files:

- ***tower_defence.py***: it is the main file in which the game starts, and every action of the game is taken, and where the development of A2C is located. The part of the code that is merely the latter's development is annotated.
- ***entities.py***: a file where all classes of the objects (enemy, physics, bullets etc) of the game are placed.

Tower_defence.py

```
import math
import random
import pygame
import entities
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

import sys

myscreen = entities.Screen_attributes()
RIGHT = 3
LEFT = 1

screen = pygame.display.set_mode((myscreen.width, myscreen.height),pygame.DOUBLEBUF)
pygame.display.set_caption('Tower Defence')

##### A2C #####
#####
#A2C: setting up the nn networks and parametres
random.seed(42)
gamma = 0.99 # Discount factor for past rewards
max_steps_per_episode = 200
eps = np.finfo(np.float32).eps.item() # Smallest number such that 1.0 + eps
!= 1.0

# actor critic network
num_inputs = 1 + (2 + 2) + (2 + 2) + (2 + 2)
#going forward or backward in the path.
# 0 : going forward
# 1 : going backwards
```



```

num_actions = 2
num_hidden = 16

inputs = layers.Input(shape=(num_inputs,))
common = layers.Dense(num_hidden, activation="tanh")(inputs)
action = layers.Dense(num_actions, activation="softmax")(common)
critic = layers.Dense(1)(common)

model = keras.Model(inputs=inputs, outputs=[action, critic])
#####

class eBrain:
    def __init__(self, enemy, trainable ):
        self.optimizer = keras.optimizers.Adam(learning_rate=0.01)
        self.mse_loss = keras.losses.MeanSquaredError()
        self.action_probs_history = []
        self.critic_value_history = []
        self.rewards_history = []
        self.running_reward = 0
        self.episode_count = 0
        self.me_the_enemy = enemy
        self.trainable = trainable
        self.step_reward = 0

    def take_an_action (self, tc_bullets):
        bstates = []
        bcnt = 0
        for b in tc_bullets:
            bstates = bstates + b.to_state_vector()
            bcnt = bcnt + 1

        while bcnt < 2:
            bstates = bstates + [0, 0, 0, 0]
            bcnt = bcnt + 1
        estate = self.me_the_enemy.to_state_vector()

        # reset the environment
        state = [self.me_the_enemy.p] + estate + bstates

        state = tf.convert_to_tensor(state)
        state = tf.expand_dims(state, 0)

        # Predict action probabilities and estimated future rewards
        # from environment state
        action_probs, critic_value = model(state)
        if self.trainable:
            self.critic_value_history.append(critic_value[0, 0])

        # Sample action from action probability distribution

```

```

        action = np.random.choice(num_actions, p=np.squeeze(action_probs))
        if self.trainable:
            self.action_probs_history.append(tf.math.log(action_probs[0, action]))

    return action

def learn(self, episode_reward):
    if not self.trainable:
        return
    # Update running reward to check condition for solving
    self.running_reward = 0.05 * episode_reward + (1 - 0.05) * self.running_reward

    # Calculate expected value from rewards
    # - At each timestep what was the total reward received after that timestep
    # - Rewards in the past are discounted by multiplying them with gamma
    # - These are the labels for our critic
    returns = []
    discounted_sum = 0
    for r in self.rewards_history[::-1]:
        discounted_sum = r + gamma * discounted_sum
        returns.insert(0, discounted_sum)

    # Normalize
    returns = np.array(returns)
    returns = (returns - np.mean(returns)) / (np.std(returns) + eps)
    returns = returns.tolist()

    # Calculating loss values to update our network
    history = zip(self.action_probs_history, self.critic_value_history, returns)
    actor_losses = []
    critic_losses = []
    for log_prob, value, ret in history:
        # At this point in history, the critic estimated that we would get a
        # total reward = `value` in the future. We took an action with log
        # probability
        # of `log_prob` and ended up receiving a total reward = `ret`.
        # The actor must be updated so that it predicts an action that leads to
        # high rewards (compared to critic's estimate) with high probability.
        diff = ret - value
        actor_losses.append(-log_prob * diff) # actor loss

