Master of Aerospace Engineering

Research Project: S3 Project report

Using conditional GANs to develop a realistic human-robot interaction simulator

Author:
Guilherme Penedo

Tutors:
Giorgio Angelotti
Caroline Chanel
Nicolas Drougard

March 2022

Declaration of Authenticity

This assignment is entirely my own work. Quotations from literature are properly indicated with appropriated references in the text. All literature used in this piece of work is indicated in the bibliography placed at the end. I confirm that no sources have been used other than those stated.

I understand that plagiarism (copy without mentioning the reference) is a serious examinations offense that may result in disciplinary action being taken.

Date: 18th of mars 2022
Signature:

[illegible]

Abstract

By modeling human behaviour a supervision policy may be planned to oversee task allocation on a human-robot system, so that this may be done in an optimal manner. Human behaviour is inherently multimodal - in a given context, various possible actions are acceptable - and so generative models and, more specifically, Generative Adversarial Networks (GANs) are used in this project to create a human-robot interaction simulator capable of generating realistic human-like actions that can then be used to estimate a supervision policy. A dataset from the "Firefighter Robot Mission" game is used, and different models were trained to generate player keypresses with a distribution approaching that of the original dataset. By using conditional GANs, game state features can be incorporated into the model, and by using LSTMs, temporal dependencies may be exploited to further improve the model. Different loss functions and validation metrics are considered and possible avenues for improvement are also discussed.

Keywords: human–robot interaction; mixed-initiative mission; generative models; generative adversarial neural networks; conditional adversarial neural networks

Contents

1 Introduction

To design effective human-robot systems one key approach is to have a dynamic assignment of tasks between the human operator and the automated system. To this end, a policy can be developed to decide when to transfer control from the human to the automated system and the reverse, based on the current state and the operator's mental condition [1].

The system's operation can be modeled as a Markov Decision Process (MDP) to optimize one such policy, as done in [2], using crowdsourced data from an online game where participants have to control a firefighting robot. This yields a probabilistic representation of possible transitions for a given system state [3].

To build upon [2], Reinforcement Learning can be used to create a generative model to mimic a human's behaviour for a particular mission state, thus allowing for the creation of a human-robot interaction simulator that can be used to further optimize a supervision policy, without the logistic difficulties inherent to observing an actual human responding to the different situations.

Since the particular shape of the data corresponding to a human's actions (the data that is to be generated) is not known *a priori*, Generative Adversarial Networks (GANs) [4] will be used on the firefighting robot dataset from [2] to sample from the interaction data's distribution without imposing a prior on its analytical form.

Human behavioural data is multimodal in the sense that, for a given context, multiple different actions are possible and perfectly reasonable. As such, common deterministic approaches that attempt to produce the average prediction which minimizes a distance metric from the real value are not adequate. By using GANs, the multimodality of the data can be preserved: for a given moment in the game there are multiple possible predictions, since sampling multiple times from the GAN will generate different outputs, for the same game state.

Conditional Generative Adversarial Networks (CGANs) [5] add additional inputs to both the generator and the discriminator, in order to allow the GAN's generated data to be conditioned on some criteria, therefore allowing for the generalization of a GAN without the need to train many different individual GANs. In this project, a CGAN will be used to condition the data generated on certain variables of the firefighting robot's game state, so that the generated data will depend on the current state of the game.

CGANs are used in this project to generate samples based on a single previous time interval/observation and, later on, Long short-term memory networks (LSTMs) [6] are also introduced. This allowed the input to include data from multiple previous time intervals, thus allowing the models to learn temporal relations in the data which may produce more accurate generated samples.

Different architectures and training methods involving GANs are considered and quantitative and qualitative methods of validation are proposed. The models developed in this project are able to, given a history of game states and control actions in the "Firefighter Robot Mission" game, generate a possible future control action that a real human could also have taken. These models may, therefore, be used to estimate a supervision policy, thus fulfilling the project's main goal.

This report is organized as follows: first, relevant Related Work is briefly discussed; then, an overview of the problem to which the project is applied is presented, followed by a recall of the work developed in the previous semesters. Following that, the main proposed model is detailed and explained, after which quantitative evaluation metrics and results are presented, followed by additional qualitative experiments on the results and, finally, the conclusion.
The code created for this project with PyTorch[1] implementations of the different models can be found on the following github repository:
https://github.com/guipenedo/horizon-ganns.
The models used were trained on ISAE-SUPAERO's PANDO supercomputer's NVIDIA A100 GPUs.

[1]https://pytorch.org/

2 Related Work

Generative Adversarial Networks (GANs) were introduced in [4] and consist of two neural networks: the generator, which takes as input random noise and generates data whose distribution shall ideally approach that of the real data, and a discriminator which classifies data as either real (belonging to the training set) or as generated. The two neural networks have opposing objectives, as the generator seeks to produce data resembling that of the training set, thus rendering the discriminator unable to distinguish between the two, while on the other hand the discriminator tries to improve its ability to predict whether the data comes from the generator or the original dataset.

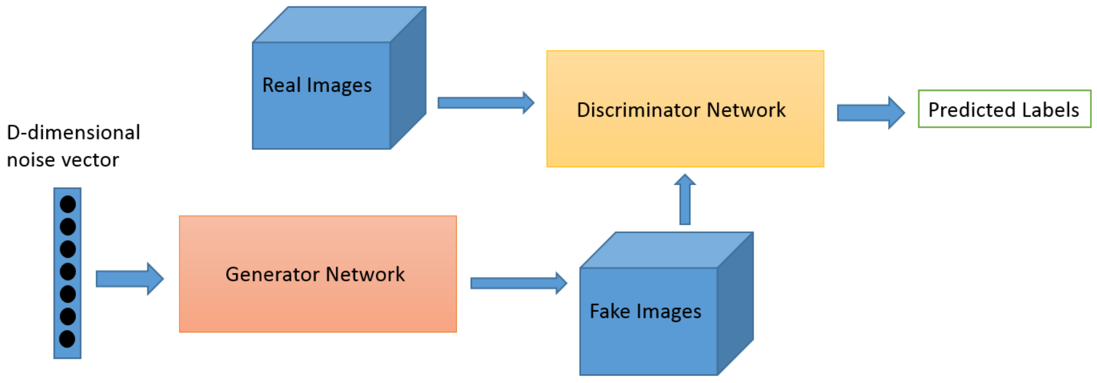


Figure 1: GAN architecture. Credit: Jon Bruner and Adit Deshpande, O'Reilly

A schematic view of this architecture can be seen on Figure 1, in which there is a dataset consisting of images and a GAN is used to generate new realistic images. As such, a generator network taking as input a noise vector and giving as output an image is used along with a discriminator which, given an image, yields as output the probability (in $[0, 1]$) of this image belonging to the "real images" dataset. This is formalized on the loss function used to train the GAN from [4]:

$$\min_G \max_D \mathbb{E}_{x \sim p_{data}(x)} \big[logD(x)\big] + \mathbb{E}_{z \sim p_z(z)} \big[log(1 - D(G(z)))\big] \tag{1}$$

In equation (1), p_{data} is the distribution the training data comes from (the real images); x is a single image; z is a latent vector (the noice vector that is the input of the generator) and p_z the distribution it was sampled from (usually a standard normal distribution). $D(x)$ corresponds to the discriminator, and is the estimated probability that sample x comes from p_{data}, while $G(z)$ is an image created by the generator for the input z (which, again, is the noise vector). The entire expression is maximized if the discriminator correctly classifies the real data with a value close to 1 and the generated data with a value close to 0, and therefore the discriminator is trained with this goal. On the other hand, the generator seeks to minimize this expression. In practice, both networks are trained in each training iteration. It should be noted that in the classic GAN formulation the generator does not have direct access to the training data and only learns through its interactions with the discriminator [7].

