PEDESTRIAN TRAJECTORY FORECASTING

ASI-CAIF 2019 SUMMER STUDENT TRAINING PROGRAM WORK REPORT

Giulio Autelitano
Stanford University
Stanford Vision and Learning Lab
giulio.autelitano@stanford.edu

January 8, 2020

ABSTRACT

This final report outlines the procedures and results of my work at the Stanford Vision and Learning Laboratory. The research focuses on pedestrian trajectory forecasting and addresses the problem of path prediction for multiple interacting agents in a scene, which is a crucial step for many autonomous platforms such as self-driving cars and social robots in space applications. To predict a future path for an agent, both physical and social information must be leveraged. The current best approach to such problem blends a social attention mechanism with a physical attention that helps the model to learn where to look in a large scene and extract the most salient parts of the image relevant to the path. This paper also covers the development of a framework to infer a the scene birds-eye view utilizing the robot's on board sensors. The problem is solved by a generative network trained on a synthetic dataset exclusively built for this porpoise.

***K*eywords** Deep Learning · Artificial Intelligence · Autonomous Vehicles · Social Navigation

1 Introduction

The following work has been developed at Stanford University between November and December 2019. Under the supervision of Professor Silvio Savarese, I was part, as a Visiting Student Researcher, of the Stanford Vision and Learning Laboratory team. The funding for the present research are provided by a Scholarship of Excellence awarded by the Cultural Association of Italians at Fermilab (CAIF) in collaboration with the Italian Space Agency (ASI).

The research topics must be aligned with ASI's mission and research interest which are mainly focused on space applications. The core of the present work is the development and implementation of machine learning algorithms to solve complex problems otherwise impossible to tackle with traditional programming methods. During the two months time frame, the objectives have been changing mostly due to a fast-pace environment of the Stanford Vision and Learning Laboratory forcing to re-plan the work several times. The overall experience allowed me to learn some state-of-the-art techniques in machine learning as well as improving my coding skills. The contributions to the project have been the creation of a package of data cleaning for the JackRabbot dataset, the completion of an algorithm to infer the top view scene from LiDAR point-cloud exclusively utilizing robot's on-board sensors and the set up of a network's structure for the inference of a semantically segmented top view scene.

Predicting the future trajectories of multiple interacting agents in a scene has become an increasingly important problem for many different applications ranging from control of autonomous vehicles and social robots to security and surveillance. This problem is compounded by the presence of social interactions between humans and their physical interactions with the scene. Social robots have also big potential in space programs such as the NASA's Astrobee. The goal of my work at Stanford is to acquire the necessary knowledge in machine learning to enrich my aerospace background with state-of-the-art techniques that will be adopted extensively in space applications.

2 Introduction to Artificial Intelligence

Artificial intelligence (AI) is a thriving field with many practical applications and active research topics. The goal of the discipline is to make intelligent software be able to automate routine tasks, understand images, allow robots to be fully autonomous and solve complex problems. There are many problems which are incredibly complicated for humans to perform, such as processing high volumes of data, computing operations with high value numbers and so on. These problems are intellectually difficult for human beings but relatively straight-forward computers problems that can be described by a list of formal, mathematical rules once discovered. On the other hand, humans are extremely fast at understating the context they are immersed in, avoid obstacles with elegance and understand emotions. The true challenge to artificial intelligence is solving the tasks that are easy for people to perform but hard for people to describe formally like problems that we solve intuitively like recognizing spoken words or faces in images. The solution to the problems allow computers to learn from experience and understand the world in terms of a hierarchy of concepts, with each concept defined through its relation to simpler concepts. Like all form of natural intelligence, the experience comes after having observed a similar situation and being capable of generalizing the abstract meaning behind it and apply the knowledge in unseen circumstances. By gathering knowledge from experience, this approach avoids the need for programmers to formally specify all the knowledge that the computer needs. The hierarchy of concepts enables the computer to learn complicated concepts by building them out of simpler ones. Figure 1 outlines these concepts.

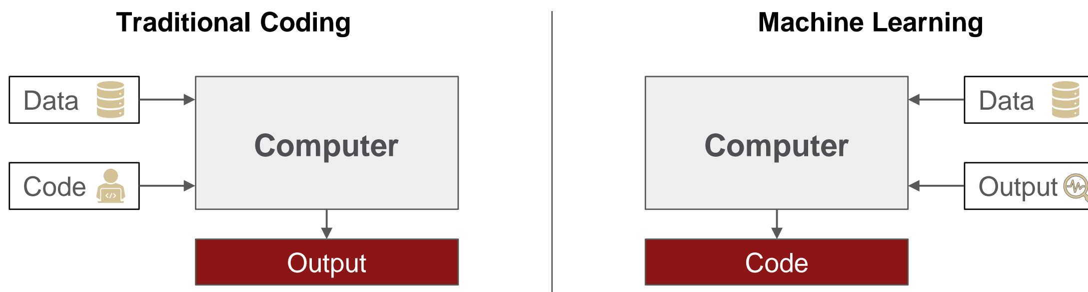

Figure 1: Traditional Coding and Machine Learning

Within the traditional coding approach, the programmer needs to manually assemble the code being exactly aware of the underlying mathematical models to solve the problem. Every policy is explicitly written and the software engineer has full authority over the code. The code is then run giving it an input and the computer outputs a result. If the output is not compliant with the expected one, the code is inspected and debugged until everything works as planned. Once a satisfactory result has been achieved, the code can be implemented and always described in each part. On the other hand, machine learning works bypassing the need of explicitly define the mathematical models of the problem. The software engineer takes care of building a proper network structure and clean the input data. The code, in this case, can be thought as an output product of the process. The data is fed trough the network and the output results compared to a ground truth. The learning process tries to tune the network's parameters to match the desired results. In this case, it is not possible to fully understand what the complicated network is doing. The underlying reasoning remains obscure and almost no control can be exerted over the process. It is thus clear how the input data is crucial for the network to be able to generate a meaningful output.

To sum up, it is possible to solve a problem in two ways: the first, by manually assigning the mathematical model which the problem rests on, or exploiting some machine learning technique to take advantage of a huge set of data. Both these approaches present pros and cons. Machine learning relief the software engineer from the need of coding some policies that may be overwhelming in situations like image recognition. It would be cumbersome to manually write an algorithm to identify a human face in a picture. Due to the high complexity of the problems that this papers aims at solving, namely the pedestrian trajectory forecasting, relying on machine learning seems the obvious, if not the only, path to follow.

3 Autonomous Vehicles Forecast

Being my work focused on human trajectory forecasting, a brief description of the market that will mainly take advantage of such break-trough is fundamental.

As reported by McKinsey & Company, autonomous vehicles (AVs) represent a major innovation for the automotive industry. While high levels of uncertainty currently surround the issue, the ultimate role that AVs could play regarding

the economy, mobility, and society as a whole could be profound. The widespread use of AVs could profoundly affect a variety of industry sectors. Upon mass adoption of AVs, drivers will have more time for everything. AVs could free as much as 50 minutes a day for users, who will be able to spend traveling time working, relaxing, or accessing entertainment. The time saved by commuters every day might add up globally to one billion hours. It could also create a large pool of value, potentially generating global digital-media revenues of billions per year for every additional minute people spend on the mobile Internet while in a car. Another great advantage of AVs regards the parking problem. AVs could change the mobility behavior of consumers reducing the need for parking space. Multiple factors would contribute to the reduction in parking infrastructure. For example, self-parking AVs do not require open-door space for dropping off passengers when parked, allowing them to occupy parking spaces that are 15 percent tighter. By mid century, the penetration of AVs could ultimately cause vehicle crashes in the United States to fall from second to ninth place in terms of their lethality ranking among accident types. Today, car crashes have an enormous impact on the US economy. For every person killed in a motor-vehicle accident, 8 are hospitalized, and 100 are treated and released from emergency rooms. The overall annual cost of roadway crashes to the US economy was $212 billion in 2012. Taking that year as an example, advanced ADAS and AVs reducing accidents by up to 90 percent would have potentially saved about $190 billion.

One of the challenges of the widespread diffusion of AVs is the necessity to make such vehicles reliable and safe. Pedestrian trajectory forecasting is crucial for road safety and this paper will focus on the techniques to tackle this challenge.

4 Machine Learning in Space Application

Artificial intelligence today plays a fundamental role in the space sector. There are many current and future applications, designed to overcome limits otherwise considered insurmountable, such as perfect orientation in space or on a planet without GPS or immediate analysis of huge amounts of data. A striking example is Phi-Sat, a cube-sat that will take pictures of our planet from a few hundred kilometers of altitude. Before sending them to Earth, however, it will be the same AI that will look at the acquired photos and decide independently which ones to discard because they are not useful for scientific purposes. Thus sending images (terabytes of data every day) will be lighter and faster. Further post processing done by artificial intelligence algorithms continues on the ground, analyzing the images coming from the satellites, from which it is possible to extrapolate every single change, even millimetric, that takes place on the Earth's surface and potentially allow for disasters to be avoided and significantly increase the safety of the population.

Furthermore, social robots could also play a crucial role for space missions. Forecasting human behavior is a necessity when dealing with human-robot interactions. AI is thus the only achievable solution to solve the problem and my work aims at developing the necessary algorithms to tackle this challenging problem.

4.1 NASA's AstroBee

A very interesting research project of social robots in space application comes from NASA's Astrobee. The robot should help astronauts in focusing on high-priority tasks while the Astrobee takes care of low-priority duties. It will allow astronauts to reduce the time spent on routine duties, leaving them to focus more on the things that only humans can do. Working autonomously, or via remote control, the robots are designed to complete tasks such as taking inventory, documenting experiments conducted by astronauts with their built-in cameras or working together to move cargo throughout the station. In addition, the system serves as a research platform that can be outfitted and programmed to carry out experiments in micro-gravity helping to learn more about how robotics can benefit astronauts in space.

The Astrobee system, shown in Figure 2, consists of three cubed-shaped robots, a control software and a docking station used for recharging. The robots use electric fans as a propulsion system that allows them to fly freely through the micro-gravity environment of the station. Cameras and sensors ensures that a perceptions algorithms can be rub and enables the autonomous navigation of the surrounding. The robots also carry a perching arm that allows them to grasp station handrails in order to conserve energy or to grab and hold items. Researchers will be able to use Astrobee to carry out investigations that will help to develop technology for future missions. Since the robots are modular and can be upgraded, the system gives researchers and scientists diverse capabilities for performing a wide range of experiments inside the station.

