UHASSELT KNOWLEDGE IN ACTION

Maastricht University

Faculteit Wetenschappen

School voor Informatietechnologie

master in de informatica

Masterthesis

Data-Driven Crowd Simulation using Neural Networks

Bram Vanherle
Scriptie ingediend tot het behalen van de graad van master in de informatica

PROMOTOR :
Prof. dr. Fabian DI FIORE

2018
2019

Faculteit Wetenschappen
School voor Informatietechnologie

master in de informatica

Masterthesis

Data-Driven Crowd Simulation using Neural Networks

Bram Vanherle

Scriptie ingediend tot het behalen van de graad van master in de informatica

PROMOTOR :

Prof. dr. Fabian DI FIORE

Contents

Abstract

In this thesis we apply the power of neural networks to data-driven crowd simulation in order to train a model that can learn the rules that govern the movement of agents in a crowd, leading to realistic looking crowd behaviour. To do so we give a thorough overview of a number of existing crowd simulation techniques, classifying them according to granularity with which they model crowds. We then go on to describe how neural networks work and how they are trained in order to use them towards crowd simulation. The focus here being on supervised learning methods, as we are creating data-driven models. We propose two neural networks that can generate new instantaneous velocities for agents given the current state of their surroundings and inner motivations. The first model being a feedforward neural network that given a description of the agents environment generates a number of possible actions at each timestep. The second proposed model is the adaptation of the feedforward model into a recurrent neural network, in order to better leverage the temporal structure of the data. Both models are trained on data gathered from areal images of crowds, with the hyperparameters being optimized by the Tree-structured Parzen Estimator algorithm. Furthermore a crowd simulation framework is described in which these two models tie in. To validate our work we defined a number of metrics that indicate realistic crowd behaviour and compared different version of our models trained with different parameters regarding the representation of the agents environment, with the already established Social Forces model. These metrics were computed for a number of different scenarios. There is a lot of variety between the performances of different models on different scenarios for different metrics, but generally we can conclude that our models manage to perform on par with the Social Forces model. With some models even outperforming the Social Forces model.

Acknowledgements

First and foremost I would like to thank my promoter, Prof. dr. Fabian Di Fiore for providing me with counseling and guidance when needed, but also for giving me a lot of freedom in choosing the direction of my research and for allowing me to work independently. Being able to research, and work on topics that I was interested in is what kept me motivated and engaged throughout the duration of this thesis.

Besides my promoter I would also like to thank dr. Jeroen Put and dr. Nick Michiels for proofreading this thesis and for providing me with insightful comments and constructive feedback.

I would also like to thank the staff – teaching or otherwise – of the University of Hasselt and the Expertise centre for Digital Media (EDM). Over the past five years I have enjoyed a rich and challenging education that has taught me so much about computer science and many other fields of science; and none of it would be possible without them.

Next, I would like to thank my parents Christa and Peter, and my step-mom Nathalie. Their support during both my masters and bachelors education has helped me tremendously and is what will eventually allow me to graduate this year.

Finally, I would like to take a moment to thank some of my fellow students, not only for their excellent teamwork on group projects or their advice on certain tasks throughout my education; but also for our time spent outside of the university doing non-school related activities, which has been instrumental staying motivated over the course of this degree.

Introduction

People have always been interested in the behaviour of human crowds, as there just is something fascinating about how many small autonomous parts with varied intentions and personalities seem to behave in a coordinated manner as one body. This has been the topic of study as soon as the late nineteenth century with LeBons book *Psychologie des Foules* [21]. Actually simulating crowds using computer models has been a more recent endeavour, only emerging in the eighties when Reynolds published his seminal work on simulated Boids [48]. The applications of this type of simulation are plenty, ranging from architecture, where the simulation of people in a building can lead to useful insights regarding the flow of humans in regular situations or emergencies; to the entertainment industry, where crowds get simulated to render large populations for movies or video games.

The two main focuses of study within the field of crowd simulation are those that focus on quality of the visualisation and those that focus on the realism of behavioural aspects [63]. With the former including tasks such as generating diverse sprites for the agents in the crowds, algorithms for creating realistic looking animations, and efficient rendering methods. Whereas in the latter researchers attempt to generate natural looking movements and interactions for the agents in the crowds based on rules or formulas. In this thesis the focus is on the behavioural aspect of crowd simulation.

With the emergence of better pedestrian tracking techniques and a wealth of areal imagery of crowds available there is no shortage of data on crowds. It is only natural for researchers to attempt to use this data to generate even more realistic looking crowds. This field of crowd simulation is called data-driven crowd simulation and there is already a lot of research done in this field. Most of these methods build a database out of samples from the collected data to then query these at simulation time in order to find appropriate actions for the agents in the simulation. These algorithms are generally computationally very expensive at runtime and fail to expose the underlying rules that define the crowds movements. We feel that a better approach would be to train a machine learning model on this data with two of the main benefits being:

- The computationally most expensive part of using a machine learning model is the training process which only needs to happen once, as opposed to query methods, where most expensive operations need to happen multiple times at each simulation step. Once we have fitted our model, generating new data is rather inexpensive, given that the model is not too complex.

- When using query methods the available data gets only compared and repeated, no real knowledge is gained about the data. As opposed to using a machine learning model, where the underlying rules can be exposed to learn a more versatile model of crowd interaction.

One of the most popular and powerful machine learning models of recent years is the neural network. This due to the fact that they can learn nonlinear dependencies and their ability to automatically learn features. We feel that the next logical step in attempting to generate realistic crowd behaviour is to apply the power of neural networks to data gathered on crowds to see if they can learn the rules that govern the motion of human in crowds. Even though this approach can lead to realistic behaving crowds the downside is that we do not gain a lot of insight on the rules themselves as neural networks act as a black box to a certain extend. But disregarding the fact that we might only gain limited knowledge about the inner workings of a crowd, a trained neural network still has a lot of potential to be an effective tool for simulating a realistic looking crowd.
The goal of this thesis is to design and train a data-driven neural network that can simulate realistic crowd movement and to check its validity compared to other models. To create such a crowd simulation model we first study the work already done in the field of crowd dynamics. We take a look at a broad number of existing methods while classifying these to gain a better oversight on what types of crowd simulation exist and to be able to provide a context for our proposed method. We then delve into the world of artificial neural networks, describing how to design and train them and what pitfalls should be avoided when doing so. Using the knowledge gained from the previous two chapters we design and train two different neural network models that can generate movement for the agents in our simulation. Furthermore, a crowd simulation framework is proposed in which these neural network models tie in, to form a complete crowd simulation method. Finally we define a number of metrics on which we will evaluate and compare our model. We compute these metrics for a number of scenarios and models and present these results.

Chapter 1

Crowd Simulation

Crowd simulation is the process of realistically simulating and visualizing the movement of a large number of agents in a scene. It is a problem that covers multiple topics: efficient rendering of a large number of models, producing realistic and diverse animations for each agent, generating natural looking movements and behaviours for the agents, etc. In this chapter a number of existing methods are studied to give an understanding of what work has already been done in the field an to use as a baseline and context for our own method.
An important aspect of crowd simulation is steering the agents to generate realistic human looking dynamic behaviours. This is often referred to as crowd dynamics and it is the topic within crowd simulation that we will focus on in this chapter and throughout this thesis. While we do so the rendering and animating of these crowds is disregarded, as simple 2D animations suffice to illustrate their movements. Even within the field of crowd dynamics there is an abundance of different methods, all applicable to different situations. Researchers have attempted to classify these methods based on two scales: the size of the crowd and the length of the simulation [73]. These scales form a 2D space that can be divided in three areas:

- models of long-term crowd phenomena
- models of short-term phenomena of huge-sized crowds
- models of short-term phenomena of medium to small-sized crowds

There are three types of crowd simulation approaches that are useful for different situations in this 2D space. These modeling approaches are classified based on the granularity with which they model the crowd. These approaches are: flow-based, entity-based and agent-based, and their positions on the classification scales are shown in Figure 1.1.
The next three sections will discuss the specifics of these methods and present some examples of them. Since they are of significance to this thesis we follow these sections up by discussions on data-driven and neural network models in crowd simulation. This is followed up by a short summary of the ways in which we can evaluate crowd simulation models.