Ideally, this minmax problem will converge to $p_{data} = p_g$, in which p_g is the distribution of the data generated by the generator, i.e., the GAN will converge so that the generator implicitly learns the distribution of the real data, or, in other words, when the discriminator is unable to tell whether a sample is from the training set or generated. In pratice, this is quite hard to achieve and convergence is still being actively researched.

Simple GANs generate data without taking into account any specific context or state information. One such example is the generation of black and white pictures representing handwritten images (based on the MNIST dataset[2]): a GAN trained with this dataset will generate new images resembling the ones it was trained with; however, there is no way to request the GAN to only generate images of a particular digit, without training 10 different GANs, one for each digit.

To solve this issue, Conditional GANs were introduced in [5]. They extend GANs by adding an additional input to both the generator and the discriminator: the condition, which will influence the outputs of the GAN.

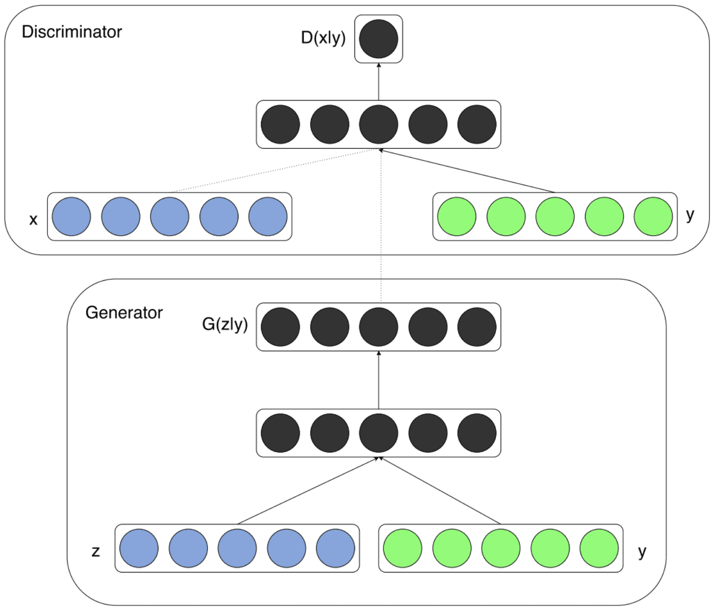


Figure 2: CGAN architecture. Credit: [5]

In Figure 2, z is, as before, a latent vector (a vector containing noise/random values), and

[2]https://en.wikipedia.org/wiki/MNIST_database

x is a vector containing a sample's data. However, a new vector, y is introduced, which is the condition. The condition is thus incorporated into both the discriminator and the generator, which discriminate and generate, respectively, under this condition. For example, for a CGAN trained with the MNIST dataset, $D(x|y)$ would not be simply the estimated probability that image x belongs to the training dataset, but the estimated probability that image x belongs to the training dataset given that it is meant to represent digit y.

Equation (2) reflects these changes to the original loss function of equation (1).

$$\min_G \max_D \mathbb{E}_{x\sim p_{data}(x)}\big[logD(x|y)\big] + \mathbb{E}_{z\sim p_z(z)}\big[log(1 - D(G(z|y)))\big] \tag{2}$$

This principle, and CGANs in general, can be adapted to have a single model generate data for different conditions or states, for instante, for the automatic or manual control modes of a human-robot system.

GAN training is often unstable, and prone to problems such as mode collapse, wherein a large range of values in the latent space (the noise vectors are drawn from this space) are mapped to the same generator's output (basically the variability of the data the generator produces is affected).

In [8] a possible avenue to mitigate these effects is introduced, by using a new loss function, based on a concept known as the Wasserstein or Earth Mover distance, as well as changing some parts of the architecture (the discriminator no longer outputs a probability, for instance). This new GAN is called the Wasserstein-GAN, or WGAN.

The new loss function, based on the Earth Mover distance, is the one in equation 3, where p_g is implicitly defined by $\widetilde{x} = G(z)$, $z \sim p_z(z)$:

$$\min_G \max_D \mathbb{E}_{x\sim p_{data}(x)}\big[D(x)\big] - \mathbb{E}_{\widetilde{x}\sim p_g(z)}\big[D(\widetilde{x})\big] \tag{3}$$

However, for these improvements to be achievable, an extra constraint must be applied to the discriminator (which in this work is called the critic): the Lipschitz constraint. This is quite difficult to optimize and the method chosen by the authors consists in clipping the values obtained.

The WGAN may sometimes still fail to converge or generate poor samples, due in great part to the weight clipping used because of the Lipschitz constraint. To overcome these problems, another work, [9], suggests changes that make weight clipping unnecessary, by introducing a new term in the loss function: gradient penalty. This new model is called WGAN-GP.

Generative Adversarial Networks (GANs) have recently also been applied to human related time series problems, such as in synthesising medical data that may be used without

anonymity concerns (MedicalGan [10], SenseGan[11] and RCGAN [12]); and in human trajectory prediction (SocialGan [13], SocialWays [14] and SophieGan [15]).
Recurrent Conditional GAN (RCGAN [12]) uses Recurrent Neural Networks (RNN) to generate medical time-series data conditioned on labels from the original data. The idea behind this base architecture is also used in this project.

Human trajectory prediction, in particular, has many similarities to the problem this project seeks to solve and, therefore, the already mentioned works on this topic are specially relevant; however, it should be noted that while these works generate the trajectory positions directly, our focus will be in the control actions taken by humans rather than on the trajectory changes which may result from them, since this project's goal is to have a model to which policy planning based on human's actions may be applied.

When it comes to validation of results, some works have attempted to use the usefulness of the generated data as an indicator of its quality, by testing on real data the performance of supervised learning models trained on the synthetic data ([12] and [16]); however, this requires the existence of a supervised learning task that may be applied to the dataset, which, as far as the author is aware, is not the case in this project's robot game problem. Additionally, the works ([13], [14] and [15]) focused on trajectory generation usually resort to validation metrics such as Average Displacement Error (ADE) and Final Displacement Error (FDE), which take into account the Euclidean distance between the generated data and the actual trajectories in the dataset; since in this project the goal was to generate the control actions and not those actions's results (specifically, the trajectories resulting from those control actions), these metrics are not adequate for quantitatively validating this work's results. Instead, the precision and recall of the generated control actions, compared to the original dataset, will be used to quantify performance.

One of the big issues GANs suffer from is mode collapse, a phenomenon in which the outputs of a GAN "collapse" into a small number of generated samples that produce a good adversarial loss. This obviously harms model performance as it affects multimodality and the variety of possible predictions. This issue is even more prevalent when we couple the adversarial loss with an L2/Mean Squared Error loss, which forces the model to reduce the Euclidian distance between the original dataset and the generated samples. The Variety Loss, used in SocialGan [13], attempts to find a balance between relying solely on the adversarial loss, which is quite slow to converge, and causing mode collapse with the L2 loss by only using the L2 loss on the closest prediction to the target from the original dataset out of a number of different sampled predictions. SocialWays dispenses with the L2 loss altogether and instead focuses on the InfoGAN architecture [17].

3 Problem formulation: Firefighter Robot Mission (FRM)

An experiment was developed in [1], in which a game, called Firefighter Robot Mission[3], was set up in order to collect behavioral and psycho-physiological features from human operators.

The game itself has a graphical user interface (GUI), pictured in Figure 3, through which the player is expected to control the movement of a robot that has a water deposit, in order to extinguish fires on nearby trees.