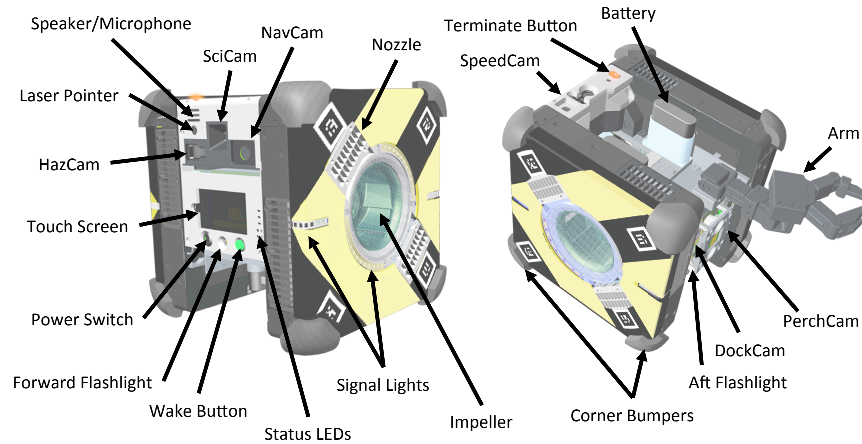

Figure 2: NASA's Astrobee

Robots will play a significant part in mission to return to the Moon as well as other deep space missions. Social robots such as Astrobee, have the capacity to become caretakers for future spacecraft, working to monitor and keep systems operating smoothly while crew are away. The space social robots must cooperate with astronauts and be able to anticipate their intentions not to collide or interrupt their work. It is thus fundamental to forecast human behaviors in a similar approach as ground autonomous vehicles adopt. The techniques adopted to forecast pedestrian trajectories are usually restricted to a flat plane. The extension to the third dimension is trivial, given that a comprehensive dataset is available to train the networks on. Astrobbe could thus be a precious source of information recording astronauts movements in micro-gravity environments. Sharing the same sensors of ground robots, it will be possible to utilize the same algorithms trained on the dataset recorded by Astrobee.

5 Machine Learning Basics

In this section I will focus on the basic concepts that constitute the fundamental principles of neural networks [1]. The difficulties faced by systems relying on hard-coded structures suggest that AI systems require the ability to acquire their own knowledge, by extracting patterns from raw data. This capability is known as machine learning. The introduction of machine learning enables computers to tackle problems involving knowledge of the real world and make decisions that appear subjective. This section will cover the elementary AI computation unit, namely the artificial neuron, how the latter is used to build a neural network and the description of the underlying algorithms that make the network be able to learn patterns and decision policies.

5.1 Artificial Neuron

The basic element of a neural network is called artificial neuron named after it's natural counterpart, the biological neuron. An artificial neuron aims at simulating a biological neuron outputting a value (usually bounded) called activation. Figure 3 shows a general artificial neuron which is a computation cell that takes multiple inputs and performs a two-step calculation to obtain the final output. The first operation is computed by summing all the inputs x_j, each multiplied by a weight w_j, and subsequently adding a bias b to the final result. This step is known as weighted sum and its result is indicated with the quantity z. At this stage in the process, the partial output can, in theory, store any real value. At first, all the weights w_j and biases b are randomly initialized from the normal distribution. It must be noted that, while only a single bias belongs to each neuron, the weights are in the same number of the neuron inputs. The learning process will aim at assigning the best weights and biases for the network to perform as desired. By varying the weights and the biases, it is possible to get different models of decision-making and thus a different output.

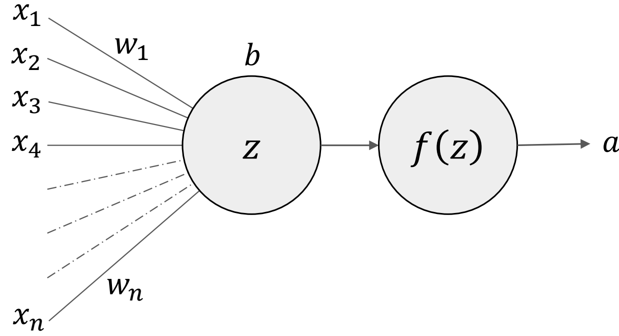

Figure 3: Machine learning unit element: Artificial Neuron

To bound the partial output z, and more in general to obtain control over it, the weighted sum is fed trough an activation function $f(z)$. The activation function output is usually called activation and the relative neuron is said to fire. There is a large variety of activation functions found in the literature, but the most common are the sigmoid, the $tanh$ and the $ReLU$ functions. The activation function mathematical expressions are expressed below:

$$\sigma(z) \equiv \frac{1}{1+e^{-z}} \qquad tanh(z) \equiv \frac{e^z - e^{-z}}{e^z + e^{-z}} \qquad ReLU(z) \equiv max(0, z)$$

The sigmoid function is continuous and derivable. It bounds the neuron activation between 0 and 1 with continue values in between. To bound the output between -1 and 1 the $tanh$ function is used. The $tanh$ possess an anti-symmetric propriety and again, is both continuous and derivable. Finally, the $ReLU$ is the most aggressive function in the positive domain outputting the untouched value of z, or zero in case z is negative. As it can be seen from the above equation, the $ReLU$ activation function presents a discontinuity in its derivative at the origin.

The activation functions have thus all different behaviors with respect to their definition, but even more important for the backpropagation algorithm is the shape of their derivatives (cfr. Eq.12 and Eq.13). Both the sigmoid and the $tanh$ functions have a zero value derivatives at the extremes of their domains which, as will be seen in the next paragraph, could initiate the vanishing gradient problem posing serious threat to learning speeds.

To recap, an artificial neuron is a mathematical function and works by accepting multiple numerical values as inputs and outputting a unique numerical value, namely the neuron activation value. The artificial neuron usually takes its name after the activation function that utilized, e.g. if a sigmoid function is adopted, the neuron is said to be a sigmoid neuron. Artificial neurons are then connected to build an artificial neural network with multiple layers containing each multiple neurons. The next paragraph will focus on how the network is assembled and the underlying mathematical principles restricted to multilayer perceptrons.

5.2 Neural Networks

In this paragraph I will present the basic network architecture of a multilayer perceptron (MLP). As suggested by the name, the elementary units of the network are artificial neurons that are arranged in a series of layers starting from the inputs to the final outputs as shown in Figure 4. When building such a network the first step is assigning the input layer which must share the same dimension of the input data. As an example, if we want to feed a picture, within an image recognition problem, the input layer dimensions will match the image pixels number. In the case of pedestrian trajectory forecasting, the input will be the coordinates of each pedestrian so that the number of input neurons will be $2N$, where N is the maximum number of agents in the scene. On the other hand, the final output must match the size of the expected result. With respect to the image recognition problem, the number of output neurons matches the number of class labels we are interested in. For the trajectory forecasting problem the output size will match the input one.

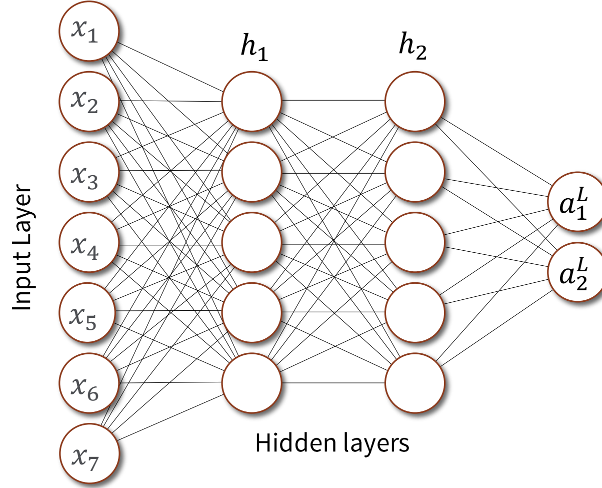

Figure 4: Multi-layer perceptron

To complete the network's architecture, the programmer is faced with the decision of selecting the number of hidden layers, and for each, the relative number of neurons. A hidden layer is any layer which lays between the input and output layers. Figure 4 shows a network with two hidden layers. Each circle represents a single neuron and is connected to each and every neuron in the preceding layers as well as the following. This architecture is called fully connected for obvious reasons.

Each neuron in layer l outputs a value a_j^l which depends on the activations of the previous layer, in case of a sigmoid activation function follows:

$$a_j^l = \sigma\left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l\right) \tag{1}$$

After the network's architecture has been chosen, the first step is to initialize the network assigning every weight and biases to each neuron. This process is often done randomly, choosing the values from a normal distribution. The input is then fed trough the leftmost layer (input layer) and forwarded trough the whole network to obtain the final output in the rightmost layer. The output of the network is the product of the so called feed-forward process. Obviously, the result of such a network will be completely random due to the initialization of the weights and no predictions can be done. The learning process will take care of tuning all the weights and biases to obtain consistent outputs $a^L(x)$ as close as possible to the desired target $y(x)$.

5.3 Loss Function

The goal is to find an algorithm to select all the weights and biases so that the output from the network approximates $y(x)$ for all training inputs x. To quantify how well the network is achieving this target it is possible to define a cost function (e.g. mean square error):

$$C(w, b) = \frac{1}{2n} \sum_x \left|\left|y(x) - a^L(x)\right|\right|^2 \tag{2}$$

In the above expression w denotes the collection of all weights in the network, b all the biases, n is the total number of training inputs, $a^L(x)$ is the vector of outputs from the network when x is input, and the sum is over all training inputs, x. In Eq.2, C is the quadratic cost function also indicated as $L2$. The cost $C(w, b)$ becomes small and approaches zero precisely when the output $a^L(x)$ is approximately equal to the ground truth $y(x)$, for all training inputs, x. The training algorithm performs well if it can find weights and biases so that $C(w, b) \approx 0$. There are many loss functions available in literature and the choice of the most appropriate one depends entirely on the problem to be tackled. The scheme of the loss function evaluation is shown in Figure 5.

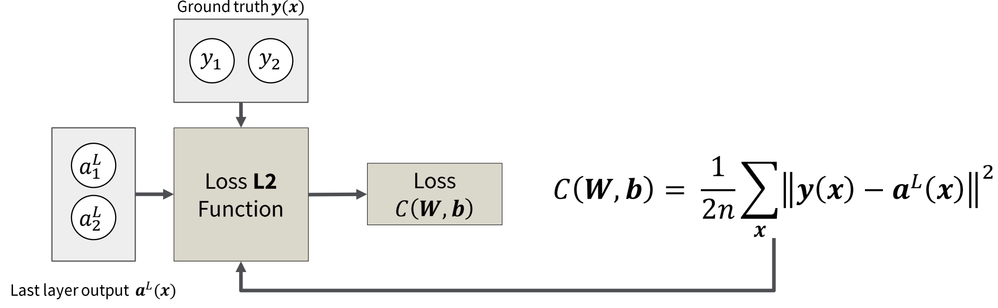

Figure 5: L2 Loss function scheme

The final aim of the training algorithm is thus to minimize $C(w, b)$ as a function of the weights and biases. In other words, to find a set of weights and biases which make the cost as small as possible. To solve this problem the stochastic gradient descent method will be exploited along with the backpropagation algorithm.

