Thesis Dissertation

VIRTUAL CROWDS SIMULATION WITH NEURAL NETWORKS

Georgios Kyrlitsias

UNIVERSITY OF CYPRUS

COMPUTER SCIENCE DEPARTMENT

May 2018

UNIVERSITY OF CYPRUS

COMPUTER SCIENCE DEPARTMENT

Virtual Crowds simulation with Neural Networks

Georgios Kyrlitsias

Supervisor

Professor Yiorgos Chrysanthou

Thesis submitted in partial fulfilment of the requirements for the award of degree of Bachelor in Computer Science at University of Cyprus

May 2018

Acknowledgments

First of all, I would like to thank each and everyone who assisted me throughout this work. First, to project's supervisor Yiorgos Chrysanthou who guided me through the project and gave me the opportunity to work on a topic that I am interested in. Second, to Panayiotis Charalambous who introduced me to the hell that data-driven crowds are and Amyr Borges Fortes Neto for his co-operation, and the opportunities he gave me. Lastly, to friends and family, for being there for me, especially George Stavrou, who had to endure me every day for the last two months.

Abstract

Virtual crowds simulation is an interesting and complex domain in computer science. Multiple approaches were used in order to achieve various crowd behaviors. In recent years, data-driven approaches came up aiming to achieve more realism in this simulations. During this work, the data-driven method used was Neural networks and the behavior we were concerned with was obstacle avoidance. In this thesis an artificial creation of data method, neural network architectures, dataset input-outputs and results will be discussed.

Contents

Glossary

agent Agent is an entity existing in a virtual crowd simulation. Each agent corresponds to one individual in the crowd. Agents in this project may be user-driven or autonomous. 4

obstacle avoidance Obstacle avoidance refers to the capability of an entity, in our case an agent to avoid obstacles while navigating an environment. 2

policy A policy refers to what the neural network algorithm learned from the data. 7

List of Figures

List of Figures

Chapter 1

Introduction

Contents

1.1 Motivation

Crowd simulation is the domain of computer science that simulates various dynamics of crowds in virtual environments. Dynamics may vary but the end goal is always the same, achieving life-like behaviors in our simulations. Dynamics may be related to crowd navigation or crowd animation. Crowd simulations have many applications. They are used in games, making virtual environments more realistic. Furthermore, they are used in movie industry creating crowds that fill an empty scene. Further applications are seen in the context of crisis training and evacuation modeling.

One of the factors that influence crowd simulations is the animation. Modeling and texturing of areas and crowds may be able to look close to real, but the most there is always a missing part, character animations. Animations are an integral part of a simulation, as it demonstrates the variety seen in human behavior. One may model a simulation and have a vast number of behaviors from the navigation standpoint, but if the agents look lifeless, the user will not get engaged in the virtual environment. Animation's significance varies though, depending on the application we use the crowds. In games or movies, the animation is crucial, whereas it is not important in the context of evacuation modeling where we want to observe how a crowd will move during an evacuation. Animation deals with the details that give the unique details needed in an environment.

But what are the dynamics that make a virtual crowd more realistic from the navigation

perspective? We all know how diverse humans are in term of behaviors. Behaviors vary depending on the environment they are currently in. The environment can be known or unknown, leading to different behavior. Moreover, the environment can be scarcely or densely populated demonstrating careless or cautious navigation in the environment. In addition, crowds demonstrate different characteristics when navigating as individuals or in groups. Furthermore, crowds in emergency situations behave in special ways, some people run, other freezes and others become heroes. People have different roles in the environment. They may follow a certain route, work etc. These are just some of the dynamics that may influence someone's behavior.

Recent research breakthrough in Artificial Intelligence and computer vision brought data-driven approaches in the spotlight. The success seen, motivated many people to try similar approaches in other computer science domains. One can only imagine what can be achieved in crowd simulations if neural networks can be used successfully. As mentioned already, crowds may exhibit a vast amount of behaviors. What if it is possible to create a model or a family of models that are able to simulate a number of these behaviors with the ability to generalize to many different situations.

1.2 Scope of the Project

The scope of this work is on simulating virtual crowds using Neural networks. This work will try to incorporate knowledge from research work done in virtual crowds with and without data-driven approaches. Previous work will be combined with original ideas that are going to come through the continuous struggle of experimenting and failing to have any positive results. Different neural network architectures, input state combinations, outputs, and environments are going to be tested in order to create a neural network approach that can imitate unique virtual crowd behaviors.

1.3 Project Aims & Objectives

This project will be aiming to provide a neural network model or family of models that will be able to deal with crowd behaviors, regarding obstacle avoidance. In order to succeed in this project, knowledge regarding virtual crowds and neural

networks will be acquired, appropriate tools, software and programming languages will be examined thoroughly. Afterward, neural networks will be trained with different datasets, architectures, and parameters, aiming towards virtual crowds that can avoid obstacles and generalize in many environments. Following steps were result-driven. A neural network family of models was trained for stationary obstacle avoidance and one more family for moving obstacles. Finally, working on this project, the realization that inputs, outputs, and data needed to be prototyped by creating artificial data was observed. Something that it was not an intention at first, but it was crucial to obtain results.

Chapter 2

Literature Review

Contents

2.1 Macroscopic

Macroscopic level modeling [1] is approaching the crowd simulation domain from a group behavior perspective. This approach treats the crowd as a whole. Consequently, crowd individual details are not taken into consideration in this approach. Instead of individual characteristics, the flow characteristics are the ones exploited. Regression models belong in the macroscopic approach, using statistically driven relations of flow variables to predict crowd flow on various occasions. Route choice models are approaches that try to determine routes to be followed from a utility perspective. The crowds choose it's route with the intention to maximize the utility of it, in order to satisfy various conditions e.g time needed, danger. Furthermore queuing models belong in the macroscopic approach as well. These models use Markov chain models to indicate how crowds move in a node to node manner. Think of a node as a room, and the link as the door.

2.2 Microscopic

Microscopic [1] level modeling is the idea of approaching virtual crowds focusing on each agent specific characteristics. Depending on the nature of the study, different characteristics of crowds may be used in order to simulate various behaviors. For instance, characteristics can be interactions between agents, agent's goal, surroundings or emotional state of the agent. Microscopic models are divided into three subcategories, cellular automata, social forces model and rule based models.

2.2.1 Cellular automata models

Cellular automata models[2] divide the simulation space into a grid. Each cell can be in two states. A cell can be either taken, meaning that an agent or an obstacle is currently at this cell or free meaning that nothing is on the cell's position. The way that agents are moving through the grid of cells is determined by probabilities that each agent have transitioning to each possible cell. The simulation updates all agent positions round by round. Agents try to move towards their intended direction, while they also satisfy all the rules set for their simulation, usually referred to as local rules. The rules used are the intelligence given to agents, in order to simulate realistic crowd behavior.

2.2.2 Social forces models

Social forces model [3] is another popular virtual crowds microscopic approach. This model is successful in simulating real-world crowd movement. This model use forces in order to simulate agent intelligence. Forces may be attractive or repulsive. Attractive forces may be the destination of the agent, other forces for things the agent is interested in, forcing the agent to move towards them before reaching his final destination. On the other hand, repulsive forces are cast from various entities in the environment that the agent would prefer to avoid. These entities could be obstacles, other agents or more case-specific entities like danger in certain areas of the environment.

2.2.3 Rule Based models

Rule-based modeling for virtual crowds is a part of a widely known domain, Ruled-Based AI. Rule base AI is commonly used in real-world applications or for simulating applications. Normally rules are formed as this example - if some conditions are satisfied, follow some predefined actions. In virtual crowds, rules may be derived from psychology research studies or by experience. This kind of rules is in the direct subcategory, because of the direct manner of setting rules and actions. Recently studies for

rule-based models, approach virtual crowds in a more indirect manner. These new approaches try to infer rules using data-driven techniques.

2.2.3.1 Direct

One of the most influential rule-based models is Reynold's simulation of flocks [4]. Flocks were filled with boids that followed rules. In this impressive work, the three rules were separation, cohesion and alignment. Separation is steering a boid in order to avoid colliding with neighbors. Cohesion helped the boids to stay close together. It was achieved by making boids follow the center of the flock. Lastly, alignment enables boids to fly in the same direction for coordinated flock behavior. In Reynold's follow up work [5], he divides the behavior in a three-layer hierarchy and tries to add more complex behaviors in the model. Layers were mentioned as action selection, steering, and locomotion. Furthermore, many common steering behaviors were presented and ways of combining them together.

2.2.3.2 Indirect-Data driven

Data-driven models infer rules in an indirect manner. These models can extract rules from data presented. Data presented can be real-world data that were captured with tracking devices, or can be artificially created data. Real-world data are more interesting to use, offering a huge variety of human behavior. Using them it is possible to simulate vast human behaviors, with drawbacks of working with black box algorithms (machine learning-neural networks) and the need for huge amount of data.

Lerner [6] Lee [7] approached virtual crowds using videos of crowds in order to teach the various crowd behavior. Lerner approach tries to determine the influences that affected the agent's behavior. Then Lerner creates a database that has examples observed. The examples store information about the influences affecting the agent's trajectory. During simulation, the query is created in a similar way with the examples stored in the database. Afterwards, a similarity function is used to determine which

example already seen is the closest one, and then the agent copies the trajectory that the example agent followed.

Lee's approach saved the data examples in the form of state-action, something seen in Machine learning algorithms as well. The examples were stored in a spatial data structure. During the simulation, similarly with Lerner's, a query having the current state of the agent was submitted, retrieving the closest matches and their actions. These actions were applied to the agent.

Recently, many methods relying on Neural Networks and Deep Learning have been introduced to navigate virtual agents, robots and self-driving cars in the real world. Even though working robots and cars have different priorities, the techniques used are very interesting and worth taking into consideration.

Long used deep learning [8] to train a policy based on carefully designed process for data creation. The process was using ORCA [9], a reciprocal velocity obstacles approach for real-time multi-agent navigation for their data creation. Long followed up with Deep Reinforcement learning approach [10], were a multi-scenario multi-stage training framework was introduced for optimizing a collision avoidance policy.

Chapter 3

Technical Review

Contents

3.1 Tools

This section provides information about the tools used. Each tool will have their own description that gives info on their capabilities, why these capabilities are useful approaching the certain research domain and reasons that they were chosen instead of their counterparts if any. Reasoning will have the criteria that the developer took into consideration in order to choose the correct tool. More often than not, the counterparts are very good solutions as well.

3.1.1 Unity Game Engine

Unity game engine is a free game engine provided by Unity Technologies. It provides both 2D and 3D capabilities for game development. Furthermore, it uses scripts that are written in JavaScript or C#, both providing the same capabilities except for occasions of language-specific libraries imported. Moreover, it is in constant development, integrating emerging technologies in no time.

3.1.1.1 Tutorials

Tutorials are commonly used among developers, assisting them to get acquainted with a specific software. Unity does very well at creating tutorials. With every big version announced, they follow up shortly with a tutorial demonstrating how a new feature is supposed to be used. Furthermore, beginners or experienced in game development,

tutorials make Unity easy to get used to and understand how it approaches the game development area.

3.1.1.2 Community

Unity engine was the first free game engine with such impressive capabilities. This played a detrimental role in its rise in popularity among game developers. As a result, Unity comes with a big community behind it. A big community adds a lot to a software in terms of usability and support.

From the research standpoint, all needs required are satisfied. Unity provides efficient rendering, Unity game engine satisfies the needs of the research area, as it provides anything simulation related.

3.1.1.3 C Sharp or Javascript

An important decision to be made was which one of the languages should be used. The decision was based on 3 criteria. First one was comfort with each language, second one the community behind each language and last but not least the future versions of Unity that bring massive changes. Comfort indicated C Sharp as the preferred language because the developer is more acquainted with object-oriented programming languages. The community seemed to be in JavaScript side even though most of it is in front-end development industry, not that helpful in terms of game development. Lastly, Unity Technologies announced in 2017.1 version of Unity that Javascript is getting in a deprecation process. Taking these into consideration the language choice was C Sharp.

3.1.2 NNSharp

NNSharp is a C# library providing the capability of running pre-trained Keras models without being dependent to python during runtime. The library is able to run most of the common layer choices provided in Keras. Its development started in April 2017 and it is open-source. It was supposed to finish with Keras and then continue with

optimization using GPUs. Unfortunately, development seems to be inactive as of November 2017.

3.1.3 Python

The machine learning approach used in this work needed a programming language that can provide everything needed, avoiding any limitations that may occur. Python is an interpreted, object-oriented and high-level language. Furthermore, python is simple with an easy syntax. As a result, it is easy to learn, especially if one has previous programming experience.