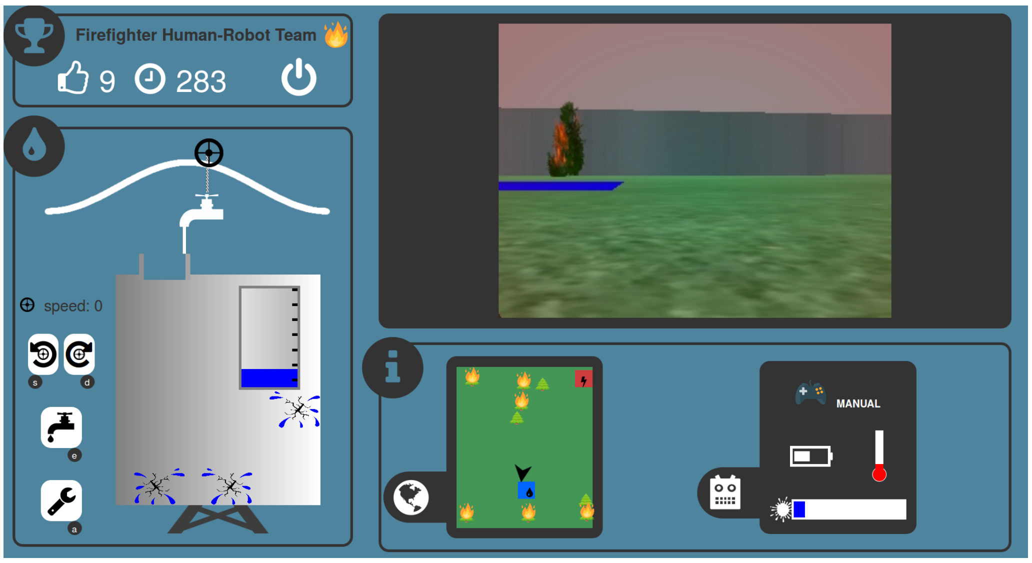


Figure 3: Game GUI. Credit: [1]

On the top right, a simulated viewpoint from the robot can be seen, while on the bottom right there are two panels: one minimap, showing the state of the 9 trees, the robot position and the places where the robot can refill its water tank (the blue square) and recharge its battery (the right square) and another panel showing the current control mode (manual or automatic, the robot's battery, temperature and water tank level). On the left, we can see the ground tank which corresponds to the blue square on the minimap, where randomly leaks appear, which the player must fix, and where the tap that refills this ground tank has to be activated while dealing with an unstable equilibrium point.

The robot's movement is controlled through the 4 arrow keys, while the space key is used to shoot water, with which the robot can extinguish the fires. The mouse is meant to be used to fix the leaks and to refill the water tank (by interacting with the left panel).

[3]http://robot-isae.isae.fr

A dataset containing game transitions: tuples containing the information about a state (which trees are on fire, battery, temperature, water levels, etc.) and player actions (keys pressed and clicks made) is available online[4]. One of the end goals of this research project is to be able to apply GAN models to data from this dataset, in order to better estimate an efficient supervision policy.

The structure of the data in this dataset is thoroughly detailed in [2]. Each game played is stored in a csv file. Each line of the csv file corresponds to a 1 second interval of the game. The data in each line contains both the game state and the human's response.

An example containing 5 consecutive lines of dataset CSV data can be seen in Tables 1, 2 and 3 (all three tables show data for the same 5 lines).

Table 1: Example CSV data from the dataset (1)

remaining_time	**robot_mode**	**alarm**	**robot_x**	**robot_y**	**robot_theta**
600	0	-1	-3.71421901945723E-06	-4.97447854286293E-06	0.000165233519510366
599	0	-1	-3.1673951070843E-06	-5.19860259373672E-06	0.000168666141689755
598	0	-1	-0.000108095133327879	5.17127300554421E-05	0.132011637091637
597	0	-1	-0.000398845179006457	0.000222028131247498	0.65968656539917
596	0	-1	-0.000375839008484036	0.000281440967228264	1.19851064682007

Table 2: Example CSV data from the dataset (2)

forest_state	**battery_level**	**temperature**	**water_robot_tank**	**water_ground_tank**	**leaks_state**
0	100	20	50	50	0
0	100	20	50	50	0
0	99	20	50	50	0
0	99	20	50	50	0
0	98	20	50	50	0

Table 3: Example CSV data from the dataset (3)

keys	**clicks**	**errors**	**shortcuts**
-1	-1	-1	-1
left	-1	-1	-1
left-left-left-left-left-left-left-left-left-left-left-left-left-left-left-left-left-left-left-left	-1	-1	-1
left-left-left-left-left-left-left-left-left-left-left-left-left-left-left-left-left-left-left-left-left-left-left-left-left-left-left-left-left-left-left-left-left	-1	-1	-1
left-left-left-left-left-left-left-left-left-left-left-left-left-left-left-left-left-left-left-left-left-back-back	-1	-1	-1

3.1 State data

The state consists on the following: the *remaining_time*, which starts at 600 (each game has a max duration of 600 seconds); *robot_mode*, one of the most important values given

[4]https://personnel.isae-supaero.fr/isae_ressources/caroline-chanel/horizon/

Table 3: Ranges for the different game state features

Feature	Range
remaining_time	$[0, 600]$
robot_mode	$\{0, 1\}$
alarm	$\{-1, ..., 6\}$
robot_x	$[-20, 20]$
robot_y	$[-20, 20]$
robot_theta	$]-\pi, \pi]$
forest_state	6 bits mask
battery_level	$[0, 100]$
temperature	$[20, 240]$
water_robot_tank	$[0, 100]$
water_ground_tank	$[0, 100]$
leaks_state	9 bits mask

the context of the experiment, which is either 0 (manual) or 1 (automatic); the robot position and heading, made up of *robot_x*, *robot_y* and *robot_theta*; the *forest_state*, a binary string (bitmask) with a 1 for trees on fire and 0 otherwise; *battery_level*; *robot_temperature*, *water_robot_tank* and *water_ground_tank* and *leaks_state*, which is a binary string similar to *forest_state*, detailing the state of the water deposit's leaks. The full list, together with the possible value ranges, are in Table 3.

3.2 Human interaction data

The data saved pertaining to the human's interaction mostly comes down to two columns: *keys*, which is a sequence of keypresses from the set $\{'front', 'back', 'left', 'right', 'space'\}$ and *clicks*, a sequence of locations clicked from the set

$$\{'left', 'right', 'push', 'wrench', 'leak\_i'_{i=1,...,9}, 'rm\_alarm'\}$$

3.3 Existing preprocessing

As the dataset has many different features/columns, one of the main difficulties it will pose will be on the way in which its data is to be encoded/normalized. In [2], the possible (x, y) positions were divided in 9 (3×3) sectors: $\{C, N, E, S, W, NE, NW, SE, SW\}$; battery and water levels were simplified to two possible qualitative labels: "nominal" and "low"; the forest_state was used to obtain a simple integer representing the number of fires, as well as some other state variables were chosen, such as a value representing the human's intention (in terms of target position). All in all, with all the variables considered, the state space has a size of 12960. This preprocessing was not used in our own work.

4 Previous Work

In the previous phase of this project (up until Semester 2), a choice was taken to focus on the control of the robot's movements and, as such, the other main features to consider were the *robot_x*, *robot_y* and *robot_theta*. In terms of generated outputs, or control actions, since the focus is the movement, the networks generated keypresses: the four arrow keys and the space key.
Ideally, the actual sequences of keypresses would be generated, but since these are variable in length, and can in some cases be quite large (as many as 40 consecutive keypresses in a single second, probably corresponding to the player holding down on the key instead of just pressing it) a simplification was implemented: instead of generating the sequence of keys, the model would generate the number of times each of the keys was pressed.

It was decided that this model should consider 1 second intervals (corresponding to one line of the csv files in the dataset), instead of a chunk of multiple second intervals. However, in an effort to introduce some information regarding the previous game state, so that player intentions could be taken into account and some sense of continuity could be considered, 3 new columns were introduced: the difference/displacement from time interval $t-1$ to time interval t in the robot's position and heading, *robot_x_diff*, *robot_y_diff* and *robot_theta_diff*.

There are 5 keys: "front", "back", "left", "right" and "space". Therefore, the network will have 5 outputs. As the Tanh activation function was chosen for the last layer of the generator, the data corresponding to these 5 keypresses had to be normalized in the $[-1, 1]$ range. The highest number of times any key was pressed in a single second was 37, and so the number 40, the closest multiple of ten greater than this value, was chosen to be mapped to 1 while 0 (the minimum for any key) would be mapped to -1.