5.4 Stochastic Gradient Descent and Back Propagation

Once the cost function has been defined, it is necessary to outline a procedure to minimize it with respect to all weights and biases (parameters) that are present in the network. The problem is thus a minimization problem where the goal is to reach the global minimum of the function. In general, C, has a great number of variables which depends on the network structure and complexity. To ease the problem definition, it is helpful to imagine C as a function of only two variables, namely h_1 and h_2. It is possible to chose a random starting point (thus a random set of h_1 and h_2) and compute locally the value of the cost function. Next, it is possible to change, by a little arbitrary amount, both the variables to lower the value of C and keep iterating this process until a global minimum has been reached. Recall that where the cost function is at a minimum, and thus approaching zero, the network is performing such that its output is the closest to the ground truth as possible. The process described above is called gradient descent and has been proven to be the most feasible way to compute the operation when the number of variables is to high to allow for an analytical solution of the problem. What described above is just a method of reasoning without a proper mathematically rigorous description. To fill the gap it is possible to approximate C with a first order expansion as expressed below:

$$dC(h_1, h_2) \approx \frac{\partial C}{\partial h_1} dh_1 + \frac{\partial C}{\partial h_2} dh_2 \tag{3}$$

The goal is to choose the correct dh_1 and dh_2 to lower the cost function and reach its minimum. To make such a choice, it helps definign dh to be the vector of changes in h, $dh = (dh_1, dh_2)^T$, where T is the transpose operation, turning row vectors into column vectors. Another useful quantity to be defined is the gradient of C being the vector of partial derivatives with respect to each variable:

$$\nabla C = \left(\frac{\partial C}{\partial h_1}, \frac{\partial C}{\partial h_2} \right)^T$$

Given the above definitions it is possible to express the change of the cost function as the inner product of the cost function gradient, ∇C and the the vector of changes, dh, as follows:

$$dC \approx \nabla C \cdot dh \tag{4}$$

In the above equation ∇C relates changes in h to changes in C. It is now necessary to choose dh such that $dC < 0$ to descent towards the global minimum. A very elegant solution, proven also to be the most effective, is to introduce a learning rate parameter, η, and multiply the gradient of C by the latter:

$$dh = -\eta \nabla C, \tag{5}$$

Combining Eq.4 with Eq.5 it is possible to express the change of the cost function as $dC \approx -\eta \nabla C \nabla C = -\eta ||\nabla C||^2$ which is finally negative and compliant with the requirement on dC. Next, it is necessary to update the values of both h_1 and h_2 assigning the new value of vector h' as

$$h' = h + dh = h - \eta \nabla C \tag{6}$$

Then, keeping to rely upon this update rule, another step is carried out. Following this procedure for a relative high number of steps it is possible to reach the global minimum of the function C. To make gradient descent work correctly, it is mandatory to choose the learning rate η to be small enough so that the first order approximation holds. On the other hand, it is not practical to assign η an excessively small value, since that will make the changes dh tiny, and thus the gradient descent algorithm will work slowly.

The above example has dealt with the particular case of a two variables cost function. Real networks have thousands, even millions, of parameters. The natural extension of the analytical description is to generalize the equations to account for an arbitrary number of weights and biases. It is straight forward to introduce the same quantities for a cost function $C = C(h_1, ..., h_k)$ of k variables. In this case, it is possible to express the gradient as $\nabla C = \left(\frac{\partial C}{\partial h_1}, ..., \frac{\partial C}{\partial h_k}\right)^T$ living in the $\mathbb{R}^k$ space. Once the gradient of C has been expressed, Eq.4 trough Eq.6 still hold and are used to update the parameters at each iteration of the gradient descent process.

The presented gradient descent approach hides a number of challenges. One of those is the problem of scalability of the method when the training set of inputs is very large. The computation of the gradients for each input could take a long time making the learning occurs slowly. By recalling the quadratic cost in Eq.2 it is obvious that it's an average over costs for individual training examples. Thus, to compute the gradient ∇C it is necessary to compute the gradients for each training input, x, and then average them. An idea called stochastic gradient descent can be used to speed up learning. The idea is to estimate the gradient ∇C by computing ∇C_x for a small sample of randomly chosen training inputs. By averaging over this small sample it turns out that we can quickly get a good estimate of the true gradient ∇C, and this helps speed up gradient descent, and thus learning. To make these ideas more precise, stochastic gradient descent works by randomly picking out a small number m of randomly chosen subset of training inputs called mini-batch. Provided the sample size m is large enough the average value of the the gradient will be roughly equal to the average over all the gradients for the full training set of dimension n:

$$\frac{\sum_x \nabla C_x}{m} \approx \frac{\sum_x \nabla C_x}{n} = \nabla C \tag{7}$$

where the second sum is over the entire set of training data. It is thus possible to estimate the overall gradient by computing the gradients just for the randomly chosen mini-batch of size m. The weights and biases for the networks are thus updated at each iteration making the network learn faster then computing the gradients for the whole set of inputs. With respect to the notation used when introducing the MLP network, it is possible to update the weights as follows:

$$w_k \rightarrow w_k' = w_k - \frac{\eta}{m} \sum_j \frac{\partial C_{x_j}}{\partial w_k} \tag{8}$$

$$b_l \rightarrow b_l' = b_l - \frac{\eta}{m} \sum_j \frac{\partial C_{x_j}}{\partial b_l} \tag{9}$$

When all the mini-batches have been evaluated and the training set of inputs is exhausted, it is said to complete an epoch of training. The process keeps going with another epoch until the number of prefixed epochs is reached. It is worth noting that the learning rate η is a hyper parameter that must be assigned in advance.

So far it has been described how neural networks can learn their weights and biases using the gradient descent algorithm. Yet, is not clear how to compute the gradient of the cost function. In the following lines I will describe a fast algorithm for computing such gradients, an algorithm known as backpropagation[1]. It is worth recalling the notation used to describe a fully connected network. It will be used w^l_{jk} to denote the weight for the connection from the k-th neuron

[1]The backpropagation algorithm was originally introduced in the 1970s, but its importance wasn't fully appreciated until a famous 1986 paper by David Rumelhart, Geoffrey Hinton, and Ronald Williams. That paper describes several neural networks where backpropagation works far faster than earlier approaches to learning, making it possible to use neural nets to solve problems which had previously been insoluble. Today, the backpropagation algorithm is the workhorse of learning in neural networks.

in the $(l-1)$-th layer to the j-th neuron in the l-th layer. The same notation is applied to biases. Sticking to the aforementioned notation, the activation a_j^l of the j-th neuron in the l-th layer is related to the activations in the $(l-1)$-th layer by the equation:

$$a_j^l = \sigma\left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l\right) \tag{10}$$

where the sum is over all neurons k in the $(l-1)$-th layer. The same equation can be expressed in a more elegant way by using the matrix notation: $a^l = \sigma(w^l a^{l-1} + b^l)$. In this case w^l is the weight matrix on layer l, the bias vector is set to be b^l, and the activation is function of the preceding layers' activations. It is clear to see the activations in the previous layer influences the activation of the subsequent neuron. Another useful quantity is the weighted sum of the input at layer l defined as $z^l = w^l a^{l-1} + b^l$. With respect to the previously defined quantities it is possible to express the activation value as a the output of $a^l = \sigma(z^l)$ where σ is a general activation function.

The goal of backpropagation is to compute the partial derivatives the cost function C with respect to any weight w or bias b in the network. To compute the desired quantities it is useful to introduce δ_j^l, which is called the error in the j-th neuron of the l-th layer. Backpropagation will give a procedure to compute the error δ_j^l, and then will relate it to $\partial C/\partial w_{ij}^l$ and $\partial C/\partial b_j^l$. By definition the error is computed as the derivative of the cost function with respect to the weighted sum:

$$\delta_j^l \equiv \frac{\partial C}{\partial z_j^l} \tag{11}$$

Backpropagation is based around four fundamental equations. Together, those equations give a way of computing both the error δ_j^l and the gradient of the cost function. Following the chain rule of partial derivatives it is possible to obtain the first equation of the backpropagation algorithm:

$$\delta_j^L = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L) \tag{12}$$

Where the letter L indicates the last layer of the network being its output. Recall that while building the network's architecture, the programmer also assigns a given and known cost function. Furthermore, all the activation functions have an analytical form such that it is possible to compute their derivatives. Given all of the above it is possible to directly compute the quantity δ_j^L when the input has been fed forward and an output obtained. Eq.12 can be also expressed in matrix form: $\delta^L = \nabla_a C \odot \sigma'(z^L)$ where $\odot$ represents the tensor product element-wise.

It is now clear how to compute the error in the final layer, the problem now shifts to the computation of the error in the generic layer l given the known error of the last layer. Again, by exploiting the chain rule it is possible to link the error of a layer with the following one as follows:

$$\delta^l = \left((w^{l+1})^T \delta^{l+1}\right) \odot \sigma'(z^l) \tag{13}$$

where $(w^{l+1})^T$ is the transpose of the weight matrix w^{l+1} for the $(l+1)$-th layer. By combining Eq.12 with Eq.13 it is possible to compute the error δ^l for any layer in the network. The process starts by using Eq.12 to compute δ^L , then apply Eq.13 to compute δ^{L-1}, then the same process continues until the firs layer is reached. This backwards computation is the reason of the name backpropagation. Finally it is possible to link the error to the partial derivatives of the cost function with respect to both weights and biases:

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l. \tag{14}$$

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l \tag{15}$$

The above equations describe the analytical formulation of the backpropagation algorithm which is based upon Eq.12 trough Eq.15. Once the learning rate η has been chosen, the above equations, along with Eq.8 and Eq.9 allows the

gradient descent algorithm to be effective in lowering the value of the cost function, thus tuning the network parameters in the learning process.

To sum up, the learning process occurs feeding an input, from the mini-batch, trough the network obtaining a first output to be evaluated against the ground truth. Next, the error of the final layer is directly obtained and the backpropagation takes place in order to compute the errors for each layer. The final step updates the parameters relying on the stochastic gradient descent algorithm. Once the process has been completed for each mini-batch, the next epoch is evaluated and so on.

Another idea for the computation of the gradients is the direct application of the definition of derivative. A small change in on parameter will eventually affect, trough a feed-forward step, the cost function, and repeating the process for each weight and bias it would be possible to obtain the whole gradient ∇C. Unfortunately, such a procedure becomes impossible once the parameters grow in numbers. On the other hand, backpropagation works obtaining the every gradients all at once at each feed-forward step. The power of this algorithm is exactly this ability, namely to dramatically speed up the process[2] of calculation, thus learning.

6 Advanced Algorithms

In this section i will focus on some of the algorithms that are used for trajectory forecasting applications. I will also describe some advanced methods for analyzing images and generate new visual contents. While the basic concepts of neural networks hold, some new techniques have been developed to obtain better results and make machine learning more capable not only of understanding, but also at generating new content from scratch.

6.1 Generative Adversarial Networks

Generative Adversarial Networks are a subset of generative models which are able to produce an entirely new content never observed before. This is in contrast with classification networks that map a given input to a prefixed labeled class. GANs typically consist of two networks (Figure 6) that compete with each other: a generator, and a discriminator. While the generator learns to generate realistic samples from input data, the discriminator is trained to discern which samples are real, and which are generated, thereby engaging in a two-player min-max game.

