1 Project Definition

1.1 Project Overview

Artificial Intelligence is currently experiencing a surge in interest because of recent major advances in both algorithms and computing power. Some recent prowess in object recognition, natural language processing or autonomous vehicles have shown how that a combination of deep learning combined with well-known algorithms can be impressively effective at problems that seemed hard in past.

Reinforcement learning is a field of machine learning that focuses on learning how intelligent agents should select their actions to maximize some notion of reward. The combination of reinforcement learning and deep learning has produced recently some large breakthrough such as AlphaGo or impressive advancement in self-driving cars.

This final project aims at applying some of these techniques to a simplified example of Air Traffic Control. The goal is to develop a set of tools that would allow an individual aircraft to choose its heading to reach its destination while avoid other aircraft.

One of the constraints that was put on this effort is that the algorithm's input is a simple "1950 's radar scope" with only the positions of the different aircraft and destination visible. The project is not meant to be included in real life air traffic management system that would include detailed flight plan information and complex radar fusion data feeds but rather as an investigation of what can be done with the ML techniques mentioned above.

Some classic computer science algorithms like maybe much more efficient/straightforward/safe for solving this project in real life conditions however this project aims to explore which transportation problems can be solved in an interesting fashion with Machine Learning.

1.2 Problem Statement

The project aims to provide a program that would assign clearances to aircraft that would allow them to reach their destination as well as avoiding crashing into each other. The policy that would select the correct action at each decision time would be derived from the learning from past experience that would be used as input for training the machine learning algorithm.

ML systems learn from data (e.g. experience) however one cannot do experiments with real life aircraft traffic. The project thus included a simulator that imitates the performance of fictive aircraft evolving in an **Environment** that includes destinations or waypoints. The **Learning Agents** will be able to choose **Actions** that will alter aircraft trajectories. Aircraft that would continue straight if the agent would not give them clearances. The agent's decisions will be based on the **Observation** perceived from the environment, in our case a sequence of integer matrices describing the positions of aircraft and waypoints. The agent gets a feedback from the environment which is a numerical **Reward** received periodically. The Reinforcement Learning framework described above is summarized on **Figure 1-1.**

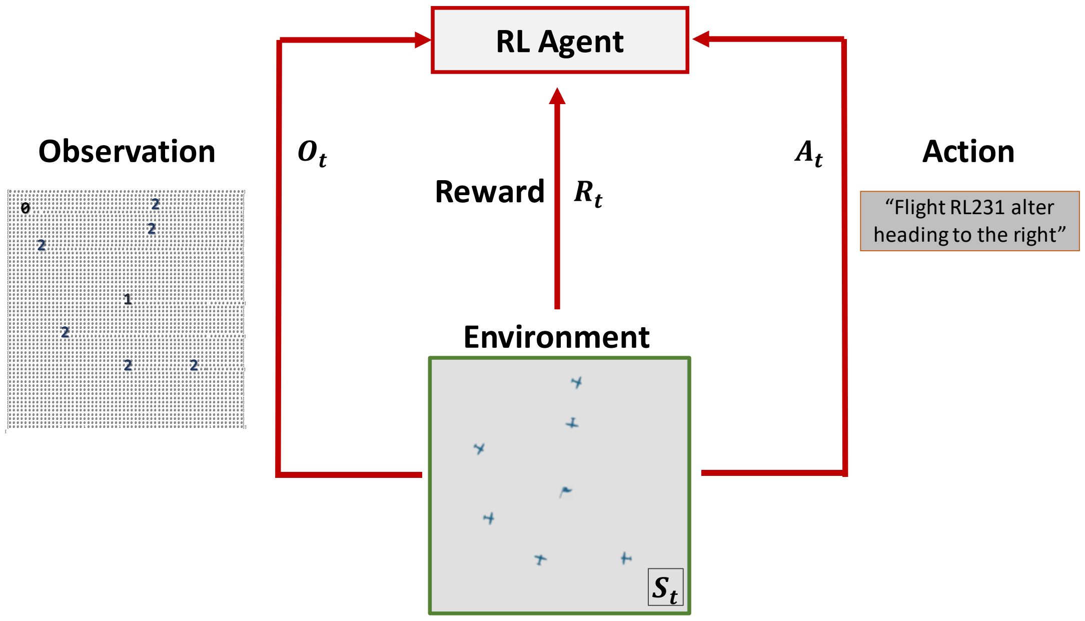


Figure 1-1: Reinforcement Learning Schematic Framework

As introduced in the Udacity Reinforcement Learning course, algorithms such as Q-learning or are promising tools to address the kind of problems. However, the complexity of the input data and the number possible situations that the agent can encounter forced to dive deeper into the RL literature to find a better way to "store" Q values (or approximate them with deep neural networks) and exploit the experience data.

It will be interesting to observe how well the agents will behave when there will be more than 1 aircraft in the same airspace. All aircraft must reach their destination but also not to crash into each other in the process. How quickly will the ML algorithms find some aircraft routing strategies to reach the destination? How well will the ML algorithm to unknow number of aircraft with new traffic configuration? Do convolutional neural networks help to generalize better due to their spatially invariant features?

1.3 Metrics

The goal of a Reinforcement Learning agent is to find the policy that maximizes the expected long-term return or discounted sum of rewards (more details in Section 2.4.1). In this project, the return is a function which depends at first approximation on the whether the agent reached its destination or crashed and how long did the episode last similarly to a video game score board.

More formally the total **return** G for an episode of duration T and a discount factor γ is the following:

$$G = \sum_{t=1}^{T} \gamma^{t-1} R_t$$

The evolution of the average total return per episode will be the most important metric for this project as this is the most comprehensive measure of "success" available.

Specialized metrics like the percent of flights that manage to arrive to their destination or **landing rate** will also be useful markers of the progress of the reinforcement agent's learning.

The agent's policy will use value estimators (neural nets) to assess which action is optimal. Checking whether the **training loss** of these estimators decreases over the iterations will be required to make sure the basic learning processes are functioning correctly. However, one particularity of Reinforcement Learning is that the value targets are not stationary until the convergence of the policy (see section 2.4). As such the training loss decrease may not be as crucial as in other Machine Learning applications.

2 Analysis

2.1 Traffic Simulator / Data Generation

As mentioned above, the project requires the use of a traffic simulator to obtain training data and for testing the agent's policy. The development of a simple air traffic simulator was necessary since no suitable lightweight open source library were found. The following section describes how the simulator was implemented. Most of the files cited below can be found in the *simulator* folder of the project's repository.

2.1.1 Simplified Traffic Simulator

The simulator is comprised of an environment where aircraft and waypoints (destinations) are created and evolve over time and some code to keeps track of the experience of the agents.

One specificity of the traffic simulator is that it allows for multiple RL agent (aircraft) to evolve, store their experience and update their policy.

The performance characteristics of the aircraft, how they move, is described in the *aircraft.py* file. An aircraft object can receive a heading clearance and will alter its heading until reaching the desired heading value. Knowing its current position and heading, the aircraft moves incrementally inside the airspace (with δ_t = 0.1s time increments) and updates its position within the environment.

Aircraft positions are initialized randomly at the edges of the airspace and their heading are adjusted to avoid exiting the airspace in the next time increments right after initialization. Initial aircraft positions are also chosen to avoid direct conflict with other aircraft that would lead to unavoidable crash.

The environment (*environment.py*) contains all the different object within the airspace (aircraft/waypoints). The environment updates each object's properties over time, assigning rewards to each individual aircraft and removing aircraft when they reached their destination, exited the airspace or crashed into each other.

2.1.2 Actions/Rewards/Observations

Agents can periodically select an action of out the following 5 choices:

- "Straight": the aircraft continues straight with its current heading
- "Left"/"Right": the aircraft receives a clearance corresponding to a full turn in the selected direction at the aircraft's turning rate during the next time step
- "Small Left"/"Small Right": same as previous but with only ¼ of the heading change

The reward value for each agent is computed each time the environment updates the position of the aircraft (*environment.step()*). The reward is computed as follow:

- +10 if the aircraft is within proximity of its destination
- -10 if the aircraft exited the airspace boundaries or crashed into another aircraft
- A small negative reward each time its heading clearance change. The value is proportional to the difference of heading clearance

The environment's state grid is the representation of the environment's current state observed by the agent. It is a 60x60 integer matrix representing the positions of aircraft, waypoint and obstacles. One specificity is that the state grid is different depending on the agent who is querying the environment's state. The 60x60 integer cell values are defined as follows:

- 0 if there is not object inside the cell (majority of the cells)
- 1 if there is a waypoint inside the cell
- 2 if the querying agent (aircraft) is inside the cell
- 3 if another aircraft is inside the cell

The environment and all its elements are created by the simulator object (*simulator.py*). The simulator keeps track of the successive environment states, agent actions and rewards.

The simulator repeatedly calls the *environment.step()* method which updates the positions of all the elements with a δ_t = 0.1s time increment. Agent choose their actions periodically, every *action_frame_period* steps. In the meantime, the simulator stores a buffer of the 3 frames (environment state grid matrix) in between actions for each agent.

The agent chooses its next action based on the 3x60x60 observation matrix $\boldsymbol{O_t}$ which is a sequence of environment states since the last action from the perspective of the agent.

As explained in Section 2.4, the project's implementation experience replay requires to store tuples of current observation $\boldsymbol{O_t}$, the selected action $\boldsymbol{A_t}$, the reward $\boldsymbol{R_t}$ received after the action and the next observation $\boldsymbol{O_{t+1}}$ called **transitions**. During experience replay of a transition, the agent will be able to estimate the value of choosing an action given the current observation.

The transitions are stored in a single object which receives the experience from every agent. The policy improvements are based on the shared experiences gathered from all the agents during the past simulations.