Regarding the game state variables to be considered, which would then be the condition to our CGAN, initially, the following variables were selected: "remaining_time", "robot_mode", "alarm", the robot position ("robot_x", "robot_y", "robot_theta", "robot_x_diff", "robot_y_diff" and "robot_theta_diff"), the forest state (which was converted into 9 individual binary values), "battery_level", "temperature" and "water_robot_tank". These values were also normalized so that they would be centered around 0 and have values between -1 and 1. This makes up a high dimensionalcondition vector, with a total of 21 variables.

A first model was created: a CGAN architecture using the standard binary cross entropy loss function (equation 2). In the hidden layers of the generator and discriminator the activation function used was the "LeakyReLU" function (Leaky version of a Rectified Linear Unit, where for $x < 0$ the function has a value of "negative slope $\times$ x" instead of 0) ; the two networks were based on dense/linear/fully-connected layers. Dropout layers were also used to improve training. However, the results were not too impressive, consisting of mostly 0 or highest possible number of keypresses.

After a few attempts, and, in particular, some further preprocessing to attempt to remove games without any keypresses whatsoever (which were heavily skewing the results towards always choosing 0 keypresses), as well as adding some simplifications (only considering keypress counts up to 10, for instance), changing the network so that for each key there are 11 outputs, and adding a softmax activation, which applies the following function to each set of 11 outputs:

$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

therefore resulting in a set of 11 numbers whose sum is 1, which may be treated as the probability of a certain key being pressed a certain number of times. Then using argmax, the number of times each key was pressed is chosen by taking the value with the highest probability.

This posed a few problems. Firstly, argmax (the function which chooses the "i" with the highest probability associated to it) is obviously not differentiable, and therefore it is not possible to apply backpropagation in this situation, rendering the training of the model impossible. The only possibility seems to be to keep the 4×11 values during training, and feed them as input to the discriminator (instead of just 4 with the actual values) but, since the real data is also fed to the discriminator as input, there is a new problem: the actual real data does not consist on probabilities for the count of each keypress, but rather on the actual number of keypresses. Simply turning this information into a one hot vector could bias the discriminator and make it artificially easier to distinguish between real and generated data. To avoid this problem, noise sampled from an exponential distribution with a rate of $\lambda = 9999$ — a distribution with a probability density function for positive numbers of $\lambda e^{-\lambda x}$, thus, for this rate, generating very small numbers that will not change the original final label — was added to the one-hot vector of the real data, which was then turned into a probability by dividing each element by the total sum of all elements.

Initially, a "vanilla" GAN architecture was used. However, later on gradient penalty (from WGAN_GP [8]) was implemented and the loss function was changed (to the one also from WGAN_GP, instead of binary cross entropy). The generator was also only trained every 5 iterations while the discriminator was trained on all iterations. After training, hundreds of thousands of samples were generated. For comparison with Figure 5, which has the key frequencies from the original dataset, the resulting frequencies of the keys in the generated data can be seen on Figure 4.

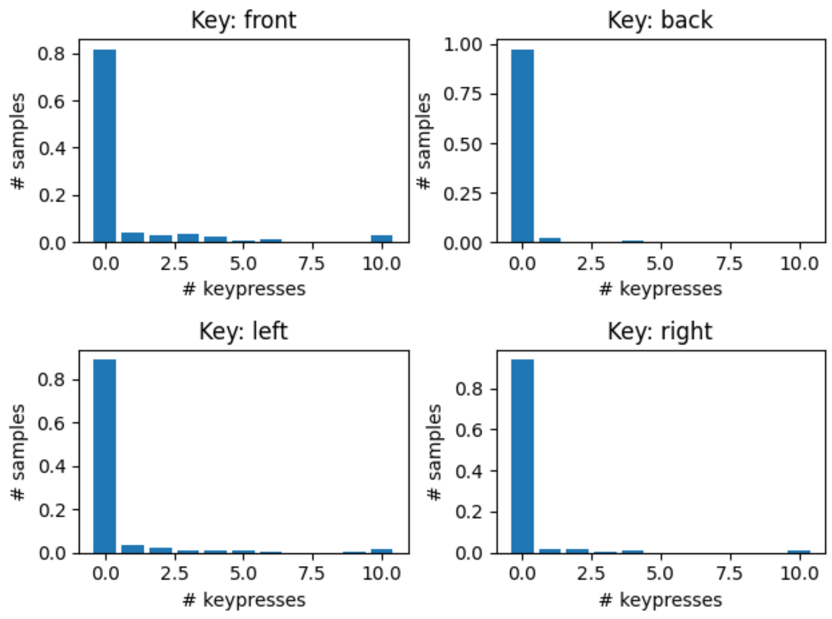


Figure 4: Frequency analysis of the keys in the generated data

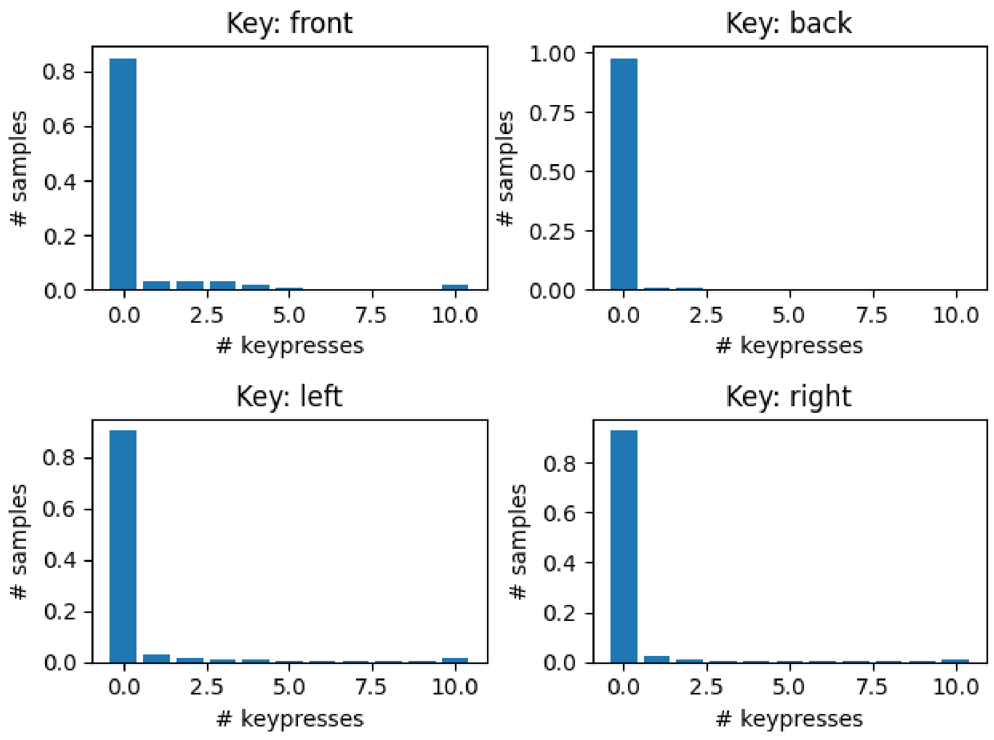


Figure 5: Frequency analysis of the keys in the original dataset (after pre-processing)

As can be seen, the frequency distributions of the generated data closely resemble those of the original dataset. From this analysis we conclude that the GAN managed to learn the underlying distribution of the dataset to a certain degree.

To compare the initial, "vanilla" GAN with the modified model following a WGAN_GP architecture, the frequency distributions for each key generated by the two models were compared to those of the original dataset. Table 4 shows the Wasserstein distance[5] between the frequency distributions of each model for a given key and that key's frequency distribution in the original dataset. It can clearly be seen that the WGAN_DP produces samples with a frequency distribution closer to that of the original dataset, as the distance between these two distributions is significant lower than with the vanilla GAN.