```

```

        # The critic must be updated so that it predicts a better estimate of
        # the future rewards.
        critic_losses.append(
            self.mse_loss(tf.expand_dims(value, 0), tf.expand_dims(ret, 0))
        )

    # Backpropagation
    loss_value = sum(actor_losses) + sum(critic_losses)
    grads = tape.gradient(loss_value, model.trainable_variables)
    self.optimizer.apply_gradients(zip(grads, model.trainable_variables))

    # Clear the loss and reward history
    self.action_probs_history.clear()
    self.critic_value_history.clear()
    self.rewards_history.clear()

#####

def append_enemies (smart_enemies, trainable_enemy_exists):

    trainable = False
    if random.random() < 0.5 and len(smart_enemies) < max_enemies:
        e = entities.Enemy((0, 0))

        # first enemy is trainable or there is no trainable agent
        if not trainable_enemy_exists :
            trainable = True

        smart_enemies.append( eBrain(e,trainable) )
    return trainable or trainable_enemy_exists

def __del__ (self):
    pass

def active_bullets_after_collision_checks (tower, smart_enemies, episode_reward, steps_count, c_episode_reward, c_step_reward):
    new_bullets = []
    hit_bullets = []
    for b in tower.bullets:
        b.move(dt)
        if b.is_in_screen():
            new_bullets.append(b)

    # the first enemy is always trainable
    trainable_enemy_exists = True
    for b in tower.bullets:
        for eb in smart_enemies:

```

```

        e = eb.me_the_enemy
        if e.is_hit(b) and not b in hit_bullets:
            e.subtrack_health()
            eb.step_reward += penalty_hit
            if (e.health<0):
                if eb.trainable:
                    eb.step_reward += penalty_death
                    ed.rewards_history.append(ed.step_reward)
                    episode_reward += ed.step_reward
                    eb.learn(episode_reward)
                    trainable_enemy_exists = False
                    episode_reward = 0
                    steps_count = 0
                    eb.trainable = False
                smart_enemies.remove(eb)
            hit_bullets.append(b)

    for b in hit_bullets:
        if b in new_bullets:
            new_bullets.remove(b)
            #del b

    return new_bullets, trainable_enemy_exists , episode_reward, steps_count,
    c_episode_reward, c_step_reward

running = True
enemies = []
smart_enemies = []
clock=pygame.time.Clock()
FRAMES_PER_SECOND=30
stopwatch_at = 0 #secs
stopwatch_timer = 0
stopwatcht_at_bullet = 2 #sec
stopwatch_timer_bullet = 0
max_enemies = 1

reward_reach_the_castle = 100
reward_moving_forward = 2
reward_moving_backwards = -0.1
penalty_hit = -10
penalty_death = -10

observer_towers = []
user_towers = []

trainable_enemy_exists = False

```

```

episode_reward = 0
steps_count = 0
episodes_count = 0
target = entities.PaintableObject()

c_episode_reward = 0
c_step_reward = 0

while running :

    with tf.GradientTape() as tape:

        dt=clock.tick(FRAMES_PER_SECOND)/1000.0 # number of seconds have
passed since the previous call.
        stopwatch_timer += dt
        stopwatch_timer_bullet += dt

        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                running = False
            elif event.type == pygame.MOUSEBUTTONDOWN and event.button == R
IGHT and not observer_towers:
                pos = pygame.mouse.get_pos()
                observer_tower = entities.Observer_tower(pos)
                if (myscreen.is_tower_position_allowed_simple(observer_t
ower)):
                    observer_towers.append(observer_tower)
            elif event.type == pygame.MOUSEBUTTONDOWN and not user_towe
rs:
                pos = pygame.mouse.get_pos()
                tower = entities.Tower(pos)
                if (myscreen.is_tower_position_allowed_simple(tower)):
                    user_towers.append(tower)
            elif event.type == pygame.MOUSEBUTTONDOWN and user_towers:
                (x,y) = pygame.mouse.get_pos()
                tower.make_bullet(x,y)

        #add a smart agent
        if stopwatch_timer >= stopwatch_at:
            trainable_enemy_exists = append_enemies(smart_enemies, traina
ble_enemy_exists)

            stopwatch_timer = 0

            new_smart_enemies = []
            two_closest_bullets = []
            for eb in smart_enemies:
                e = eb.me_the_enemy