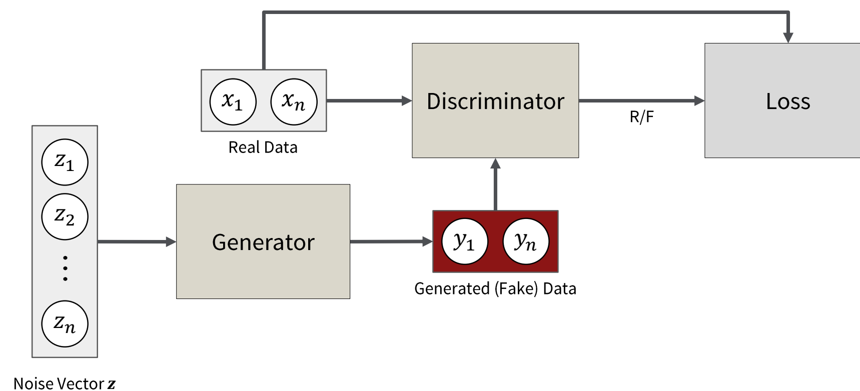

Figure 6: GAN block scheme

The generator is a neural network that takes as input a random set of variables (usually white noise) and returns, once trained, an output to match the target. Being the task very complex (e.g. generating a photo-realistic picture) the clever idea behind GANs relies on training the generator with another neural network called discriminator. The latter takes as input the generated data and returns as output the probability of this data to be real or synthetic. Both the networks are then trained jointly with opposite goals. The generator tries to fool the discriminator, so the generative neural network is trained to generate data as close to the real as possible. On the other hand, the discriminator is forced to detect synthetic data. At each iteration of the training process, the weights of the generative network are updated in order to increase the realism of the output, whereas the weights of the discriminator are updated to maximize the precision of the classification between real and fake. Both networks try to beat each other (thus "adversarial") improving their

[2]This speedup was first fully appreciated in 1986, and it greatly expanded the range of problems that neural networks could solve. That, in turn, caused a rush of people using neural networks.

performances at each iteration. From a game theory point of view, this is called a mini-max two-players game, where the equilibrium state corresponds to the situation where the generator produces data indiscernible from the ground truth and the discriminator predicts the validity of the output with a 50% of probability. To formalize the above description it is necessary to introduce some notation. The generator takes a random input $z \sim \mathcal{N}(\mu = 0,\ \sigma^2 = 1)$ and returns as output the generated data $x_g = G(z)$. On the other hand, the discriminator randomly accepts as an input either a true data x_t or the generated data x_g and outputs the probability $D(x)$ of the input being real or fake data. The expected value of the discriminator is:

$$E(G, D) = \mathbb{E}_x [\log D(x))] + \mathbb{E}_z [\log(1 - D(G(z)))] \tag{16}$$

The goal of the generator is to fool the discriminator whose objective is to distinguish between true and generated data. So, when training the generator, the goal is to maximize this error while minimizing it for the discriminator:

$$\max_G \min_D E(G, D) \tag{17}$$

To sum up, GANs can be trained to generate realistic outputs that mimic real data. The generated data is not a copy of the dataset but closely matches its patterns. For the present work, GANs will be used to generate pedestrian trajectories to be as close as possible to the real ones.

6.2 Recurrent Neural Networks

Every network analyzed above is time independent. This means that each output won't be influenced by the time history the network has gone trough. For tasks that are time dependent this approach poses high limitations. Humans don't start their thinking from scratch every second. Any input that humans receive is elaborated also as a function of the previous instants in time thus relying on temporal history. Traditional neural networks can't do this, and it seems like a major shortcoming. Recurrent neural networks address this issue. They are networks with loops in them, allowing information to persist. Once the network has been trained, in traditional MLPs, the input is fed trough each layer with a feed-forward step to obtain the desired output. If another input is fed, the output won't be aware of the previous input thus abandoning any temporal connection. In some applications, such temporal history is fundamental. As an example, in speech recognition, humans read acquiring knowledge from previous words in the sentence and building the meaning of the phrase upon that information. For trajectory forecasting is fundamental to preserve the temporal history of each agent in the scene to be able to predict their future steps.

Recurrent neural networks are intrinsically time dependent due to their chain structure. The output of the network is then also used as an input for the next step thus allowing for time history to be propagated. This chain-like nature reveals that recurrent neural networks are intimately related to time sequences. The standard recurrent neural networks (RNN) lack the capability of storing an information for long period of time, and this could be a great limitation in some cases. Long Short Term Memory networks are a special kind of RNNs, capable of learning long-term dependencies. The basic chain structure of an LSTM network is shown in Figure 7.

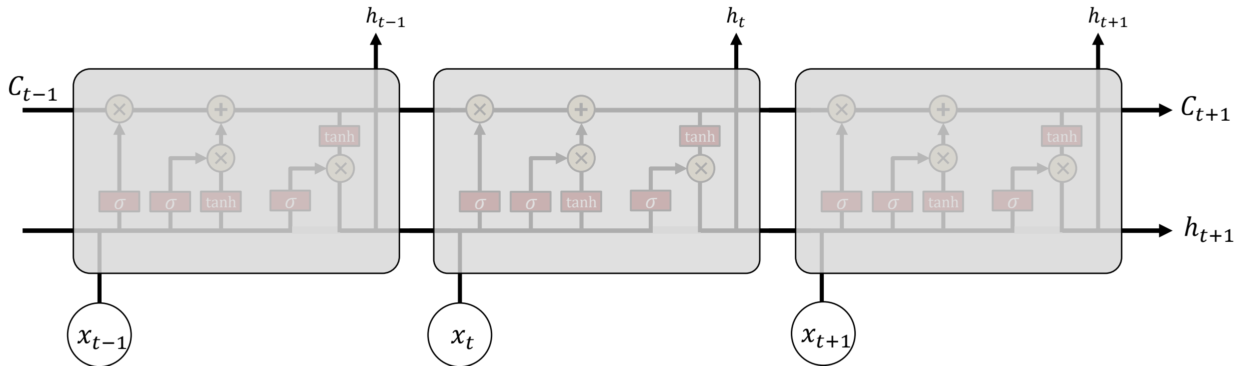

Figure 7: LSTM Network

LSTMs are explicitly designed to avoid the long-term dependency problem. All recurrent neural networks have the form of a chain of repeating modules of neural network. They are based upon four neural network layers, interacting

in a very special way. In Figure 7, each line carries an entire vector, from the output of one node to the inputs of others. The circles represent pointwise operations, while the red boxes are neural network layers. Lines merging denote concatenation, while a line forking denote its content being copied. The key to LSTMs is the cell state, the horizontal line running through the top of the diagram. It runs straight down the entire chain, with only some minor linear interactions. It's very easy for information to just flow along it unchanged [2]. The cell state propagates the context information throughout all the blocks preserving information in time. The LSTM does have the ability to remove or add information to the cell state, regulated by structures called gates. Gates are a way to optionally let information through. They are composed out of a sigmoid neural net layer and a pointwise multiplication operation. The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through. A value of zero means deletes a content, while a value of one leaves the cell state untouched. An LSTM has three of these gates, to protect and control the cell state.

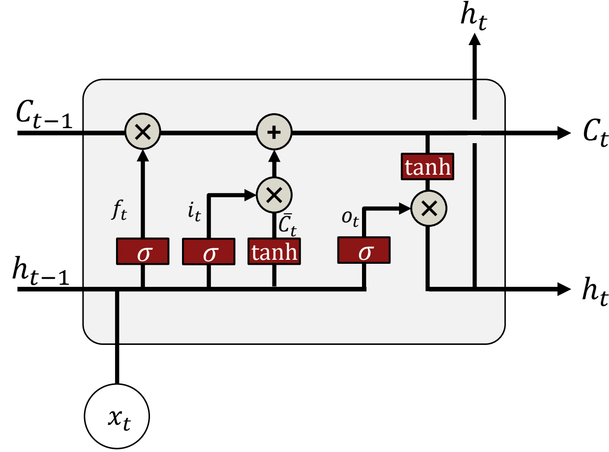

Figure 8: LSTM unit cell

With respect to Figure 8, the first step of an LSTM is to decide what information to delete from the cell state. This decision is made by a sigmoid layer called the "forget gate layer." It accounts for h_{t-1} and x_t, and outputs a number between 0 and 1 for each element in the cell state C_{t-1} according to the following equation:

$$f_t = \sigma \left(W_f \cdot [h_{t-1}, x_t] + b_f \right) \tag{18}$$

The next step is to decide what new information to store in the cell state. This step is divided in two parts. First, a sigmoid layer called the "input gate layer" decides which values to update. It uses as input the concatenated vector of both h_{t-1} and x_t like the forget layer. Next, a tanh layer creates a vector of new candidate values, $\bar{C}_t$, that could be added to the state. Both this operations are expressed below:

$$i_t = \sigma \left(W_i \cdot [h_{t-1}, x_t] + b_i \right) \tag{19}$$

$$\bar{C}_t = \tanh \left(W_C \cdot [h_{t-1}, x_t] + b_C \right) \tag{20}$$

Once the above quantities have been computed they will be combined to create an update to the state. To update the cell state C_{t-1} to new value C_t is just a matter of simple operations already possessing every intermediate result. The previous steps already decided how to perform such update. To compute C_t, C_{t-1} is multiplied by f_t, deleting (forgetting) the information that have been considered not useful. Then is possible to add the value $\bar{C}_t \cdot i_t$ which is the new candidate value, scaled by how much to update each state value. To sum up, the updated cell state is computed as follows:

$$C_t = C_{t-1} \cdot f_t + \bar{C}_t \cdot i_t \tag{21}$$

The last step is to generate the final output. This output will be based on the cell state, but will be a filtered version. First, a sigmoid layer is computed to decide what parts of the cell state to output. Then, we put the cell state through

tanh (to push the values to be between -1 and 1) and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to.

6.3 Convolutional Neural Networks

Convolutional neural networks have become the standard in image recognition. This section aims at describing how the convolutional process extracts the most relevant features in an image. In a standard MLP network an image should be fed by creating a column vector storing all the pixel intensity values concatenating adjacent rows. It is clear that the spatial information of an image is compromised. Furthermore, a small tilt of the input picture could fire completely different neurons not allowing for a good generalization of the image recognition process. To avoid such problems convolutional neural networks have been developed. Convolutional neural networks use three basic ideas: local receptive fields, shared weights, and pooling.

It is helpful to think about convolutional network as having the first layer composed by square neurons with same size of the input image.

(a) Convolution step 1

(b) Convolution step 2

Figure 9: Convolution process

The input pixels will be connected to subsequent layers of hidden neurons. In CNN not every input pixel is connected to every hidden neuron. Instead, connections have a small area, localized regions of the input image. To be more precise, each neuron in the first hidden layer will be connected to a small region of the input neurons, in Figure 9 a 3×3 region (kernel), corresponding to 9 input pixels. That region in the input image is called the local receptive field for the hidden neuron. Each connection learns a weight. And the hidden neuron learns an overall bias as well. The next step is to slide the local receptive field across the entire input image as depicted in Figure 9. For each local receptive field, there is a different hidden neuron in the first hidden layer called feature map. And so on, building up the first hidden layer. Note that the size of the first hidden layer is reduced by the following formula:

$$O = \frac{I - K}{S} + 1 \tag{22}$$

Where O is the size of the output, I the input one, K is the kernel's dimension and S is the stride which is the number of pixels that the kernel is slid of across the input picture at each step.