Model used to generate samples	Key to compute Wasserstein distance			
	front	back	left	right
Vanilla GAN	0.04833	0.03722	0.05902	0.10258
WGAN_GP	0.00674	0.00207	0.00371	0.00299

Table 4: Wasserstein distance for two types of GANs

In this initial part of the project, these overall good results for the samples generated from the WGAN_GP model were still somewhat limited, on three main aspects:

- Condition/game state - The model created did not account for conditional data as was originally intended. Moving forward, it is important to have a model conditioned on at least the robot_mode and possibly positional data.
- Validation - The validation process, in which directly sampling from the generator was performed to compare frequency distributions with the original dataset, is not possible once we have a model conditioned on different real valued parameters, such as the position. A more robust validation process was needed.
- Observations history - The current approach only takes into account the time interval immediately before when making a prediction. Incorporating a larger amount of previous observations could lead to better results.

[5]https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.wasserstein_distance.html

5 Proposed Models and Preprocessing

5.1 Dataset

In this final phase of the project (Semester 3), some simplifications were made to the way the dataset is processed. Games without a single keypress continued to be excluded. Instead of trying to generate the number of times the player pressed a certain key in a 1 second interval, which often depends more on internet connection delay than on actual intent, and keeping in mind that in a lot of cases this amount was 0 or quite close to 1 (looking at Figure 5 we see that most keypresses involve touching one key less than 5 times), the control actions were turned into binary variables: we only consider whether a key (the "front" key, for example) was pressed or not in a time interval (which are all 1 second) - we now only have two possible values to generate per key. Additionally, the games that were not excluded were processed in a sliding window fashion such that each sample consisted of 6 consecutive recordings. Of these 6 recordings in each sample, the first 5 are, in this work, called *observations*, as they will be used as input to the models that will attempt to predict what happens on the last time interval, the *prediction*, in terms of what keypresses were pressed. This preprocessing of the original dataset results in a total of 270630 samples. For reference, in these 270630 samples and, therefore, $270630 \times 5 = 1353150$ total key outputs, 95700 are positive (the key in question was pressed) and 1257450 were negative (the key in question was not pressed) for a ratio of 0.07 positives.

The focus remains on the movement related variables and, as such input features are "robot_mode", "robot_x", "robot_y" and "robot_theta". The forest state variables (which trees were on fire) were also considered on some of the tests.

5.2 Model Details

The main proposed model uses an observations encoder with LSTM layers to encode the game state and keys pressed information from multiple previous time intervals into a single vector and the usual Generator and Discriminator networks, made up of fully connected layers. A diagram of the model's architecture can be seen in Figure 6.

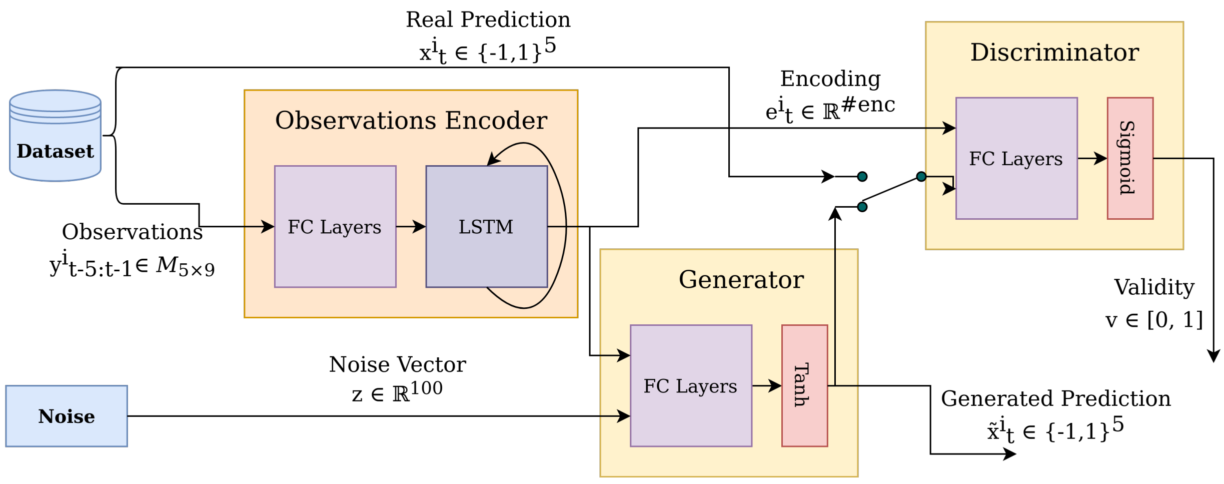


Figure 6: Architecture Diagram of main GAN model

Observations Encoder: the observations, which consist of the 4 state variables and the 5 keys' presses for the 5 preceding time intervals, are encoded by the "Observations Encoder": these 9 variables are first projected onto a higher dimensional space using a fully connected layer with a LeakyReLU activation function, before going into 2 LSTM layers. The final hidden state of the last LSTM layer is the output and this encoded observations state will be the condition of our GAN. The LSTM layer allows for simple generalization - for instance, if instead of 5 observations we wanted 10, the network would not have to be changed.

Generator: it has two inputs, the observations encoding, which is the GAN's condition, and the noise/latent vector, which is a 100 dimensional vector with values sampled from a normal distribution $z \sim \mathcal{N}(0, 1)$. These two inputs are concatenated and then go into a series of fully connected blocks. Each block has a dropout layer, a fully connected/linear layer and a LeakyReLU activation. The last block has a 5 dimensional output followed by a Tanh activation, so that we may obtain 5 values in the $[-1, 1]$ range, one for each key (we consider that a key was pressed if the generated value is positive and not pressed otherwise).

Discriminator: similarly to the generator, it also receives the observations encoding as an input, but the second input is a prediction (a set of 5 values in the $[-1, 1]$ range, one for each key), either the actual prediction from the dataset or one obtained from the generator. Also like the generator, the two inputs are concatenated before going through a series of fully connected blocks, with the difference that the last block has a single number as output, followed by a Sigmoid activation. This network's output is the probability (as the number is in the $[0, 1]$ range) of the prediction given as input being a real prediction for the given encoded observations. It is this value that is then used for adversarial training.

The network's hyperparameters as well as some training parameters were optimized using

Table 5: Model's layers description

Observations Encoder	Generator	Discriminator
Dropout layer Linear layer (9 $\rightarrow$ 256) LeakyReLU LSTM layer (256 $\rightarrow$ 256) LSTM layer (256 $\rightarrow$ 256)	Dropout layer Linear layer (356 $\rightarrow$ 512) LeakyReLU Dropout layer Linear layer (512 $\rightarrow$ 256) LeakyReLU Dropout layer Linear layer (256 $\rightarrow$ 128) LeakyReLU Dropout layer Linear layer (128 $\rightarrow$ 5) Tanh	Dropout layer Linear layer (261 $\rightarrow$ 256) LeakyReLU Dropout layer Linear layer (256 $\rightarrow$ 128) LeakyReLU Dropout layer Linear layer (128 $\rightarrow$ 64) LeakyReLU Dropout layer Linear layer (64 $\rightarrow$ 1) Sigmoid

Optuna[6]. The full layer description is in Table 5. The probability for each Dropout layer was always 0.3, while the LeakyReLU activations had a negative slope of 0.2. Each layer's input and output sizes are inside parenthesis (input $\rightarrow$ output).

5.3 Loss functions and training process

For training, the Adam optimizer [18] was used. The generator learning rate used was 0.0005 while the one for the discriminator was 0.0001. The observations encoder was jointly trained with the discriminator and generator, so that the resulting encoding may simultaneously help the generator generate good quality samples (the generator's training objective is to "fool" the discriminator into thinking its outputs are real, thus this encoding will help generate outputs with higher discriminator validity value) and help the discriminator distinguish real and generated samples.

5.3.1 Adversarial Loss

When it comes to the loss functions used, the usual GAN adversarial loss function (adapted for CGANs) was used (Equation 2). During actual training this loss function is split into two: the generator only trains with the last portion, seeking to improve the validity of its output, while the discriminator uses the full expression. The condition (y), in this case, is our encoding vector (e).