The project code provides the option of storing the return from the moment $\boldsymbol{t}$, $\boldsymbol{G_t} = \sum_{i=t}^{T} \gamma^{i-t} R_i$, instead of the immediate reward $\boldsymbol{R_t}$. $\boldsymbol{G_t}$ requires waiting for the end of the simulation to calculate the sum of discounted rewards since the moment $\boldsymbol{t}$ of the observation $\boldsymbol{O_t}$.

The simulator sends a signal to update the agents' policy when the agents select their actions. This update signal is then processed by the **trainer** object (*main_train.py*) which handle the experience data processing and policy parameter update.

One important particularity of the project data sets is that there a not static as the data sets use for most machine learning projects. The data set evolves as the agents' policy changes. There is a strong dependency between the training data set (experience) and the past policies as the agent explores different parts of the environment state domain depending on the actions recommended by its current policy.

2.2 Data Exploration

As mentioned in Section 2.4.1, the data used for the training of the agents, called experience in the RL setting, is comprised of $(\boldsymbol{O_t}, \boldsymbol{A_t}, \boldsymbol{R_t}, \boldsymbol{O_{t+1}})$ tuples that describe the current observation $\boldsymbol{O_t}$ from which the agent based the selection of its next action $\boldsymbol{A_t}$ and the subsequent reward $\boldsymbol{R_t}$ received by the agent and the next observation $\boldsymbol{O_{t+1}}$. of the environment state. The agent uses $\boldsymbol{O_t}$ to estimate which action is the most promising to get an optimal return on the long run.

The dataset is thus dynamically generated as the agent interacts with the environment. This is a challenge because contrary to standard supervised learning datasets the target values (Q-values for example) are not time-invariant as they change as the agent's policy improves. An action that would have seemed as a bad choice 10,000 policy updates ago could now be the optimal action given the current policy that would make better choices afterward. As described in Section 2.4.2.1, the dynamic nature of the training data may lead to some instable learning but techniques like target networks help to solve this challenge.

There is a strong dependence between samples. Indeed, the agent's experience at time step t+1 is strongly dependent on its experience at time step t. Random experience sampling is implemented during training to break the dependence between samples from the same simulation.

The environment state is only partially observed by the agents. For example, one agent is not told which clearance were received by the other nor their current heading, it can only observe the evolution of their past positions. More broadly, one agent considers the other agents as part of the environment even if they may share the same policy. The environment's state evolution depends on all the policies from the different agents while one agent can only perceive the three last 60x60 position grids.

The characteristics of the tensor observation $\boldsymbol{O_t}$ depends on the number of agent on the frame. The 6x60x60 tensor is sparse since there are non-zero values only where the aircraft and the waypoint are located. Running simulations with initially 3 aircraft in the environment and a random policy for each agent, the average value of the 3x60x60 tensor is 2.13×10^{-3} and the standard deviation is 7.25×10^{-2}. These values were used to normalize the input of the policy's estimators.

Using the same simulation settings, the average reward is -6.80 and the standard deviation is 5.42. The reward was also normalized to improve the stability of learning.

2.3 Exploratory Visualization

The project includes a type of simulator (*vizsimulator.py*) with a pygame interface, inspired by the smartcab Udacity assignment, that allows to follow the movement of aircraft within the environment and understand which actions are selected by the policy.

Some example of simulations with random policy and convolutional neural network policy can be watched here. The video shows the evolution of the total score (all agents included), decreasing with time and jumping +10 or -10 depending on the final state of each aircraft.

Figure 2-1 is a histogram of the immediate reward $\boldsymbol{R_t}$ received by individual aircraft when running 3-aircraft simulation with a random policy. The most frequent reward value range is 0 or slightly lower (~90% of transitions) which corresponds to the rewards received while the aircraft is moving in the environment without reaching a final state. The -10 reward value range correspond to the transitions when the aircraft reached a "bad" final state: exiting the airspace or crashing into another aircraft. The +10 correspond to the "good" final state of reaching the aircraft's destination, however we can note that with a random policy this is pretty rare (in this setting 9.4% of aircraft reach their destination).

Figure 2-2 shows the distribution of the long-term returns, $\boldsymbol{G_t} = \sum_{i=t}^{T} \gamma^{i-t} R_i$ received from any time step **t** until the end of the simulation. The discount factor $\boldsymbol{\gamma_t}$ value is set at 0.95. The exponential decay of the final reward (+-10) can be observed on the graph.

The negative outcome (crash/exit) return values range from -10 to -2. This is explained by the fact that the total duration of a simulation for an aircraft can reach 30 time steps as shown in Figure 2-3 and the final reward decay would lead to such values (.95^30= .21)

Most of the information regarding the expected return from a state is thus contained in the value of the final state and the information "decays" with time.

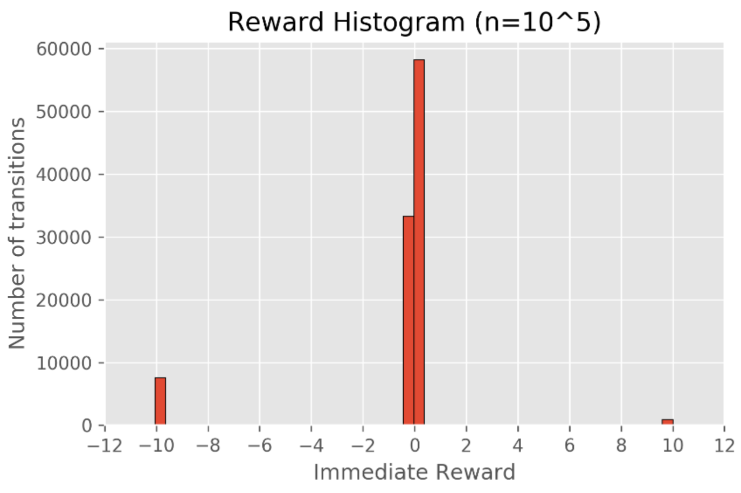


Figure 2-1: Reward (R_t) Histogram (3 aircraft with random policy simulations)

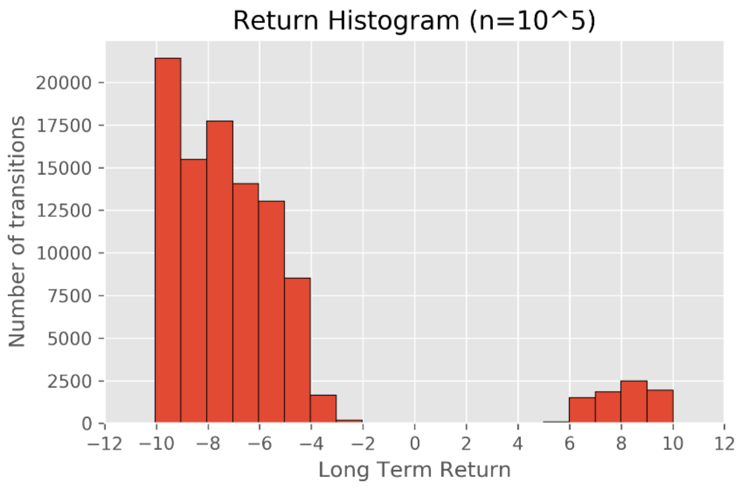


Figure 2-2: Return (G_t) Histogram (3 aircraft with random policy simulations; gamma = .95)

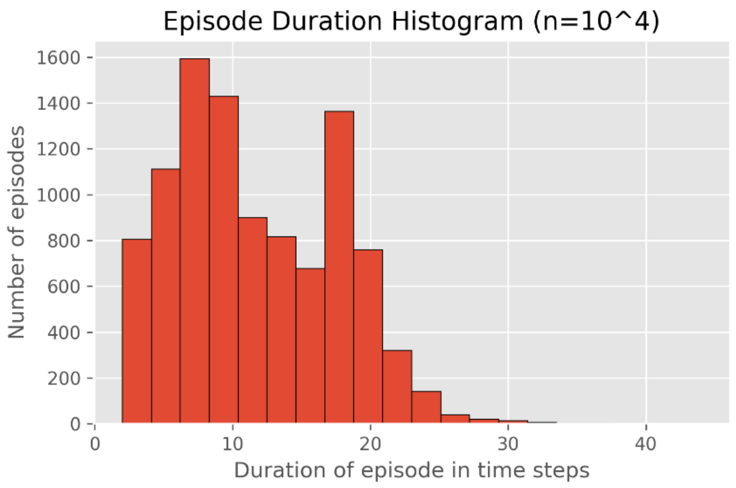


Figure 2-3: Histogram of the simulation (episode) duration for an individual agent (3 aircraft with random policy simulations)

2.4 Algorithms and Techniques

Algorithms and techniques used in the project are thoroughly discussed and properly justified based on the characteristics of the problem.

As mentioned in Section 1, agents will learn to pick the right actions through the utilization of Reinforcement Learning techniques applied on the experience gathered from the simulated environment (see Section 2.1). Due to recent advances in the effectiveness neural networks, the project will use several neural networks architectures as the main function estimators (with different depth, activation functions, convolutional units…).

2.4.1 Reinforcement Learning[1]

Fundamental components

The goal of a reinforcement learning agent is to maximize the return $\boldsymbol{G_t}$ by choosing the optimal actions given the observations of the environment perceived by the agent. This problem is a sequential decision making problem, the actions may have long term consequences and the reward may be delayed into the future.