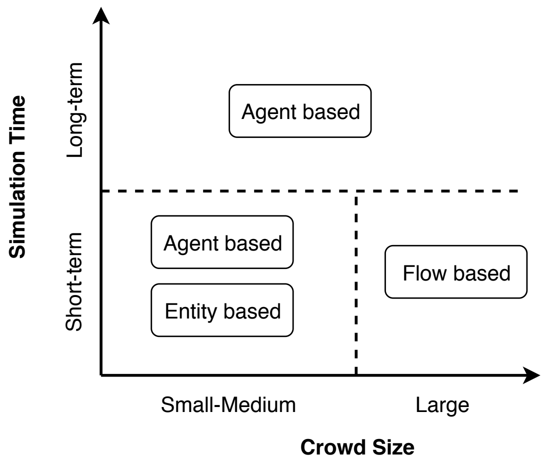


Figure 1.1: Graph showing the two scales crowd simulation approaches can be classified on and the position of the three major approaches on these scales.

1.1 Agent-based

Agent-based methods model the agents in the simulation as autonomous, interacting individuals. Each agent has a certain degree of intelligence they use to make decisions based on their environment. Broadly, one can identify first-order algorithms, where the dominant cue for interaction is the position of the agents, and the more recent second-order algorithms, which started predicting where and when collisions will take place in order to avoid them with anticipation [69]. In this section a number of methods from both categories are presented to get an insight in the different ways of achieving agent-based crowd dynamics.

1.1.1 First-order algorithms

In first-order agent-based algorithms, the interaction rules are based on the agents positions. This makes these models easy to understand, implement and extend. Despite their simplicity these models are still capable of generating believable movement for their agents. Some examples are explored here as they offer a look into some of the more simple cues for crowd interaction along with an overview of the history of crowd simulation, as these methods where some of the first around.

Reynolds' Boids

One of the first works done in this category and in crowd simulation in general was Reynolds' model for Flocks, Herds and Schools [48]. In his seminal work, regarding the simulation of flocks of boids (bird-oid objects), Reynolds described behaviours that correspond to the opposing forces of collision avoidance and the urge to join the flock.

These behaviours are expressed as three rules (in order of precedence), that lead to simulated flocking:

1. **Collision avoidance:** avoid collisions with nearby flockmates, by steering away from close flockmates
2. **Velocity matching:** attempt to match velocity with nearby flockmates, by steering towards the average heading of local flockmates
3. **Flock Centering:** attempt to stay close to nearby flockmates, by steering to move towards the average position (center of mass) of local flockmates

An illustration of how these behaviours look in a flock is given in Figure 1.2. The boids are given a scripted path to follow and their movements along this path are calculated by applying the above three behaviours. To avoid objects a steer-to-avoid method is implemented. The boid only considers objects in front of it by checking for collisions with its local Z-axis. Working in local perspective space, it finds the silhouette edge of the obstacle closest to the point of eventual impact. A radial vector is computed which will aim the boid at a point one body length beyond that silhouette edge. The model has since been extended in several different ways to incorporate different natural effects, such as fear [13].

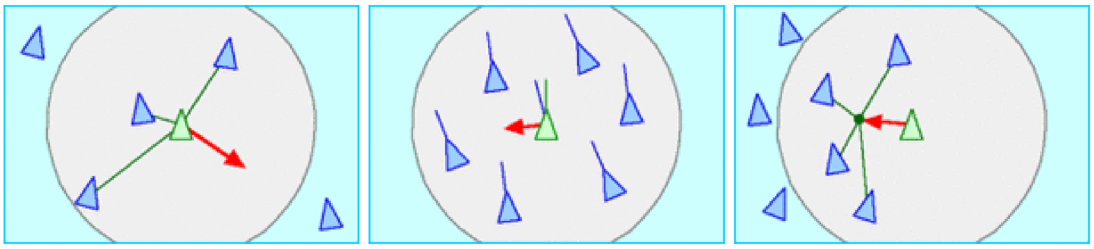

(a) Collision avoidance (b) Velocity matching (c) Flock centering

Figure 1.2: Three behaviours of Reynolds' Boids model

Musse and Thalmann's Group Inter-relationship model

Initial research in the field of human crowd simulation began in 1997 with Daniel Thalmann's supervision of Soraia Raupp Musse's PhD thesis as they presented a model of crowd behaviour to simulate the motion of a generic population in a specific environment [41].
Their method is based on describing the crowd behaviour through the group interrelationships. All agents can belong to a group and can change groups based on their emotional parameters and the group's parameters. The group parameters are specified by defining the group's goals, number of autonomous virtual humans in the group and the level of dominance from each group. This is followed by the creation of virtual humans based on the groups' behaviour information. The individual parameters are: a

list of goals and individual interests for these goals (originated from the group goals), an emotional status (a random number), the level of relationship with the other groups (based on the emotional status of the agents from a same group) and the level of dominance (which follows the group trend).
They go on to describe a number of scenarios in which agents can change groups or change their individual parameters. Agents can, for example, change groups when the emotional status of the new group is more similar to theirs compared to their old group. Or the emotional status can be changed in the encounter of two autonomous virtual humans in a random process. Only when both have a high value for the domination parameter, one virtual human is chosen in a random way to reduce its emotional status. In this case a polarization between two leaders will follow. In any other case, both virtual humans must assume the highest emotional status between them. This model is complemented by a collision detection system. An overview of this complete method is shown in Figure 1.3.

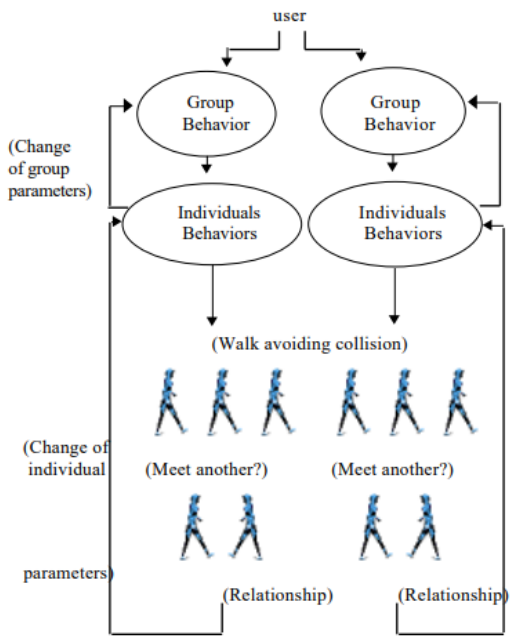


Figure 1.3: Overview of Musse and Thalmans's Group Inter-relationship model.

Social Forces

Further research started to consider agents as particles subjected to several forces such as repulsion from neighbours and attraction to their goal. One of these works was the Social Force model for pedestrian dynamics [24]. Their method assumes that the motion of an agent is influenced by three components: the agent's drive to reach their goal, the repulsiveness from neighbouring agents and obstacles and the attractiveness of specific

agents and objects. An example of some of these forces applied to an agent is shown in Figure 1.4.
The agents tendency to move towards his goal or desired direction is given by the following acceleration term:

$$\mathbf{F}_\alpha^0(\mathbf{v}_\alpha, v_\alpha^0 \mathbf{e}_\alpha) = \frac{1}{\tau}(v_\alpha^0 \mathbf{e}_\alpha - \mathbf{v}_\alpha) \tag{1.1}$$

with $\mathbf{e}_\alpha$ the agents desired direction, v_α^0 the agents desired speed, $\mathbf{v}_\alpha$ the agents current speed vector and τ the relaxation time.
The repulsive effect of agent β on agent α can be represented by vectoral quantities:

$$\mathbf{F}_{\alpha\beta}(\mathbf{r}_{\alpha\beta}) = -\nabla_{\mathbf{r}_{\alpha\beta}} V_{\alpha\beta}[b(\mathbf{r}_{\alpha\beta})] \tag{1.2}$$