Python may be an easy language to use, but its true strength is in its vast library support. Most things one may think will probably have a library. Especially in machine learning domain, there are plenty of libraries to integrate into a project. Examples of Tensorflow, Keras, Theano(Deep Learning) and SciKit-Learn are the first that come to mind. It's worth noting the big community behind python and each individual library community and support as well.

3.1.3.1 Keras

Keras [11] is a high-level neural networks API, written in Python with the capability of running on top of TensorFlow, CNTK, or Theano. It was developed focusing on fast experimentation, avoiding all the Neural network specific needed. User friendliness modularity and extensibility enables easy and fast prototyping. Furthermore, it makes use of Nvidia GPUs, providing faster training times. Moreover, the library is open-source and actively developed. On its Git-hub page there bugs are reported through the issue tracker and fixed constantly.

3.1.3.2 SciKit-Learn

SciKit-Learn [12] is a free machine learning library for Python. It provides simple and efficient tools for data mining and data analysis. It supplies algorithms for classification, regression, clustering, dimensionality reduction, model selection and

preprocessing. It is actively developed for ten years and has built a huge community around it. Scikit-Learn is able to scale very well using the spark framework, but during this project, it is not a feature that can be exploited. A downside of Scikit-Learn is that it doesn't make use of GPUs.

3.1.3.3 Conclusion

The library that was chosen is Keras. The reason for this choice was that the approach taken towards Virtual crowds would try to use Neural networks and Deep Learning. Even though Scikit-Learn offers far more algorithmic solutions in a lot of different areas, it doesn't offer as much as Keras for the scientific approach followed. On the other hand, Scikit-Learn may be used in order to use some of the pre-processing and dimensionality reduction algorithms provided.

3.1.4 Anaconda

Anaconda is a free and open source distribution of the Python programming language for data science and machine learning related applications, offering simple package management and deployment. It provides fast and easy library dependency handling, the creation of more than one virtual environments enabling developers to use variable setups to work with depending on their goal. For instance, each environment may use a different back-end for Keras like Theano or Tensorflow[**tensorflow2015-whitepaper**] mentioned earlier.

One of the reasons that Anaconda was used is that python libraries based on C, that normally can't work on Windows are available through Anaconda. Adding the compiler needed and Blas library is not child's play and it is time-consuming. Without Anaconda, one would need to work on both Linux and Windows, Linux for training Neural Networks and Windows for building the simulation engine. Afterward, to test the results it is necessary to have socket communication or open a Unity build of the simulation engine in Linux using Wine.

3.2 Unity (C#)-Python Communication

Working with Unity demands interprocess communication with Python. The communication was needed in order to propagate the states of all the agents in the simulation from Unity to Python. Python receives the states, creates the array that has the states of each agent in a row. Then it inputs the array to the Neural network and receives the output. The output is then propagated to Unity so agents update their action. In the following subsections there is information about the form of the data exchanged, the data types considered and the communication ways that the developer worked with and which one worked the best.

3.2.1 State data form

In order to establish a stable communication between the two processes, a form of data was needed. Two protocols were created, because of the usage of two different data types used, float and string. The protocol concerning strings had agent states separated by a semicolon, and each states single value separated by a comma. The same way of communication was used in order to send and receive values. The protocol used for exchanging float values had as first value the number of all the state features together followed by each feature's byte form. Between the two protocols exchanging float values was more efficient, due to excessive memory waste of mismanaged strings.

3.2.2 Inter-process communication

There is a limited number of ways to communicate between two different processes that use different languages. In order to find the more efficient way, the developer tried the following approaches: Sockets, Process spawning inside C# and Pipes.

3.2.2.1 Sockets

A socket is one endpoint of a two-way communication link between two programs running on the network. A socket is bound to a port number so that the transport protocol layer can identify the application that data is destined to be sent to. Sockets were the first communication implemented as it's the most common way for inter-process communication. Integrating sockets as a way of communication was easy. For string protocol mentioned using a certain encoding schema as ASCII or UTF8 was enough and for float protocol getting the byte representation worked as well. Even though integrating was easy, the need for fast communication wasn't satisfied. As a result actions for each state were late making the agents seem "slow". Sluggish agents made the evaluation of Neural network approaches hard to evaluate due to misleading actions taken.

3.2.2.2 Process Spawning

Process spawning is the procedure where you initiate a new process from an existing one. In order to make this work in C#, one has to open a python process with the .py file needed an argument. Communication between the two processes is possible by redirecting the standard input-output of the python process. After that, it is possible to send the states from unity through the standard input and receive the actions through the standard output. The process spawning approach satisfied the fast communication needs. Using this approach made the evaluation of the Neural networks possible as doubts about actions being in time were eliminated.

Even though it provided fast communication, there were some problems encountered. Spawning processes did not work the same way on every computer, resulting in the same project not working on different computers. Python process from C# on different computers with the same Anaconda configuration yields exceptions related to library dependencies(Keras). It wasn't possible to find what was the problem, speculating to be different versions of Windows(Worked on Windows 10, didn't on Windows 7). Moreover, another downside of process spawning is that you can't have the process load all the neural network models needed beforehand resulting in slowing

down the work-flow.

3.2.2.3 Named Pipes

Named pipes is an extension of the traditional pipe concept found in Unix and other Unix-like systems as Linux. Pipes work in a FIFO manner and they may be unidirectional or bidirectional. The usage of pipes come with some limitations. Pipes are not part of the current official version of .Net used in Unity. Recently Unity gives the option to use .Net 4.6 that is still in experimental stage with all the dangers that this may bring. Integrating named pipes needed to create two unidirectional pipes. Choosing unidirectional pipes provides more reliable communication. From python side Win32 library was used, providing file-like usage of the pipe. Furthermore one pipe was created in C# and one in Python. This way it was possible to make python wait Unity to create it's pipe first and have them synchronize without much hassle. Important note for using pipes is that after reading the input one may need to bring file pointer at the start of the file.

Named pipes usage resulted to faster communication of all approaches used. It is more difficult to use than the other approaches because of the big difference between the libraries used. C# library had different functionality and provided different capabilities than the Win32 library in Python. Furthermore, resources on how to use the pipes in each language were hard to find. In spite of its difficulties, the time needed is well worth it. Faster than any other approach used, can run and initialize before the simulation is on providing faster work-flow and is also portable to use on different setups.

3.2.2.4 Conclusion

As it is already mentioned named pipes were the best approach used. Named pipes and process spawning where integrated into a later part of this work, when feature states started to grow bigger. That was when communication overhead was taken more seriously. Having that in mind, it is worth noting that the socket

approach may have been incorrectly used, resulting in inefficient communication due to inexperience.

Chapter 4

Methodology

Contents

This section provides information on the work-flow followed during this work and the crucial parts of it. First, let's see a summary introducing the steps followed.

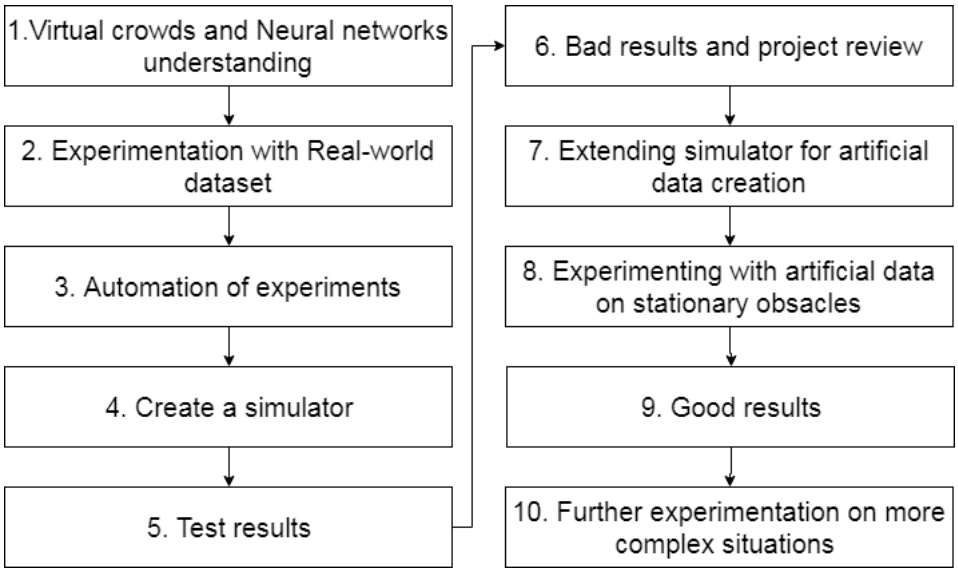


Figure 4.1: Overview of the project.

The initial step was to get a general understanding of the research domain working on and acquire knowledge related to neural networks as it was the approach followed in this work. After getting a decent understanding of the topic and on neural networks, starting some experimentation with a real-world dataset was the following step. During working on this dataset, learning Python, libraries like Keras, experimenting with plotting data in various ways, clustering data, and more others common techniques was the immediate step. Time efficiency was of importance so after getting acquainted with the workflow, automation of training neural networks in order to find the best neural network architecture was next. Results seemed acceptable, so we moved forward and tried creating a simple simulation that is going to use the best trained Keras models. Seeing the first real results, brought to the developer's understanding that common ways of evaluating the training error may not be enough to base result quality on.

Bad results deemed necessary to think of another approach that may help using real-world data efficiently. The idea of creating artificial data came up. Using Unity, a simulation engine that could be used to create data as well as simulate trained neural networks was created. The simulator engine offered a way to create data using a various set of features. Creating artificial data proved to be a good idea, some interesting results started to come up. More capabilities were added to the simulator and further experimentation was next.

That was a small summary of how this work was produced. More details on the individual parts of this work will be given in the following subsections.

4.1 Project assumptions and decisions

This subsection is going to report the assumptions made working on this project. These assumptions are made in order to be able to use usual terms like **meters** in the text, aiming for easier comprehension of the text. Many of the assumptions set are unrealistic, but this was not the priority of the project, as exploring obstacle avoidance capabilities was not influenced.

Unity has it's own world space, defining transforms that are used to save object

position, rotation, and scale in the 3D world. Transformations are manipulated in the 3D world with X, Y, Z axes. Values of each axis have continuous values. As an assumption, a change equal to one on a certain axis is equal to a **one meter change** in the real world. Furthermore, agents are represented by capsules with a diameter of one meter.

A decision about velocity was made. Most common approaches on virtual crowds use the velocity vector. The velocity vector stores information of velocity on x,y,z axes. Instead of using the velocity vector, a speed with maximum and minimum value limit was used and for steering, the agent rotates. One more detail to take into consideration is the speed limits considered. The maximum limit was set to $4\,\mathrm{m/s}$ and minimum $0\,\mathrm{m/s}$. User-driven agents would accelerate to a max speed and then keep that speed until a situation of slowing down occurs. This is not a realistic scenario, because different agents may have a different maximum speed and people generally seem to keep a certain pace from which each one can accelerate/decelerate to a certain maximum/minimum speed.

4.2 Neural Networks

Neural networks were the approach used in order to teach virtual agents obstacle avoidance behaviors. Even though neural networks were an integral part of this project, a small fraction of possible layer types, architectures, optimizers and activation functions were used, because the aforementioned parts of this project were prioritized.

4.2.1 Layer types

4.2.1.1 Dense - Fully connected

The Dense layer as referred in Keras or most commonly fully connected layer is the most common layer type used in neural networks. A fully connected layer consists of a variable number of nodes and each individual node is connected with all the nodes of the previous layer.

This layer type was mostly used in this project and is the main layer used in the best

neural network models trained shown in 5.2. The main factors for using this layer was the performance limitations for a real-time application and it's simplicity. Moreover, due to the vast number of variables in this project and the small amount of data, the decision to keep this layer as the standard for experimenting was made.

4.2.1.2 LSTM

Lstm [13] is a special kind of recurrent neural network and currently one of the most popular. Recurrent neural networks try to connect previous information to the present task, in this case predicting the best action to be taken from the agent for a collision-free navigation in the environment. Lstm networks are capable of learning long-term dependencies as they were explicitly designed for this purpose.

As mentioned in 4.3 we keep information about the past in our data we need to use this information for better predictions. Lstm was used at a time of uncertainty during this work, where most results were negative. The decision to stop experimenting with Lstm was made, due to insufficient understanding of the problem and the layer, low amount of data and not even goal following results. In future work, lstm will be exploited thoroughly, as they seem to be one of the best bets, because of the importance of past and future in the problem of obstacle avoidance.