The return G_t from time step t for an episode of duration T and a discount factor γ is the following:

$$G_t = \sum_{i=t}^{T} \gamma^{i-t} R_i$$

As noted in Section 2.2, in this project the environment is only partially observable. Hence the agent's state is different from the environment state, but the agent still has to make decisions with its own representation of the state of the environment. For the rest of this section, we will use the agent's state S_t interchangeably with its observation of the environment O_t

[1] In addition to the Udacity resources, I went in detail through the University College London Reinforcement Learning course material for this section (http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html)

The agent's policy $\boldsymbol{\pi}$ represents the agent's behavior. The policy maps the agent's state S_t to the action A_t that the agents would select. The policy can be deterministic where $A_t = \pi(S_t)$ or stochastic such that $\pi(a|s) = \mathbb{P}[A_t = a \mid S_t = s]$.

The RL techniques that are used in this project are model-free. The agent does have a model of which next state S_{t+1} is likely to occur if it takes the action A_t and does not have a prediction of the values of the reward R_t neither.

Value Function Learning

The action-state value function $\boldsymbol{q_\pi(s,a)}$ is the expected return starting from state **s**, taking action **a** and then following policy $\boldsymbol{\pi}$**.** The expectation is computed over the policy $\boldsymbol{\pi}$, this means that $\boldsymbol{q_\pi(s,a)}$ depends all the possible subsequent states and actions when following policy $\boldsymbol{\pi}$. Evaluating the function value of a policy is called **policy evaluation.**

$$\boldsymbol{q_\pi(s,a) = \mathbb{E}_\pi[G_t | \, S_t = s \, , A_t = a]}$$

The optimal action-state value function is the $\boldsymbol{q_*(s,a)}$ is the maximum action-state value function over all policies, $\boldsymbol{q_*(s,a) = \max_\pi q_\pi(s,a)}$ **.**

Once the optimal action-state value function $\boldsymbol{q_*}$ is known, the policy that selects greedily action $\boldsymbol{a^* = \operatorname*{argmax}_a q_*(s,a)}$ at each time step is optimal.

One key element is that we have to be able to have a good estimation of the action-state values $\boldsymbol{q_\pi}$ given a policy $\boldsymbol{\pi}$ (policy evaluation). Monte-Carlo (MC) and Temporal Difference (TD) techniques learn $\boldsymbol{q_\pi}$from experience under policy $\boldsymbol{\pi}$ by updating incrementally the action-state values toward a proxy target of $\boldsymbol{\mathbb{E}_\pi[G_t | \, S_t = s \, , A_t = a]}$.

Monte-Carlo learning updates $\boldsymbol{q(A_t,S_t)}$ toward the actual return $\boldsymbol{G_t}$. MC learning estimates $\boldsymbol{G_t}$ by using directly the empirical mean of $\boldsymbol{G_t}$. The estimation of the G_t can only be made at the end of the episode. MC will be used in this project since the time duration of each is reasonably low max 30 time steps see Section 2.3). The Monte-Carlo $\boldsymbol{q(A_t,S_t)}$ incremental update with a learning rate $\boldsymbol{\alpha}$ is the following:

$$\boldsymbol{q(A_t,S_t) \;\leftarrow\; q(A_t,S_t) \;+\; \alpha \,[\, G_t - q(A_t,S_t)\,]}$$

Backing up one step in the future, $\boldsymbol{q_\pi(s,a)}$ can be decomposed into the Bellman expectation equation:

$$\boldsymbol{q_\pi(s,a) = \mathbb{E}[R_t + \gamma \, q_\pi(S_{t+1},A_{t+1}) | \, S_t = s \, , A_t = a]}$$

Temporal Difference learning updates $\boldsymbol{q(A_t,S_t)}$ **the estimated return** $\boldsymbol{R_t + \gamma \, q(S_{t+1},A_{t+1})}$. We can note that the TD learning target uses the current approximation $\boldsymbol{q(A_t,S_t)}$ of the state-action value function $\boldsymbol{q_\pi(A_t,S_t)}$**.** Contrary to MC learning, the update does not have to wait the end of the episode to be accomplished. The Temporal Difference $\boldsymbol{q(A_t,S_t)}$ incremental update with a learning rate $\boldsymbol{\alpha}$ is the following:

$$\boldsymbol{q(A_t,S_t) \;\leftarrow\; q(A_t,S_t) + \; \alpha \,[\, R_t + \gamma \, q(S_{t+1},A_{t+1}) - q(A_t,S_t)\,]}$$

The Q-Learning algorithm uses a similar target to the TD learning target, however the q value for the next step is not the one chosen from the experience but rather the maximum value for all the available actions (off-policy):

$$\boldsymbol{q(A_t, S_t) \leftarrow q(A_t, S_t) + \alpha \left[R_t + \gamma \max_a q(S_{t+1}, a) - q(A_t, S_t) \right]}$$

Deep Q-Networks (DQN) use deep convolutional networks (see Section 2.4.2) as function estimator $\boldsymbol{q(a, s; \theta)}$. The $\boldsymbol{\theta}$ parameter vector is tuned in order to minimize the mean squared error between $\boldsymbol{q(A_t, S_t; \theta)}$ and the Q target $\boldsymbol{R_t + \gamma \max_a q(S_{t+1}, a; \theta)}$.

The pseudo code of the basic DQN algorithm[2] used in this project can be found below (Algorithm 1). Sections 2.4.2 and 2.4.2.1 will add more precision to the implementation of Algorithm 1 and add improvements that will lead to Algorithm 2.

Algorithm 1

```
Initialize replay memory $D$ with $I$ transitions sampled with a random policy
Initialize action-value function $Q$ weights
for episode = 1, M do
    Initialize $s_1$ and normalize it $\varphi_1 = \varphi(s_1)$
    for t = 1, T do
        Increment time step $t$ += 1
        with probability $\varepsilon$ eps select a random action at
        otherwise select $a_t = \operatorname{argmax}_a q(\varphi_t, a; \theta)$
        Execute action $a_t$ in the simulator and observe reward $r_t$ and state $s_t$
        Set $s_{t+1} = s_t$ and preprocess $\varphi_{t+1} = \varphi(s_{t+1})$
        Store transition $(\varphi_t, a_t, r_t, \varphi_{t+1})$ inside $D$
        if t is a multiple of $F$ do
            Sample random minibatch of $m$ transitions $(\varphi_j, a_j, r_j, \varphi_{j+1})$ from D
            Set $y_j = \psi(r_j) + \max_a q(\varphi_{j+1}, \mathrm{a}; \theta)$
            Perform a gradient descent step on $(y_j - q(\varphi_{j+1}, a_j; \theta))$
        end if
    end for
end for
```

2.4.2 Deep Learning[3]

Artificial Neural Network Components

The estimation of the action-state values is performed by learning the value from experience (see Section 2.4.1). Neural Networks (NN) are one the most powerful sets of Machine Learning algorithm that will enable us to perform supervised learning on the experience of the agents (see Algorithm 1).

[2] Directly inspired from Playing Atari with Deep Reinforcement Learning, V. Mnih, 2013, arXiv:1312.5602

[3] In addition to the Udacity resources, I went in detail through the Stanford CS231n course material for this section (http://cs231n.stanford.edu/)

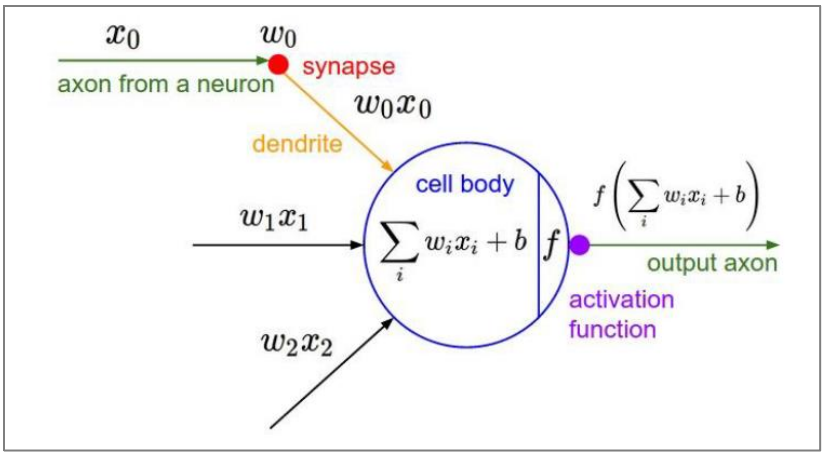


Figure 2-4: Artificial Neuron Representation

As depicted in Figure 2-4, Neural Networks are loosely inspired from the brain's neuronal cells. Neurons are arranged in a network architecture by connecting the outputs of the neurons from the previous layer to the input of the neurons of the next layer.

Each neuron receives a set of scalar inputs $\boldsymbol{X = x_1, x_2 \dots x_n}$, the outputs of the previous layer, which are linearly combined with the weights $\boldsymbol{W = w_1, w_2 \dots w_n}$ and bias $\boldsymbol{b}$. The scalar $\boldsymbol{W^T X + b}$ is then passed into an activation function $\boldsymbol{f}$ which produces the output of the neuron. The neural network is parametrized by the set $\boldsymbol{\theta = (W_k, b_k)_{k=1..K}}$ of the weights and biases (omitted for the rest of the section) of the $\boldsymbol{K}$ neurons also called units.

The output values of the last layer represent the predictions of the neural network. In this project, neural networks will be used as estimators of the state-action values.