```

```

tc_bullets = []
all_bullets = []

for ot in observer_towers:
    all_bullets = all_bullets + ot.bullets
for ut in user_towers:
    all_bullets = all_bullets + ut.bullets
tc_bullets = e.find_two_closest_bullets(all_bullets)

action = eb.take_an_action(tc_bullets)

e.p += ((-1)** action) * e.r_and_u.u.magnitude *dt

if e.p >= myscreen.MAX_DIST:
    eb.step_reward += reward_reach_the_castle

    if eb.trainable:
        trainable_enemy_exists = False
        eb.learn(episode_reward)
    else:
        new_smart_enemies.append(eb)
        e.route( )
        if action == 0 :
            eb.step_reward += reward_moving_forward
        else:
            eb.step_reward += reward_moving_backwards
        for ot in observer_towers:
            if random.random() < 0.05 and len(ot.bullets) <ot.max
_bullets and stopwatch_timer_bullet >= stopwatch_at_bullet:
                ot.make_bullet(e)
                stopwatch_timer_bullet = 0

smart_enemies = new_smart_enemies

trainable_not_deleted_ut = True
trainable_not_deleted_ot = True
for ut in user_towers:
    (ut.bullets, trainable_not_deleted_ut, episode_reward, steps_
count, c_episode_reward, c_step_reward) = active_bullets_after_collision_chec
ks (ut, smart_enemies, episode_reward, steps_count, c_episode_reward, c_step_
reward)
    trainable_enemy_exists = trainable_enemy_exists and trainable
_not_deleted_ut
    for ot in observer_towers:
        (ot.bullets, trainable_not_deleted_ot, episode_reward, steps_
count, c_episode_reward, c_step_reward) = active_bullets_after_collision_chec
ks (ot, smart_enemies, episode_reward, steps_count, c_episode_reward, c_step_
reward)
        trainable_enemy_exists = trainable_enemy_exists and trainable
_not_deleted_ot

```

```

for ed in smart_enemies:
    if ed.trainable:
        ed.rewards_history.append(ed.step_reward)
        episode_reward = episode_reward + ed.step_reward

steps_count +=1

if steps_count == max_steps_per_episode :
    for ed in smart_enemies:
        if ed.trainable:
            ed.learn(episode_reward)

    episode_reward = 0
    steps_count = 0
    episodes_count += 1

for ed in smart_enemies:
    if ed.trainable:
        ed.step_reward = 0

## PLOT
screen.fill( myscreen.background_colour)
myscreen.display_route(screen)
for ut in user_towers:
    ut.display(screen)
    for b in ut.bullets:
        b.display(screen)

for ot in observer_towers:
    ot.display(screen)
    for b in ot.bullets:
        b.display(screen)

for eb in smart_enemies:
    e = eb.me_the_enemy
    e.display(screen)

target.display(screen)

pygame.display.flip()

pygame.quit()