[6]https://optuna.org/

5.3.2 Variety Loss

As discussed above, some works using GANs for time series problems have opted to also use, in addition to the adversarial loss, an L2/Mean Squared Error loss. This loss function results in faster convergence but may easily cause mode collapse: having multiple inputs result in a "collapse" of the possible outputs into a single one, thus harming multimodality and the range the model is able to cover.

SocialGAN [13] has introduced Variety Loss in GANs' training. This is a variation of the L2 loss wherein, for each sample/training batch, k noise inputs are used to get different output values and only the one with the shortest L2 distance to the actual prediction is used in the optimization loss, as demonstrated by Equation 4.

The practical implication is that, unlike the simple L2 loss, this loss function allows "variety" of the model's outputs while still "nudging" the model towards making more realistic predictions.

This loss function was also used in this project's model to train the generator, with a weight of 0.6 (meaning that the variety loss was added to the original adversarial loss of the generator after being multiplied by 0.6).

$$\mathcal{L}_{variety} = \min_{k} \|x_i - \widetilde{x}_i^{(k)}\|_2 \tag{4}$$

6 Model Validation

Validation is one of the main issues when it comes to GANs and, specifically, with GANs applied to non image related tasks. The training losses by themselves are often unreliable as a validation criteria, and the well established usual validation metrics such as the Inception Score [19] or the Frechet Inception Distance (FID score) [20] are specific to image applications.[21] Nevertheless, having a quantitative evaluation metric is useful to compare different models, for early stopping and preventing the model from overfitting to the training set.

6.1 Quantitative Evaluation Metric

Under the problem formulation defined for this project, and with the change to the way the keypresses are processed - for each key only whether the key was pressed or not for a given time interval is considered - we have obtained one binary output per key. As mentioned above, of the 1353150 individual keypresses ground truths/targets, only 7% of them were positive. In other words, in a lot of samples no keys were pressed, and so we have an unbalanced problem, where most outputs are likely to correspond to a key not being pressed. Therefore, a metric such as accuracy (comparing one by one whether for each key the generated output matched the ground truth) is not useful as a model that, for instance, would only output 0s (no key presses) would have a 90%+ accuracy. Therefore, considering a keypress as the positive class and not pressing a key as the negative class, precision and recall metrics were defined:

$$\text{Precision} = \frac{\text{correct generated keypresses}}{\text{total generated keypresses}} \tag{5}$$

$$\text{Recall} = \frac{\text{correct generated keypresses}}{\text{total dataset keypresses}} \tag{6}$$

The quantity “correct generated keypresses” is the number of keys that the model generated that were “correct”, that is, that matched the original sample’s future keys pressed; “total generated keypresses” is the total number of keypresses that the model generated; and “total dataset keypresses” refers to the number of keys pressed in the original sample.

Precision is, therefore, a measure of the percentage of the keypresses that we generated that matched the original dataset. On the other hand, recall is a measure of how many of the keypresses in the original dataset we were able to find. There is a trade-off between these two metrics: by making fewer positive predictions precision can be increased while recall decreases and by making a lot of positive predictions precision will likely be low while recall will be high (many of the positives will be false positives but we will likely find many of the original dataset’s positives). The F1-score combines these two metrics into a single number and is defined as the harmonic mean of these two metrics (Equation 7). This is the main metric used in this project to quantify a model's performance.

$$\text{F1-score} = \frac{2 \times \text{precision} \times \text{recall}}{\text{precision} + \text{recall}} \tag{7}$$

As the model's generator ends with a Tanh activation layer, which yields numbers in the $[-1, 1]$ range, a generated output number > 0 is considered as a positive (the key was pressed) whereas a number ≤ 0 is considered as a negative (the key was not pressed).

6.2 Evaluation Methodoloy

As a model should never be tested on the same data that was used for its training, and to ensure that when the dataset is split into validation and training sets, the performance evaluation is not heavily dependent on the chosen division, cross validation [22] was used. Using Scikit-learn's [23] , the dataset was split into 4 disjoint folds. Then, 3 of the folds were used as a training set while the remaining fold was used for validation (precision, recall and F1-score metrics were computed on this fold for a model trained on data from the other 3 folds). By changing in turn which fold is used as a validation set, we obtain 4 F1-scores, one for each validation set, which can then be averaged to yield the model's final evaluation score.

Another issue to consider is that since stochastic models (such as GANs) are able to generate a set of different predictions for each sample, computing metrics on a single prediction does not properly represent the model's performance. Following the methodoloy presented in [13], out of k validation was performed: for each sample, k predictions are generated by sampling different random noise vectors; then, the prediction which is closest to the ground truth is used for validation. Using this method we can preserve the GAN's multimodality while properly quantifying the model's outputs correctness.

6.3 Baselines

The model detailed in Section 5.2 was compared against the following baselines:

- *GAN-GP* A Wasserstein GAN [8] trained with gradient penalty [9] without any conditions. This model generates outputs without considering the previous game state at all. The generator was trained once for every 5 training iterations of the critic (the equivalent of the discriminator in wgans).
- *CGAN-GP* Same as above, but the game state (robot_mode, coordinates and keys pressed) of the time interval immediately preceding the current one are used as a condition to generate the next time interval's keypresses.

Table 6: Quantitative evaluation results for all models. F1-score, precision and recall values were averaged over batches and folds; then, the epoch with the highest average F1-score was chosen (Best F1-score for a model). The precision and recall values shown are the ones associated with that epoch. The best values are displayed in **bold**.

Model	**GAN-GP**	**CGAN-GP**	**RCGAN-V1**	**RCGAN-V5**	**RCGAN-V20**
Best F1-score	19.95%	48.82%	61.56%	62.50%	**62.55%**
Precision/Recall	14.07%/34.27%	68.48%/37.93%	**72.92%**/53.27%	70.78%/**55.95%**	72.89%/54.77%

A choice was made to not use the Wasserstein GAN's loss on the RCGAN model, as it didn't seem to lead to a significant performance improvement.

Each of these models, as well as the RCGAN detailed in Section 5.2, were out of k validated, with $k = 5$ samples, and were trained for 100 epochs. The performance metrics values were averaged over testing batches and folds; then, the epoch with the highest average F1-score was selected (and its associated precision and recall). For the RCGAN-V**V**, **V** is the number of samples used in the variety loss. The results obtained for the different models can be seen in Table 6. Three different **V** values were considered: 1, 5, 20. From the results, it seems that this parameter did not have a great influence, which is likely due to the nature of the problem, as while we are generating binary labels, works in trajectory prediction such as [13] which have had success with the variety loss were generating coordinate positions (real numbers) where the L2 loss may have a stronger impact. Looking at GANGP, it seems clear that not having access to any observational/previous game state data has a great impact in model performance, as this model has an extremely low F1-score. On the other hand, CGANGP has a significantly higher score, not too far from that of the RCGAN models. The difference observed may be due to the fact that RCGAN has access to more observational data, as it takes into account the full 5 preceding seconds and not only the last one.

7 Experiments

Additional experiments and data analysis were performed. Taking the model with the highest performance, "RCGAN-V20", generating 100 predictions for each sample in the training set (made up of the fold in which it wasn't trained on), and associating the last position of the robot on each sample with the keys the model generated for it, it can be seen that, unfortunately, the model does not seem to ever generate keypresses for the "back" and "space" keys. This is likely due to the fact that these keys' prevalence in the dataset are extremely low, and so the model focused on optimizing more common keys, such as the "front" and the side keys ("left" and "right").

In Figures 7, 8 and 9, we can see the results obtained, where the game plane has been discretized into 10 by 10 regions, and where the value for each region is the proportion of keypresses generated while in that region (example, 0.5 for the front key would mean that in half the times there was a sample where the robot was in that position, the model generated a front keypress).