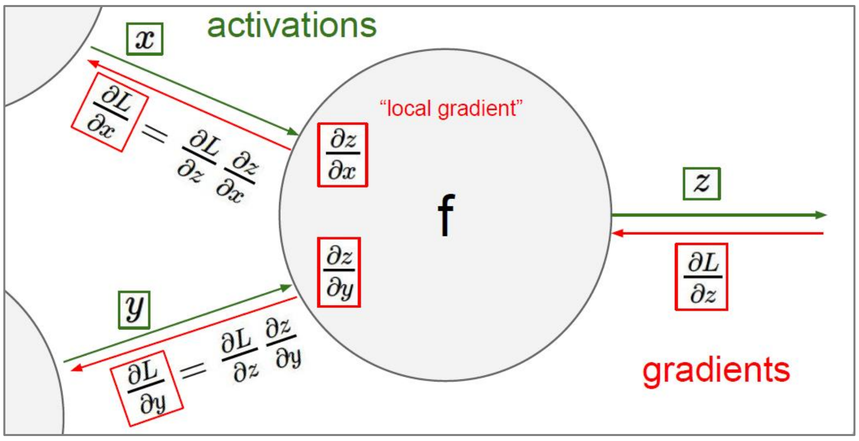
The input values that will feed the first layers of the neural network will the normalized state representation $\boldsymbol{\varphi(s_j)}$ which is a 3x60x60 tensor.

The succession of matrix multiplications and activations functions starting from the network's input until its output is called the forward pass. The forward pass is the step when the neural network makes a prediction $\boldsymbol{\hat{y}_j}$. This prediction is then evaluated against the target value $\boldsymbol{y_j}$ with the *l2-error* $\boldsymbol{l_j = (y_j - \hat{y}_j)^2}$.

Weight Update

The error signal is then used to update the network's weights through a process called backpropagation or backward pass. The goal of the backward pass is to minimize the loss for a batch of N data points $\boldsymbol{L = \frac{1}{N}\sum_{j=1}^{N} l_j}$ and is achieved through Stochastic Gradient Descent (SGD).

Gradient descent aims at minimizing the loss function $\boldsymbol{L}$ by updating each weight in the opposite direction of $\boldsymbol{\Delta W_k = \frac{\partial L}{\partial W_k}}$. The computation of the gradient is made easy by the fact that neural networks are comprised of successions of matrix multiplications and differentiable activation functions. As shown in Figure 2-5, the recursive application of the chain rule ($\boldsymbol{\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial x}}$) along all the units of the networks starting from the output layer through the first layer allows to compute each gradient.



activations
x
$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial x}$
"local gradient"
$\frac{\partial z}{\partial x}$
f
$\frac{\partial z}{\partial y}$
z
$\frac{\partial L}{\partial z}$
y
$\frac{\partial L}{\partial y} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial y}$
gradients

Figure 2-5: Backpropagation of gradient into a unit

The standard gradient descent update can be written as $\boldsymbol{x} = \boldsymbol{x} - \boldsymbol{\alpha dx}$ where $\boldsymbol{\alpha}$ is the learning rate. This naïve version is not the most efficient way to use the gradient information to minimize the loss function. SGD may not move fast enough in shallow gradient directions while moving too fast in steep gradient directions, this leads to some jittering of the gradient updates.

The Adam update introduces a first order moment $\boldsymbol{m}$ and second order $\boldsymbol{v}$ term that speed up the gradient update in shallow directions and dampen it in steep gradient directions. The Adam gradient update can be written as:

$$\boldsymbol{m} = \boldsymbol{\beta_1}\,\boldsymbol{m} - (\boldsymbol{1} - \boldsymbol{\beta_1})\boldsymbol{dx}\,, with\ 0 < \beta_1 < 1$$
$$\boldsymbol{v} = \boldsymbol{\beta_1}\,\boldsymbol{v} - (\boldsymbol{1} - \boldsymbol{\beta_2})\boldsymbol{dx^2}\,, with\ 0 < \beta_2 < 1$$

$$\boldsymbol{x} = \boldsymbol{x} - \frac{\boldsymbol{\alpha}\,\boldsymbol{m}}{\sqrt{\boldsymbol{v}} + \boldsymbol{\varepsilon}}\,, with\ \varepsilon\ very\ small$$

Convolutional layers

Convolutional layers enable neural networks to learn powerful image features from training data. A convolutional layer is comprised of a filter which is convolved with the image over all its spatial locations (see Figure 2-6). The same filter is used for dot products with the successive small chunks of the image. The ensemble of dot products at combined into an activation map. Generally, several separate filters are used to get a stack of activation maps.

Part of the strengths of convolutional layers resides in the fact that the weights of the filter are shared for all the spatial locations on the image. The sharing of the weights allows for some spatial invariance on the same

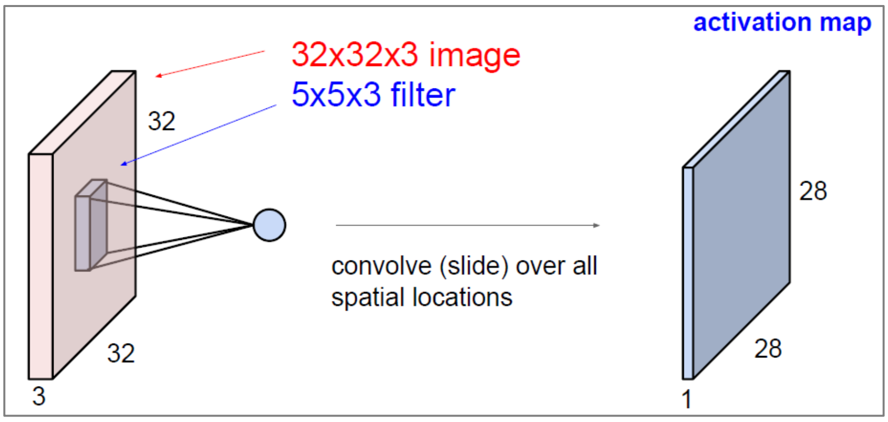


Figure 2-6: Convolution of a 5x5x3 filter on a 32x32x3 input

Neural Network Architectures

Two main neural network architecture were used for this project. Both networks aim at estimating the state-action values given the environment's observation tensor (see Section 2.1.2).

Instead of having one estimator per action, the network architectures have a 5-dimension output correspond to the action-values of the 5 possible actions. This allows to train the same network and learn the same features for all actions which is more data efficient but requires the definition of a new training loss function (see Section 3.2).

The first architecture (std_nn) is 3 hidden-layer neural network with 128 units at each layer. The input is a 10,800-dimension vector which corresponds to the flatten 3x60x60 environement state tensor. The architecture requires storing and updating 1.4 million scalar weights.

Table 2-1: std_nn Architecture Layer Description

Layer	Input	Nb Units	Activation	Output	Nb Weights
fc1	10,800	128	Leaky ReLU	128	10800*128 = 1.38m
fc2	128	128	Leaky ReLU	128	128*128 = 16.4k
fc3	128	128	Leaky ReLU	128	128*128 = 16.4k
fc4	6x6x64	512	Linear	5	128*5 = 640

The second architecture (conv_nn) is a typical architecture for convolutional neural network with 3 convolutional layers ending with a fully connected layer. The network's input is the stacked environment state tensors from the latest time steps. The architecture uses 1.2 million scalar weights.

Table 2-2: conv_nn Architecture Layer Description

Layer	Input	Filter size	Stride	Nb filters	Activation	Output	Nb Weights
conv1	3x60x60	6x6	3	32	Leaky ReLU	19x19x32	3*6*6*32 = 3.5k
conv2	19x19x32	4x4	2	64	Leaky ReLU	8x8x64	32*4*4*64 = 32.8k
conv3	8x8x64	3x3	1	64	Leaky ReLU	6x6x64	64*3*3*64 = 36.9k
fc1	6x6x64	-	-	512	Leaky ReLU	512	6*6*64*512 = 1.18m
fc2	512	-	-	5	Linear	5	512*5 = 2.5k

Both architectures have a similar number of scalar parameter (around 1 million) but as we will see in Section 4, they yield very different performances.

On specificity of these architecture is the use of Leaky ReLU as activation functions instead of the standard ReLU. This choice was made because of the changing nature of the learning targets values that change while the policy is improving. When using stand ReLU units, some part of the neural network may be definitely "dead" if the input of a ReLU is negative and no gradient can flow back during backpropagation. Leaky ReLU do not shut down completely if it receives negative inputs but scaled them down with a small positive coefficient.

2.4.2.1 Additional Techniques

Target Network

Algorithm 1 has a learning target $\psi(r_j) + \max_a q(\varphi_{j+1}, \mathrm{a}; \theta)$ which is dependent to the current network parameters θ which are to be updated. This correlation causes an oscillatory behavior which can slows down or make impossible the learning of the policy.

Freezing the target network weights θ^- for some number of time steps (10,000 for instance) allows to stabilize the learning process. The learning loss function becomes $[\psi(r_j) + \max_a q(\varphi_{j+1}, \mathrm{a}; \theta^-) - q(\varphi_{j+1}, a_j; \theta)]^2$, the network weights θ are updated following this loss and agents still use the current network parameters θ for choosing their next actions.

Prioritized Replay[4]

The dilemma of exploration versus exploitation can be alleviated by a better use of the experience replay. Indeed, all transitions are not of equal importance when learning a policy, some transitions may be frequent and updating the policy parameters on those transitions may not improve the policy dramatically while some transitions that are infrequent with unexpected rewards (landing or crashes) can be more useful.

A good proxy for the "usefulness" of a transition is its Temporal Difference error (see Section 2.4.1) which is used to update the policy weights. A high TD error means that the approximate value expected by the current action-state value estimator was very difference from the one observed in the transitions.

Algorithm 1 samples transitions from the experience uniformly. Prioritized replay samples transitions more often if there TD errors are higher. More precisely, we define the transition priority $p_i = |\delta_i| + \varepsilon$ where δ_i is the TD error and ε a small positive constant (10^{-2} for instance). The probability at which a transition is sampled is given by:

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

Where α is a positive number controlling how much prioritization is applied. If $\alpha = 0$, then we have uniform sampling.