As been said each hidden neuron has a bias and $K \times K$ weights connected to its local receptive field. To introduce the shared biases and weights it is possible to refer to the next equation:

$$c_{ij} = \sigma\left(\sum_{l=0}^{4}\sum_{m=0}^{4} w_{l,m}a_{j+l,k+m} + b\right) \tag{23}$$

Where σ is the neural activation function, b is the shared value for the bias and $w_{l,m}$ is an array of shared weights. This means that all the neurons in the first hidden layer detect exactly the same features, just at different locations in the input image. As an example imagine that the weights and bias are such that the hidden neuron can pick out a vertical edge in a particular local receptive field. That ability is also likely to be useful at other places in the image. And so it is useful to apply the same feature detector everywhere in the image. The map from the input layer to the hidden layer is thus called feature map. The shared weights and bias are often said to define a kernel. The programmer have the ability do define both the number of convolutional layers and the number of feature maps to extract from each layer. It is common to start from a single picture with the highest dimension and proceed along the network with the extraction

of multiple feature maps. The network will learn things related to the spatial structure. Another advantage of sharing weights and biases is that it greatly reduces the number of parameters involved in a convolutional network.

In addition to the convolutional layers just described, convolutional neural networks also contain pooling layers. Pooling layers are usually used immediately after convolutional layers. What the pooling layers do is simplify the information in the output from the convolutional layer by extracting, in max-pooling, the maximum activation in the kernel input region.

6.3.1 Transposed Convolution

The previous paragraph focused on convolutional networks. The main idea is to extract relevant feature maps containing the most relevant information of the original image. The feature maps have reduced dimensions and are usually in great number. The transposed convolution relies on the opposite procedure. It takes as an input some feature maps and upscale them to generate a bigger and richer image. The transposed convolution can then be exploited to generate images from fragmented pieces of visual information.

7 Path Prediction

Path prediction consists in forecasting the future positions of agents (e.g. humans or vehicles) within an environment [3]. Formally defined, human trajectory forecasting is the problem of predicting the future navigation movements of pedestrians (namely their x and y coordinates on a 2D map representation), given their prior movements and additional contextual information about the scene. It is assumed that the route taken by each pedestrian is influenced by the location of other humans and the physical constraints on its path, as well as its own goal, which is to some extent encoded in its past course of movements. For any particular scene, the inputs to the model are twofold [4]: the scene information, in the form of a top-down view image of the scene, I_t, and the previously observed trajectory within the scene of each of the N currently visible pedestrians:

$$X_i = \left\{ \left(x_i^t, y_i^t\right) \in \mathbb{R}^2 | t = 1, ..., t_{obs} \right\} \forall i \in \{1, ..., N\}$$

Given all above inputs and the ground truth future trajectory for each pedestrian between t_{pred} and t_{obs} time steps, i.e.:

$$Y_i = \left\{ \left(x_i^t, y_i^t\right) \in \mathbb{R}^2 | t = t_{obs} + 1, ..., t_{pred} \right\} \forall i \in \{1, ..., N\}$$

the goal is to learn the underlying distribution which can generate feasible samples for their future trajectories, i.e.:

$$\hat{Y}_i = \left\{ \left(\hat{x}_i^t, \hat{y}_i^t\right) \in \mathbb{R}^2 | t = t_{obs} + 1, ..., t_{pred} \right\} \forall i \in \{1, ..., N\}$$

7.1 Baseline: S-GAN

To obtain a baseline for comparison I used, as a benchmark, the trajectory forecasting algorithm S-GAN [5]. The aforementioned algorithm does not take into account the physical context of the scene thus not requiring any image as an input. It works analyzing the time history of each agent in the scene storing for each the coordinates with respect to a fixed reference frame. The positions are then encoded utilizing an LSTM encoder and the forecast trajectories are generated using a GAN module. When published, S-GAN was the state-of-the-art algorithm and it has been trained and optimized using available datasets including ETH [6] and UCY [7]. These datasets consist of real-world human trajectories with rich human-human interaction scenarios. The evaluation metrics are the following:

1. Average Displacement Error (ADE): Average $L2$ distance between ground truth and our prediction over all predicted time steps.
2. Final Displacement Error (FDE): The distance between the predicted final destination and the true final destination at end of the prediction period T_{pred}.

To further refine the baseline and obtain a more realistic scenario, the model has been trained on the JRDB dataset. To do so I had to create a package to allow the dataset to be red by S-GAN converting each pedestrian label (manually labeled) to a readable x and y coordinate converted in meters. The package is called JRDB Converter and allows also to convert LiDAR point-cloud scans to a top view image as explained later in the report. The converted data has been used as the training set for the model with the results expressed in the Table 1.

		Training Set			
		JRDB-In	JRDB-Out	ETH (worst)	ZARA2 (best)
Eval Set	JRDB-In	**0.08 / 0.18**	**0.08 / 0.18**	0.14 / 0.28	0.12 / 0.24
	JRDB-Out	**0.07 / 0.15**	**0.06 / 0.13**	0.14 / 0.29	0.12 / 0.24
	ETH	1.66 / 2.85	1.02 / 1.82	0.72 / 1.31	0.52 / 0.94
	ZARA2	0.64 / 1.28	0.37 / 0.78	0.32 / 0.67	0.31 / 0.64

Table 1: S-GAN Baseline results

The dataset have been split in two different sets: 80% has been allocated for the training while the remaining 20% has been used to validate the model and obtain the final results in Table 1. The JRDB Dataset has been further split in two blocks dividing indoors from outdoors scenes. In the table above each column represents the the training set which the networks has been trained on. Next, each model is tested on the evaluation set as expressed by each row. Every training set name derives from the dataset that has been trained on while the Eval Set refers to the evaluation set been used to obtain the results. There is no appreciable difference between the models trained with indoors data with respect to outdoor ones. I highlited in bold the results obtained using the new JRDB dataset resulting in outstanding performances outperforming the best models of S-GAN.

7.2 SoPhie: An Attentive GAN for Predicting Paths Compliant to Social and Physical Constraints

One of the current state of the art algorithms to forecast pedestrian trajectories is SoPhie. I decided to exploit this algorithm that take as an input the top view scene as seen from an above drone not usually available in real case scenarios. The algorithm addresses the problem of path prediction for multiple interacting agents in a scene, which is a crucial step for many autonomous platforms such as self-driving cars and social robots. SoPhie is an interpretable framework based on Generative Adversarial Network (GAN),which leverages two sources of information, the path history of all the agents in a scene, and the scene context information, using images of the scene. To predict a future path for an agent, both physical and social information must be leveraged. SoPhie blends a social attention mechanism with a physical attention that helps the model to learn where to look in a large scene and extract the most salient parts of the image relevant to the path. Whereas, the social attention component aggregates information across the different agent interactions and extracts the most important trajectory information from the surrounding neighbors. SoPhie also takes advantage of GAN to generates more realistic samples and to capture the uncertain nature of the future paths by modeling its distribution [8]. The architecture of the network is shown in Figure 10.

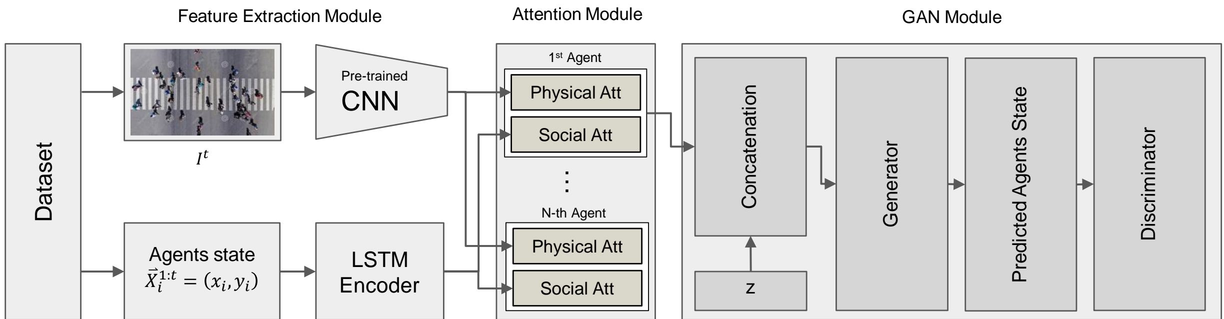

Figure 10: SoPhie block diagram

As shown in the block architecture one of the network's input is a birds-eye view RGB picture obtained from a stationary drone flying above the scene. A more realistic case scenario doesn't allow to take advantage of such solution. The next steps are thus to build a consistent input for SoPhie to allow for on-board sensors to be the only source of information for the network. The proposed solution is detailed in the next section.

8 Trajectory forecasting using on-board sensors

The trajectory forecast problem is too complex to be solved with the conventional coding. Trying a direct and analytical approach to infer pedestrian movements would require a profound understanding of human behavior and an underlying

analytical foundation to build the code upon. On the other hand, a machine learning approach could provide a better result by analyzing a huge set of data of real pedestrian trajectories to extract a plausible generalization policy. Recent works on the problem utilize as an input a top down view of the scene, plus the coordinates of each pedestrian extracted with various methods. The network will use as the visual input both the information coming from the cameras and LiDAR sensors. The agents coordinates are inferred by an online algorithm (e.g. YOLO [9]) from the same images and fed to the network for every agent in the scene. The extended proposed architecture is shown in Figure 11.