where repulsive potential $V_{\alpha\beta}(b)$ is a monotonic decreasing function of b with equipotential lines having the form of an ellipse that is directed into the direction of motion. A pedestrian is also repulsed by borders of objects, therefore border B evokes a repulsive effect on agent α described by:

$$\mathbf{F}_{\alpha B}(\mathbf{r}_{\alpha B}) = -\nabla_{\mathbf{r}_{\alpha B}} U_{\alpha B}(||\mathbf{r}_{\alpha B}||) \tag{1.3}$$

with a repulsive and monotonic decreasing potential $U_{\alpha B}(||\mathbf{r}_{\alpha B}||)$ and $\mathbf{r}_{\alpha B} = \mathbf{r}_\alpha - \mathbf{r}_B^\alpha$, where $\mathbf{r}_B^\alpha$ denotes the nearest point on border B.
The attractive effects effects $\mathbf{F}_{\alpha i}$ at places $\mathbf{r}_i$ can be modelled by attractive, monotonic increasing potentials $W_{\alpha i}(||\mathbf{r}_{\alpha i}||, t)$ as such:

$$\mathbf{F}_{\alpha i}(||\mathbf{r}_{\alpha i}||, i) = -\nabla W_{\alpha i}(||\mathbf{r}_{\alpha i}||, t) \tag{1.4}$$

The agents total motivation is denoted as $\mathbf{F}_\alpha(t)$, and is found by computing the sum of the agents own driving force and all attractive and repulsive forces for all objects and agents in the simulation. Furthermore the agents line of sight is considered, as they can only be influenced by things they can see. Finally a fluctuation term is introduced that takes random variations of behaviour into account. We can now define the change in velocity for an agent as:

$$\frac{d\mathbf{w}_\alpha}{dt} = \mathbf{F}_\alpha(t) + fluctuations \tag{1.5}$$

These first-order algorithms are easy to implement and extend, causing them to be widely used. Groups can be convincingly modelled, however, locally, agents' trajectories are not always very convincing. Second-order methods attempt to solve some artefacts found in these approaches.

1.1.2 Second-order algorithms

The methods described in the previous section base their collision avoidance behaviour on the locations of other agents, but this way they do not make use of all the available information. The methods in this section improve on the performance of the first-order

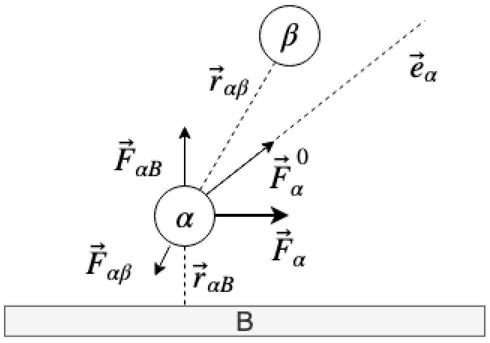


Figure 1.4: An example of an agent in the social forces model. The agent feels repulsive forces from neighbour β and wall B, while following desired direction $\mathbf{e}_{\alpha}$

models by making use the instantaneous velocities of other agents to linearly extrapolate their future trajectories and predict future collisions, creating better collision avoidance. There are multiple ways of using neighbours' speeds to improve collision avoidance, two of those are: generating repulsive forces from future collisions and generating collision free velocities. Both these approaches are briefly introduced in this section, accompanied by an example of a method using these approaches.

Repulsive-forces from Future Collisions

This type of second-order algorithm predicts where collisions between agents will take place in the future and generates repulsive forces on these agents to steer them away from the potential collision.
One such method was developed by Karamouzas et al. [30] in their Predictive Collision Avoidance Model for Pedestrian Simulation. Their work can be seen as an extension of the Social Force model with an improvement in the way repulsive forces from pedestrians are computed.
The first force applied to pedestrians is the force that drives them to their goal, this is done in the same manner as in the Social Forces model (Equation 1.1). Next, the agents are repulsed from walls, the force applied by wall B on a pedestrian α is defined by:

$$\mathbf{F}_{\alpha B} = \mathbf{n}_B \frac{d_s + r_\alpha + d_{\alpha B}}{(d_{\alpha B} - r_\alpha)^k} \tag{1.6}$$

this is only applied if $d_{\alpha B} - r_i < d_s$, with $\mathbf{n}_B$ the normal vector of the wall, $d_{\alpha B}$ the shortest distance between pedestrian α and wall B, d_s the safe distance that the pedestrian prefers to keep from walls and r_α the radius of pedestrian α.
Each pedestrian has a personal space, modelled by a disk that is defined by its radius ρ_α. A collision occurs when pedestrian β invades into the private space of pedestrian α at some time t_c:

$$\exists\, t_c \geq 0 \,|\, d_{\alpha\beta} \leq \rho_\alpha + r_\beta \tag{1.7}$$

where $d_{\alpha\beta}$ denotes the distance between the pedestrians' centers. To avoid potential collisions within a certain anticipation time t_a an evasive force $\mathbf{F}_e$ is applied to pedestrian α. To compute this force, all potential collisions need to be predicted. This is done for pedestrian α by first computing its desired velocity, which is the sum of its current velocity with the force applied by the walls and by the pedestrian's goal. Using this desired velocity an estimation of the pedestrian's future position can be made. This process is repeated for all of the agents that can be seen by α, the only difference being that only the pedestrians current velocity is used to estimate the future position, as pedestrian α does not known the inner forces working on other agents. Pedestrian β intersects with pedestrian α if it lies in or intersects the personal space of α. To easily check if this is the case the Minkowski sum is performed between the disk representing personal space of pedestrian α and the disk representing pedestrian β as this reduces the problem to a ray-disc intersection test:

$$||\mathbf{x}_\beta - (\mathbf{x}_\alpha + \mathbf{v}t)|| = \rho_\alpha + r_\beta \tag{1.8}$$

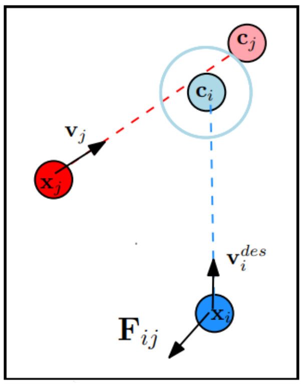
with $\mathbf{x}_\alpha$ and $\mathbf{x}_\beta$ the positions of the pedestrians and the relative estimated velocity between α and β. This equation has to be solved for t. When there is no or one solution then no collision takes place, if there are two solutions for t and at least one of them is positive, pedestrian β can be added to the set of pedestrians set for collision with α. These pedestrians are sorted by increasing collision time and the first N are kept. The direction of the evasive force applied to α to avoid pedestrian β is given by the following unit vector:

$$\mathbf{n}_{\alpha\beta} = \frac{\mathbf{c}_\alpha - \mathbf{c}_\beta}{||\mathbf{c}_\alpha - \mathbf{c}_\beta||} \tag{1.9}$$

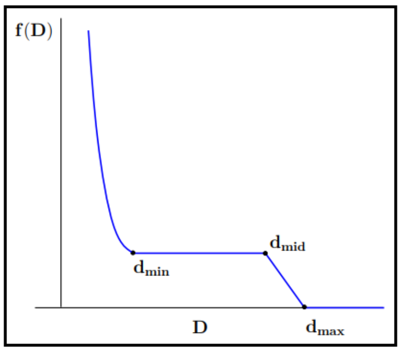
where $\mathbf{c}_\alpha$ and $\mathbf{c}_\beta$ are the positions of the pedestrians at the time of the collision, this direction is illustrated in Figure 1.5a. The magnitude of the force is approximated by piecewise function $f(D)$ with:

$$D = ||\mathbf{c}_\alpha - \mathbf{x}_\alpha|| + (||\mathbf{c}_\alpha \mathbf{c}_\beta|| - r_\alpha - r_\beta) \tag{1.10}$$

An example of how function f looks is given in Figure 1.5b. The threshold d_{max} determines the start of the avoidance maneuver, whereas d_{min} defines the beginning of an impenetrable barrier between the pedestrians. The threshold d_{mid} regulates the start of the constant part of the function. This part is used to eliminate jerky behaviour.
To compute the total evasive force exerted on pedestrian α, the evasive force of each pedestrian β in the set of N pedestrians with future collisions is iteratively applied to pedestrian α. Once one future collision's evasive force is applied it is checked whether α still is set to collide with any of the other pedestrians. If this is the case the evasive force for the next pedestrian is computed and added to the total evasive force on α and the process is repeated. If no more collisions occur the total evasive force $\mathbf{F}_e$ is set to the average of all the evasive forces that were applied to α. Experiments have confirmed that by applying the forces sequentially smoother and more realistic avoidance behaviour is achieved, as forces are only added if they are still required to avoid further collisions.