As explained in Section 2.4.1, we aim at learning the expected action-state value of a policy using stochastic gradient updates to a non-linear estimator (neural network). This technique requires the training to have the same distribution as the expected value. However, sampling transitions

[4] Prioritized Experience Replay, T. Schaul ,19 Nov 2015, arXiv:1511.05952v2

with the method described above changes the distribution. To correct this bias, we introduce Importance Sampling weights $w_i = (\frac{1}{N\,P(i)})^{\beta}$, where N is the total number of transitions, to perform a weighted gradient update.

Revised Learning Algorithm

Algorithm 1 was enhanced with the additional techniques described in sections 2.4.2 and 2.4.2.1. The changes are highlighted in Algorithm 2 in orange. They include the use of Adam for stochastic gradient updates, target network for learning stability and prioritized replay for a more efficient use of the experience.

Algorithm 2

```
Initialize replay memory $D$ with $I$ transitions sampled with a random policy
Initialize action-value function $Q$ weights $\theta$ and $\theta^-$
for episode = 1, M do
    Initialize $s_1$ and normalize it $\varphi_1 = \varphi(s_1)$
    for t = 1, T do
        Increment time step $t$ += 1
        with probability $\varepsilon$ eps select a random action at
        otherwise select $a_t = \underset{a}{\operatorname{argmax}}\, q(\varphi_t, a;\ \theta)$
        Execute action $a_t$ in the simulator and observe reward $r_t$ and state $s_t$
        Initialize transition priority at max_td_error
        Set $s_{t+1} = s_t$ and preprocess $\varphi_{t+1} = \varphi(s_{t+1})$
        Store transition $(\varphi_t, a_t, r_t, \varphi_{t+1})$ inside $D$
        if t is a multiple of $F$ do
            Sample using the priority distribution minibatch of $m$ transitions $(\varphi_j, a_j, r_j, \varphi_{j+1})$ from D
            Set $y_j = \psi(r_j) + \max_a q(\varphi_{j+1}, a; \theta^-)$
            Perform a gradient descent step on $(y_j - q(\varphi_{j+1}, a_j; \theta))$
            using Adam and the Importance Sampling weights
            Update the priority of the sampled transitions
        end if
    end for
end for
```

2.5 Benchmark

For some domains like the Atari emulators, there are a lot of RL research available with well documented benchmarks. Since the learning environment was created uniquely for this project, there is no pre-existing policy examples available.

The default benchmark policy is the random policy. An agent using this policy selects uniformly at random its next action.

As we will see in Section 4, the performance of the random policy is, as expected, quite low for any configuration of the environment (number of agents/configuration of waypoints). However, this benchmark was very useful during the development of the learning algorithm to quantify if the algorithm was “actually doing something”.

3 Methodology

3.1 Data Preprocessing

As described in Algorithm 2, the (3x60x60) state tensors s_i are normalized (φ) as soon as they are observed by the agent and then saved into the experience replay data structure. Every component of the tensor is normalized using the same values derived from Section 2.2, such as $\varphi(s) = \frac{s-0.00213}{0.0725}$. It was observed that the distribution of the state tensor value remains approximately the same throughout the environment configurations because of its sparsity (see Section 2.1.2).

In the same fashion, the immediate reward values are also normalized using the mean and standard deviation observed on the initial set of transitions $\psi(\mathrm{r}) = \frac{r-avg\_r\_init}{std\_r\_init}$. As we will see in the next section, the reward normalization process can be optimized further by being more dynamic.

The experience replay data structure is initialized by running simulations using the random policy. The number of initial simulations is chosen to match approximation the size of the experience replay, in the order of 10^5 transitions.

3.2 Implementation

3.2.1 Code Architecture

The code is divided into several folders:

- **simulator** contains the objects that are required to run the simulation environment mainly aircraft objects, environment objects and the simulator that hold all these elements. There is fa subclass of simulator (*vizsimulator.py*) that allow to visualize the movement of the aircraft on a pygame interface using the .png files stored in the icons folder.
- **policy** contains the agent's policy object which holds the method called during the simulator to select actions as well as separate definition files (using Keras) for the value estimators (neural networks architectures) used by the policy. The policy class was defined in as generic Reinforcement Learning policy class, the *dqn* class is a subclass where the estimator definition, training loss and update procedure was specified.
- **experience** contains files that are related with the sampling of transitions from experience using prioritized replay and the plotting of performance metrics as the agents' experience is building up.

There are 3 "main' scripts that allow to either learn a new policy (*main_train.py*), test the performance of an existing policy (*main_test.py*) and visualize the actions of a policy with a pygame interface (*main_viz.py*). Each script contains a Trainer object which holds all the simulation/learning para meters, simulations, policies and experience replay memory.

The **metrics** folder contains the successive dashboards and csv files the depicts key training performance metrics.

The **models** folder holds the saved policy model weights files created during training that can be utilized by the *main_viz.py* or *main_test.py* to evaluate the performance of the models after training

The icons folder is where the aircraft and waypoint icons .png files used by the pygame visualization in *main_viz.py* are stored.

3.2.2 Neural Network Training

Training Loss

The neural network initialization (std_nn or conv_nn, see 2.4.2) is performed using the Keras library in *estimator.py*.

The neural network has 5 scalar outputs corresponding to the state-action values of the 5 valid actions that agents can select for a given state entered as input of the neural network.

When selecting its next action the agent "feeds" the current state into the neural network and selects the actions with the highest value. During policy update when replaying a transition, the goal is to correct the weights of the neural network following the training loss for the action that was selected in the experience. However, a single transition gives information on the state-action value of the state and selected action ("Left" for example) but does not impact directly the state-action values for the other actions ("Straight", "Right" …).

A "masked" loss was introduced to address the issue of propagation the loss gradient on for the action selected in the transition.

Efficient Computations

Training neural networks with more than 1 million parameters is computation intensive. Especially for Reinforcement Learning where the target values are non-stationary, being able to run forward and backward propagation fast was necessary for making the training of a reasonable policy for multiple agents even feasible.

First, the code was designed to use "vectorization" and limit to usage of loops for policy estimation and improvement as much as possible.

GPU are also known to process matrix multiplications operations much faster than CPU. While the code works for both CPU and GPU, the training of the models was performed for on a GPU. Since I don't own a computer with such equipment, an Amazon Web Service P.2 instance was set up with the proper deep learning NVIDIA (cudnn) libraries on a NVIDIA K80 GPU to facilitate the training and model iterations.

Finally, it was shown[5] that using lower precision float numbers allows to get a significant increase in speed and limited memory footprint while keeping the same model performance. 16-bit float were used during training (instead of 34-bit). Reducing the float precision was very helpful since it allowed to store much more transitions in the experience replay memory during training and allowing to sample more transitions into the GPU during policy update which helped for learning stability.

[5] Training deep neural networks with low precision multiplications, M Courbariaux, 2015, arXiv:1412.7024

3.2.3 Learning Scenarios

Several learning scenarios were used as the basis the basis for agent training. Each scenario corresponds to a given setting of the environment, or “the rules of the game”. The different learning environment settings include the number of aircraft initially in the environment and the behavior of the waypoints.

The following scenarios were explored:

1. 1 aircraft with 1 fixed waypoint; essentially a test to check whether the learning algorithm works properly
2. 3 aircraft with 1 fixed waypoint; introducing several agents and observing if policy converges to a satisfactory solution
3. 3 aircraft with 1 “random” directional waypoint; introducing more randomness into the environment

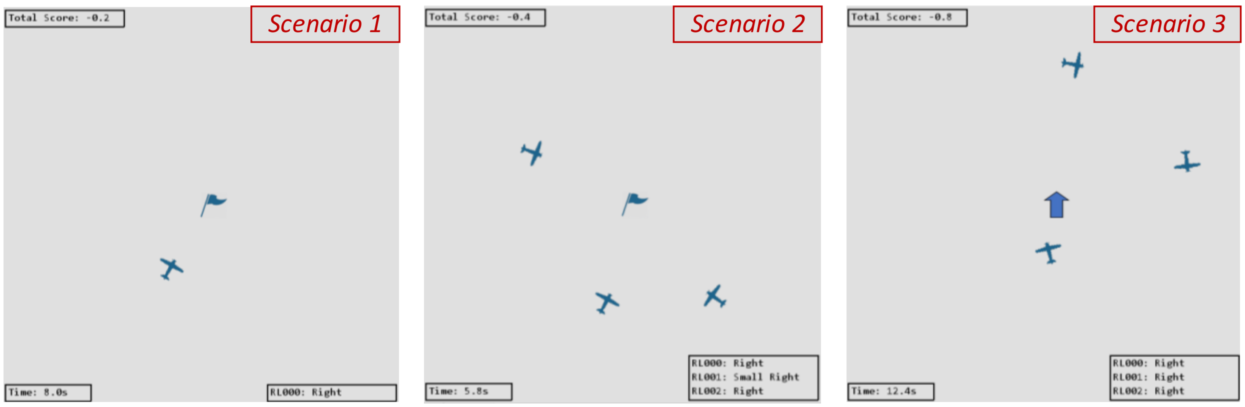


Figure 3-1: Learning Scenarios Pygame Visualizations

3.2.4 Learning Dashboard

During the main policy training, a dashboard is regularly plotted to follow the progress of the learning. As shown in Figure 3-2, the dashboard displays the evolution of metrics related to the agents’ performance as well as the neural networks training target and loss.