```

Entities.py

```
import math
import pygame

class Vector:
    def __init__(self, mag, th):
        self.magnitude = mag
        self.th = th

    def to_cartesian(self):
        return (math.cos(self.th)*self.magnitude,
                math.sin(self.th)*self.magnitude)

    def from_cartesian(self, cx, cy):
        self.magnitude = math.sqrt(cx*cx + cy**2)
        self.th = math.atan2(cy, cx)

    def from_cartesian_from_screen(self, cx, cy):
        fs_cx, fs_cy = self.from_screen(cx,cy)
        self.magnitude = math.sqrt(fs_cx*fs_cx + fs_cy**2)
        self.th = math.atan2(fs_cy, fs_cx)

    def __add__(self, other):
        if not isinstance(other, Vector):
            raise Exception("Invalid add type")
        sx, sy = self.to_cartesian()
        ox, oy = other.to_cartesian()
        rx = sx + ox
        ry = sy + oy
        mag = math.sqrt(rx**2 + ry**2)
        th = math.atan2(ry, rx)
        return Vector(mag, th)

    def go_to(self, toV):
        if not isinstance(toV, Vector):
            raise Exception("Invalid add type")
        from_x, from_y = self.to_cartesian()
        to_x, to_y = toV.to_cartesian()
        rx = to_x - from_x
        ry = to_y - from_y
        mag = math.sqrt(rx**2 + ry**2)
        th = math.atan2(ry, rx)

        return Vector(mag, th)

    def __str__(self):
        return "<{}, {}>".format(self.magnitude, self.th)

    def to_screen (self):
```



```

scr_attributes = Screen_attridutes()
fToScreen = scr_attributes.factor_to_screen
(x,y) = self.to_cartesian()
return fToScreen*x, fToScreen*y

def from_screen (self, cx, cy ):
scr_attributes = Screen_attridutes()
fToScreen = scr_attributes.factor_to_screen
return cx/fToScreen, cy/fToScreen

class PhysicsObject:
def __init__(self, r = Vector(0, 0), u = Vector(0, 0)):
self.r = r
self.u = u

def apply_force(self, f, dt):
fx, fy = f.to_cartesian()
ux, uy = self.u.to_cartesian()

ux = ux + fx * dt
uy = uy + fy * dt

x, y = self.r.to_cartesian()

x = x + ux*dt
y = y + uy*dt

self.u.from_cartesian(ux, uy)
self.r.from_cartesian(x, y)

def to_state_vector(self):
r = list(self.r.to_cartesian())
u = list(self.u.to_cartesian())
return r + u

def __str__(self):
return "[r = {}, u={}]".format(self.r, self.u)

class PaintableObject:
def __init__(self):
rV = Vector(0.0, 0.0)
rV.from_cartesian_from_screen(800, 640)
self.r = rV
self.colour = (255, 0, 38)
self.radius = 1.5
self.thickness = 23

def display(self, screen):
(x,y) = self.r.to_screen()
scr_attributes = Screen_attridutes()

```

```

        to_sc = scr_attributes.factor_to_screen
        pygame.draw.circle(screen, self.colour, (int(x), int(y)), int(self.radius*to_sc), self.thickness)

class Enemy:
    def __init__(self, position):
        new_rV = Vector(0.0, 0.0)
        new_rV.from_cartesian_from_screen(position[0],position[1])
        new_uV = Vector(2, 0.0) # 2 m/s
        self.r_and_u = PhysicsObject(new_rV, new_uV)
        self.radius = 0.3125

        self.colour = [ (255, 214, 214), (240, 175, 175), (255, 184, 184),
            (255, 138, 138),(255, 84, 84), (222, 73, 73), (255, 54, 54), (219, 9
, 9), (194, 31, 31), (163, 7, 7)]
        self.thickness = 5
        self.p = -1
        self.health = 9 # if changed then the colour list must be extended as
well

    def display(self, screen):
        (x,y) = self.r_and_u.r.to_screen()
        scr_attributes = Screen_attridutes()
        to_sc = scr_attributes.factor_to_screen
        pygame.draw.circle(screen, self.colour[self.health], (int(x), int(y))
, int(self.radius*to_sc), self.thickness)

    def route(self):
        scr_attributes = Screen_attridutes()
        p = self.p
        dy_in_route= scr_attributes.dy_in_route #metres
        enemy_size_offset = self.radius

        # p metres
        if p<0 :
            self.p = 0
            return (0, 0)

        count_inner_set =0
        offset=0
        offset_standard = scr_attributes.width_meters #screenX_metres
        previous_height = -dy_in_route
        new_x_metres=0.0
        new_y_metres=0.0
        for check in scr_attributes.check_points:
            if( check - offset == offset_standard): #left or right
                previous_height += dy_in_route