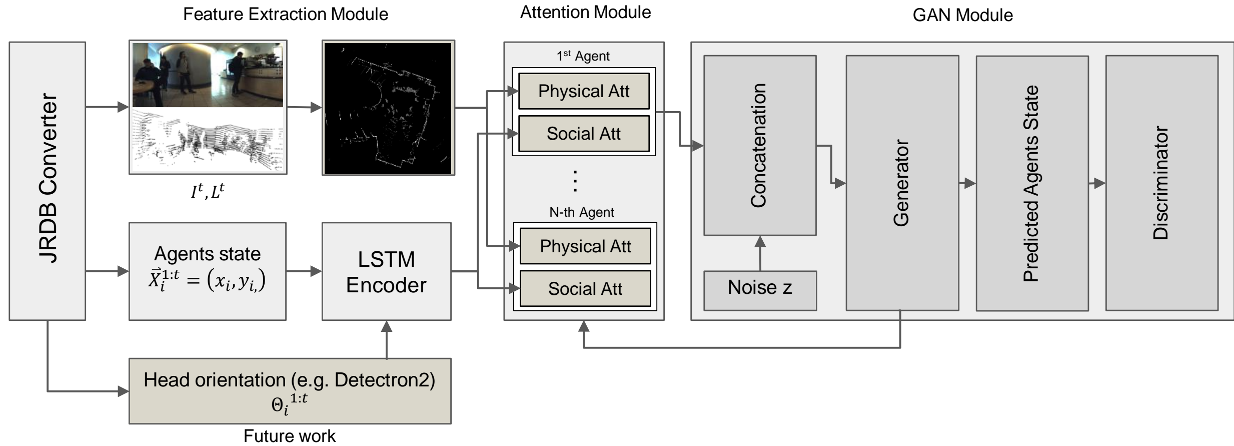

Figure 11: SoPhie Extension block diagram

The usage of a birds-eye view image as the main visual input stems form the necessity of avoiding any occlusion problem that would arise if on board sensors would be used. For real life application though, it is unpractical to deploy a drone to follow the robotic agent broadcasting the live top view image as seen from above. It is then necessary to adopt a method of obtaining the same information utilizing only the provided sensors. With this in mind, I faced the problem of deciding which path to cover: one possibility could be modifying one of the state-of-the-art trajectory forecasting algorithm to rely exclusively on on-board sensors, the other possibility would be reconstructing a synthetic birds-eye view of the scene. I decided to adopt the latter approach being the birds-eye view a valuable source of information also for other tasks a robot could be asked to complete (e.g. following a human agent, obstacle avoidance, etc.). A comprehensive and omniscient top view scene allows for a deeper awareness of both physical and social context. This information can then be fed to any trajectory forecasting network that requires a top view scene as an input (e.g. SoPhie, Social-BiGAT, S-GAN).

8.1 TopNet: a birds-eye view generative network

To obtain a more informative input for the trajectory prediction algorithms, a segmented birds-eye view can be generated using on-board sensors as an input. The most difficult part of the process is the hallucination of occluded areas. Human beings are extremely capable in such a task, mainly exploiting prior experiences to infer the most probable scene. If a human is standing behind a table with his lower body occluded we can easily infer legs position with some confidence. The same concept applies with the physical environment like hallucinating a cross road approaching an intersection or infer a building shape from a side view. The goal is to make the network able of hallucinating occluded areas training the former with some ground-truth omniscient maps.

The approach chosen to solve the current problem has been modeling a supervised convolutional neural network as shown in Figure 12.

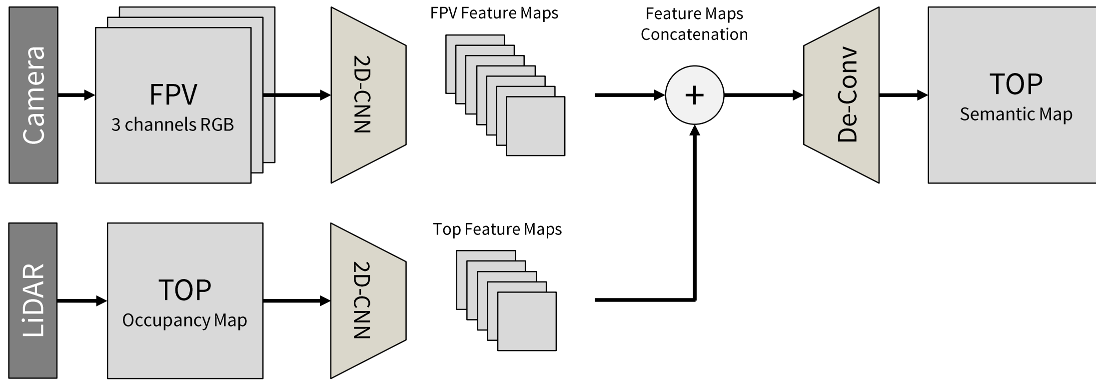

Figure 12: Top View Net Block Diagram

The network is composed of two separate branches: one for the first person view input (3 channels RGB) and the other branch to account for LiDAR information. The architecture is based on a feature extractor module, utilizing a first series of convolutional layers, a concatenation node, and a series of shared de-convolutional layers to generate the final visual output. The camera input is used to obtain a rich and dense information regarding the observed scene. It will be used to extract the semantic features of the context thus obtaining a first estimation of sections to be hallucinated (e.g. a door opened on a corridor will trigger the hallucination, to some extent, of the non observable corridor parts). Furthermore, the first person view also serves to generate the textures to be used in the top view representation. On the other hand, the point-cloud extracted from LiDAR can be projected on a two dimensional plane obtaining a first rough estimate of the shape of the environment. The latter information will serve, once concatenated with the RGB input, as a support to reconstruct a richer birds-eye view.

The input to the network is thus a 4-channels pseudo-image as shown in Fig. 13. The four channels consist in the standard RGB input plus an additional fourth layer being the inferred top view coming from LiDAR point cloud.

(a) RGB First Person View

(b) MONO Top View

Figure 13: Network 4-Channels Input

As it can be seen, the input images don't share the same dimensions. Furthermore, while the camera image have a fixed resolution, the LiDAR projected top-view can be tuned on demand utilizing some hyper parameters. The programmer can chose both the resolution and the extension, in meters, of the area of interest. This freedom and could arise some problems during real time applications. The laser rays can reach very far distances if not blocked by any solid surface. In Figure 13b one can observe that some rays have traveled a big distance due to the presence of several windows in the scene. It is clear that those "external" boundaries are not relevant for local motion planning. It will be necessary thus to implement a policy to be able of cropping the picture to a relevant portion of the whole point-cloud. For the first attempts a naive approach has been followed by manually selecting the area given the specific environment.

To train the network and rely on the stochastic gradient descent method it is mandatory to create an automated procedure to feed both the input and the desired target to proceed with the supervised training. The first approach followed consisted in ignoring at first the dataset and focusing on the automation problem to feed the network.

A custom data-loader has been coded to load and automatically create the mini batches for the following training steps. The datalaoder also performs all the necessary transformations on the input images, like cropping and normalization. Furthermore, the dataloader is also capable of augmenting the dataset slightly rotating or deforming the pictures. This capability is extremely handy to augment the number of data points and avoid the network to learn some peculiarities that may be present in the original dataset. Another technique of data augmentation is the horizontal flip of the images. The input is constructed by the concatenation of the three RGB channels of the first person view with the mono top view channel. Every channels is thus converted to a torch tensor, normalized and cropped to the same sizes to obtain a consistent input. The final input is shown in Figure 14.

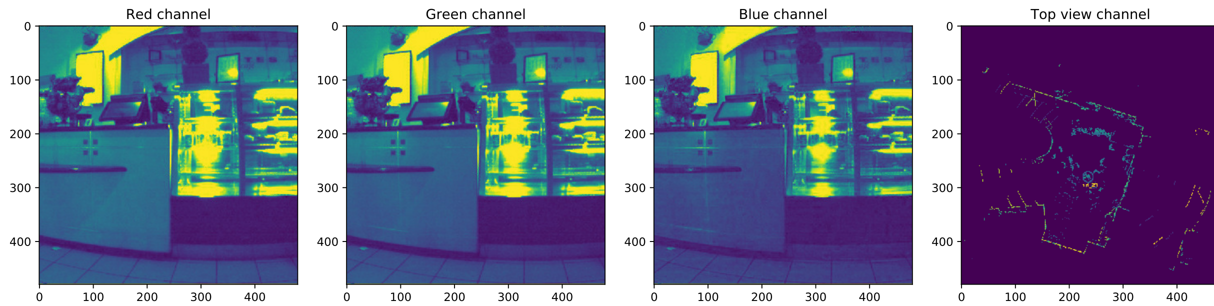

Figure 14: Normalized Input Channels

In the above picture the yellow color represents areas with high intensity values, while blue zones are the darkest. The input is normalized meaning that each intensity value is bounded between an interval of -1 and 1.

To start debugging the code and conduct a sanity check of the network's architecture, as a proof of concept, a first training has been conducted using a full black picture as a target. This naive test's only purpose is to obtain a visual proof of the output of the network, if any. The network has been coded using the PyTorch library [10] to take advantage of the powerful implementation of the backpropagation algorithm. The fed input consists in a 4-channels (pseudo-image) generated using random white noise extracted form the normal distribution $z \sim \mathcal{N}(\mu = 0,\ \sigma^2 = 1)$ as seen in Figure 15. Each pixel in every input channel has an intensity value between -1 and 1 divided in 255 intervals (8-bit images). The same proprieties applies to the real image inputs.

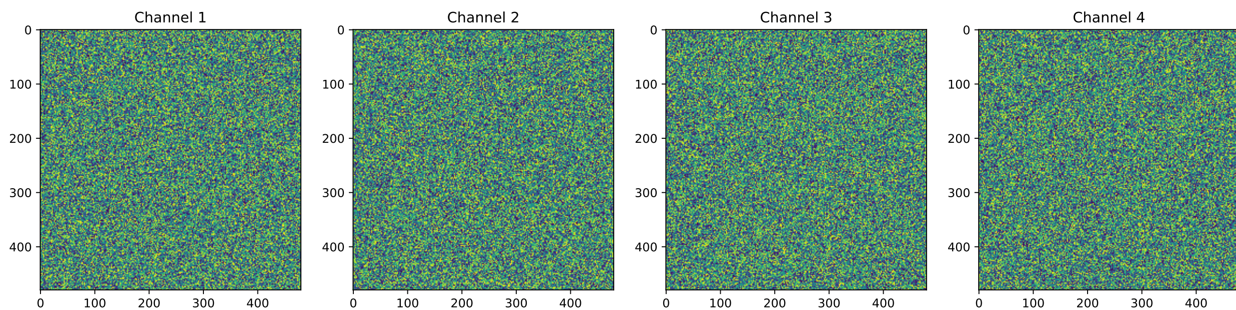

Figure 15: Random noise 4-channels input

Following the first branch of the network shown in Figure 12 the input goes trough a series of convolutional layers to extract the feature maps. Next, the information is used to generate a final output by means of some de-convolutional layers. Giving the network a black target picture to be recreated it should be able to learn all the weights of the convolutional layers to output an empty image being forced by a L2 mean square error function against the target. In this example, the loss is computed on pixel intensity values on the final output. The output image should become

darker for each training epoch given that the back propagation algorithm is working properly. The hope is to obtain a completely black image after a relative small number of epochs. The results of such trial can be evaluated in Figure 16.

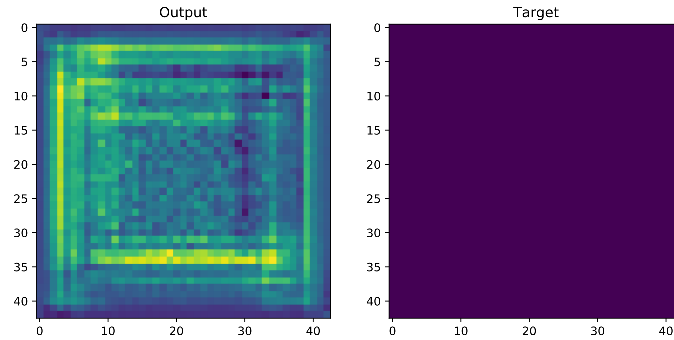

Figure 16: First training results: Output vs Target

Unfortunately, the results of the test were not satisfactory. Even tough the network is capable of providing a visual output, the latter is not consistent with the target by any means. This may be due to a poor convergence of the loss function caused by the vanishing gradient problem. Furthermore, the time necessary to complete 500 training epochs was around 17 minutes on the local workstation. To speed up the process, a GPU-based approach has been adopted. Pytorch allows for a simple implementation of CUDA, resulting in outstanding time performance improvements as shown in Figure 17 where the GPU approach (blue line) has cut the computation time to 2 minutes per training.