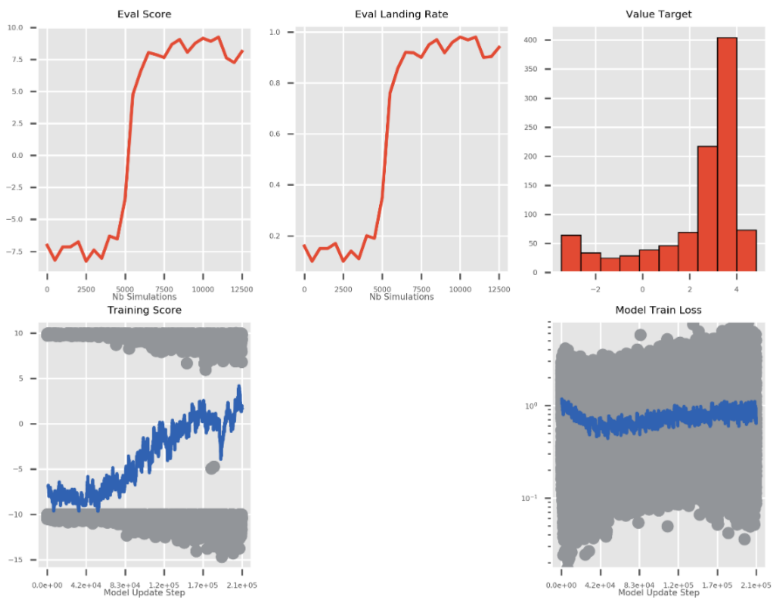


Figure 3-2: Training Dashboard (Scenario 1 run)

The "Eval Score" and "Eval Landing Rate" subplots are representative of the performance of the policy at given time during training. The policy evaluation is performed by setting the epsilon parameter close to 0 meaning that all actions are derived directly from the policy's model. Typically, 100 simulations are launched and the average score for all the agents as well as the average landing rate are display on the two subplots. While both metrics a closely related, difference in trends can add useful information on policy performance. Indeed, we want to distinguish whether an aircraft finally reaches its waypoint after 200 time steps or only 10 time steps and whether it reaches its waypoint at all.

The "Training Score" subplot shows the total score for all agent for the simulations launched during training. The grey circles are parts of a scatter plot of the scores and the blue line is the moving average. Figure 3-2 is example of a dashboard produced when running Scenario 1 simulations with only 1 aircraft. We can clearly see two cluster of scores values around +10 and -10 corresponding to the fact that the aircraft reached its waypoint or not. Additionally, as the policy improves (and epsilon decays to zero), some scores go slightly lower than the clusters representing the fact that aircraft spend more time on the environment as explained in Section 2.1.2, the agent receives a small negative reward each time it steers in the environment.

The "Value Target" subplot displays a histogram of the current model's value target estimated by sampling uniformly 1,000 transition from experience replay. The value target dramatically evolves as the model changes. Performing "sanity" checks on this subplot allows to make sure the model is not showing signs of divergence.

The "Model Train Loss" subplot shows the evolution of the training loss observed during the policy updates. The grey circles are parts of a scatter plot of the training loss and the blue line is the moving average. This is the first subplot used to check if the model training is performing adequately. It is reasonable to expect that the training loss must decrease during training and an increasing loss must be an alarm. It is the case here since the target values are always evolving with the improvement of the policy, it is not alarming to see the loss increasing overall. However, the training loss must be decreasing in between the target model updates (Section 2.4.2.1) similarly to Figure 3-3.

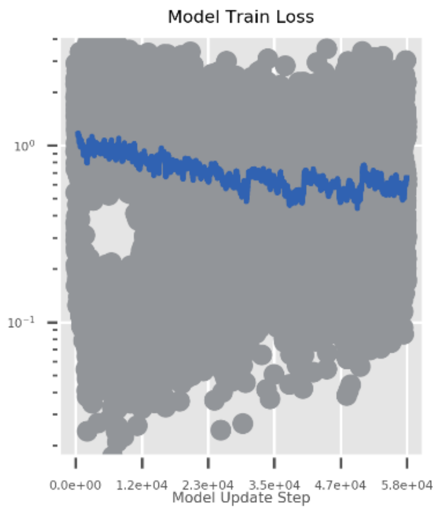


Figure 3-3: Model training loss behavior during first update steps

3.3 Refinement

The following section describes the successive results and improvements for each learning scenario. The code and all dashboards related to the different runs can be found in the ***report runs*** folder.

Scenario 1

In Scenario 1, a single aircraft is evolving in an environment where a single waypoint is located. This is the simplest scenario possible for this project. Therefore Scenario 1 was used as the “unit test” for all modification to the learning script.

The random policy, choosing randomly the agent’s actions, yield a landing rate of the 10.6% and an average score -8.11. This policy is used to initialize the experience replay before starting the training. If the experience replay was empty at the beginning of training, the agent would overfit its first experiences.

The std_nn architecture gives satisfactory performance after run approximately 150k simulations and 1.8m update steps. The best landing rate is 86.7% with an average score of 7.01.

The conv_nn architecture converges to a “near perfect” policy much faster. Indeed, a landing rate >85% is reached after only 7k simulations, which is before the epsilon parameter reaches 50%. In other words, this training setting allows to generalize even while the experience replay is comprised of transitions where the agent took half or more than half of its action randomly. The policy reaches a landing rate of 96.4% and an average score of 8.83.

Due to its much slower convergence (20x), the std_nn architecture seems to “remember” more than “generalize” when compared to the conv_nn architecture. One interpretation is that the conv_nn generalizes the spatial relationship of the input (see 2.4.2) while the std_nn has no clue whether the aircraft is at a location close to where the action-state value is already well estimated, the std_nn must learn it for each possible location almost as if we were using a tabular version of Q-learning.

The std_nn architecture will not be used for the rest of its report due the costs (on AWS) associated with training such a slow model.

However, as shown in Figure 3-4, both the std_nn and conv_nn, the learning process is not stable after the policy reaches its high point. When visualizing the policy after the performance decrease has occurred, we can observe that the agent is stuck in loops (see video).

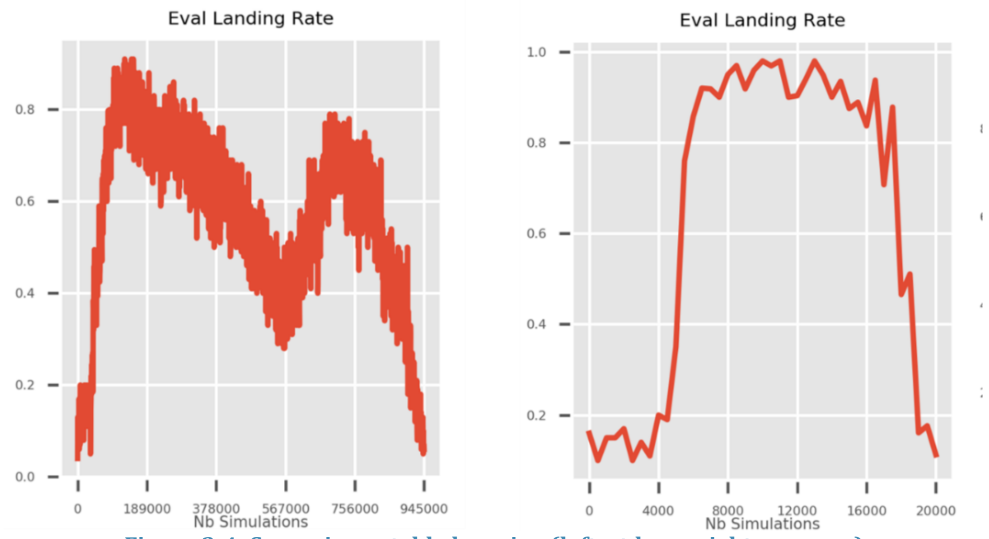


Figure 3-4: Scenario unstable learning (left: std_nn ; right: conv_nn)

One solution that was found effective was to adaptively normalize the observed reward value at the same frequency as the fixed network weight update rate. The reward values are normalized with $\psi(\mathrm{r}) = \frac{r - avg\_r}{std\_r}$ with *avg_r = .95*avg_r + .05*latest_mean* and *std_r = .95*std_r +.05*latest_std* such as *latest_mean* and *latest_std* are computed over the latest transitions in experience replay memory.

At the end of training, when the policy yields positive rewards, adaptive reward normalization allows to separate more accurately the "good" rewards from the "best" reward such as the policy does not diverge into suboptimal behaviors.

When the reward normalization is used with the conv_nn architecture yields the best results for Scenario 1. The final landing rate is 97.2% and an average score of 9.23. The behavior of this policy can be watched here and its performance dashboard is shown on Figure 3-5.

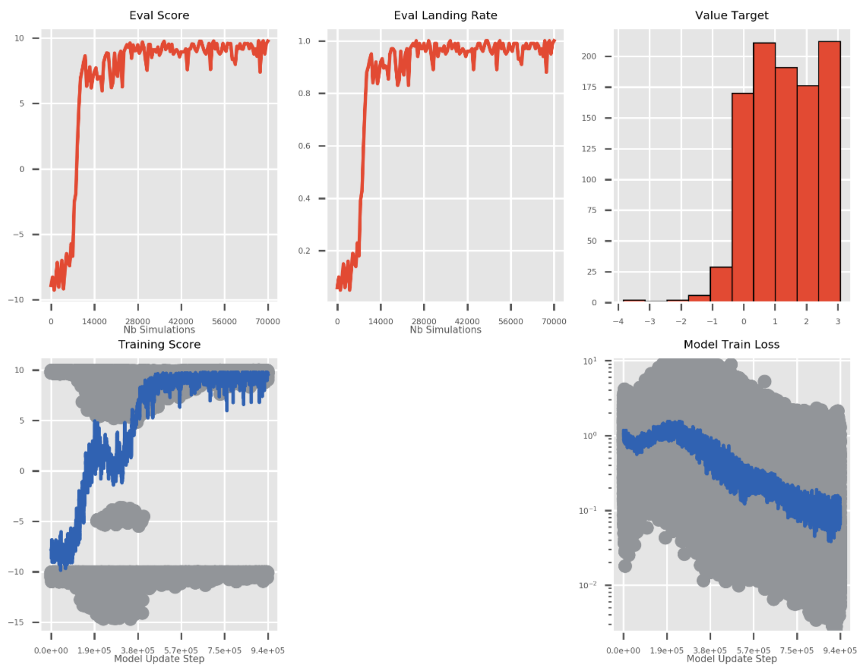


Figure 3-5: Scenario 1 conv_nn with adaptive reward

Scenario 2

In Scenario 2, 3 aircraft are evolving simultaneously in an environment where a single waypoint is located. This scenario allows to assess whether the policy can maximize the chance of landing for an individual aircraft while interacting with itself.