```

```

if (p>=offset and p<check):
    if count_inner_set ==0: #going right(->) -.
        new_x_metres = p-offset
        new_y_metres = previous_height
        if new_y_metres == 0:
            new_y_metres +=enemy_size_offset
        elif new_y_metres == scr_attributes.height_meters:
            new_y_metres -=enemy_size_offset
    elif count_inner_set ==1: #going down from right(|) |.
        new_x_metres = offset_standard - enemy_size_offset
        new_y_metres = previous_height + (p - offset)
    elif count_inner_set ==2: #going left(<-) .-
        new_x_metres = offset_standard - (p-offset)
        new_y_metres = previous_height
    elif count_inner_set ==3: #going down from left(|) .|
        new_x_metres = 0 + enemy_size_offset
        new_y_metres = previous_height + (p - offset)
    (x,y) =self.r_and_u.r.to_cartesian()
    dx = new_x_metres - x
    dy = new_y_metres - y
    self.r_and_u.r.from_cartesian(new_x_metres, new_y_metres)
    self.r_and_u.u.th =math.atan2(dy, dx)

count_inner_set +=1
if count_inner_set ==4:
    count_inner_set =0
offset = check

def to_screen(self, x, y, factor_to_screen):
    self.x = factor_to_screen*x
    self.y = factor_to_screen*y

def is_hit (self, b ):
    (ex,ey) = self.r_and_u.r.to_cartesian()
    (bx,by) = b.r_and_u.r.to_cartesian()
    #distance between center of enemy and bullet point left up
    #distance between center of enemy and bullet point left down
    #distance between center of enemy and bullet point right up
    #distance between center of enemy and bullet point right down
    return (math.sqrt( (ex - bx)**2 + (ey - by)**2 ) < self.radius
            or math.sqrt( (ex - bx)**2 + (ey - (by + b.r_height))**2 ) <
self.radius
            or math.sqrt( (ex - (bx + b.r_width))**2 + (ey - by)**2 ) < s
elf.radius
            or math.sqrt( (ex - (bx + b.r_width))**2 + (ey - (by + b.r_he
ight))**2 ) < self.radius )

def subtrack_health(self):

```

```

        self.health -= 1

    def find_two_closest_bullets(self, bullets ):
        d_close = 5
        two_closest_bullets = [] # size 2, bullet-
distance. Closest bullet is in [0], second closest in [1]
        (xe, ye) = self.r_and_u.r.to_cartesian()

        first_closest_bullet = None
        fcb_d = 5.1
        second_closest_bullet = None
        scb_d = 5.1
        for b in bullets:
            (xb, yb) = b.r_and_u.r.to_cartesian()
            d = math.sqrt( (xe-xb)**2 + (ye-yb)**2 )
            if (d <= d_close):
                if (d < fcb_d):
                    second_closest_bullet = first_closest_bullet
                    scb_d = fcb_d
                    first_closest_bullet = b
                    fcb_d = d

                elif d < scb_d and d is not fcb_d:
                    second_closest_bullet = b
                    scb_d = d

            if (first_closest_bullet is not None):
                two_closest_bullets.append(first_closest_bullet)
            if (second_closest_bullet is not None):
                two_closest_bullets.append(second_closest_bullet)

        return two_closest_bullets

    def to_state_vector(self):
        return self.r_and_u.to_state_vector()

class Screen_attridutes:
    def __init__(self):
        self.background_colour = (119,136,153)
        (self.width, self.height) = (800, 640)
        self.width_meters = 50.0
        self.height_meters = self.width_meters * self.height / self.width
        self.max_loop_sets = 5
        self.h_in_set = 2
        self.dy_in_route = self.height_meters/ (self.max_loop_sets*self.h_in_
set)

        self.MAX_DIST = (self.width_meters + self.dy_in_route) * self.h_in_se
t * self.max_loop_sets + self.width_meters
        self.factor_to_screen = self.width//self.width_meters
        self.check_points = self.calculate_checkpoints()