(a) Direction of the repulsive force used to steer pedestrians away from each other.



(b) Piecewise function that approximates the magnitude of the evasive force.

Figure 1.5: Key concepts of Karamouzas et al. Predictive collision avoidance model.

Collision-Free Velocities

Whereas the previous methods looked for upcoming collisions between agents to generate forces to steer them away from the collision, this type of method solves collisions by looking for a velocity to assign each agent so that they would not have any collisions in the near future.
One such method was developed by Paris et al [44], who generate new orientations and velocities for agents by exploring the reachable space for that agent in any direction and for a range of speed values, and search for possible collisions with neighboring agents. Their model works by first considering all neighbouring dynamic pedestrians for an agent, deducing a set of speed and orientation values that allow collision free motion. The same is done for static obstacles, to then merge the valid solution ranges. Finally the solution regions are scored and compared to select the most suitable one.
To find valid orientation ranges that avoid collisions with dynamic neighbours, time needs to be discretized. This is done by considering successively adjacent intervals having different durations: $[0, k^0\Delta t], [k^0\Delta t, k^1\Delta t], [k^1\Delta t, k^2\Delta t] \dots$ with Δt defining the precision of the discretization, and k ensuring that subsequent time intervals are always larger, since less precision is required for events further in the future. Considering one neighbouring agent at the time we compute a range of orientations for each time interval that could lead to a collision with that agent in that time interval. For each of these orientation sections the critical speeds $\mathbf{v}_1$ and $\mathbf{v}_2$ are computed, the maximal speed the agent is allowed to move to avoid a collision and the minimal speed the agent needs to move to avoid collision respectively. These values are computed using the following formulas:

$$\begin{aligned}\mathbf{v}_1 &= \min_{t=t_1}^{t_2}\left((||\overrightarrow{\mathbf{p}_\alpha \mathbf{p}_\beta(t)}|| - R)/t\right)\\ \mathbf{v}_2 &= \max_{t=t_1}^{t_2}\left((||\overrightarrow{\mathbf{p}_\alpha \mathbf{p}_\beta(t)}|| + R)/t\right)\end{aligned} \tag{1.11}$$

with $t1$ and $t2$ the begin and end times of the orientation range, $\mathbf{p}_\alpha$ the position of the reference agent, $\mathbf{p}_\beta(t)$ the linear extrapolation of the position of the neighbour agent at time t and R the sum of radii of the agents, increased by an extra security factor. This leaves us with a number of orientation sections defined by time interval $[t1, t2]$, orientation interval $[\theta_1, \theta_2]$ and critical speeds $\mathbf{v}_1$ and $\mathbf{v}_2$, an example of this is shown in Figure 1.6. Finally overlapping sections of the orientation ranges are merged to create a number of adjacent non-overlapping sections.

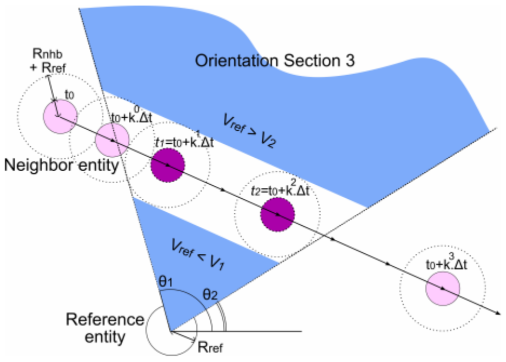


Figure 1.6: Example of a reachable space sectioning for a given time-interval.

As was the case for dynamic entities, a number of orientation ranges are computed for the static obstacles in the scene, only the process is simpler, as time is no longer considered. To compute this set of ranges the obstacle – which is modeled as a line – needs to be subdivided. For this point $\mathbf{p}_0$ is first computed, the point on the line closest to the reference agent. Points $\mathbf{p}_1$ and $\mathbf{p}'_1$ are defined so that the length $\mathbf{p}_0\mathbf{p}_1 = \mathbf{p}_0\mathbf{p}'_1 = k^0 \Delta t v_{ref}$, points $\mathbf{p}_2$ and $\mathbf{p}'_2$ are defined so that $\mathbf{p}_0\mathbf{p}_2 = \mathbf{p}_0\mathbf{p}'_2 = k^1 \Delta t v_{ref}$ and so on. The result of this is shown in Figure 1.7. For each orientation section i only maximal speed $\mathbf{v}_1$ is computed:

$$\mathbf{v}_{1,i} = ||\overrightarrow{\mathbf{p}_i\mathbf{p}_\alpha}||/t_{2,i} \tag{1.12}$$

minimal speed $\mathbf{v}_2$ is set to $+\infty$ as it has no meaning.

The final step of their method is to extract a solution move from the generated orientation sections. This is done by first weighting each computed section by a cost function. This cost function considers the following factors:

- The speed ranges $\mathbf{v}_1$ and $\mathbf{v}_2$ need to be close the the agents desired speed and need to be within the allowed speed range.
- Orientation limits of the section need to be close to the desired orientation of the agent.
- Required acceleration needs to be as small as possible to generate smooth movement.

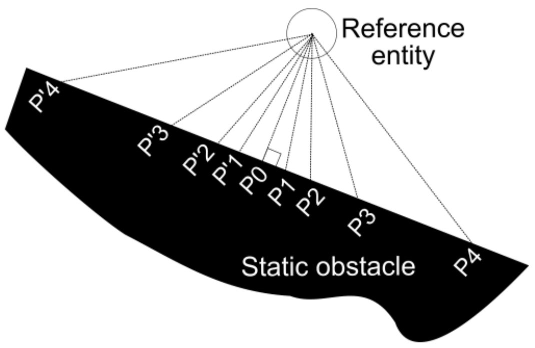


Figure 1.7: Sections computation for static obstacles

- The closer the time interval is in the future the more confident we can be in the cost.

At this point there are as many sets of sectionings as there are static (obstacles) and dynamic (agents) entities, all these weighted sections are now merged in the way described earlier. The cost of each subsection created is the sum of all the sections that were superposed and split to create it. We now select the section with the lowest value for the cost function, and compute an orientation and velocity – in the ranges available in this section – that are the closest to the desired ones by the agent.

Agent based approaches are able to generate very complex and unique behaviours for all agents in a crowd, with them being able to handle difficult scenarios. These approaches are ideal when the crowd is small displaying a lot of detail in the separate agents trajectories. Most of these models are however, computationally very expensive, especially the second-order ones. This makes them less than practical for large crowds. Since we require less detail in larger crowds it makes no sense to sacrifice this computational cost for higher quality trajectories.

1.2 Entity-based

Entity-based approaches model agents as homogeneous entities whose movements are influenced by some global/local laws that are introduced to represent various physical/social/psychological influences on an individual's movement in a crowd.
Most entity-based methods work using cellular automata. Cellular automata are discrete models consisting of a regular grid of cells, each in one of a finite number of states, such as on and off. These can be applied to crowd simulation by considering the space of our simulation as a two dimensional cellular automata. A cell can be occupied by an agent or by an obstacle, and the algorithms in this section of crowd simulation define the rules that cause an agent to switch cells.

One such model is Schadschneider's Cellular Automaton Approach to Pedestrian Dynamics [53]. This method divides the space in $40 \times 40\,\text{cm}^2$ squares that can all be occupied by zero or one agent. At each timestep it it computed for every agent where he should move, this is done using the following components.
Preference Matrix $\mathbf{M}_{ij}$: Each particle is given a preferred walking direction. From this direction a 3×3 preference matrix is constructed that contains the probabilities for an agent to move in a certain direction. Figure 1.8 shows this transition matrix and the directions the matrix cells represent.