The random policy, choosing randomly the agent's actions, yield a landing rate of the 9.9% and an average total score over the 3 aircraft of -24.72. The performance dashboard shows the total score for the 3 aircraft, this allows to understand better the evolution of the policy.

Using the setting that worked best on Scenario 1 (conv_nn with adaptive reward), the model training loss diverged regardless of the learning rate utilized (see Figure 3-6). The best performance was only a 23.1% landing rate and an average score of -17.49. The policy can be watched here.

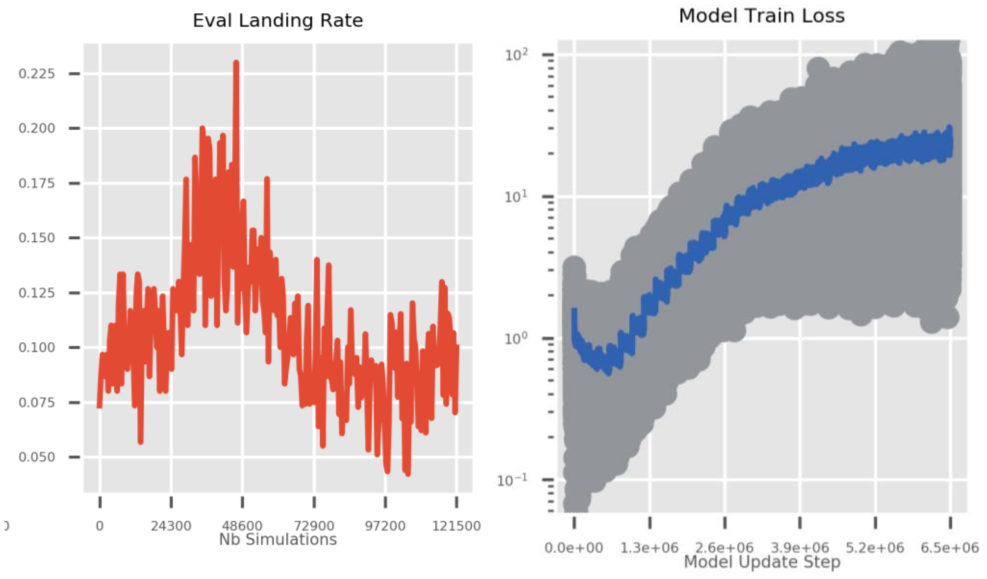


Figure 3-6: Scenario 2 First setting – Landing Rate and training loss

Compared to Scenario 1, the experience replay of Scenario 2 contains more "surprising" transitions since collisions between aircraft can happen anywhere in the environment. In Scenario 1, agents received rewards at well-defined locations on the environement (edges and the waypoint).

Monte Carlo (MC) learning was found to be more suitable for this scenario. As described in Section 2.4.1, MC learning utilizes final return $G_t = \sum_{i=t}^{T} \gamma^{i-t} R_i$ instead of the Temporal Difference $R_t + \gamma\, q_\pi(S_{t+1}, A_{t+1})$ or the Q-learning target $R_t + \gamma \max_a q(S_{t+1}, a)$. Using Monte Carlo learning, Algorithm 2 becomes the revised version Algorithm 2Bis where the final return G_t for each transition is computed at the end of the episode and the learning target is replaced by G_t.

Algorithm 2bis

```
Initialize replay memory $D$ with $I$ transitions sampled with a random policy
Initialize action-value function $Q$ weights $\theta$ and $\theta^-$
for episode = 1, M do
    Initialize $s_1$ and normalize it $\varphi_1 = \varphi(s_1)$
    Initialize episode memory M
    for t = 1, T do
        Increment time step $t$ += 1
        with probability $\varepsilon$ eps select a random action at
        otherwise select $a_t = \underset{a}{\operatorname{argmax}} q(\varphi_t, a;\ \theta)$
        Execute action $a_t$ in the simulator and observe reward $r_t$ and state $s_t$
        Initialize transition priority at max_td_error
        Set $s_{t+1} = s_t$ and preprocess $\varphi_{t+1} = \varphi(s_{t+1})$
        Store transition $(\varphi_t, a_t, r_t, \varphi_{t+1})$ inside $M$
        if t is a multiple of $F$ do
            Sample using the priority distribution minibatch of $m$ transitions $(\varphi_j, a_j, g_j, \varphi_{j+1})$ from D
            ~~Set $y_j = \psi(r_j) + \max_a q(\varphi_{j+1}, a; \theta^-)$~~
            Perform a gradient descent step on $(g_j - q(\varphi_{j+1}, a_j; \theta))$
            using Adam and the Importance Sampling weights
            Update the priority of the sampled transitions
        end if
    end for
    Compute final returns $g_t$ and store $(\varphi_t, a_t, g_t, \varphi_{t+1})$ inside D
end for
```

Using Algorithm 2Bis with the conv_nn architecture and adaptive return (not reward) normalization, some significant gains in performance were obtained. The landing rate achieved was 82.4% and the average score was 18.61. The satisfactory policy can be visualized here.

Compared to Q learning, Monte Carlo provides an unbiased estimate of the expected value for a given state-action it is based on the episodes' final return while Q learning utilizes bootstrapping from the next state value approximation. The fact that collisions may happen anywhere on the environment may cause the "bootstrapped" approximations of Q learning to diverge over time. This may be solved by increasing even more the duration while the target network is frozen, reduce the learning rate, increase the size of the experience replay memory and/or slow down the decay of the epsilon parameter. However, no practical solutions to stabilize Q-learning were found.

The MC learning policy can be watched here. While the policy provides acceptable performance, the fact that aircraft often reach the waypoint simultaneously seems un-natural. This is not the policy's fault but rather the experiment scenario.

Scenario 3

In scenario 3, the directional waypoint is introduced. This kind of waypoint has a heading attribute and an agent reaches successfully the waypoint only if it current heading is the close to the waypoint's heading.

The random policy, choosing randomly the agent's actions, yield a landing rate of the 9.7% and an average score -24.84 for scenario 3.

Using MC learning and the conv_nn architecture, the policy achieved a 70.3% landing rate and average score of 8.68. The policy can be watched here.

A new convolutional architecture, conv_nn+, was experimented with to increase performance. The new architecture doubles the numbers of filters at each convolutional layer and double the number of units in the fully connected layer. As shown on Table 3-1: conv_nn+ architectureTable 3-1, conv_nn+ has around 5 million scalar which is quadruple the size of the conv_nn architecture.

Table 3-1: conv_nn+ architecture

Layer	**Input**	**Filter size**	**Stride**	**Nb filters**	**Activation**	**Output**	**Nb Weights**
conv1	3x60x60	6x6	3	**64**	Leaky ReLU	19x19x64	3*6*6*64 = 6.9k
conv2	19x19x64	4x4	2	**128**	Leaky ReLU	8x8x128	64*4*4*128 = 131.0k
conv3	8x8x128	3x3	1	**128**	Leaky ReLU	6x6x128	128*3*3*128 = 147.5k
fc1	6x6x128	-	-	**1024**	Leaky ReLU	1024	6*6*128*1024 = 4.7m
fc2	1024	-	-	5	Linear	5	1024*5 = 5.1k

Using MC learning and the conv_nn+ architecture, the policy achieved a 68.1% landing rate and average score of 8.67. Increasing the size of the neural network did not bring any performance improvement.

4 Results

4.1 Model Evaluation and Validation

The models' performance (landing rate/average score) for the different scenario was evaluated with the evaluation script *main_test,* that would run more than 1000 simulations per evaluation.

Table 4-1: Scenario 1 Results

Model #	**Architecture**	**Adaptive Reward**	**Nb simulations until best score**	**Learning Behavior**	**Landing Rate**	**Score**
1	**random**	N/A	N/A	Stable	10.6%	-8.11
2	**std_nn**	No	150k	Unstable	86.7%	7.01
3	**conv_nn**	No	10.5k	Unstable	96.4%	8.83
4	**conv_nn**	Yes	28k	**Stable**	**97.2%**	**9.23**

Table 4-2: Scenario 2 Results

Model #	Architecture	Adaptive Reward	Learning Type	Nb simulations until best score	Learning Behavior	Landing Rate	Score
1	**random**	N/A	N/A	N/A	Stable	9.9%	-24.72
2	**conv_nn**	Yes	DQN	48k	Unstable	23.1%	-17.49
3	**conv_nn**	**Yes**	**MC**	**200k**	**Stable**	**82.4%**	**18.61**

Table 4-3: Scenario 3 Results

Model #	Architecture	Adaptive Reward	Learning Type	Nb simulations until best score	Learning Behavior	Landing Rate	Score
1	**random**	N/A	N/A	N/A	Stable	9.7%	-24.84
2	**conv_nn**	Yes	MC	**175k**	**Stable**	**70.3%**	**8.68**
3	**conv_nn+**	Yes	MC	170k	Stable	68.1%	8.67

When increasing the number of aircraft in the environment from 3 to 4, the Scenario 2 and 3 best models proved to be somewhat robust. The landing rate for Scenario 2 with 4 aircraft is 71.6% compared to 82.4% previously and landing rate for Scenario 3 became 55.8% from 70.3%

One could argue that having a landing rate lower 99.9% would be unacceptable for an air transportation project. However, this is a just an exploration of the benefits of some ML techniques and the model results seem reasonable when visualizing the policy. Some improvements are definitely needed as described in section 5.3.