4.2.1.3 Dropout

Dropout [14] layer is one of the most popular approaches to regularization today. Research shows that using dropout layers as regularization method in neural networks better generalization is accomplished. Dropout layers work simply by deactivating a fraction of the nodes on a particular layer during training time, opting to trade training loss for generalization. The fraction of nodes that are going to be deactivated is variable and need to be fine-tuned at a point that over-fitting is avoided.

4.2.1.4 Batch Normalization

Batch Normalization [15] is a layer provided in keras that can be used in between other layers normalizing the activations of the previous layer. It normalizes the activations of the previous layer at each batch by applying a transformation that maintains the mean activation close to 0 and the activation standard deviation close to 1. In this project, this layer was used to normalize the input data.

4.2.2 Activation functions

In neural networks, the activation function of a node defines the output of that node given an input or set of inputs. There are many different activation functions, each used more commonly with different layer types and losses. During this project, the following activation function was used, without any of them being eliminated from the experimenting process. Each one may produce better results with different architectures and losses. The activation functions are Relu, Sigmoid, and Softmax. In this project, relu is used only on hidden layers, sigmoid on both hidden and output layers and softmax on output layer only. More specifically when training classifiers softmax was used mostly with few exceptions and for regressors sigmoid.

4.2.3 Optimizers

Optimizers are optimization algorithms that are used in order to minimize the error of the neural network during training. Only two algorithms were considered most of the time, Adam and Stochastic gradient descent(SGD) [16]. Adam [17] algorithm is an extension of SGD and popular at the time of this work. In our case, Adam was the way to go. The decision was result driven, as SGD could not handle the complexity of the data as well as Adam algorithm. As with other parts of this project, SGD stopped being part of the experiments after constantly producing sub-par results relative to the results produced using Adam.

4.2.4 Loss functions

A loss function is a function that is responsible for computing the error of the network. Different errors will have a considerable effect on the network's performance. There are loss functions for regression, classification, and embedding. The first type is used to train networks for continuous value prediction, second for category classification as predicting discrete actions for an individual and third type deals with problems that measure inputs being similar or dissimilar.

In this project, regression and classification losses were taken into consideration. For regression mean squared error(MSE) was used and for classification binary cross-entropy and categorical cross-entropy was used. In some cases, MSE was used for categorical data as well. It behaved better with the noisy artificial data.

4.3 State inputs

State inputs are the individual parts of a full input state used for the neural networks. Trying different state inputs and combining many of them together was something that needed a lot of time to work with. It's difficult to state how important was to find good state inputs that will produce decent results. Even if a state input idea seems great on paper, results can be far from what one expected. Something logical for humans, will not work the same on learning algorithms like neural networks. Furthermore, some architectures may be incapable of learning the complexities existing in the data. Other architectures may learn too much, providing very good training and validation error, but look illogical during simulations.

One more detail that has to be specified is that for each state input, a variable number of time-steps is saved as well. Time-step is the previous state inputs calculated during the past frame. The motivation behind time-steps was to be able to give more information to the neural network, for it to extract valuable details that will lead to better results. Furthermore, each state input can avoid using all time-steps saved and instead skip a variable number of them.

The state inputs that were used in different occasions, successful or not, will be stated

below. Information on what was the reason to use this state, pitfalls that each state input may have as well as a summary and details of the implementation.

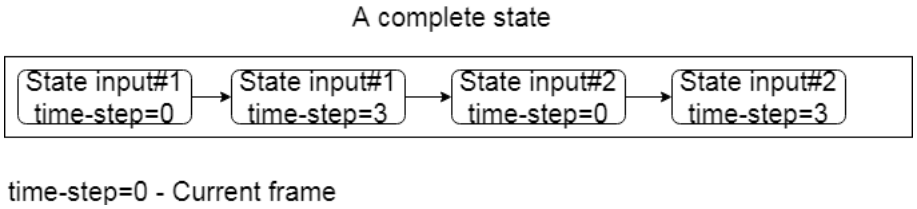
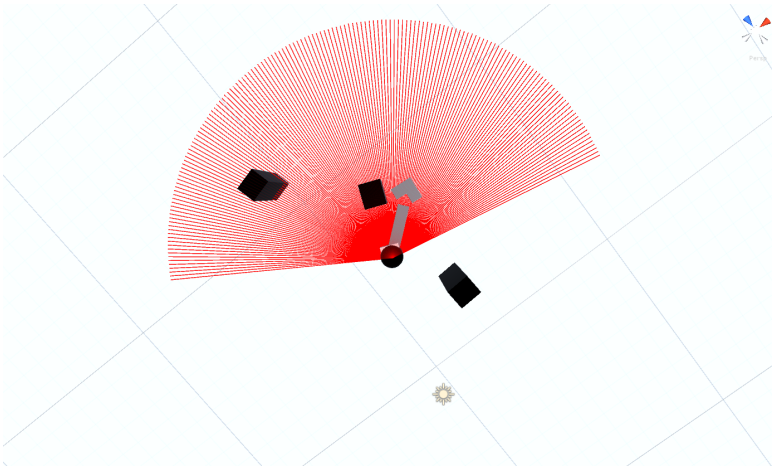
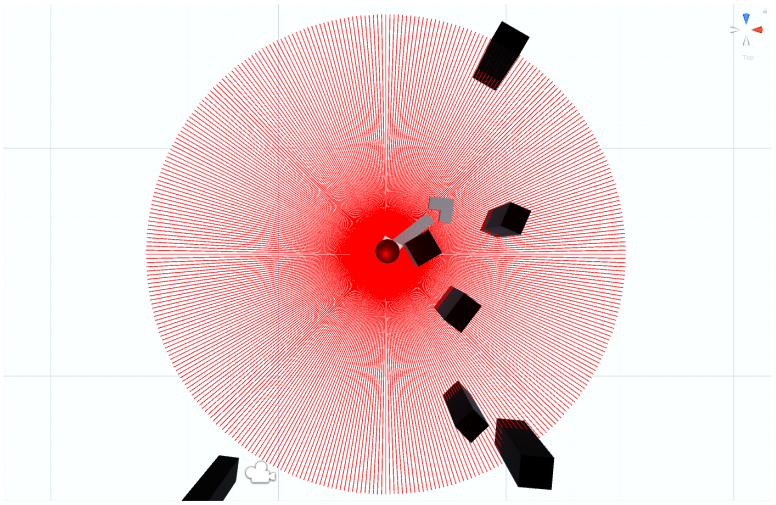


Figure 4.2: A complete state representation.

4.3.1 State input #1



(a) Rays casted limited.



(b) Rays casted all around.

Figure 4.3: Demnostration of rays casted.

Summary: This state input cast rays around the agent looking for collisions. It is used in order to have info about the agent's surroundings and using it to learn obstacle avoidance.

Details: Rays are cast around the agent in a symmetrical manner. The rays are *oriented* towards the agent's *front*. So ray count left and right of the agent is always same. When a collision is found, the angle where the ray was cast indicates the index, and the collision's distance the value. Information is saved in an array of float values. The array resets every new frame, setting all values to a constant *maximum distance* predefined constant value. For flexibility, this state input was coded in a way that enables grouping nearby angles together. Group size can be of any value. Having big group size, reduce the features of this input and a small group size, increases the features.

When using a group of rays, the ray with the minimum collision distance is the one used. Furthermore, rays were cast from left to right. For example if one wants to cast only on 160 degrees (4.8(a)) first ray will be cast at -80 and last at 79.

Observations: This state was used successfully as part of a combination of state inputs, used to train a policy for stationary obstacle 5.1 avoidance and goal following. This state alone did not yield good results on its own.

4.3.2 State input #2

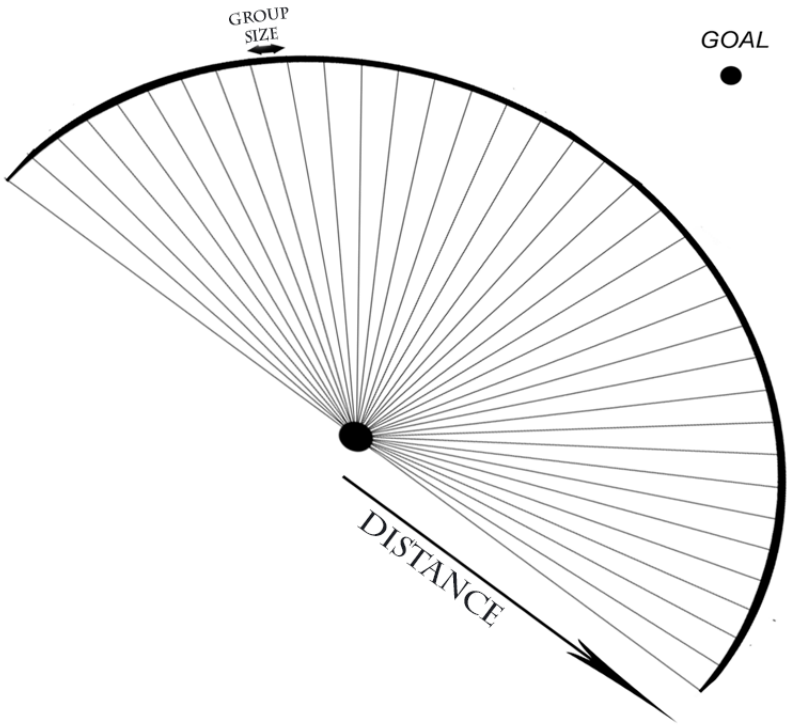


Figure 4.4: A representation of the rotated state.

Summary: This state input cast rays around the agent looking for collisions in a similar manner with 4.3.1. It is used together with 4.3.1. The idea behind using this was to decouple the complexity of the data, making it easier for the neural network to learn.

Details: Rays are cast around the agent in a symmetrical manner. The rays are *oriented* towards the agent's *goal*! Except for this detail, the way collisions are handled are the same with 4.3.1.

Instead of casting new rays and making the simulation heavier, the current implementation uses the rays cast from 4.3.1. For this implementation to succeed, the angle that each ray was cast at has to be adapted. If rays were not cast at a certain angle of the rotated state, this angle takes the default distance value predefined.

Observations: This state was used successfully as part of a combination of state inputs, used to train a policy for stationary obstacle avoidance and goal following.

4.3.3 State input #3

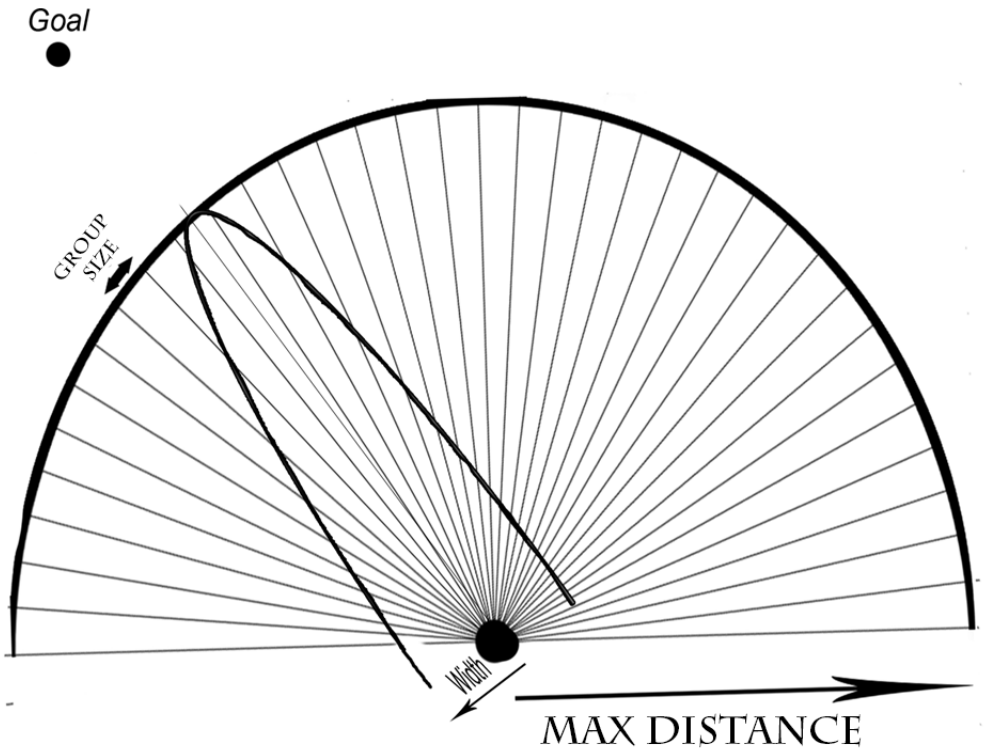


Figure 4.5: A representation of the ellipse state.