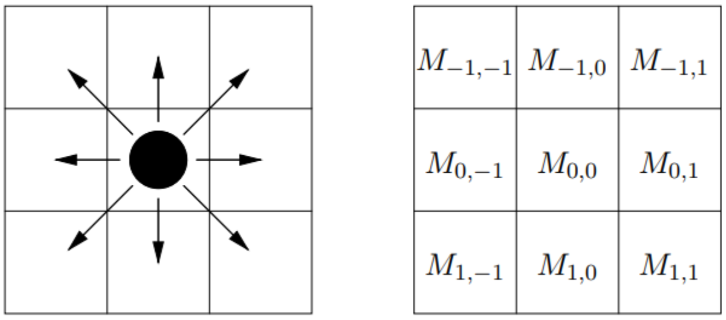


Figure 1.8: Illustration of an agents preference matrix and the directions the cells in the matrix represent.

Floor field: The floor field is a second grid underneath the grid occupied by the agents. The floor field models the interactions among agents and between agents and buildings by modifying the transition probabilities. There are two types of floor fields:

1. **Dynamic floor field $\mathbf{D}_{ij}$:** This is a virtual trace left by agents as they pass over cells by increasing their transition probability. This is done to achieve crowd interaction effects such as lane formation. The dynamic floor field is subject to decay en diffusion to make the traces disappear.

2. **Static floor field $\mathbf{S}_{ij}$:** This field does not evolve with time and is not changed by the presence of pedestrians. It is included to model the attractiveness of particular locations such as emergency exits or shop windows.

Having defined the three factors that influence an agents decision making we can formulate the probability p_{ij} that an agent moves in the (i, j) direction:

$$p_{ij} = N\mathbf{M}_{ij}\mathbf{D}_{ij}\mathbf{S}_{ij}(1 - n_{ij}) \tag{1.13}$$

where N is a normalization factor to ensure that all probabilities sum to one, and n_{ij} is the occupancy of cell (i, j) which is zero if the cell is empty and one if there is an agent in this cell. If more than one particle share the same target cell, one is chosen according to the relative probabilities with which each particle chose their target. This particle moves while its rivals for the same target keep their position.

Entity-based models allow us to simulate reasonably large crowds while still being able to observe individual movement. This movement is however, less detailed than that of the agents in agent-based approaches. For very large crowds these methods might still be too computationally expensive, the solution here is to completely sacrifice individuality of the agents by employing a flow-based method.

1.3 Flow-based models

Flow-based approaches simulate only abstract behaviour of large crowds instead of modeling individualistic behaviours. Crowds are modeled as a continuous flow as agents do not react to inputs from their surroundings. Typically, with the flow-based approach, vector fields are used to represent the impact of various environmental factors on the movement of the crowd, and the movement of a crowd is described using some differential equations. To form up these vector fields and differential equations, certain hypothesis and statistical assumptions are needed, whose validity are often debatable [73]. In this section we discuss three diffent approaches to flow-based crowd simulation: methods using graphs, methods using a grid of tiles and methods using the equations of fluid mechanics.

1.3.1 Graph models

A basic example of one of these graph-using flow-based methods is EVACNET4 [32], a simulation tool designed to model building evacuations. The model uses a description of a building and an initial occupation to calculate an optimal evacuation of the building, minimizing the evacuation time. Buildings in EVACNET4 are modeled as graphs in which the nodes represent building components such as rooms, halls and stairs. The arcs represent the passageways between these components.
For each node an initial amount of people and capacity need to be defined, the capacity is the upper limit on the number of people that can be contained in the building component the node represents. For each arc an arc traversal time and arc flow capacity need to be specified. The traversal time is the number of time periods it takes to traverse the passageway the arc represents. The arc flow capacity is the upper limit on the number of people that can traverse the passageway the arc represents per time period. Figure 1.9 shows the network description of a two story building.
Information about the evacuation procedure is calculated using an advanced capacitated network flow transshipment algorithm, a specialized algorithm used in solving linear programming problems with a network structure. EVACNET4 is a tool that offers great flexibility in terms of architecture, but it fails to model behavioural aspects of crowds besides just movement. And even the movement is greatly simplified as the speed of the crowd is treated as a piece wise constant, since no vector fields and differential equations are used.

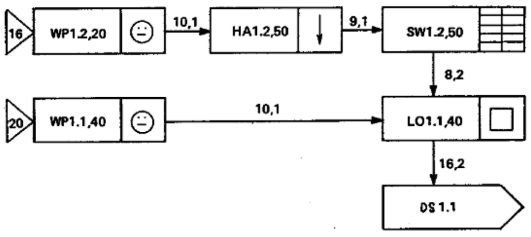


Figure 1.9: EVACNET4 Representation of a two story building.

1.3.2 Tile-based models

Tile-based algorithms discretise the environment into a grid, for which we can precompute the motion at each tile. An example of this is Cenney's Flow Tiles [9]. Each flow tile defines a small, stationary region of velocity field. They can be pieced together to form large stationary fields and then used to drive various flows such as fluids and crowds. In terms of crowd modeling and simulation, the flow tiles technique aims at producing visual effects of crowd motion rather than studying the dynamics of crowd.
Tiles are characterised by four corner velocities and four edge fluxes. These edge fluxes adhere to the following constraint to ensure divergence free tiles: $f_{bottom} + f_{left} = f_{top} + f_{right}$. In a divergence-free field there are constraints upon the velocity values. This is exploited to decrease the storage cost. Every 2D divergence-free incompressible field can be represented as the curl of a scalar function, the stream function $S(x, y)$, multiplied by a vector in the third dimension, $\mathbf{z}$:

$$\mathbf{v}(x, y) = \nabla \times (S(x, y)\mathbf{z}) \tag{1.14}$$

This both reduces the memory cost from $2n(n+1)$ to $(n+1)^2$ for an $n \times n$ cell grid, and makes it is simpler to construct tiles with specific fluxes across the edge. The velocity $(\dot{x}, \dot{y})$ at point (x, y) is found by interpolating the stream function to find $S(x - 0.5, y)$, $S(x + 0.5, y)$, $S(x, y - 0.5)$ and $S(x, y + 0.5)$. Then:

$$\begin{aligned} \dot{x} &= S_{x,y-0.5} - S_{x,y+0.5} \\ \dot{y} &= S_{x+0.5,y} - S_{x-0.5,y} \end{aligned} \tag{1.15}$$

This process is illustrated in Figure 1.10. The values of the stream function can be initialized using just the flux at the edges and velocities in the corners, which are already given for each tile. Consider filling a tile of size $n_x \times n_y$, with indices into the stream function in the domain $(-0.5, \ldots, n_x + 0.5) \times (-0.5, \ldots, n_y + 0.5)$. Using the fluxes assigned to this tile, we set the corner values:

$$\begin{aligned} S_{-0.5,-0.5} &= 0 \\ S_{n_x+0.5,-0.5} &= S_{-0.5,-0.5} + f_{bottom} \\ S_{-0.5,n_y+0.5} &= S_{-0.5,-0.5} + f_{left} \\ S_{n_x+0.5,n_y+0.5} &= S_{-0.5,n_y+0.5} + f_{top} \end{aligned} \tag{1.16}$$

Using the given velocities and Equation 1.15 we can set the three stream values surrounding each corner. For example:

$$\begin{aligned} S_{0.5,-0.5} &= S_{-0.5,-0.5} + \dot{y}_{0,0} \\ S_{-0.5,0.5} &= S_{-0.5,-0.5} - \dot{x}_{0,0} \\ S_{0.5,0.5} &= S_{-0.5,0.5} + \dot{y}_{0,0} \end{aligned} \tag{1.17}$$

The above construction around each corner results in 16 known stream values. A bicubic Bezier patch is used to fill the remainder of the grid. A linear system is solved to find the control points for the patch that interpolates the values around the corners, the patch is than evaluated at other grid locations to determine the remaining stream function values. The result is a smooth, continuous flow within the tile. Having described the dynamics of the flow tiles, the problem of laying out tiles to meet continuity and boundary conditions remains. To solve this the paper describes a tiling algorithm that returns a set of tiles given a partial tiling and an unfilled location. The selection operation guarantees that the tiling can be completed if one of the selected tiles is used in the given location.
In tile-based algorithms like Chenney's, agents are advected following the same flow fields causing them to have no local collision-avoidance. Which could lead to unrealistic behaviour, especially at higher crowd densities. It also makes it difficult to model scenarios in which groups of agent have specific goals or where intersecting flows occur.