For the front key, in Figure 7, we can see that it has an overall high prevalence and is well spread over most regions. On the other hand, the right and left keys are not as evenly distributed. As can be seen in Figure 8, there seems to be a higher proportion of right keypresses on the left side of the board, while the opposite happens for the left key (Figure 9). This seems intuitive: when a player is on the border of the map he will likely try to move towards the center.

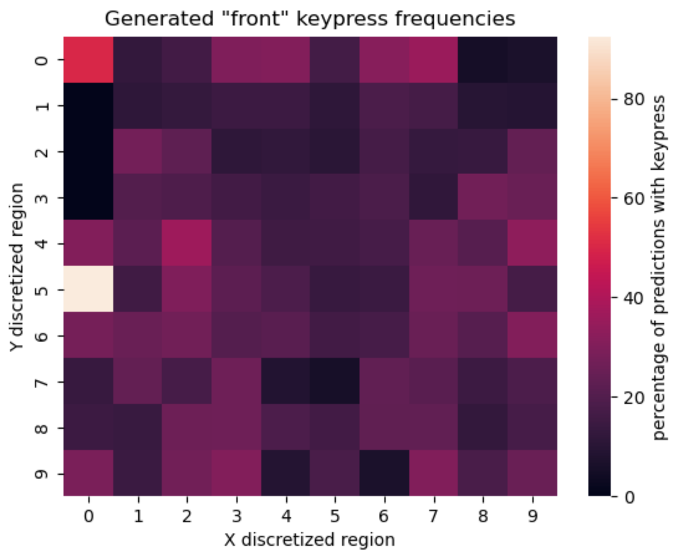


Figure 7: Geographical frequencies for front key

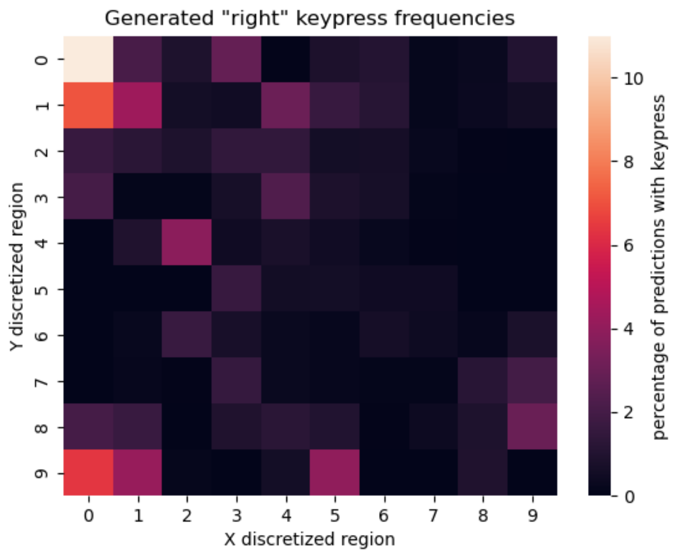


Figure 8: Geographical frequencies for right key

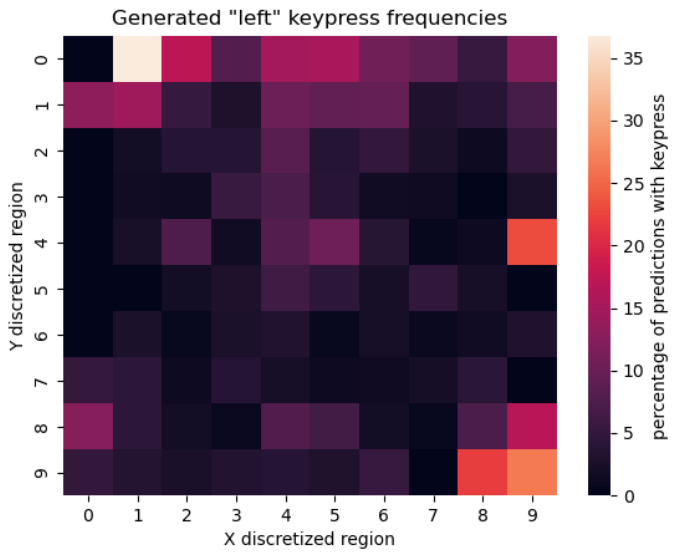
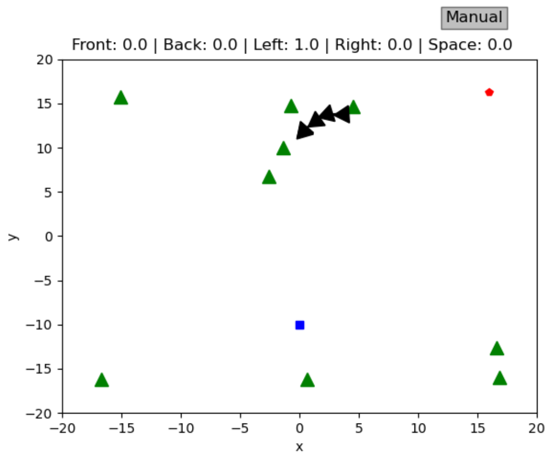
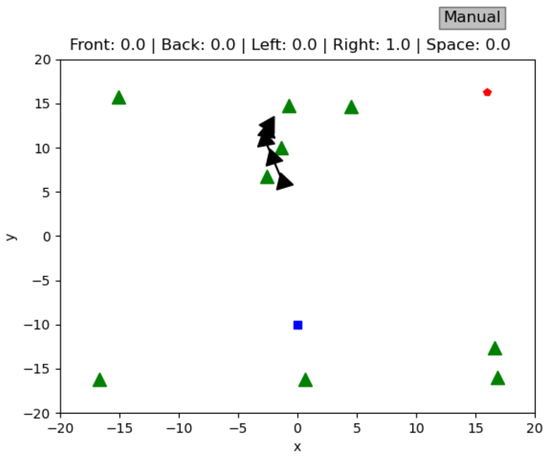


Figure 9: Geographical frequencies for left key

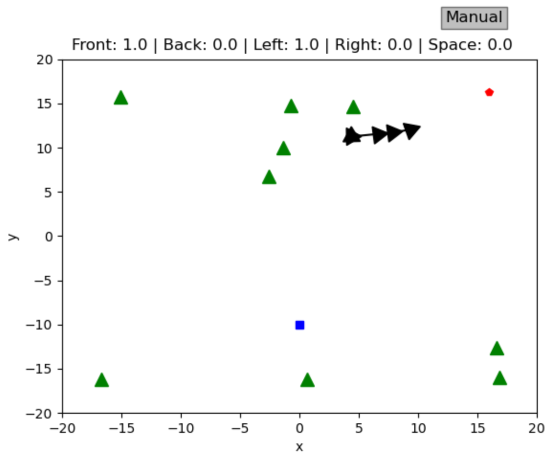
In order to visualize some samples generated by the model, the robot's 5 consecutive positions were plotted on the game map as black triangles facing in the same direction the robot was facing. We can see 4 different samples in Figure 10. The green triangles represent the trees, the blue square the water refill tank and the red one the battery recharge location. All images generated correspond to the manual control mode, and the model's outputs (the generated keys) are above the map, where a value of “1.0” means the key was pressed and a value of “0.0” that it was not.



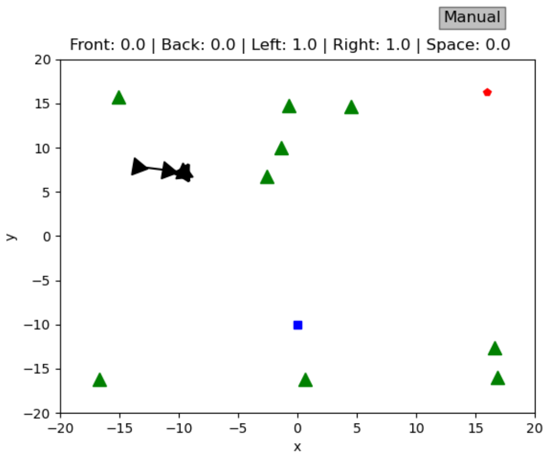
(a) Sample with left keypress



(b) Sample with right keypress



(c) Sample with front and left keypress



(d) Sample with right and left keypress

Figure 10: Samples generated by the model

We can see, for example, on Figure 10a, that the model seems to have taken into account the leftwards curve the robot made, and has predicted the left key as being pressed. Similarly, we see the same happen to the sample in Figure 10b with a right keypress continuing the rightwards turn. On the sample in Figure 10c, the front and left keys are pressed in conjunction. Finally, on sample 10d, as can be seen from the black triangles, there seems to have been a sudden change in direction and it is not clear in which direction the player was intending to go, and the model has generated both the left and right keys being pressed. This combination of left and right keys seems to be a rare occurrance in the generated data.