Summary: This state indicates whether a collision was found between the agent and the goal. The state used for this one seems unusual but produced good results. The idea behind using this was to decouple the complexity of the data, making it easier to the neural network to learn.

Details: Rays are cast around the agent the same way with 4.3.1. This time the features are binary. It returns ones for collision found in the ellipse shown above, and zeros for collisions outside the ellipse or for no collision at all.

Instead of casting new rays and making the simulation heavier, the current implementation uses the rays cast from 4.3.1. For this implementation to succeed, the collisions should also be passed from an ellipse equation showing whether the collision is inside the ellipse or outside it. Maximum distance, width and ray group size are constant values set before running the simulation.

$$f(x,y) = \begin{cases} 1 & \text{if} \quad \frac{x^2}{W^2} + \frac{y^2}{D^2} \leq 1 \\ 0 & \text{otherwise.} \end{cases} . \tag{1}$$

Observations: This state was used successfully as part of a combination of state inputs, used to train a policy for stationary obstacle avoidance and goal following. Taking this state input out of the combination, returned worse results. The agent would still avoid the obstacles, but it was not as smooth as with this state.

4.3.4 State input #4

Summary: This state input cast rays around the agent looking for collisions. It is used in order to have info about the agent's surroundings and using it to learn obstacle avoidance.

Details: Rays are cast around the agent the same way with 4.3.1. This time the features are binary. It returns ones for collision found in a certain distance from the agent. The distance is defined before the execution. This state can be used multiple times having different distance definition each time.

Once more, instead of casting new rays the current implementation uses the rays cast from 4.3.1.

This state is used in the moving agent neural networks created demonstrated in 5.2. It was used in combination with 4.3.6, because of the low detail used in most cases. Low detail resulted at times in bad behaviors because when other agents are very close to the agent they appear to be the same. This results in unorthodox behaviors, with the agent moving towards the occupied area. Using this state does not yet seem to be effective and in the future, it may be stripped of the project.

4.3.5 State input #5

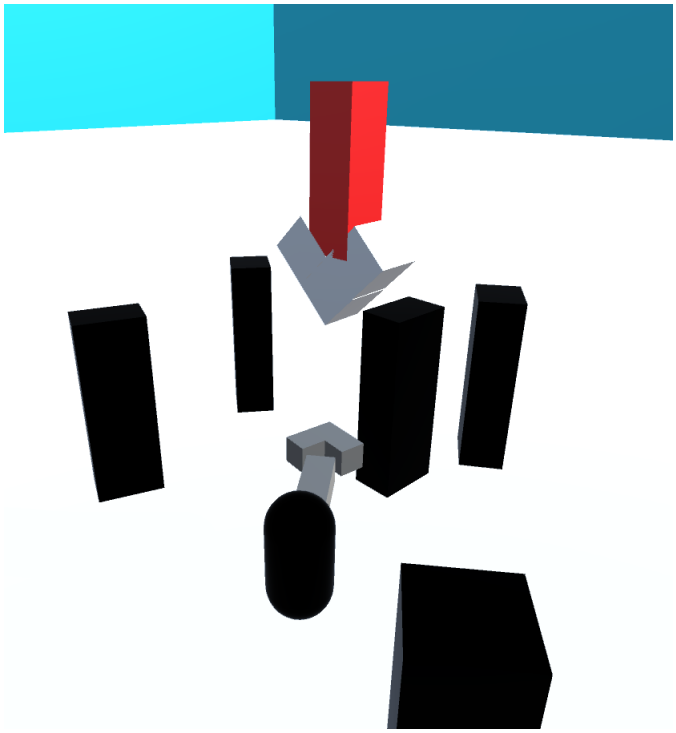


Figure 4.6: How angle to goal is visible during simulation

Summary: This state input indicates the angle where the goal is relative to the agent. This input is used to enable the policy that is going to be learned, to successfully follow the goal.

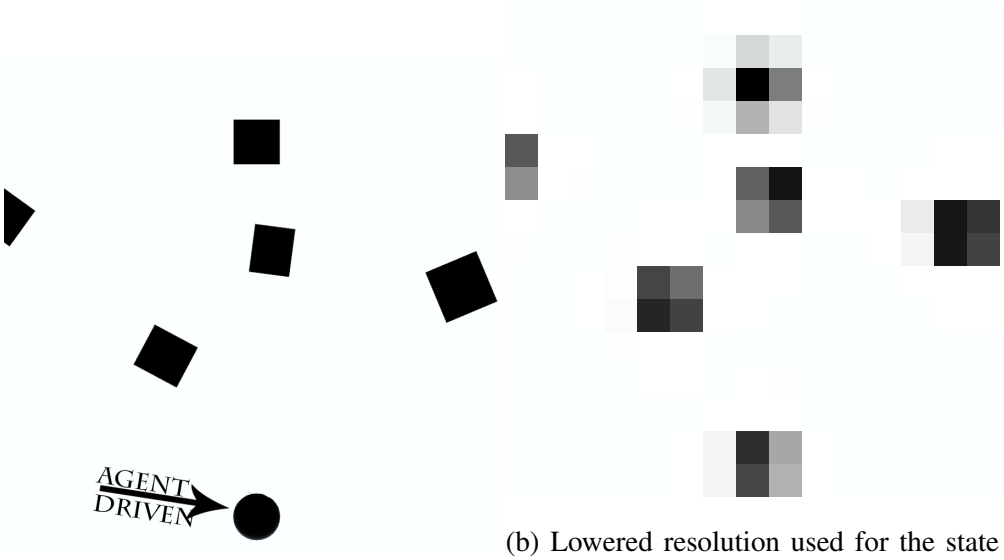
Details: For calculating the angle the following equation was followed.

$$angle = \text{atan2}(x, y) * (180/\pi)$$

Observations: On most of the failed policies, this feature was the only one that it's intention was understood from the neural network. The outputted policy would result in good goal following but, inadequate obstacle avoidance. This was fixed with balancing the data, adding more situations of obstacle avoiding examples. Furthermore, when using this feature through time, a small time interval between each time-step produced bad results. This probably happened because turning in order to avoid a collision was associated with the change of this value through time, rather than the

environment state.

4.3.6 State input #6



(a) Normal resolution.

(b) Lowered resolution used for the state input.

Figure 4.7: A representation of the state.

Summary: This state input renders the top orthographic view of the area the agent moves. It is used in order to have info about the agent's surroundings and using it to learn obstacle avoidance.

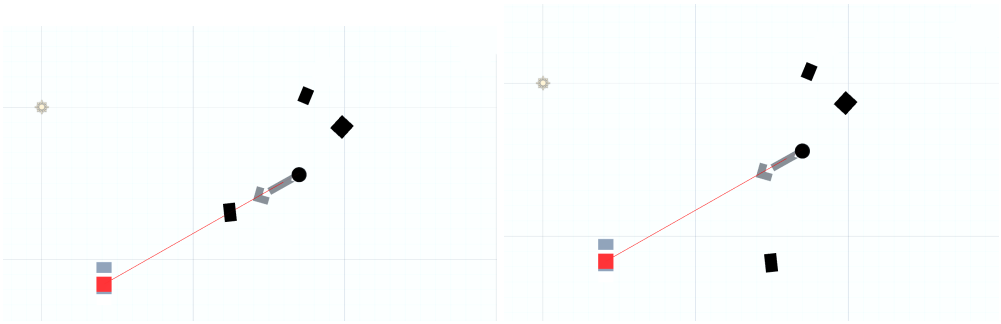
Details: The top view is taken from a camera placed above the agent. The camera field of view may be modified for different experiments. Furthermore, the resolution of the image can vary, depending on the detail needed. It is important to note that adding more detail increase the features of this state input exponentially.

This was implemented using the orthographic camera provided in Unity. This is not the best approach, but it was the faster to implement at the time. In order to get the top view, each time the camera has to render it's view and then take the render-texture that the view was rendered on and take all the pixels. Then the gray-scale values of each pixel are taken and after filtering them we get binary value.

Observations: This state was the best state input used for policies dealing with moving obstacles. Increasing the detail of the picture comes with the need for more

data to get good results. During experiments, pictures with 10x10 pixels to 30x30 were used.

4.3.7 State input #7



(a) Goal not directly visible. (b) Goal directly visible.

Figure 4.8: A representation of the visible goal state.

Summary: This state input is just indicating whether the goal is directly visible. The motivation to use this simple feature was to see whether is possible to introduce different behavior to the agent when the goal is directly visible and when it is not.

Details: A ray is cast towards the goal. The single feature gets binary values. One for not visible goal, zero for visible.

Observations: It was interesting to see that it is possible to model different behaviors just by using one more simple feature. If you think of it, this is similar to teaching the neural network to behave differently if a condition is true or false. You may set the conditions, but the actions to each condition are going to be learned through neural networks.

4.3.8 State input #8

Summary: This state input provides info for the distance from the goal. The motivation behind using this feature is because people tend to act differently when they are near or far from the goal.

Details: Euclidean distance from between the agent and the goal is calculated.

$$distance = \sqrt{(x2-x1)^2+(y2-y1)^2}$$

Observations: Using distance to goal state was a difficult state to use. There were many times that distance to the goal would ruin the results. No clear conclusion was made though about what was the problem. Except for bad results, distance to goal provided an unexpected positive result. While working on stationary obstacle avoidance 5.1, distance to goal helped the network understand on some occasions that the goal is behind a wall. This was a result that was made possible with the combination of the states used in 5.1.

4.4 Action outputs

Action outputs are the actions taken for every training example given. Prior working on this topic, it was difficult to understand how much difference can the outputs make on the policy learned. Some outputs may have lackluster generalization. Generalization that makes one think whether there is something wrong with something he tested a dozen of times. Some outputs resulted on really noisy actions but following the goal fine, other outputs would move clean, avoid collisions, but still do dumb decisions in terms of following the goal. Details on the most important output states will be given below, with some of the advantages and disadvantages observed.

Before going on each action output an important detail has to be stated. In the following subsections there will be a differentiation on turning left and right. A common approach would use one continuous rotation value. The problem with this approach happened a lot of times during experimentation. Having two similar situations that have exactly opposite actions would cancel them out when using a regression algorithm. This situation changes when you strip the rotation rate prediction from turning left or right prediction or when the actions are discretized.

4.4.1 Action output #1

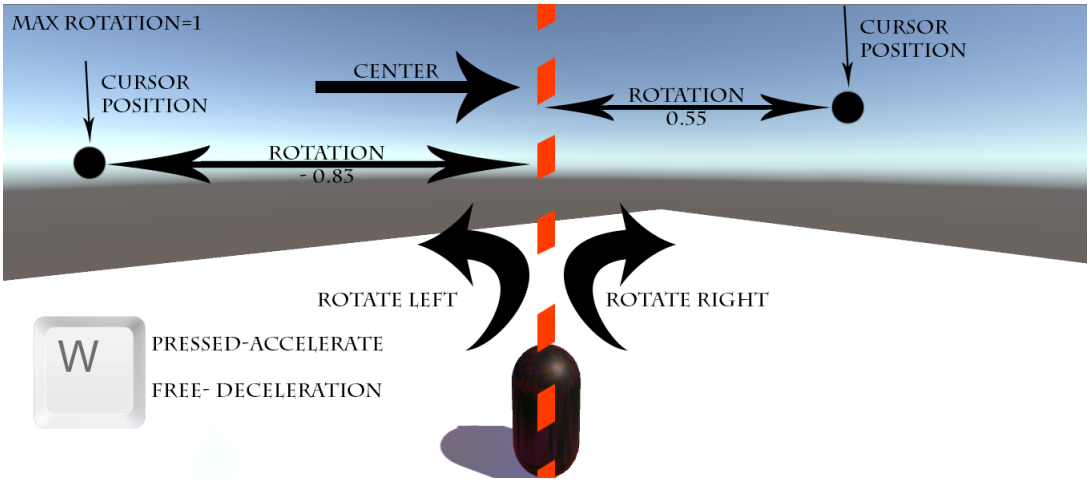


Figure 4.9: Reference figure for better understanding of the output mechanics.

This action output is saved in the data example created in two values. One value is a continuous regarding rotation and takes values $-1 \leq rotation \leq 1$. The other value is binary and is related to whether the agent accelerates or decelerates.

As seen in 4.9 rotation value is taken from the mouse x position in the screen, relative to the center of the screen. It is important to understand that the value is not based on mouse acceleration. The absolute rotation rate increases linearly while moving the mouse towards the sides. Opposite action yields the opposite results.