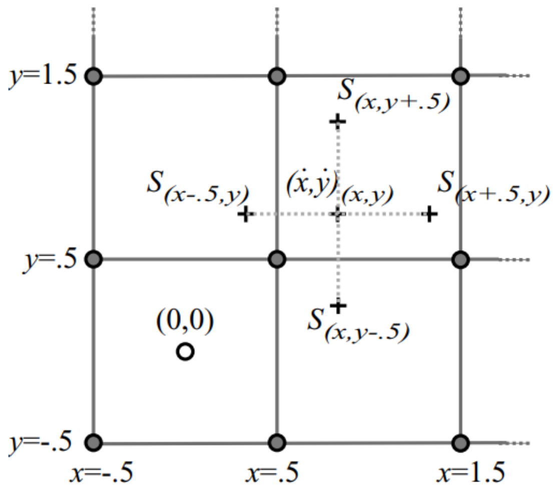


Figure 1.10: The stream function method for representing divergence-free velocity fields.

1.3.3 Fluid models

Another approach to flow-based methods is to consider the intelligence of crowd flows. In these works, crowds were modeled as a fluid with thinking ability in an attempt to include factors like pedestrians' moving tendency in a crowd.

Hughes' fluid model

A basis for a lot of these fluid models was made by Hughes [27] who – in consultation with behavioural scientists – derived a pair of nonlinear partial differential equations that govern the flow of a crowd, representing it as a continuous density field. The system is driven by an evolving potential function, defined so as to guide the density field optimally toward its goal. The equations are based of three hypotheses governing the motion of a crowd:

1. The speed at which pedestrians walk is determined solely by the density of the surrounding pedestrians, the behavioural characteristics of the pedestrian and the ground on which they walk.
2. Pedestrians have a common sense of the task (called potential) that they face to reach their common destination, such that any two individuals at different locations having the same potential would see no advantage in exchanging places.
3. Pedestrians seek to minimize their estimated travel time but temper this behaviour to avoid extreme densities. This tempering is assumed to be separable, such that pedestrians minimize the product of their travel time as a function of density.

The above hypotheses lead to the basic governing equations for the flow of a single pedestrian type walking on isotropic topography. These equations are:

$$-\frac{\partial \rho}{\partial t} + \frac{\partial}{\partial x}\left(\rho g(\rho) f^2(\rho)\frac{\partial \varphi}{\partial x}\right) + \frac{\partial}{\partial y}\left(\rho g(\rho) f^2(\rho)\frac{\partial \varphi}{\partial y}\right) = 0 \tag{1.18}$$

and

$$g(\rho)f(\rho) = \frac{1}{\sqrt{\left(\frac{\partial\varphi}{\partial x}\right)^2 + \left(\frac{\partial\varphi}{\partial y}\right)^2}} \tag{1.19}$$

where φ is the remaining travel time, which is a measure of the remaining task (called potential), ρ the density of the crowd, $f(\rho)$ is the speed of the pedestrians as a function of the density, $g(\rho)$ is a factor related to the discomfort of the crowd at a given density, and (x, y, t) denotes the space and time coordinates. With these governing equations it is necessary to specify the forms of the speed and discomfort functions $f(\rho)$ and $g(\rho)$. A full derivation of these equations was given by Hughes in *"A continuum theory for the flow of pedestrians"* [26].

Continuum crowds

Hughes only investigates analytic properties of his equations and does not discuss simulation. This was done by Trueille et al. [64] in their work: Continuum Crowds.
Continuum Crowds is a realtime motion synthesis model for large crowds. The model is inspired by Hughes' work in the sense that it uses a similar potential function to guide crowds towards their goals. The model unifies global path planning and local collision avoidance into a single optimization framework. Four hypotheses are made about crowds:

1. Each agent is trying to reach a geographical goal.
2. Agents move at the maximum speed possible.
3. Agents prefer some places to others.
4. Agents will choose a path by minimizing three factors:
    - Distance travelled
    - Time spent
    - Discomfort felt

   how important each of these factors are, is different for each group of agents.

Because it is impossible to evaluate the value of the fields at each point the simulation environment is discretised into a 2D grid. For each tile in the grid a number of fields are evaluated in two categories: per environment fields and per group fields. These values are either stored as a value for the cell itself or as a value pertaining to the transition between the cell and one of its neighbouring cells. The per environment fields are constant for each agent in the cell and only need to be computed once per timestep. They include:

- **Height field**, containing the terrain height for each field as well as slope between the cells
- **Discomfort field**, the discomfort value for each cell, specified by the designer and can vary over time
- **Density field**, the density of the crowd at each cell. Each person has a non-zero contribution to the density of the four closest cells.
- **Average velocity field**, containing the average velocity of the crowd at each cell

The agents in the crowd are divided into groups, where every person in the same group shares a common goal. The following fields are evaluated for each grid tile for each group:

- **Speed field**, contains the speed that a person can travel from one cell to another, stored at the face of each cell. The speed is made up of two components: the topographical speed, which depends solely on the slope of the terrain and the flow speed, which depends on the average speed of the crowd.
- **Cost field**, contains the cost of travelling from one cell to another. The value depends on the three factors discussed in hypothesis four.
- **Potential field**, contains the potential at each cell, and also the gradient of potentials between cells. the potential is defined as an implicit eikonal equation: $\|\nabla\phi\| = C$, with ϕ the potential and C the cost. This equation is not directly

solvable, but it can be evaluated using the fast marching method [55]. After applying this numerical method, each cell has a potential, which in turn can be used to calculate the potential gradient between cells.

- **Velocity field**, contains the velocity at each cell face. The velocity is calculated by multiplying the speed by the potential gradient.

Once the velocity field has been computed, the velocity of each person in the crowd can be assigned, by interpolating between the velocity values in the surrounding cells. Each person's position is then updated using Euler integration. Figure 1.11 shows an overview of the algorithm.

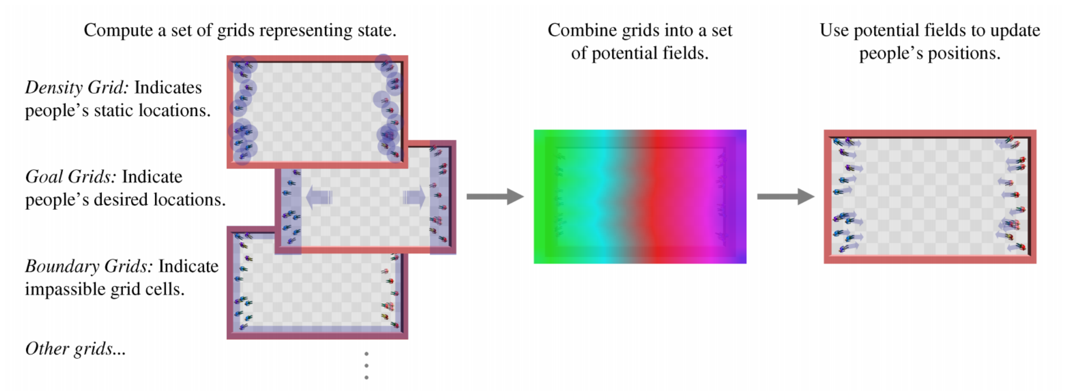

Compute a set of grids representing state.
Density Grid: Indicates people's static locations.
Goal Grids: Indicate people's desired locations.
Boundary Grids: Indicate impassible grid cells.
Other grids...
Combine grids into a set of potential fields.
Use potential fields to update people's positions.