8 Conclusion

Multiple models were introduced over the course of this project in an attempt to properly create a generator of human actions. The CGAN and RCGAN models, with its conditional features, are able to take into account the previous game history, incorporating information such as the robot's position, to more accurately predict reasonable future actions. Additionally, by using Recurrent Neural Networks (RNNs), the RCGAN model may exploit multiple game instants to further improve the predictions.

As it stands, the RCGAN model may be used iteratively to simulate an entire game: by starting with 5 predictions, provided that a causal relationship may be formalized between the binary keypress actions and actual in game changes caused by them, the new predictions generated and their consequences may be used as observations for the next predictions. Unfortunately, this causal link is not easy to establish as there is no clear way to extrapolate an unordered set of keys pressed in a 1 second window to the normal gameplay where a user may press many keys in any order in less than one second.

Another alternative, which can easily be implemented, is to extend the current RCGAN network by adding an extra LSTM layer on the discriminator and by iteratively using an LSTM cell on the generator to immediately generate predictions not for one but for multiple consecutive seconds. Provided that the time window is of moderate size, the generated sequences may be useful even without having to compute changes to the game state.

In terms of the quantitative validation performed, it should be noted that even with the approach that was followed, given the nature of the precision and recall metrics, their trade-off relation and, that effect on the F1-score, there is likely a theoretical limit to the maximum F1-score that may be obtained for this problem by a model. While being a good indicator of overall performance and extremely useful in, for example, hyperparameter optimization, the F1-score's simple application does not necessarily guarantee the quality of a model. Finding good quantitative metrics for this problem is a difficult task, and it may be that other, more appropriate metrics may be defined in the future.

Further work may be applied to the study and experimentation of different GAN architectures and loss functions, such as, for example, the Info-GAN architecture used in [14], as well as to additional validation metrics and strategies. Given the unbalanced nature of the dataset which, as discussed before, has around 7% positives (keypresses) and an even lower percentage when we consider uniquely the "space" or "back" keys, another avenue for improvement would be to collect additional high quality data, which may be defined as complete games from players who have understood the game and how it works, and that actively interact with the controls.

References

[1] Caroline Ponzoni Carvalho Chanel, Raphaëlle N. Roy, Frédéric Dehais, and Nicolas Drougard. "Towards Mixed-Initiative Human-Robot Interaction: Assessment of Discriminative Physiological and Behavioral Features for Performance Prediction". In: *Sensors, Special issue Human-Machine Interaction and Sensors* 20.1 (Jan. 2020), pp. 1–20. DOI: `10.3390/s20010296`. URL: `https://oatao.univ-toulouse.fr/25267/`.

[2] Jack-Antoine Charles, Caroline P. C. Chanel, Corentin Chauffaut, Pascal Chauvin, and Nicolas Drougard. "Human-Agent Interaction Model Learning Based on Crowdsourcing". In: *Proceedings of the 6th International Conference on Human-Agent Interaction*. HAI '18. Southampton, United Kingdom: Association for Computing Machinery, 2018, pp. 20–28. ISBN: 9781450359535. DOI: `10.1145/3284432.3284471`. URL: `https://doi.org/10.1145/3284432.3284471`.

[3] Mausam and Andrey Kolobov. "Planning with Markov Decision Processes: An AI Perspective". In: *Synthesis Lectures on Artificial Intelligence and Machine Learning* 6.1 (2012), pp. 1–210. DOI: `10.2200/S00426ED1V01Y201206AIM017`. eprint: `https://doi.org/10.2200/S00426ED1V01Y201206AIM017`. URL: `https://doi.org/10.2200/S00426ED1V01Y201206AIM017`.

[4] Ian J. Goodfellow et al. *Generative Adversarial Networks*. 2014. arXiv: `1406.2661 [stat.ML]`.

[5] Mehdi Mirza and Simon Osindero. *Conditional Generative Adversarial Nets*. 2014. arXiv: `1411.1784 [cs.LG]`.

[6] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[7] Antonia Creswell, Tom White, Vincent Dumoulin, Kai Arulkumaran, Biswa Sengupta, and Anil A Bharath. "Generative adversarial networks: An overview". In: *IEEE Signal Processing Magazine* 35.1 (2018), pp. 53–65.

[8] Martin Arjovsky, Soumith Chintala, and Léon Bottou. *Wasserstein GAN*. 2017. arXiv: `1701.07875 [stat.ML]`.

[9] Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron Courville. *Improved Training of Wasserstein GANs*. 2017. arXiv: `1704.00028 [cs.LG]`.

[10] Saloni Dash, Andrew Yale, Isabelle Guyon, and Kristin P Bennett. "Medical time-series data generation using generative adversarial networks". In: *International Conference on Artificial Intelligence in Medicine*. Springer. 2020, pp. 382–391.

[11] Moustafa Alzantot, Supriyo Chakraborty, and Mani Srivastava. "Sensegen: A deep learning architecture for synthetic sensor data generation". In: *2017 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. IEEE. 2017, pp. 188–193.

[12] Cristóbal Esteban, Stephanie L Hyland, and Gunnar Rätsch. “Real-valued (medical) time series generation with recurrent conditional gans”. In: *arXiv preprint arXiv:1706.02633* (2017).

[13] Agrim Gupta, Justin Johnson, Li Fei-Fei, Silvio Savarese, and Alexandre Alahi. “Social gan: Socially acceptable trajectories with generative adversarial networks”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 2255–2264.

[14] Javad Amirian, Jean-Bernard Hayet, and Julien Pettré. “Social ways: Learning multi-modal distributions of pedestrian trajectories with gans”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*. 2019, pp. 0–0.

[15] Amir Sadeghian, Vineet Kosaraju, Ali Sadeghian, Noriaki Hirose, Hamid Rezatofighi, and Silvio Savarese. “Sophie: An attentive gan for predicting paths compliant to social and physical constraints”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2019, pp. 1349–1358.

[16] Ali el Hassouni, Mark Hoogendoorn, and Vesa Muhonen. “Using generative adversarial networks to develop a realistic human behavior simulator”. In: *International Conference on Principles and Practice of Multi-Agent Systems*. Springer. 2018, pp. 476–483.

[17] Xi Chen, Yan Duan, Rein Houthooft, John Schulman, Ilya Sutskever, and Pieter Abbeel. “Infogan: Interpretable representation learning by information maximizing generative adversarial nets”. In: *Advances in neural information processing systems* 29 (2016).

[18] Diederik P Kingma and Jimmy Ba. “Adam: A method for stochastic optimization”. In: *arXiv preprint arXiv:1412.6980* (2014).

[19] Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. “Improved techniques for training gans”. In: *Advances in neural information processing systems* 29 (2016).

[20] Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, and Sepp Hochreiter. “Gans trained by a two time-scale update rule converge to a local nash equilibrium”. In: *Advances in neural information processing systems* 30 (2017).

[21] Ali Borji. “Pros and cons of gan evaluation measures”. In: *Computer Vision and Image Understanding* 179 (2019), pp. 41–65.

[22] *Scikit-learn: Cross-validation.* URL: `http://web.archive.org/web/20080207010024/http://www.808multimedia.com/winnt/kernel.htm` (visited on 03/23/2022).

[23] F. Pedregosa et al. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.