```

```

self.x_y_checkpoints = self.calculate_x_y_checkpoints()

def calculate_checkpoints(self):
    dy = self.dy_in_route
    dx = self.width_meters

    check_points=[]
    for i in range(self.max_loop_sets*2):
        check_points.append((i+1)*dx +i*dy)
        check_points.append((i+1)*dx +(i+1)*dy)
    check_points.append((self.max_loop_sets*2+1)*dx +self.max_loop_sets*2
*dy)
    return check_points

def calculate_x_y_checkpoints(self):
    dy = self.dy_in_route
    dx = self.width_meters
    check_points = []
    x = 0
    y = -dy

    for i in range(self.max_loop_sets):
        for i in range(4):
            if i%2==0: #going down
                y+=dy
            elif i == 1:#goint right
                x +=dx
            else:
                x -=dx

            v= Vector(0,0)
            v.from_cartesian(x,y)
            check_points.append(v)

        y+=dy
        v= Vector(0,0)
        v.from_cartesian(x,y)
        check_points.append(v)
        x +=dx
        v= Vector(0,0)
        v.from_cartesian(x,y)
        check_points.append(v)
    return check_points

def display_route(self, screen):
    color = (131, 148, 166)
    metre_checkpoint = self.x_y_checkpoints
    screen_check_points = []
    for v in metre_checkpoint:
        (x,y) = v.to_screen()

```

```

        screen_check_points.append((x,y) )

pygame.draw.lines(screen, color, False, screen_check_points, 15 )

def is_tower_position_allowed_simple(self, tower):
    (xt, yt) = tower.r.to_cartesian()
    yt = yt - tower.symmetrical_distance #( the middle of total height o
f triangle)
    dmin = 2* tower.symmetrical_distance
    dy = self.dy_in_route
    print("xt = {} yt = {}. dmin = {}".format( xt, yt,dmin))

    remainder = yt % dy

    if remainder < dmin or dy - remainder < dmin :
        return False
    elif xt < dmin or xt > self.width_meters - dmin:
        return False
    else: return True

def is_tower_position_allowed(self, tower):

    (xt, yt) = tower.r.to_cartesian()
    yt = yt - tower.symmetrical_distance #( the middle of total height o
f triangle)
    dmin = 2 * tower.symmetrical_distance
    x0e = 0
    y0e = 0
    e = Enemy((0, 0))
    ue = e.r_and_u.u.magnitude

    k = 0
    previous_distance = 100
    while k<50:

        const_a = - 2 * (yt - y0e) * ue
        const_b = 2 * (xt - x0e) * ue
        const_c = math.atan2(const_a,const_b)
        const_d = 1 / ( 2 * ue**2)

        th = k * math.pi - const_c

        t = const_d * (const_a * math.sin(th) - const_b * math.cos(th))
        x = x0e + ue * math.cos(th) * t
        y = y0e + ue * math.sin(th) * t
        dx = xt - x0e - ue * math.cos(th) * t
        dy = yt - y0e - ue * math.sin(th) * t