Figure 1.11: Overview of the Continuum Crowds algorithm.

The sacrifice of individuality in flow-based approaches makes these methods ideal to simulate large crowds with a high density, as it makes them computationally very efficient since no complex calculations need to happen at agent level. This does however lead to unrealistic simulations when the individual behaviour within the crowd in normal or emergency situations is a concern. It is clear from the previous three sections that before using a crowd simulation method one should first determine the context in which it will be used, as all methods offer different benefits, useful for different situations.

1.4 Data-driven models

The previously described models all use a set of rules or formulas to simulate the behaviour of humans, but this often limits the behavioural complexity of the models or leads to unnatural looking simulations. Data-driven models attempt to learn crowd behaviours from real-world sources such as motion capture or camera footage, in the hopes of creating more complex and realistic looking simulations. In this section we present two models that indicative of how most the data-driven models work. It should be noted that this is not a different category form the previous three as these models still fall under one of the main classes. The two methods presented here fall under the agent-based category.

1.4.1 Query method

In Crowds by Example [35], Lerner et al. propose an example based method of crowd simulation. They construct a database of examples from a set of trajectories extracted from video images of a real crowd. An example is the behaviour of a specific person and the given surroundings that influenced that behaviour. During the simulation they look for similar situations to those of the agents in their simulation. These situations are then extracted from their database.
An example consists of a segment of the trajectory the agent in question is following and of a configuration of all the factors that might have influenced the trajectory. These can be other agents or obstacles. For each influencing factor the path is also stored (if it is an agent) along with the amount of influence this factor is believed to have. Each example is defined using a local coordinate system where the agent is at (0,0) facing the Y-axis. The influence of other agents is determined by the distance between them and whether they are in front of behind them, furthermore the influence is evaluated over a time period.
During the simulation the process repeatedly considers for each of the participating agents whether a new trajectory is required. This is the case if the agent has either depleted the last trajectory segment it was given, or if the agents surrounding configuration has changes significantly since the last time it was assigned a trajectory. An overview of this process is shown in Figure 1.12.

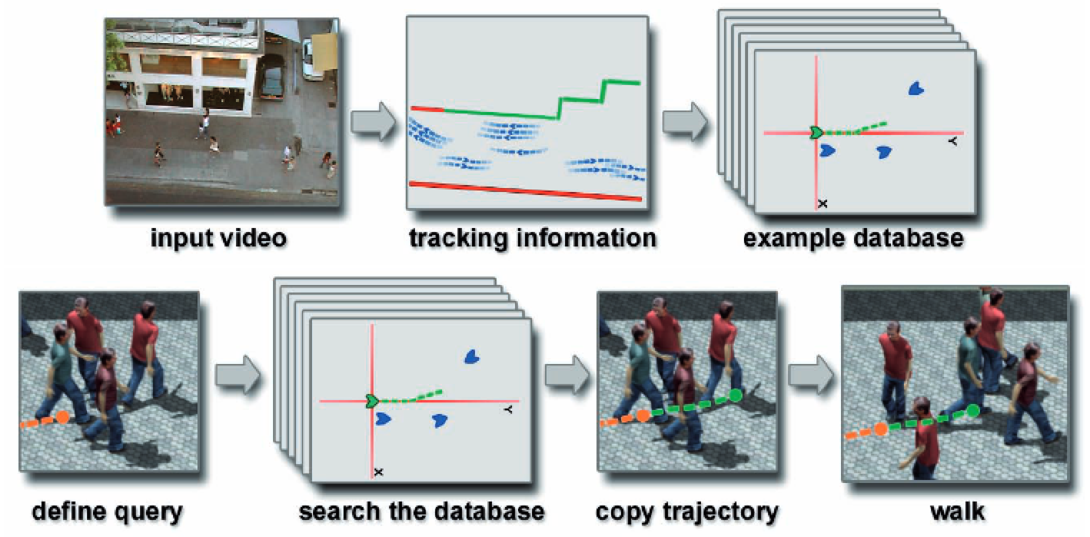


Figure 1.12: An overview of the Crowds by Example method

1.4.2 Locally weighted linear regression

Lee et al. [34] proposed a method based on applying locally weighted regression to state-action pairs of agents in a similar situation to the current agent at runtime to generate

actions.
To do this they extract trajectories from real life data, along with annotations of high-level behaviour patterns. From these trajectories state-action pairs are extracted. With the states being made up of the agents own speed, the formation of the agents neighbourhood, a pivot (objects or locations of interest) and the intended moving direction. The action is simply the instantaneous velocity at the next moment. Principle component analysis [29] (PCA) is used to reparameterize the state space in a low-dimensional space for efficient learning of behaviour models. PCA is a mathematical procedure that transforms a number of possibly linearly correlated variables into a, hopefully smaller, number of uncorrelated variables called principal components. This removes unnecessary data which makes it easier for the network to be trained.
The behaviour model is a combination of high-level behaviour models and low-level action models. A certain high-level behaviour model consists of a set of low-level action models. The state-action trajectories acquired from the processed video are segmented and classified into groups according to their annotated behaviour patterns, and each group is used to learn a corresponding action model. That action model being a function that takes state and produces an action for that state.
These action models are constructed using locally weighted linear regression [1]. This means that given a specific state for which we want to generate an action, we first search a number of samples similar to this state within the current high-level behaviour model, and fit our linear regression model to these state-action pairs. Similar samples could still have more than one output possibility, this is a problem, since locally weighted linear regression requires the model to be Markovian. To solve this problem k-means clustering is applied to the set of similar samples to sort them into clusters with similar output actions. A cluster is then probabilistically selected. On this cluster the locally weighted linear regression is applied to generate an appropriate action.
The high-level behaviour model is implemented as a finite state machine, in which each action model corresponds to a state. To reproduce global behaviour patterns observed in training video, the transition conditions between action models and control parameters are encoded, such as the average duration of staying in each action model. These control parameters are measured from crowd video.

1.5 Neural Network Methods

A number of works already exist that apply the power of neural networks to the problem of crowd simulation. In this section a few of these works are presented to give an overview of the progress already made in the field. If the reader is not familiar with the concepts of neural networks it is recommended to first read Chapter 2. The methods presented here are all based on supervised learning as this is the approach taken in this thesis. However, there exist reinforcement learning approaches to crowd simulation, such as Lee et al. [33] who propose an actor (policy) trained by a critic (value function), both represented by deep neural networks. Another interesting example is the work of Widmer [68], who used the genetic algorithm NEAT (Neuro-evolution of Augmenting Topologies [62]) to evolve

different generations of agents navigating the scene, controlled by neural networks.
There are two distinct methods in supervised-learning neural network methods. The first type makes use of linear regression to generate new actions based on the current state of the agent, whereas the second type classifies the current state of the agent to then pick an action form the samples form the database that match this category. We present a number of methods from both approaches in the rest of this section.

1.5.1 Regression Methods

Wei et al. [66] proposed a feedforward neural network method that uses supervised learning to fit behaviours from real crowd data to a crowd simulation. Firstly trajectories are extracted from crowd videos, these trajectories are used to generate the state-action pairs with which the neural network is trained.
Each state is defined by dividing the field of view around the agent at a certain moment in N parts. For each part the relative position relationship between the closest neighbour in that part and the agent is included in the state. Figure 1.13 shows a visual representation of how the agents FOV is divided. In order to include previous movements of the agents neighbours this information is repeated in the state for all the previous M frames. Besides this, the agents current speed and angle between the current speed and the agents ultimate goal is also part of the state. The state is thus defined as $state \in \mathbb{R}^{M \times N+2}$. The action consists of the speed in the x and y direction in the next step. The state vector will contain a lot of null items as agents will often not have a neighbour for a specific area. To combat this principal component analysis is applied to the state to reduce the dimensionality.
Because two agents with roughly the same state will often pick different actions due to the intricate nature of the human thought process, the neural network can have trouble learning from the data as it contains contradictions. Consider the situation in which a person is walking towards a goal that is straight ahead of them. On their path is a person standing still, to avoid collision some people would pass the person via the left and some people pass via the right side. Both are valid options and could appear in our dataset. To combat this Wei et al. used K-nearest neighbour clustering on the cosine similarity of the states of each sample. Each sample is also given the actions of the k state-actions pairs with the most similar states. This leads to the neural network outputting multiple actions.
At run-time the state for each agent is computed and fed into the trained network. The network outputs several action options and an appropriate one is chosen by sequentially going through all of the actions and selecting one that does not collide with other agents. If none is found the current action is kept, which will lead to a collision.