Figure 17: Loss during training: GPU (blue line) vs CPU (orange line)

Further analysis on the code revealed that the the final layer neuron outputs were not bounded within the same range of the target image resulting in a non consistent loss evaluation. This problem has been tackled utilizing a tanh activation function applied to the final layer. To improve network's performances the following steps were made: adding $ReLU$ activation functions between the de-convolution layers and a final $tanh$ layer to bound the final layer output in the range $O \in [-1, 1]$. The final bulk of the code, relative to the network's architecture, is presented below using the PyTorch implementation. The convolutional layers are built passing multiple parameters such as: the number of input channels, the number of output feature maps, the kernel size, the stride and the padding. The following MaxPool layers accept, as the required input, the size of the kernel, while the ReLU activation is a pre-built function not requiring any additional inputs. All the inputs are considered as hyper-parameters and will be eventually tuned on the final network architecture based on the results on the real dataset. The forward function, which is the one that propagates the input trouh all the network's layers, is defined as the sequence of the encoder and the decoder applied to the input x being the 4-channels pseudo-image. PyTorch is capable of condensing the main part of the network in a very limited number of code lines resulting for me the best approach to the problem.

```
class autoencoder(nn.Module):
    def __init__(self):
        super(autoencoder, self).__init__()
```

```
        self.encoder = nn.Sequential(
            nn.Conv2d(4, 16, 3, stride=3, padding=0),
            nn.ReLU(True),
            nn.MaxPool2d(2, stride=2),
            nn.Conv2d(16, 8, 3, stride=2, padding=1),
            nn.ReLU(True),
            nn.MaxPool2d(2, stride=1))

        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(8, 16, 3, stride=2),
            nn.ReLU(True),
            nn.ConvTranspose2d(16, 8, 5, stride=3, padding=0),
            nn.ReLU(True),
            nn.ConvTranspose2d(8, 3, 2, stride=2, padding=0),
            nn.Tanh())


    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)
        return x
```

Furthermore, an improvement in the target picture consists in utilizing the input itself to obtain a more realistic case. Eventually, the full semantic top view image will be utilized as a target. The previous changes constitute a naive implementation of a convolution auto-encoder network. An auto-encoder is the combination of an encoder function, which converts the input data into a different representation, and a decoder function, which converts the new representation back into the original format. Autoencoders are trained to preserve as much information as possible when an input is run through the encoder and then the decoder. An input image is forwarded trough the convolution layers to extract the most important feature maps and then a series of transposed convolution layers are used to up-sample the feature maps and reconstruct the image with less but more salient information. The input images have been chosen among a subset of the JackRabbot Dataset. The loss function is again a $L2$ Mean Square Error function comparing the output and target picture pixels intensity on a single channel. In Figure 18 we can appreciate a substantial improvement compared to previous approach. Two sets of images are shown: the first picture is the target and the following is the network's output (inferred image).

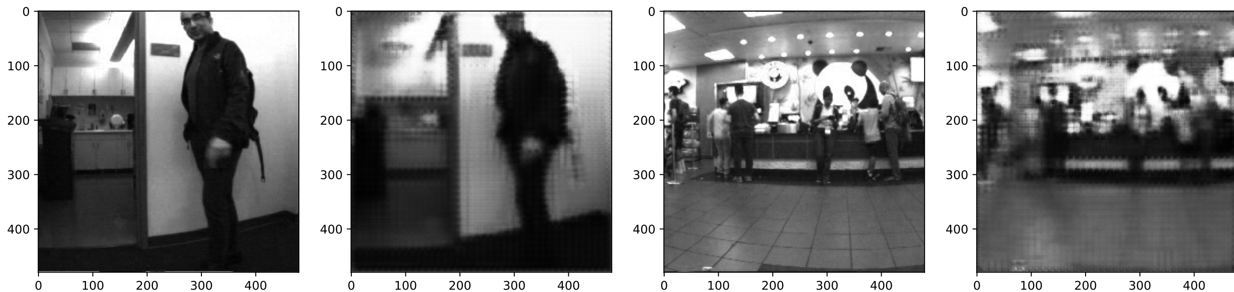

Figure 18: Autoencoder first-person view results

It's worth stressing that the output of the network is the result of a learning process trying to extract the most relevant feature from the input image (e.g. sharp corners, edges, etc.) and reconstruct a final output with the more relevant information. The first example of Figure 18 highlights the physical boundaries and successfully identifies a person in the scene. Unfortunately the networks is not able to provide the head orientation of the agent which could be a problem in a social navigation problem. In the following example the physical boundaries are still correctly inferred as well as the number and position of each agent in the scene. The results suggest that the proposed architecture is a good candidate for the final network. To further confirm this statement, another test has been conducted on the top view scene. The main idea is to help the network with a high fidelity birds-eye view provided by a two dimensional projection of the point-cloud extracted for LiDAR sensors. The image should help the network understanding the rough shape of the environment and the first person view will eventually add the semantic information to the scene. The results of the same network are shown in Figure 19.

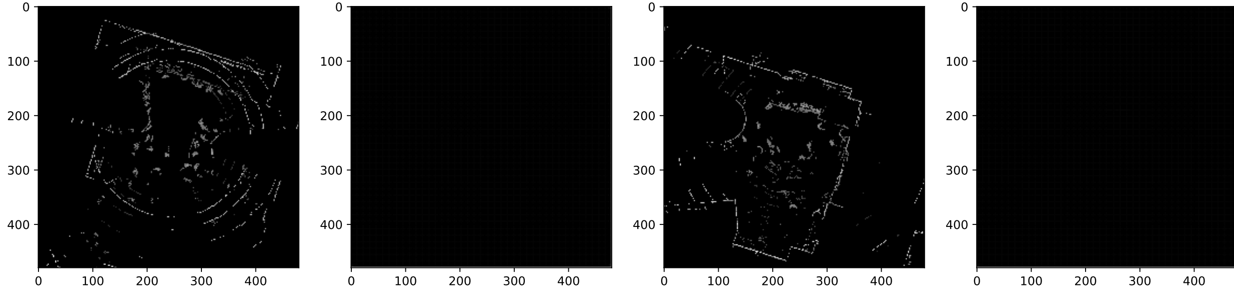

Figure 19: Autoencoder top view with 2d convolution

The results clearly show that the network is not capable of extracting the information fed trough the projected point-cloud image. This is probably due to a noisy and poorly detailed image. The two-dimensional convolution layers are not able to generate representative feature maps like edges or corners. Being the input image a monochromatic picture, and providing only a single information, namely the distance from the sensor, it is possible to convert the two dimensional tensor in a 1-dimensional vector layer storing the pixel intensity values of the whole image. PyTorch allows for a easy conversion of the two-dimensional tensor to a vector using the torch.view() method . The resulting vector is built concatenating each row of pixels in a single column vector. Given a $n \times n$ pixels square image, the resulting column vector will have dimensions $n^2 \times 1$. The input is then fed forward trough a series of one-dimensional convolution layers and the subsequent one-dimensional de-convolutions to generate the final output. To visualize the final image it is necessary to convert the output vector to a tensor with the same size of the desired output. In Figure 20 it is possible to evaluate the results of the new network architecture.

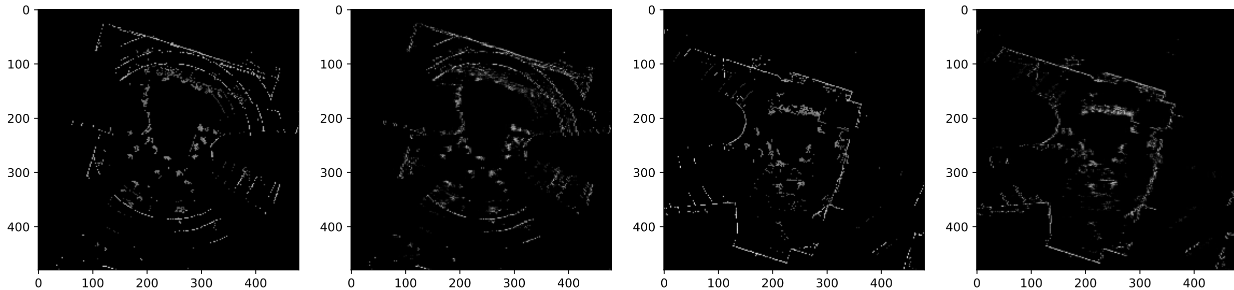

Figure 20: Autoencoder top view with 1d convolution

With respect to the former architecture the present architecture is now able to reconstruct the top view image although with some introduced noise. The output images have yet less structure and definition if compared to the target. It is worth recalling that the LiDAR image is only supposed to help the network obtaining a better understanding of the physical constraints so that the most important features to be highlighted are the outline of the environment like walls. As a comparison, in Figure 21 the loss evolution are evaluated for both first-person and top-view images. It can be seen that the first-person converges at a slower speed and achieves a higher error value. This is probably due to more complex information to be analyzed compared to a mono chromatic and sparse image like the LiDAR counter part.

The above results show that the first convolutional networks are capable of extracting key features from the inputs. It is safe to assume that a concatenation of the output of the convolutional layers can be used to be the input of the transposed convolution layer series for the generation of a richer birds-eye scene. The goal is now to provide the network with a good dataset that provides both a photo-realistic FPV as well as a LiDAR representation of the scene. It is then crucial to obtain a ground truth for the birds-eye view to be used as target in the learning process. The optimization of network's hyper parameters will then be conducted on the final architecture to obtain the best possible results. The aforementioned dataset will be constructed using a Gibson Environment [11] as reported ahead.