Acceleration value is taken from pressing certain button binded for this action, in this case, W. One may argue that people do not accelerate or accelerate with constant acceleration and they may keep a certain speed. The answer to this, is first, the simulation used to create artificial data is built around agent optimal speed, that when this speed is achieved, he stops accelerating and keeps that speed. Second, even though user action during data creation is binary, neural network results are still continuous. Depending on the loss function, using the probability given without thresholding it gives unexpectedly good results.

This action output form comes with some advantages and disadvantages. Starting with the good characteristics. The agent can move delicately around stationary obstacles giving a nice feel. Of course, this nice feel is occasionally lost due to

noisy output from the network. Noisy output is a problem, but not as big as problems on generalization. Training with data from similar situations but with different action outputs resulted in a vast difference in generalization.

4.4.2 Action output #2

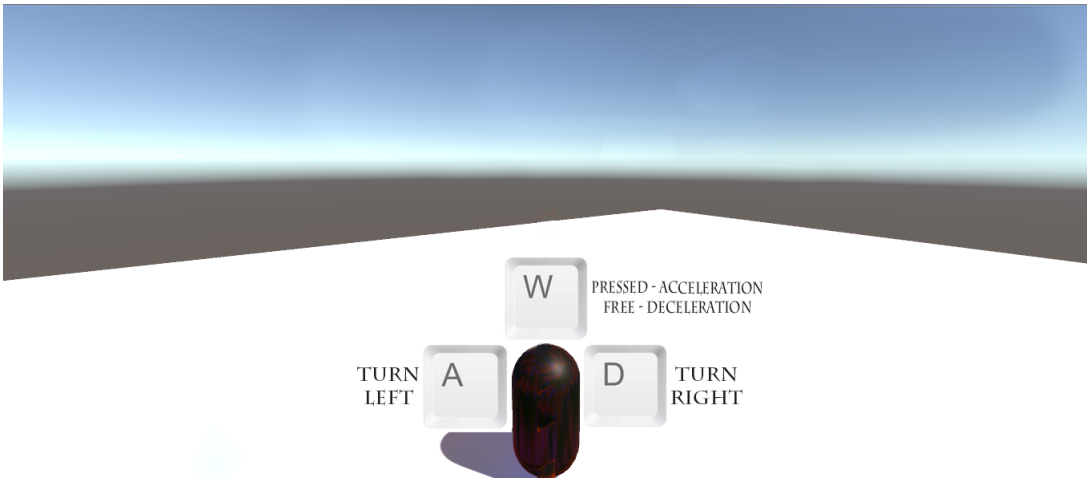


Figure 4.10: Reference figure for better understanding of the output mechanics.

This action output is saved in the data example created with 3 binary values. A value for turning left or not, value for turning right and a value for acceleration or deceleration.

As seen in 4.10 all the actions are taken from a certain key binding that is defined in the simulators code. Rotation related bindings are A and D keys, and only one of the two can be used at a time. It is important to note that the amount of rotation applied when pressing the key binding is constant. This approach does not make as much sense as the previous one but there is a reasoning behind it. This idea seemed very interesting for generalization impact. Turning at a constant rate would provide similar differences between input states through time resulting in easier to learn data.

Acceleration value is taken from pressing certain button binded for this action, in this case, W. One may argue that people do not accelerate or accelerate with constant acceleration and they may keep a certain speed. The answer to this, is first, the simulation used to create artificial data is built around agent optimal speed, that

when this speed is achieved, he stops accelerating and keeps that speed. Second, even though user action during data creation is binary, neural network results are still continuous. Depending on the loss function, using the probability given without thresholding it gives unexpectedly good results.

This action output form comes with some advantages and disadvantages. Starting with the good characteristics. The agent avoids other moving agents or obstacles quite effectively while following the goal. Movement, as expected, does not look nice in the eye, but at least it is not noisy. Another drawback is the goal following. Even though the agent reaches the goal, it does so looking unnatural. Last but not least, this action state provided better generalization, which was the motivation behind.

4.5 Artificial data

During this work, the need for artificial data came up. Artificial data are data created by the software created. This data may be generated by AI agents or user-driven agents like games. To give a better understanding, one may build a scenario of multiple moving agents. Each agent has its own goal that he follows. The agent may have complex AI enabling it to avoid imminent collisions or just move towards the goal of ignoring everything. In case of data generated by AI agents, all agents will generate an input-output example, predefined from the software used. In case of data generated from a user-driven agent, usually, only the user-controlled agent will provide an input-output example.

4.6 Real world datasets

Real-world datasets are datasets that were created with motion tracking devices. Usually, people creating datasets from real-world examples set tracking devices at places where constant crowd movement is evident. Tracking devices output each individual's position in the area in a frame by frame manner. The limit of time in between frames is determined from the tracking device capabilities. When one receives the tracking device output, it's possible to simulate the crowd behavior in a simulation

engine. The ability to replay the crowd behavior enables a developer to create input-output examples for every frame recorded with the tracking device. A simulator engine that can create artificial data should be able to have some modifications to follow real-world tracked agents.

4.7 Artificial data creation

This work revolved around the idea of creating artificial data. As mentioned in the summary, the first approach used was to use real-world preprocessed data in order to train neural networks and simulate the results. Failing repeatedly using this approach, the need to understand better the training data and their impact emerged. By creating artificial data, a lot of different states and outputs-actions could be tested. Furthermore, different environment setting could be used to create data. Different settings provide a further understanding of what data can produce a good obstacle avoidance policy.

Throughout the course of this project, only the user-driven method of creating data was exploited. Even though the amount of data that can be created is far less than the data created with multiple AI agents in the scene, it was more interesting for us to see if the model will be able to learn user-driven data. If you think about it, data created using an AI algorithm will always follow certain rules resulting in the same actions under the same situations. But does this actually happens with people? More often than not, one will act differently under the same situation, providing a diverse range of actions. Testing state combinations and output forms with user-driven data may be able to provide better results, as this approach takes into consideration the human factor. Think of this approach as a way of prototyping input-outputs and neural network architectures, aiming to apply them to real-world generated datasets.

Creating artificial data, especially user-driven was time-consuming. The need for more data examples brought data augmentation on the table. Data augmentation is the process where the state and the output in a data example are mirrored. Of course, using data augmentation, we assume that in the mirrored environment the user would indeed take the mirrored decision as well, even though we know that in reality people

actions are related to their dominant leg, arms etc. Creating data faster was a priority and seeing that it did not hinder the results in any way, data augmentation was used. This should be taken with a grain of salt though, as the experiments were not oriented towards observing changes due to data augmentation.

4.8 Environment setting

Environment setting is the environment geometry, agents, and obstacles that populate the simulation during the creation of artificial data and testing trained policies. Creating good environment settings turned out to be detrimental for good results. Environment setting ideas came mostly from survival like games, where the player does the same thing over and over until he loses, or just quits.

The first environment setting used was an environment having only agents that follow their goals without avoiding any imminent collisions. The user controlling one of the agents generated data by following a goal while avoiding imminent collisions. The goals were generated in a fixed size square in order to keep agents near each other. Furthermore, goals had to be generated in a certain distance from the agent. When agents were at a distance lower than a threshold set, the goal relocates to a new position. Distance was set in the environment variables, one indicating minimum distance and another one for maximum distance.

This setting being the first one tested didn't really yield any positive results. At the time of testing this environment and failing, attention was on the state input part of the data. Thinking that the environment wasn't much of a problem, a version similar to it was used afterward. Failing repeatedly to get good results with moving agents, lowering the bar and trying stationary obstacles was next. The environment was modified. Instead of moving obstacles, a random environment was generated in the fixed size square and again the user had to navigate avoiding the stationary obstacles.

After failing again with the combination of the environment setting and the input states used, the decision to make a clean start was made, to simplify everything. Simplifying the environment was achieved by stripping everything related to the goals.

Furthermore, the environment was not randomly created at the start. Instead, walls were positioned around the area. These changes served the goal of creating a neural network that can act and avoid collisions only, without goal following. Last change was the dynamic spawning of stationary obstacles in front of the agent at a constant rate. Obstacles where spawned in a range of values, so obstacles can at any distance from the agent. This would help create data from a lot of similar situations. Sometimes it is not enough to just steer, it also needs to slow down. Learning that was the first step that something worked, even though simple. The previous environment setting was creating more complex data to be used with the learning algorithms and the data variety generated was poor.

These first results highlighted the importance of the environment and how it affects data quality. In the next step following the obstacle avoidance neural network, was to incorporate the goal again in the data generated. In the new environment setting goal came back the same way they worked before. Dynamically generated stationary obstacles were still used, but spawning more obstacles. Now an obstacle is spawned always in front of the agent, and a random number of obstacles(bounded) where spawned around the agent. Moreover, obstacles were spawned at a constant time-interval and where destroyed after 10 seconds. This setting helped to generate a quality dataset in a short amount of time(10 minutes). The dataset generated combined with certain input states provided a very impressive policy for goal following and obstacle avoidance.

Further experimentation with the environment setting was building scenarios. Each scenario is one environment setting, that changed through time during the simulation. Instead of using a single environment, more now could be used. For instance, one series of scenarios could start with a simple goal following scenario, while obstacles did not spawn. The next scenario would provide more complexity in the environment. The following scenarios follow in the same manner. At this stage, moving obstacles are back in the environments. Scenarios now may now imitate real-world settings. For example, moving obstacles will move parallel to the user agent, with the same direction, opposite direction or both. In addition, scenarios were added where moving obstacles move perpendicular to the user's agent from only one or both sides.

Except for multiple scenarios programmed, different behaviors for the moving obstacles were created as well. The need for more interesting data deemed necessary to add minor characteristics to the obstacles. The characteristics are not something special. Behaviors added are slowing down, steering, slowing down and steering and having different speed ranging between zero and Max speed threshold. More behaviors can, of course, be programmed, but this was not a priority. These changes are still new and not thoroughly tested, so clear conclusions are yet to be made. Gut instinct though says that further experimentation would be worth it.

This subsection tried to give an insight on the approach used from the environment setting perspective. A part of this work that was taken lightly at the start, but through experimentation, it seems that it plays a vital role in the creation of a quality dataset. In a few words, to create a good dataset, the environment should provide a big variety of examples, using dynamically spawned stationary/moving obstacles at random locations with various behaviors. Randomness offer examples that would not show up at a constant rate with a predefined environment.

4.9 Data variety and quality

Data variety and quality were mentioned multiple times due to its vital role in getting good policies. This subsection will be a short one, providing some characteristics that provide a good dataset. Characteristics are not going to be referring to input states. Instead, they are going to be general. These characteristics where derived from experimenting and good to have in mind when using simple neural network architectures, similar to the ones used in this work. The characteristics may apply to other architectures, but it is not tested yet.

There are not many data characteristics needed and they are simple as well. When creating an artificial dataset for data-driven virtual crowds, it is helpful to have well **balanced** data. To better understand balanced data, here is an example of not balanced data. Imagine creating data that the agent mostly follows his goal and rarely avoids an imminent collision. This will not result in a good policy. Most of the times the learning algorithm will stop on local optima that will enable the agent to

follow the goal religiously.

Next characteristic needed is to have example **variety**. Variety is what will help the policy learned to generalize in different situations better. To achieve good data variety some imagination is needed. The scenarios explained before, where the way to provide variety followed in this work. It is important to note though, that adding variety, makes more difficult balancing the data, but it is not as noticeable as the example mentioned above.

One may argue that with a better understanding of the learning algorithm, different parameters and analyzing node activity can help get better results from datasets that don't have these two characteristics and its true. But still, one needs a good foundation to get the best results, and data quality is that foundation.

4.10 Virtual Crowds Simulator engine

Simulation of Virtual crowds deemed necessary to create a simulator engine providing necessary parts of the workflow. The simulator engine was an integral part of this work, that's why a big amount of time was invested in building, refactoring and adding more features to it. Initially, it's role was in testing trained neural networks. To achieve this, the simulator had to be able to produce the state inputs of an agent. For better usage input states should be easily extensible and able to be combined together. In order to achieve this goal, the simulator engine should be able to communicate with python, exchanging state inputs and receiving the outputs. As it was stated in the summary, testing of neural networks was not enough anymore. The simulator was extended to support artificial data creation for further experimentation. Implementation details are stated below.