4.2 Justification

As mentioned in Section 2.5, the random was chosen as the performance benchmark. As shown in the previous, every scenario best model achieves significantly better landing rate and score performance than the benchmark over the 1,000-simulations evaluation runs. The models pass easily the benchmark's performance threshold.

However, some qualitative observations on the models' performance can be made.

The failed landings in Scenario 1 (video) with the best policy corresponds often to the aircraft being initialized close to two edges of the airspace. The aircraft fails to perform a full turn to get out of the corner.

In Scenario 2 (video), two aircraft can be initialized with conflicting headings too close to each other, the crash is unavoidable. Another observation is that the third aircraft sometimes "shy away" from the waypoint area and even exit the airspace at the beginning of the simulation.

In Scenario 3 (here), aircraft sometimes does circulate in the same direction (clockwise/counterclockwise) which may create unsolvable trajectory conflicts.

5 Conclusion

5.1 Free-Form Visualization

As noted in Section 2.4, the goal of this project's models is to estimate what is the value of a given state compared to another state given the policy, if not the exact state values. Visualizing the state values estimated by the policy's model allows to have a "feel" of the quality of the policy.

For each scenario's best policy, several thousand simulations were ran with only one aircraft in the environment and the transitions stored in the experience were then post-processed. The aircraft position on the last frame (60x60 pseudo state) of each state (3x60x60) receives the maximum value (over possible actions) estimated by the policy's model. A 60x60 heatmap with the mean pseudo state values is then plotted.

As expected, the Scenario 1 value map (Figure 5-1) has an axial symmetry with increasing state value closer to the center. The center is empty since the simulation stops when the aircraft reaches the waypoint.

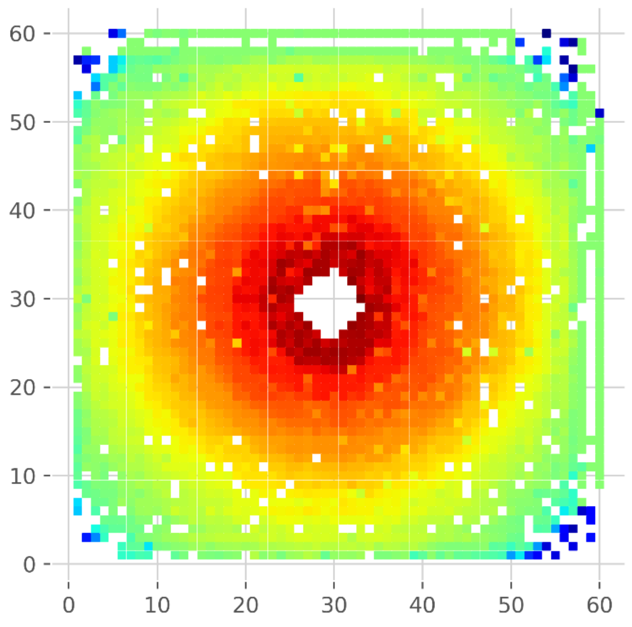


Figure 5-1: Scenario 1 Value Map

A "fake" aircraft was introduced in the environment's state grid (40,40) position when evaluating the Scenario 2 value map (Figure 5-2). As the policy tends to guide the aircraft directly toward the center of the environment, there is a low value state area is present next to the "fake" aircraft on the outer part of the airspace while there is a high value area on the inner part of the environment despite the proximity to the fake aircraft.

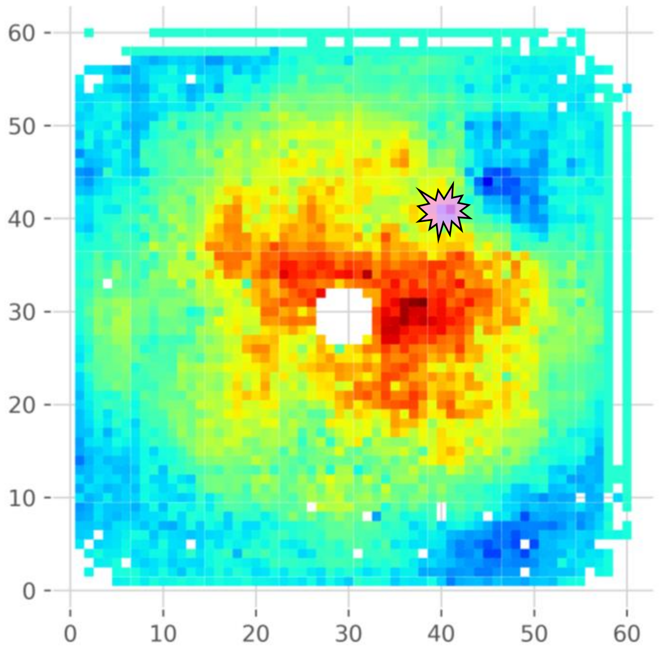


Figure 5-2: Scenario 2 Value Map ("fake" aircraft at [40,40])

The Scenario 3 value map was estimated by fixing the waypoint's heading to 180 degrees. As shown on Figure 5-3, the presence of the fake aircraft in Scenario 3 "shuts down" entirely the right part of the value map.

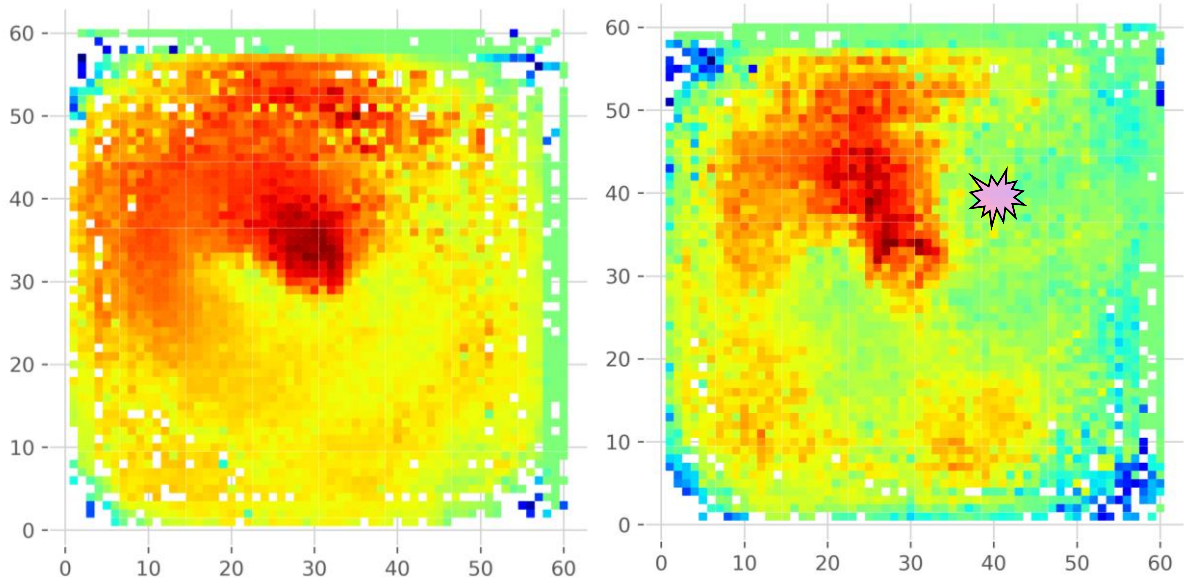


Figure 5-3: Scenario 3 Value Map (right: 180 deg waypoint / left: "fake" aircraft at [40,40])

5.2 Reflection

The project has successfully illustrated that some simplified examples of wayfinding and conflict avoidance in the context of multi-agent transportation could be learnt directly from experience. The development of the traffic simulator allowed to have a flexible learning environment that allowed to analyze several learning scenarios.

The policy training process evolved from a standard DQN setting to a more complex system including prioritized replay and adaptive reward normalization. Monte Carlo learning was found to be more effective than Q learning in the more advanced learning scenarios.

The convolutional architecture proved to be crucial at catching spatial relationships between the aircraft and waypoint. Through an extensive number of policy iterations on the experience replay data, the model was even capable of remembering “strategies” to handle multiple agent conflicts.

However, training large neural networks while using the networks for creating new experience data proved to be very time consuming. For instance, some models required, for scenario 2 and 3, more than 2-3 days of training time on an AWS GPU instance. This fact limited the ability of optimizing the numerous model hyperparameters and clearly slowed down the model development process.

5.3 Improvements

As stated above, the training speed is one of the major bottlenecks of this project. Optimizing the efficiency of the training would bring a lot of benefits to the model iteration capability. The way prioritized replay works could be optimized further by avoiding computing the probabilities too often and use buckets of probability instead of exact values.

The conv_nn neural network architecture works well for identifying spatial pattern but does not have the ability to advantage of the time dependencies between frames and agents action. A Long Short-Term Memory (LSTM) network could be put on top of the convolutional layers to add the possibility of processing time based relationships.

Policy gradient method such as Actor-Critic could also be explored. The projects code would allow to implement this method easily but creating another Policy subclass instead of the DQN class.

A more polished version of this project could include hardcoded “safety” rules along with the Machine Learning brain. Instead of placing all the responsibility of agent’s actions on the Machine Learning models, a programmatic set of “safety rules” could override the models’ decisions if required for pre-determined safety reasons. This type of “hybrid” system would require extensive expert knowledge to set the safety rules as well as thorough validation/test to rule out any conflict between the set of rules and the Machine Learning models.