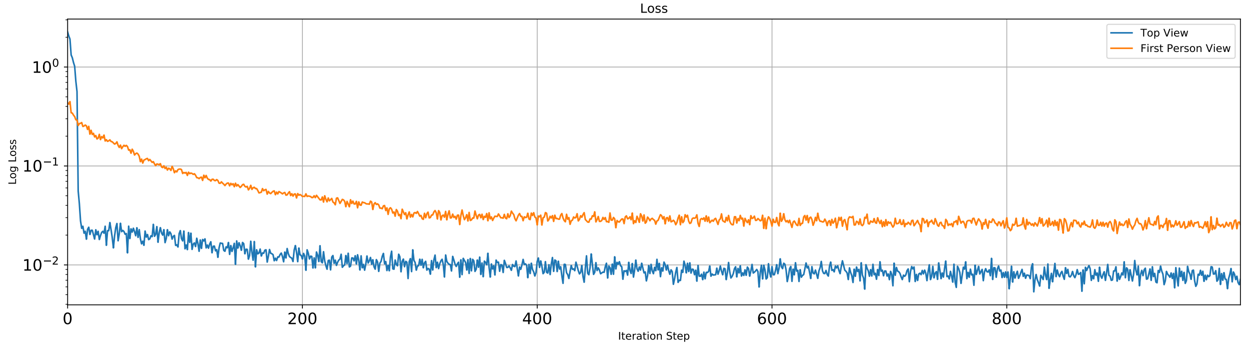

Figure 21: Loss plot

A further development for the presented approach takes advantage of a third information that the LiDAR sensor is capable of providing, namely the occupancy information. The basic concept behind an occupancy grid map is the representation of three different domains: the free space, the solid boundaries and the unknown space. A LiDAR sensor projects n laser rays, by the spinnign of a mirror, and when a non transparent surface is hit, the ray bounces back and the relative point distance is stored in the point-cloud as a tuple of coordinates $\vec{r}_i = (x_i, y_i, z_i)$ with respect to the sensor position. The full pointcloud at time t can be described as follows: $P_t = \{\vec{r}_i\}_{i=1}^{n}$, where n is the number of projected rays. It is possible to distinguish the three domains as follow:

- $\hat{r}_i < \vec{r}_i$: Free space (White)
- $\hat{r}_i = \vec{r}_i$: Solid boundary (Black)
- $\hat{r}_i > \vec{r}_i$: Unknown space (Grey)

For every domain the visual map can be constructed using a unique color. The white space is free, the solid boundaries are represented with black lines (thickness is set in advance) and the unknown areas are shown in light gray. This representation allows to add one more information, namely the unknown domain, to the LiDAR top view image as shown in Figure 22. The white rays that seem to escape from the images are often generated because of windows unable to back-scatter the laser beams. The occupancy grid map representation is clearly more valuable and "human-friendly" representation of the top view scene. Feeding more information to the network is a key aspect to obtain better results. The better the initial guess is the more easily the ouptut will be consistent with the grund truth.

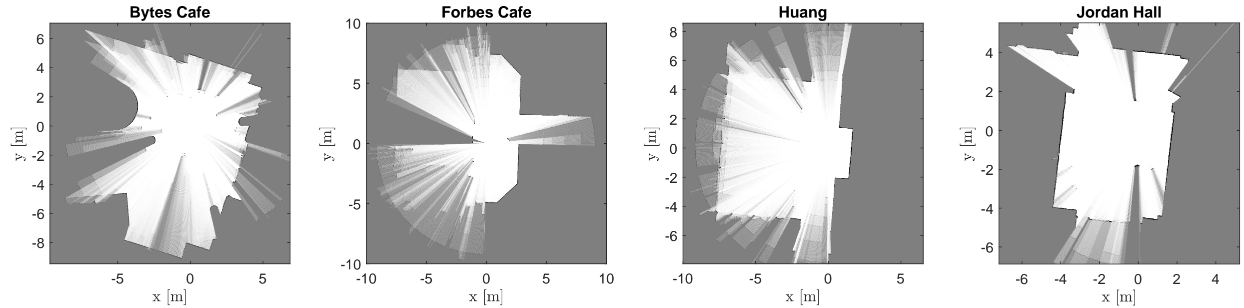

Figure 22: Occupancy Maps

Still, the same problems of the previous approach arise with this representation such as the area of interest to be set a-priori and the resolution of the desired image. In the above picture every scene has been scaled to nicely fit for a clearer visual result.

8.2 Gibson Environment Dataset

To train the network, and allow it to hallucinate hidden parts f the scene, it is crucial to provide it with a good dataset for the trainign process. Unfortunately, the JRDB dataset does not provide a ground truth reference for the top view scene. Usually every trajectory forecasting algorithm uses as an input a birds-eye view to overcome occlusion problems. To

make the network able to generalize it is necessary to provide the same input structure of the real robot, plus a ground truth reference as a target for the loss function. To obtain the datased, a Gibson Environment has been used to output the required images.

Gibson environment is a platform to explore active and real-world perception. It is composed of a graphic engine and a physics simulator. To extract the data needed to train the network I only focused on the renderer. The package provides a big set of indoor environments that have been recorded stitching recorded images on top of a 3D mesh of the buildings. The result is a photorealstic 3D environment which is explorable. The package already comes with a pre-built camera view that will be exploited as the first person view input. The main adjustment to such a camera has been the tuning of the parameters to account for different focal lengths and angle of view to match the robot's on-board camera. The other robot's sensor to simulate is the LiDAR. To create a synthetic LiDAR the 3D mesh was utilized. The final necessary ingredient to complete the dataset is the groundtruth. I utilized an orthogonal top view centered on the robot position on the ground. To generate a big amount of data, I had to automate a navigation path and record each synthetic sensor input in addition to the top view. The dataset is thus assembled recording the robot navigation path trough multiple environments to obtain the necessary diversity of scenarios. As a proof of concept I recorded the inputs from a static scene to obtain some preliminary results and evaluate their validity. The results are shown in Figure 23.

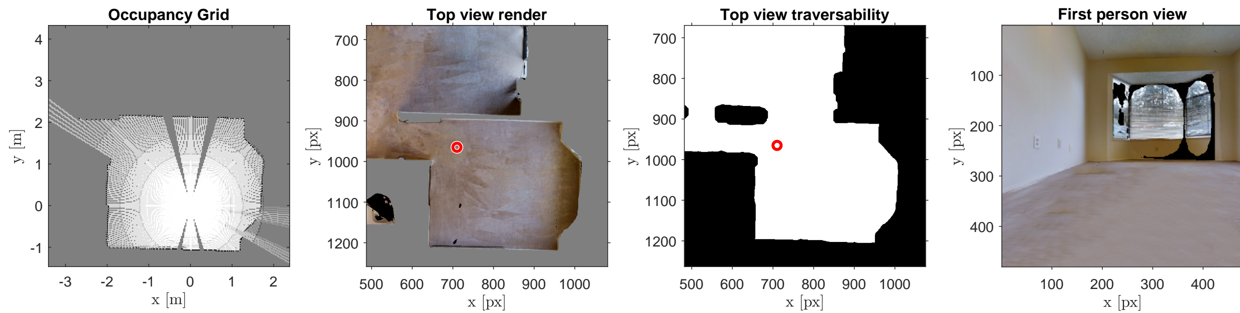

Figure 23: Gibson environment dataset: proof of concept

In Figure 23 the first picture represents the occupancy grid that will be fed trough the LiDAR branch of the network to help the generator understanding the scene physical constraints like walls. The last picture is the first person view sharing the same focal parameters of the real robot's camera. This image will be utilized to extract the visual information and learn the scene characteristics. The first person view will also help the network to understand the parts to be hallucinated. The expected result is shown in the second picture, where the top view is reproduced. Furthermore, I also export a traversability map (third picture), to account for future improvements of the learning process. The most difficult part is to make the first three representation consistent with each other. Every map should perfectly align with the others in order to ease the learning process. To do so, it is necessary to account for the robot position at each time step and center the map with the origin coincident to its position. Once all of the above problems have been solved it is possible to collect the final dataset and start the training process.

8.3 Training the Network

The following work will focus on the trainign process. Once the dataset will be recorded it will be fed as input to the network to start the learning process. It is expected that a lot of work will be necessary to improve network's results tuning the hyper parameters like the number of convolutional layers or the kernel sizes. Furthermore, another possible modification is to exploit a wider field of view, and possibly a full 360 degree visualization instead of the first person view. The aforementioned work will be done during the third month of my stay as a conclusion to the project.

9 Conclusions

The work done at Stanford's Vision and Learning Laboratory, under the supervision of Prof. Silvio Savarese, allowed me to acquire some knowledge in machine learning to enrich my aerospace background with state of the art techniques that will be adopted extensively in space applications. For the results obtained, I was proposed to extend my stay in the laboratory to keep working on the aforementioned problem. The proposal represents for me a great honor and I'm very grateful for the opportunity that ASI and CAIF have given me.

Contents

1 Introduction **1**

2 Introduction to Artificial Intelligence **2**

3 Autonomous Vehicles Forecast **2**

4 Machine Learning in Space Application **3**

4.1 NASA's AstroBee . . . 3

5 Machine Learning Basics **4**

5.1 Artificial Neuron . . . 4

5.2 Neural Networks . . . 5

5.3 Loss Function . . . 6

5.4 Stochastic Gradient Descent and Back Propagation . . . 7

6 Advanced Algorithms **10**

6.1 Generative Adversarial Networks . . . 10

6.2 Recurrent Neural Networks . . . 11

6.3 Convolutional Neural Networks . . . 13

6.3.1 Transposed Convolution . . . 14

7 Path Prediction **14**

7.1 Baseline: S-GAN . . . 14

7.2 SoPhie: An Attentive GAN for Predicting Paths Compliant to Social and Physical Constraints . . . 15

8 Trajectory forecasting using on-board sensors **15**

8.1 TopNet: a birds-eye view generative network . . . 16

8.2 Gibson Environment Dataset . . . 22

8.3 Training the Network . . . 23

9 Conclusions **23**

References

[1] Michael A. Nielsen. Neural networks and deep learning. *Determination Press*, 2015.

[2] Chris Olah and Shan Carter. Attention and augmented recurrent neural networks. *Distill*, 2016.

[3] Amir Sadeghian, Ferdinand Legros, Maxime Voisin, Ricky Vesel, Alexandre Alahi, and Silvio Savarese. Car-net: Clairvoyant attentive recurrent network. 2017.

[4] Vineet Kosaraju, Amir Sadeghian, Roberto Martín-Martín, Ian Reid, S. Hamid Rezatofighi, and Silvio Savarese. Social-bigat: Multimodal trajectory forecasting using bicycle-gan and graph attention networks, 2019.

[5] Agrim Gupta, Justin Johnson, Li Fei-Fei, Silvio Savarese, and Alexandre Alahi. Social gan: Socially acceptable trajectories with generative adversarial networks. 2018.

[6] Stefano Pellegrini, Andreas Ess, and Luc Van Gool. Improving data association by joint modeling of pedestrian trajectories and groupings. pages 452–465, 09 2010.

[7] Laura Leal-Taixé, Michele Fenzi, Alina Kuznetsova, Bodo Rosenhahn, and Silvio Savarese. Learning an image-based motion context for multiple people tracking. 06 2014.

[8] Amir Sadeghian, Vineet Kosaraju, Ali Sadeghian, Noriaki Hirose, S. Hamid Rezatofighi, and Silvio Savarese. Sophie: An attentive gan for predicting paths compliant to social and physical constraints. 2018.

[9] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 779–788, June 2016.

[10] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[11] Fei Xia, Amir R. Zamir, Zhi-Yang He, Alexander Sax, Jitendra Malik, and Silvio Savarese. Gibson env: real-world perception for embodied agents. In *Computer Vision and Pattern Recognition (CVPR), 2018 IEEE Conference on*. IEEE, 2018.