4.10.1 Flow

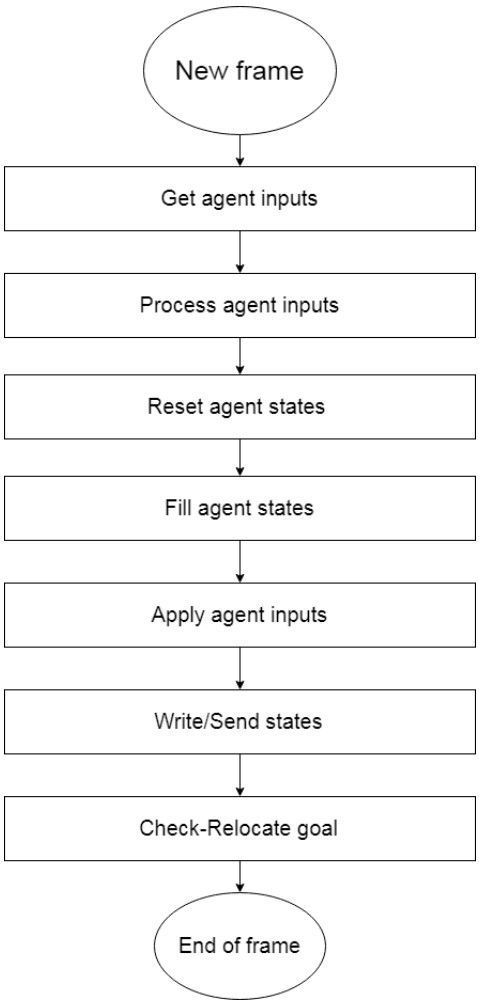


Figure 4.11: Flow followed for the engine.

In figure 4.11 a high-level view of the flow is shown. This is the sequence of actions used for the virtual crowd simulation. The flow was chosen taking into consideration unity implementation specifics and the time needed for sending states and getting outputs from the neural networks. The flow was the same for creating artificial data and simulating neural network policies.

First, it is important to understand some details behind this flow. Unity engine at the end of each frame's logic, proceed to update physics of each object moved. Then it proceeds to render the scene and show it to the user. When the new frame is shown, the user may react to the current environment setting. At this point, there were some pitfalls.

The first pitfall was to apply the inputs first and then fill the states. This is not a good approach because of the way unity physics work. Even if you move an object through code, you cannot use any state that is physics related because it will not be updated and other states that are physics independent will update resulting in inconsistencies.

The second pitfall observed was the difference in behavior of data creation and neural network simulation. Figure 4.12 shows the problem. Creating artificial data can follow the flow as needed because the action can be direct. But in the case of the neural network, there is communication overhead. The state seen in the current frame will get associated action during the next frame. There is a way to fix this. After creating the data, just during data preprocessing shift the action taken one frame before. This accounts for human reactions as well.

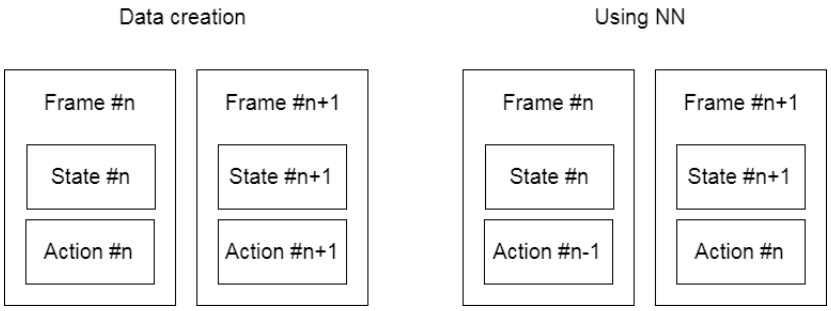


Figure 4.12: Pitfall demonstration.

4.10.2 State implementation

The states shown in the state section were coded into the simulator following some good practices. This section will not go into detail about the implementation of each state. Instead, it will give an insight on how all the states are implemented in order to enable the simulator engine to be extensible.

Extensibility was achieved by determining a state interface that provides necessary functions for a state to be used flawlessly in the engine logic. Functions deemed necessary were related to initializing a state, saving previous states, returning the state in different types and retrieving the number of features of each state input. After the interface was ready, a generic abstract class implemented the interface

functions. The abstract class was then derived for the implementation of each state input. For each state implementation, only unique details had to be implemented. Furthermore, this made possible to add more states and make different combinations of states for experimentation due to Unity being component based.

4.10.3 Agent behavior

Agent behavior is related to the action output section. For the implementation of each action output, an interface was established. The interface provided functions that are used for processing and applying the inputs. This approach was used in order to provide extensibility and flexibility to the engine user. This way it was possible to add new agent behaviors without having to change anything in the engine's logic.

4.10.4 Goals

For the implementation of goals in the simulation engine, two-goal managers where created. A goal manager that provided the goal to agents asking for one and an agent related goal manager attached to each agent.

Think of the first one as the parent goal manager. Its role is to return goals to agents asking, taking into consideration their position and other characteristics as minimum/maximum distance. Furthermore, it creates goals following predefined sequences for evaluation purposes. In case of sequences, agents are given a number so the manager can build the sequence. As for the second agent related manager, its main purpose in the system is to keep track of the agent's distance from the goal. A new goal is retrieved when the distance from the current goal is lower than a threshold predefined.

4.10.5 Obstacle spawning

Dynamic creation of obstacles was something analyzed in environment setting section. First of all obstacle spawning is not seen in the engine main flow 4.11, because it's execution order is not crucial. Obstacle spawning implementation enables agents to

be assigned to it. Agents attached have obstacles spawned around them with various ways already seen in 4.8. But how was this coded into the simulation engine?

An obstacle manager was created that was responsible for the obstacle creation tasks. The manager has many different obstacle prefabs that can be spawned. Scenarios were mentioned in 4.8. A simple implementation of scenarios was to create a settings class that would hold different variables that are related to obstacle positions, spawn timers, obstacle behaviors and a number of obstacles. Furthermore, the manager keeps a counter which is used in a simple if statement to choose different predefined settings through time. A default setting is chosen when no all predefined settings are finished. t counter value is equal to the frames passed in the simulation.

Chapter 5

Results

Contents

5.1 Static obstacles

With the stationary obstacle results a paper was published. The paper was on Virtual Environment Navigation Assisted by Neural Networks. The neural network architecture, inputs-outputs and dataset used to produce this static obstacle avoidance policy used in the paper will be presented. The NN was used as an assistance to user navigation in a Virtual environment.

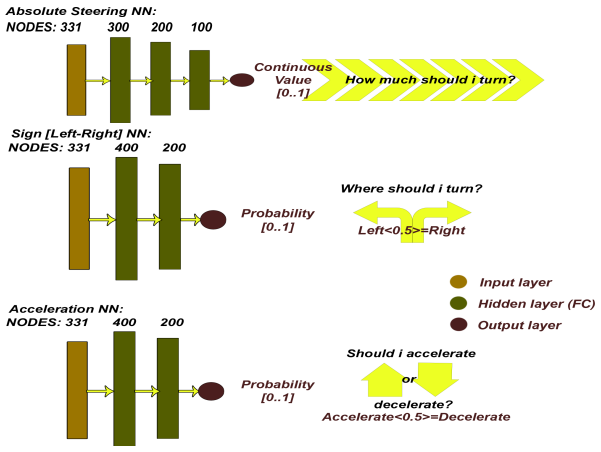


Figure 5.1: Neural networks and their role in agents behaviour.

In this approach 3 neural networks are used to solve the problem of collision avoidance. We use one NN for absolute value of steering angle the agent should use, one NN for the sign of the steering applied (i.e., negative or positive for left or right respectively) and one NN for deciding acceleration (one) or deceleration (zero).

For the absolute steering NN topology 3 hidden layers were used that have 300 neurons, 200 neurons and 100 neurons respectively with dropout layers between each of them and one between the last hidden layer and the output layer. The dropout rates are 0.8, 0.6, and 0.5 respectively. The output layer has one node for the steering value. The three hidden layers use the RELU activation function and the output layer uses the sigmoid function. This output is in the [0, 1] range and then we scale it back and combine it with the sign predicted by the other NN to get the correct rotation.

For the acceleration and sign NNs Topology the same topology was used. For this aspect of our system we used 2 hidden layers that have 400 and 200 nodes respectively with dropout layers between them with 0.6 and 0.5 dropout rates respectively. The output has one node for the respective prediction. The hidden layers use the RELU activation function and the output uses sigmoid activation function.

For all the NNs, the ADAM optimization algorithm was used. The parameters used were the default parameters provided from Keras library. For loss functions Mean Squared Error was used for the absolute steering NN and Binary Cross Entropy for the acceleration and sign NN.

The dataset used had 70000 training examples and it was created with user-driven artificial data creation. The environment setting used was spawning static obstacles in front and around the agent with increasing complexity over time. The number of features used for the input were 331 and they were associated with 2 outputs 4.4.2. The features keep information for 3 frames, current frame and the two before it. The input features consisted of:

1. Angle to goal - (4.3.5) - 3 feature.
2. Distance to Goal (4.3.8). - 3 features
    - Max Distance: $8\,\mathrm{m}$.
3. Forward rays (4.3.1). - 108 features
    - Scanning Area: 180°.
    - Divisions - Groups: 36.

- Groups of: 5°.
- Ray max distance: 6 m.

4. Towards goal rays (4.3.2). - 108 features
    - Scanning Area: 180°.
    - Divisions - Groups: 36.
    - Groups of: 5°.
    - Ray max distance: 6 m.
5. Ellipse (4.3.3). - 108 features
    - Scanning Area: 180°.
    - Divisions - Groups: 36.
    - Groups of: 5°.
    - Ray max distance: 6 m.
6. Visible/Invisible goal (4.3.7). - 1 feature only, the only feature that does not hold information from previous frames.

This combination of architecture, dataset and inputs-outputs resulted in good static obstacle avoidance. The resulting policy was able to avoid obstacles by close margins and only rarely the agent would collide with an obstacle. The cases that the agent collided were mostly related to the detail of the rays cast. The agent would slightly hit an object, but it would never have a head on collision. Another property of this policy it was the tendency to not always follow the safest path, but instead follow the path that was better aligned to its goal. Moreover, it was very interesting to see that it was able to find very narrow paths. A detail should be stated though, narrow passages could be detected when they are around 1-2 m away. Seeing the 4.8, we can see that two rays have bigger distance between them when they get far from the ray cast start point.

Increasing difficulty of the environment during the data creation, obstacles that looked more like walls were created and the policy learned was able to move around them

very well. Observing the policy and its capabilities further, we saw an unexpected capability. The policy could understand whether the goal was behind or in front of an obstacle.

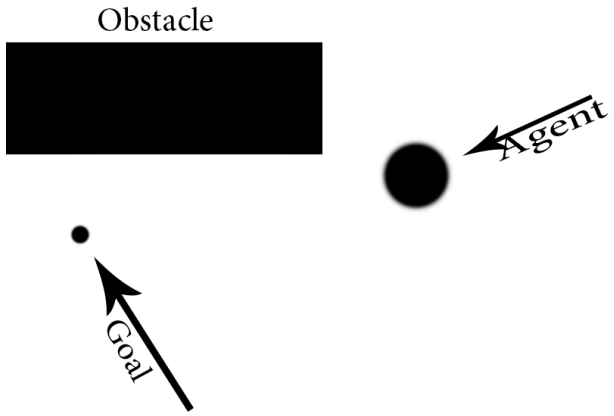


Figure 5.2: Static result problem #1.

This policy showed many interesting behaviors, but some weaknesses were observed. The two most important weaknesses are going to be stated. As seen in 5.2 the agent tries to turn left, but there is an obstacle at its left. The agent is stopped and tries to turn left, but when it turns then turns right again going back to its initial position. The agent does not realize that its stopped, and that its safe to turn towards the goal. After a number of tries the agent most of the time succeeds and move towards the goal. In some cases the agent will move around the obstacle trying to find an opening.

The second weakness seen was the inability to turn around when reaching a dead end. This inability was not a weakness exactly, as this capability was not in the target behavior. Seeing some unexpected path-finding capabilities, it was intriguing to see if the agent would be able to find its way toward its goal in an environment with big walls and dead ends. Testing in such environment revealed the inability of recovering

from dead ends.

5.1.1 Conclusion

Overall the policy learned was quite interesting to analyze and understand. A good behavior was learned with decent movement and not many problems. Of course, there are a improvements that can be done. It was clear that the policy did not have speed awareness. Using speed in the input features added more complexity in the data and it was more difficult to balance them as well, as a result it was not used. After this results, experimentation switched towards moving agents, because it was the initial goal of this project.