```

```

        d = math.sqrt( dx**2 + dy**2)

        if d < dmin:
            return False
        elif previous_distance < d:
            return True
        elif previous_distance>d:
            previous_distance = d
            x0e += ue * math.cos(th) * t
            y0e += ue * math.sin(th) * t

        k +=1

class Tower:
    def __init__(self, position):
        rV = Vector(0.0, 0.0)
        rV.from_cartesian_from_screen(position[0],position[1])
        self.r = rV
        self.colour = (204,153,0)
        self.symmetrical_distance = 0.625
        self.bullets = []
        self.max_bullets = 10

    def display(self, screen):
        (x,y) = self.r.to_screen()
        symmetrical_distance_toScreen = self.symmetrical_distance * Screen_a
ttributes().factor_to_screen
        pointlist_3 = [(x - symmetrical_distance_toScreen, y), (x + symmetri
al_distance_toScreen, y), (x, y - 2*symmetrical_distance_toScreen)]
        pygame.draw.polygon(screen, self.colour, pointlist_3, 0)

    def make_bullet(self, to_x, to_y):
        toV = Vector(0.0, 0.0)
        toV.from_cartesian_from_screen(to_x,to_y)
        if len(self.bullets) < self.max_bullets:
            self.bullets.append(Bullet(self.r, toV

class Observer_tower:
    def __init__(self, position):
        rV = Vector(0.0, 0.0)
        rV.from_cartesian_from_screen(position[0],position[1])
        self.r = rV

        self.colour =(44, 0, 176)
        self.symmetrical_distance = 0.625
        self.bullets = []
        self.max_bullets = 10

```

```

def display(self, screen):
    (x,y) = self.r.to_screen()
    symmetrical_distance_toScreen = self.symmetrical_distance * Screen_a
    ttridutes().factor_to_screen
    pointlist_3 = [(x - symmetrical_distance_toScreen, y), (x + symmetrical_distance_toScreen, y), (x, y - 2*symmetrical_distance_toScreen)]
    pygame.draw.polygon(screen, self.colour, pointlist_3, 0)

def make_bullet(self, e ):
    if len(self.bullets) < self.max_bullets:
        #(x,y) = self.r.to_cartesian()
        b = Bullet(self.r, Vector(0,0))
        (xt,yt) = self.r.to_cartesian()
        ub = b.r_and_u.u.magnitude

        (xe,ye) = e.r_and_u.r.to_cartesian()
        (uxe,uye) = e.r_and_u.u.to_cartesian()

        # after solving the motion equations of e,b
        # to find the collision point/ angle of speed of the bullet, as
        e, b: eythygrammi omali kinisi
        # (yt - ye) * uxe - (xt - xe) * uye = (yt - ye) * ub * cos(th) -
        (xt - xe) * ub * sin(th)
        # (yt - ye) * uxe - (xt - xe) * uye - (yt - ye) * ub * cos(th) +
        (xt - xe) * ub * sin(th) = 0
        # A -B cos(th) - C sin(th) = 0
        # A - sqrt(B^2 +C^2) * sin (th + arctan(B/C)) = 0
        # th = arcsin(A/D) - E

        const_a = (yt - ye) * uxe - (xt - xe) * uye
        const_b = (yt - ye) * ub
        const_c = -(xt - xe) * ub

        const_d = math.sqrt (const_b**2 + const_c**2)
        const_e = math.atan2(const_b,const_c)

        th = math.asin(const_a/const_d) - const_e

        b.r_and_u.u.th = th

        self.bullets.append(b)

class Bullet:
    def __init__(self, fromV,toV):# ,position, to):
        r = fromV.go_to(toV)

```



```

v = Vector(8,r.th)
self.r_and_u = PhysicsObject(Vector(fromV.magnitude,fromV.th), v)
self.r_width = 0.3125
self.r_height = 0.3125
self.colour = (71, 70, 57)#(1,1,3)
self.thickness = 15

def display(self, screen):
    (x,y) = self.r_and_u.r.to_screen()
    scr_attributes = Screen_attridutes()
    to_sc = scr_attributes.factor_to_screen
    pygame.draw.rect(screen, self.colour, pygame.Rect((x, y, int(self.r_w
idth * to_sc), int (self.r_height *to_sc))), 0)

def move(self,dt):
    self.r_and_u.apply_force(Vector(0,0),dt)

def is_in_screen (self):
    #upper screen limit
    scr_attributes = Screen_attridutes()
    (x,y) = self.r_and_u.r.to_cartesian()
    return (y + self.r_height >0
            and y < scr_attributes.height_meters
            and x < scr_attributes.width_meters
            and x + self.r_width >0)

def to_state_vector(self):
    return self.r_and_u.to_state_vector()

```