5.2 Moving agents

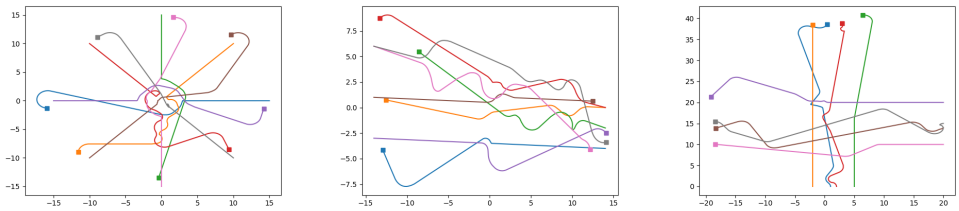
In this subsection results related to moving agents will be discussed. The results shown below are not at the level needed. Throughout this section, details on input states, NN architecture and comments on the results will be given. In order to demonstrate the results better, trajectories of agents were plotted in three different scenarios; Circle, Opposite and Crossing. More on the agent's behavior will be stated during the simulation. The neural networks were trained with three different datasets, as information collected from each result changes were proposed on the environment setting used for creating the data. Datasets for ease will be referred as first, second and third from their chronological order.

Before mentioning the results, some technical details added during the simulations need to be stated. All neural networks use the outputs of 4.4.2 and you will be able to see the observations stated in the figures. Due to the form of the outputs used, some changes were introduced. Neural network predictions did not work as expected, thresholding the value to 0 or 1 was not made at 0.5, instead, the threshold had to be fine-tuned for steering and accelerating. Furthermore, some randomization was introduced, decreasing or increasing the probability of steering or accelerating. The randomization range was fine-tuned.

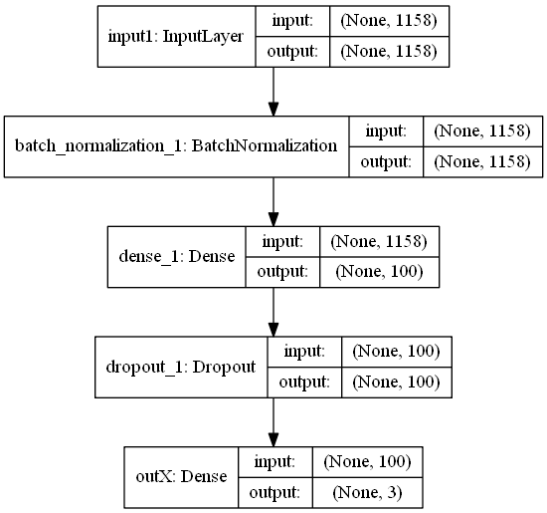
Before going into details about the results, some details about the scenarios used are needed. First of all, all scenarios are simulated using eight agents. The scenarios were chosen while having in mind to check for certain behaviors. These behaviors were related to perpendicular movement behaviors, head-on collision avoidance, and parallel movement.

Circle scenario is a scenario where the agents are spawned on a circle keeping symmetry. Keeping symmetry deems necessary to use an even number of agents. This way each agent will have another agent in front of him at a distance equal to the circle's diameter, in our case 30m. The opposite scenario is a situation where agents are spawned in two sides and each agent has to move towards the opposite side. The crossing scenario tries to create situations that agents will have perpendicular movement.

5.2.1 Results for dataset #1



(a) Circle. (b) Opposite. (c) Crossing



(d) Neural network Architecture

Figure 5.3: Results in Scenarios and NN Architecture #1

For this results the first data set taken into consideration for moving agents was used. The data set was created with user-driven data, under a mostly random environment setting. The input states combination chosen accumulated 1158 features. The output used was the one mentioned in this (4.4.2) section. The input state consisted of:

1. Angle to goal - Current frame state (4.3.5) - 2 feature.
2. Area picture - Seven frames state skipping two timesteps e.g 0,~~1,2~~,3,~~4,5~~,6 etc(4.3.6). - 700 features
   - Dimensions: 10 by 10.
   - Area: 10 m by 10 m as well.

3. Distance to Goal - Eight frames state skipping 4 timesteps e.g 0,~~1,2,3,4~~,5 etc(4.3.8). - 8 features
    - Max Distance: $35\,\mathrm{m}$.
4. Ellipse - Seven frames state skipping two timesteps e.g 0,~~1,2~~,3,~~4,5~~,6 etc(4.3.3). - 112 features
    - Scanning Area: 160°.
    - Divisions - Groups: 16.
    - Groups of: 10°.
    - Ray max distance: $8\,\mathrm{m}$.
5. Three Binary collision rays states - Seven frames state skipping two timesteps e.g 0,~~1,2~~,3,~~4,5~~,6 etc(4.3.4). - 336 features
    - Scanning Area: 160°.
    - Divisions - Groups: 16.
    - Groups of: 10°.
    - Ray max distance: $1\,\mathrm{m}$-$2\,\mathrm{m}$-$3\,\mathrm{m}$.

The neural network selected for this results was a shallow architecture with only one hidden layer. The hidden layer used Relu activation function. The dropout layer had 0.92 drop rate. Using considerably lower drop rate resulted to high over-fitting. The neural network was trained with MSE loss function.

5.2.1.1 Circle Scenario

Observing circle scenario gave a better understanding on what the neural network learns. All agents would come near each other while moving forward towards their goal. When noticing the other agents they would slow down, with some of them stopping. Some of them start first to rotate towards free space and then proceed to avoid the crowded area. After two to three agents find their way, normal flow is reestablished and everyone reaches it's goal.

5.2.1.2 Opposite Scenario

During the opposite scenario, more weaknesses of the policy are observed. Reactions to imminent head on collisions were observed. The reactions though were mostly late, resulting in avoiding collisions unnaturally. Furthermore, agents moving in parallel have awkward decision making. The agents occasionally overreacted to agents moving in parallel with them, some times slowing down.

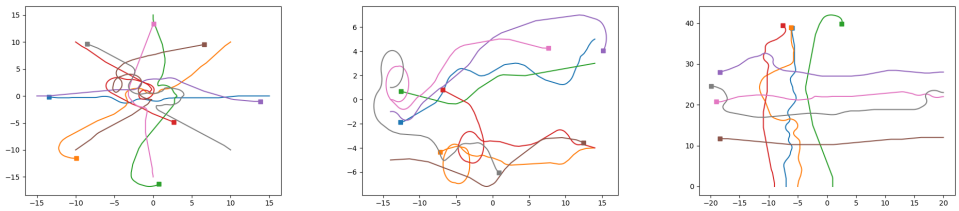
5.2.1.3 Crossing Scenario

In crossing scenario, the weakness of parallel move was seen one more time. As a new addition to the already seen problems, the agents does not act as expected when other agents have near perpendicular movement, taking into consideration the data used for training and the actions taken. In more detail, while data were created, the user acted by decelerating or turning in the opposite direction relative to the other agent. This was not seen, in this results. Instead, agents would come near each other, slow down and then rotate towards free space in order to avoid the collision.

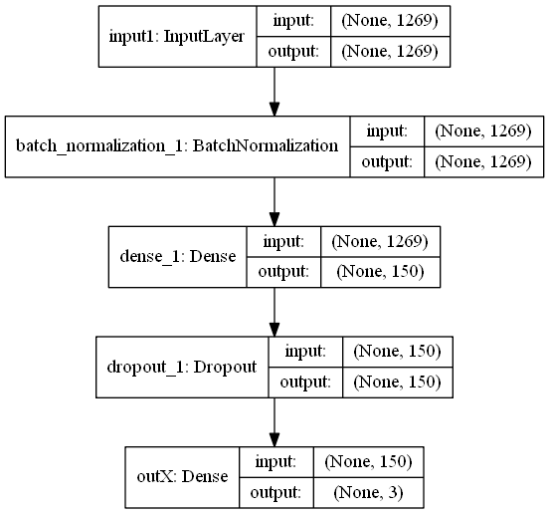
5.2.1.4 Result conclusion

This results were really interesting to see. Let's not forget that the results seen in this simulation are a based of generalization and not specifically trained agents for the tasks seen, as the environment was completely random and most of the time with a higher number of obstacles. What was taken from this results, was the ability to generalize well with the format of outputs stated 4.4.2.

5.2.2 Result for dataset #2



(a) Circle. (b) Opposite. (c) Crossing



(d) Neural network Architecture

Figure 5.4: Results in Scenarios and NN Architecture #2

For this results the second data set taken into consideration for moving agents was used. The data set was created with user-driven data, under a more controlled, but still random environment setting. The input states combination chosen accumulated 1269 features. The output used was the one mentioned in this (4.4.2) section. The input state consisted of:

1. Angle to goal - Current frame state (4.3.5) - 1 feature.
2. Area picture - Seven frames state skipping three timesteps e.g 0,~~1,2,3~~,4,~~5,6,7~~,8 etc(4.3.6). - 700 features
    - Dimensions: 10 by 10.
    - Area: 10 m by 10 m as well.

3. Distance to Goal - Eight frames state skipping 4 timesteps e.g 0,~~1,2,3,4~~,5 etc(4.3.8). - 8 features
    - Max Distance: $35\,\mathrm{m}$.
4. Ellipse - Seven frames state skipping three timesteps e.g 0,~~1,2,3~~,4,~~5,6,7~~,8 etc(4.3.3). - 112 features
    - Scanning Area: 160°.
    - Divisions - Groups: 16.
    - Groups of: 10°.
    - Ray max distance: $8\,\mathrm{m}$.
5. Four Binary collision rays states - Seven frames state skipping three timesteps e.g 0,~~1,2,3~~,4,~~5,6,7~~,8 etc(4.3.4). - 448 features
    - Scanning Area: 160°.
    - Divisions - Groups: 16.
    - Groups of: 10°.
    - Ray max distance: $1\,\mathrm{m}$-$2\,\mathrm{m}$-$3\,\mathrm{m}$-$4\,\mathrm{m}$.

As shown, the only changes made state wise, were the addition of on more Binary collision rays state, looking for collisions in four meters and stripping away the time-step in angle to goal state. This was mostly done for experimenting and results as shown below are not satisfying.

The neural network selected for this results was a shallow architecture similar to the one used in previous results. One hidden layer was used. The hidden layer used Relu activation function. The dropout layer had 0.9 drop rate. Using considerably lower drop rate resulted to high over-fitting. The neural network was trained with MSE loss function.

5.2.2.1 Circle Scenario

Observing circle scenario gave a better understanding on what the neural network learns. All agents would come near each other while moving forward towards their goal. When noticing the other agents they would slow down, with some of them stopping. But with this network, the agents behaved really awkward. Instead of finding an open space near their goal, they instead turned around and got further away from the crowded area and then turning again towards the goal. This was interesting at first until seeing the output of opposite scenario.

5.2.2.2 Opposite Scenario

During the opposite scenario, one can see strange actions. It seems that agents when they first turn towards a certain direction they tend to replicate the same action in the following frames, eventually doing a full circle before going reorienting themselves towards the goal.

5.2.2.3 Crossing Scenario

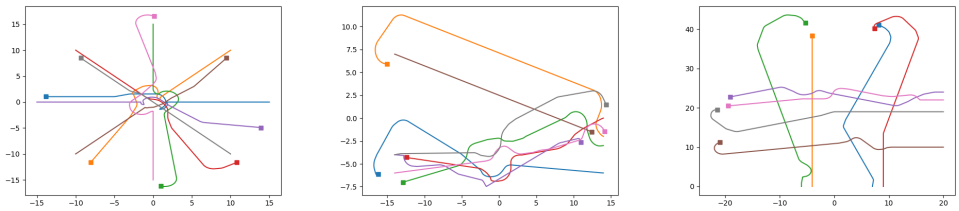
In crossing scenario, the weakness of parallel movement seen before, seems to be a bit better than the previous results. The same weakness with perpendicular movement is seen in this scenario again.

5.2.2.4 Result conclusion

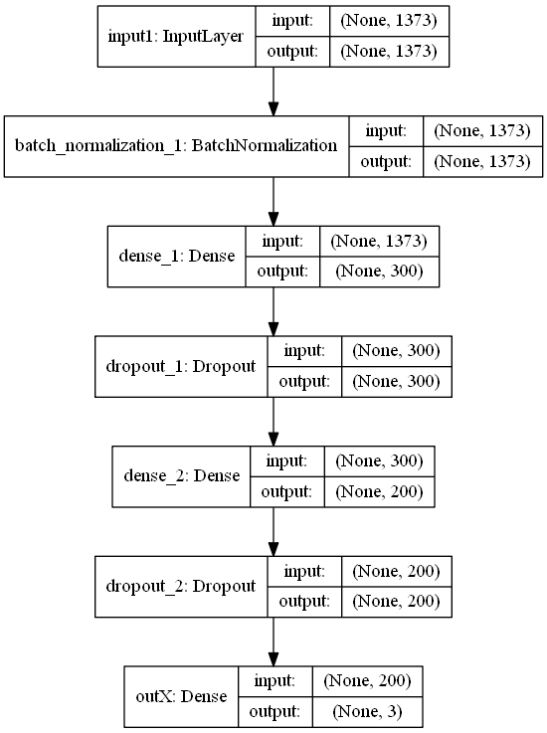
These results were not the ones expected. Instead of improving the policy, things seem to get worse. The changes on the environments setting done did not help at all. More control over the environment seems to be needed to balance data better, recreating various situations than need specific actions. Last but not least, seeing the agent in various examples insist on the previous action taken, resulting to the agent doing a full circle before orienting again towards the goal needs to be taken into consideration. It seems that the neural network is able to learn the differences

between two consecutive frames of area related states. As a result it may take this differences as an indication of steering.

5.2.3 Results for dataset #3



(a) Circle. (b) Opposite. (c) Crossing



(d) Neural network Architecture

Figure 5.5: Results in Scenarios and NN Architecture #3

For this results the third data set was used for moving agents was used. The data set was created with user-driven data, with the environment being more controlled, creating perpendicular , parallel and opposite movements. Random environment was used only during half of the data examples. The input states combination chosen accumulated 1373 features. The output used was the one mentioned in this (4.4.2) section. The input state consisted of:

1. Angle to goal - 3 frames state skipping nine timesteps (4.3.5) - 3 features.

2. Area picture - Six frames state skipping five timesteps e.g 0,~~1,2,3,4,5~~,6 etc(4.3.6). - 864 features
    - Dimensions: 12 by 12.
    - Area: $10\,\mathrm{m}$ by $10\,\mathrm{m}$.
3. Distance to Goal - Eight frames state skipping 4 timesteps e.g 0,~~1,2,3,4~~,5 etc(4.3.8). - 8 features
    - Max Distance: $45\,\mathrm{m}$.
4. Ellipse - Six frames state skipping five timesteps e.g 0,~~1,2,3,4,5~~,6 etc(4.3.3). - 96 features
    - Scanning Area: 160°.
    - Divisions - Groups: 16.
    - Groups of: 10°.
    - Ray max distance: $8\,\mathrm{m}$.
5. Four Binary collision rays states - Six frames state skipping three timesteps e.g 0,~~1,2,3,4,5~~,6 etc(4.3.4). - 384 features
    - Scanning Area: 160°.
    - Divisions - Groups: 16.
    - Groups of: 10°.
    - Ray max distance: $1\,\mathrm{m}$-$2\,\mathrm{m}$-$3\,\mathrm{m}$-$4\,\mathrm{m}$.

The most important changes in this state is the additional detail in the area picture and the intervals between frames by increasing the skipping value. The bigger interval was used in order to have biggest differences between the frames saved, hoping the NN will be able to make use of this detail.

The neural network selected for this results had two hidden layers. The hidden layers used Relu activation function. The first dropout layer had 0.5 drop rate and the last one 0.9 drop rate. If both drop rates are high, the network cannot learn and if the

last dropout is lower, the NN is over-fitting.

5.2.3.1 Circle Scenario

Observing the circle scenario, the behavior was similar with the behavior seen in 5.2.1. The agents would slow down when near each other. Next agents started to rotate towards free area trying to avoid the crowded area. After 2-3 agents leave the area, the remaining agents are able to move towards their goal.

5.2.3.2 Opposite Scenario

In the opposite scenario, results are more stable than before. The agents avoid each other most of the time. The problem of late reaction is still there. Unfortunately changing the intervals between each frame used did not help with this problem.

5.2.3.3 Crossing Scenario

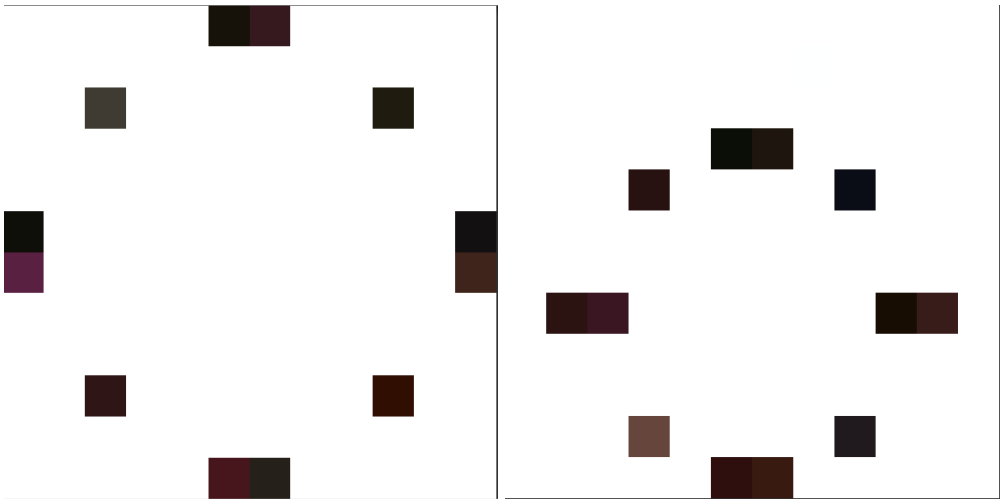
In crossing scenario some positive behaviors where observed. Agents during crossing scenario behave better during perpendicular movement situations. Of course, this behavior is not stable, but seeing some of the training examples be used during the simulation is nice. Parallel movement is better as seen during opposite scenario.

5.2.3.4 Result conclusion

These results brought some positive vibes to this project. The examples created for this simulation seem to influence positively the agent behaviors. Furthermore, it is still very difficult to see whether different intervals between frames can influence the results positively.

5.2.4 Moving agent conclusions

As already mentioned this results was not at the level expected. It was still interesting to state some of the changes done, trying to improve the results. Unfortunately, the improvements are small or non existent and drastic changes have to be done to improve this results. For now some observations from these results will be stated.



(a) Current frame (b) After 0.5s.

Figure 5.6: Representation with resolution: 12 by 12.

Regarding the states, one of the weaknesses seen in the results seems to be directly related to the states. Think of the state 4.3.6 mapping the area around the agent. As stated in results, the area covered was $10\,\mathrm{m}$ by $10\,\mathrm{m}$. But we should remember that this area is not that big if we take into consideration the maximum speed $4\,\mathrm{m/s}$. The problem seen in circle scenario may be directly related to the limited view of agents, resulting in slow reactions by default. In figure 5.6 we see the exact frame when the opposite agent is at a distance of $9\,\mathrm{m}$. Even if the agent starts to slow down at that point, most probably the agents will almost be very close together and the possible actions are limited to decelerating. In addition, the lack of detail is detrimental. The agent will find it very difficult to find openings due to the inaccuracy of the state representation. This observation is really important to be taken into consideration in future work.

Another question-mark raised during these experiments is using only simple neural

networks and as the environment and simulations got more difficult it seems that this type of neural networks lacks the capabilities to deal with temporal data. Temporal data are not usually used with this kind of neural networks but it was interesting to see, that even simple neural networks have the ability to learn some of the complexities seen in crowd specific data.

Chapter 6

Conclusions

Contents

This project experimented on virtual crowds using Neural networks. A number of conclusions were derived from this work. The most important conclusions are going to be mentioned.

Creating artificial data was detrimental. This approach has a lot of potential. Throughout this research artificial data helped in prototyping different state-input combinations, output forms, and environments. The prototyping process was responsible for understanding various aspects that have a huge impact. Experimenting and observing the results brought in the surface the importance of data balance and variety. Brainlessly creating artificial data will not help. User-driven artificial data could be a subfield of data-driven virtual crowds.

From the neural network's perspective, simple architectures were only used. These algorithms had the ability to learn a very good policy on stationary obstacles and not as good moving obstacles one. It was intriguing to see the capabilities of neural networks on a temporal problem. At a certain point, neural networks seem to hit a ceiling and more algorithms need to be considered.

As a final note, imitating crowd behavior is demanding. The range of possible actions and the data complexity is vast. The way people perceive their surroundings, memory, emotions and many other characteristics influence human behavior. In this project, we stripped a lot of these characteristics but results were still not at the level expected.

6.1 Future work

During this project, we scratched the surface of virtual crowds. This work was oriented towards obstacle avoidance but in the future, more elements of human behavior are going to be researched. The future can be divided into two parts.

A first part that will try to utilize the useful knowledge acquired in this work, aiming to improve the obstacle avoidance aspect presented. An obstacle avoidance policy that will be able to transfer human behavior in virtual agents. A good policy has a detrimental role for further extending the capabilities of an agent. Another thought would be to try using reinforcement learning to learn the behavior needed, as it is able to learn the temporal properties of the problem.

The second and more interesting part will be working on adding variety to the behaviors that the AI agents will be able to exhibit. Agents driven by their emotional state and agents that are able to understand the danger and act accordingly are some of the examples that come to mind. The behaviors and their applications are endless.

References

[1] Nuria Pelechano, Jan Allbeck, and Norman Badler. *Virtual Crowds: Methods, Simulation, and Control (Synthesis Lectures on Computer Graphics and Animation)*. Morgan and Claypool Publishers, 2008. ISBN: 1598296418, 9781598296419.

[2] Jessurun A.J. Dijkstra J. Timmermans H.J.P. "A Multi-Agent Cellular Automata System for Visualising Simulated Pedestrian Activity." In: *Theory and Practical Issues on Cellular Automata. Springer, London* (2001).

[3] Helbing and Molnár. "Social force model for pedestrian dynamics." In: *Physical review. E, Statistical physics, plasmas, fluids, and related interdisciplinary topics* 51 5 (1995), pp. 4282–4286.

[4] Craig W. Reynolds. "Flocks, Herds and Schools: A Distributed Behavioral Model". In: *SIGGRAPH Comput. Graph.* 21.4 (Aug. 1987), pp. 25–34. ISSN: 0097-8930. DOI: 10.1145/37402.37406. URL: http://doi.acm.org/10.1145/37402.37406.

[5] Craig Reynolds. "Steering Behaviors For Autonomous Characters". In: (1999).

[6] Lerner Alon, Chrysanthou Yiorgos, and Lischinski Dani. "Crowds by Example". In: *Computer Graphics Forum* 26.3 (), pp. 655–664. DOI: 10.1111/j.1467-8659.2007.01089.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2007.01089.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2007.01089.x.

[7] Kang Hoon Lee et al. "Group Behavior from Video: A Data-Driven Approach to Crowd Simulation". In: *Eurographics/SIGGRAPH Symposium on Computer Animation*. Ed. by Dimitris Metaxas and Jovan Popovic. The Eurographics Association, 2007. ISBN: 978-3-905673-44-9. DOI: 10.2312/SCA/SCA07/109-118.

[8] P. Long, W. Liu, and J. Pan. "Deep-Learned Collision Avoidance Policy for Distributed Multi-Agent Navigation". In: *ArXiv e-prints* (Sept. 2016). arXiv: 1609.06838 [cs.AI].

[9] Jur van den Berg, Ming Lin, and Dinesh Manocha. "Reciprocal Velocity Obstacles for Real-Time Multi-agent Navigation". In: (May 2008), pp. 1928–1935.

[10] Pinxin Long et al. “Towards Optimally Decentralized Multi-Robot Collision Avoidance via Deep Reinforcement Learning”. In: *CoRR* abs/1709.10082 (2017). arXiv: 1709.10082. URL: http://arxiv.org/abs/1709.10082.

[11] François Chollet et al. *Keras*. https://keras.io. 2015.

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

[13] Sepp Hochreiter and Jürgen Schmidhuber. “Long Short-Term Memory”. In: *Neural Comput.* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. URL: http://dx.doi.org/10.1162/neco.1997.9.8.1735.

[14] Nitish Srivastava et al. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. In: *J. Mach. Learn. Res.* 15.1 (Jan. 2014), pp. 1929–1958. ISSN: 1532-4435. URL: http://dl.acm.org/citation.cfm?id=2627435.2670313.

[15] S. Ioffe and C. Szegedy. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”. In: *ArXiv e-prints* (Feb. 2015). arXiv: 1502.03167 [cs.LG].

[16] J. Kiefer and J. Wolfowitz. “Stochastic Estimation of the Maximum of a Regression Function”. In: *Ann. Math. Statist.* 23.3 (Sept. 1952), pp. 462–466. DOI: 10.1214/aoms/1177729392. URL: https://doi.org/10.1214/aoms/1177729392.

[17] D. P. Kingma and J. Ba. “Adam: A Method for Stochastic Optimization”. In: *ArXiv e-prints* (Dec. 2014). arXiv: 1412.6980 [cs.LG].