A crowd simulation scene does not only contain other agents, there are also obstacles like walls that need to be avoided by agents. The previous paper did not take path planning to navigate around these obstacles into account.
Song et al. [60] propose a method that uses the angle between the agents moving direc-

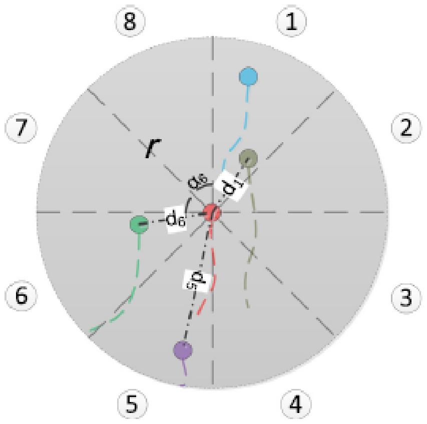


Figure 1.13: Illustration showing the FOV for identifying the closest neighbour for each section. The FOV with radius of r is evenly divided into $N = 8$ sections. d_n is the distance from the nearest neighbor in section n [66].

tion and the current goal of the precalculated A* path instead of the angle between the current moving direction and goal.

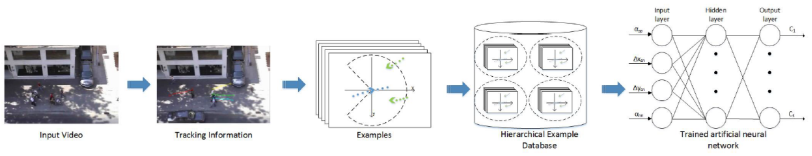
Instead of using the relative position relationship in the state this method uses the relative positions and speeds of the five closest neighbours in a set field of view (FOV). This way it is not necessary to include information of the previous frames in the state, as this can be estimated by the speeds. This leads to a decrease in the state space size.

The walking or running directions of pedestrians are continuous. This incurs the following problem: it is impossible for an artificial neural network to learn all the possible directions because it is continuous. The solution to this problem is to introduce a transfer layer. In this layer all the pedestrians' directions are translated counter-clockwise to the x-axis, leaving only a speed scalar. All the other parameters like the A* direction, output speed and information about the neighbours is rotated by the same angle.

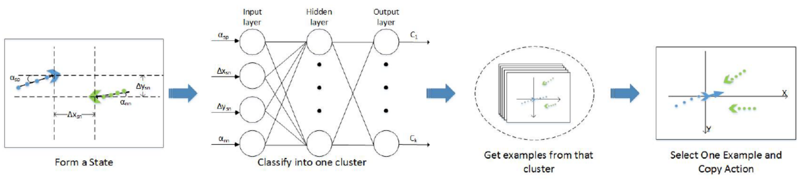
1.5.2 Classification methods

Instead of using the neural network as a method to generate the velocity at the next step, Zhao et al. [72] trained a classifier to identify examples with a similar state from their database when simulating agent behaviour. They generate state-action pairs in a fashion similar to the previous two methods. These examples are then grouped in clusters by their state similarity using agglomerative hierarchical clustering algorithm [12]. This algorithm works bottom-up, starting with individual objects as clusters, the two closest clusters are merged until only one cluster remains. This algorithm ends with one cluster that is the base of a tree-structure that needs to be partitioned into a number of clusters. The silhouette coefficient [49] is used to obtain accurate partitioning. This is a measure of how similar an object is to its own cluster compared to other clusters and is used to evaluate the clustering result at different cutting levels and select the most precise one

among them.
After clustering, every example in the database is assigned its cluster membership. An artificial neural network classifier is trained with example states as input and their cluster membership as output. During the simulation this network is used to find examples similar to to the agents current state. A suitable action is then chosen from all of the actions in this cluster by picking one that avoids collision with other agents. The processes in this model are summarized in Figure 1.14.
This methods uses neural networks in a more traditional way as it picks an option from a discrete space, where as the previous two papers generate a 2D vector from a continuous state space. This makes the network easier to train, the downside of this method is that all of the actions for each cluster have to be kept in a database at simulation time. An algorithm for choosing a correct action also needs to be implemented.



(a) First the simulation samples get extracted from input videos and clustered into groups. A neural network classifier is then trained to classify new samples in these groups.


Form a State
Input layer
Hidden layer
Output layer
Classify into one cluster
Get examples from that cluster
Select One Example and Copy Action

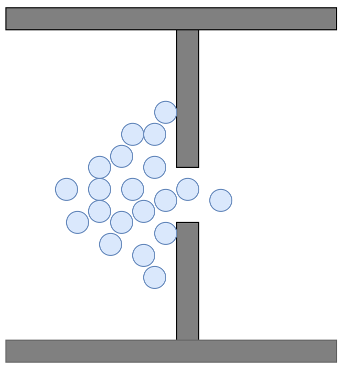
(b) During simulation states are extracted from the simulation, the classifier determines which cluster the state belongs to and an appropriate action is selected from that cluster.

Figure 1.14: The two processes used in the model proposed by Zhao et al. [72].

1.6 Evaluation of Crowd Simulation methods

An important part in the process of developing a crowd simulation method is evaluation. The requirement of simulating a realistic crowd is quite subjective and ambiguous, making evaluating and comparing crowd simulation methods no trivial task.
A first option is the qualitative analysis of the simulation results of crowd scenarios for which we know how a human crowd would behave. One such scenario is where multiple agents attempt to pass through a doorway at the same time, this is shown in Figure 1.15a. In this scenario a real crowd would pile on in a circular formation while only a few agents slowly seep through. Another such scenario is when two groups in a crowd run into each other. In this case the agents will display line-formations as shown

in Figure 1.15b.
Another approach to validating crowd simulation methods is the comparison to data captured of real crowds. This can be done by a number of metrics, such as the difference between simulated and captured agents or more advanced entropy metrics that compare the actions taken by simulated and recorded pedestrians at each step of their trajectories [22]. Chattaraj et al. [8] defined a fundamental diagram that illustrates the speed of agents in a flow as a function of their surrounding density which can be used to compare between simulated and captured crowds.
In an attempt to standardize the evaluation of steering algorithms, and by extension crowd simulation techniques, Singh et al. [59] proposed a framework consisting of a number of metrics and test scenarios that can be used to compare between different algorithms.


(a) Validation scenario where many agents attempt to pass through a doorway

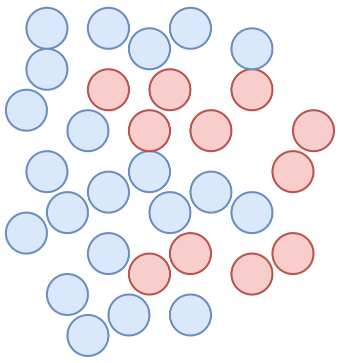

(b) Validation scenario where two groups of agents move in the opposite direction and form lanes.

Figure 1.15: Two examples of scenarios in which human crowds display certain behaviour that can be used to validate crowd simulation methods.

Chapter 2

Artificial Neural Networks

In our method for crowd simulation we will use an artificial neural network (ANN) to generate the movements of the agents in our crowd based on their surroundings. In order to effectively design an ANN for our purposes we will first need to explore the general concept of ANNs and how to train them.
"Artificial neural networks" is a relatively loose term referring to mathematical models which have some kind of distributed architecture [3]. The neural network itself is not an algorithm, but rather a framework for many different machine learning algorithms to work together and process complex data inputs. The power of neural networks comes from the fact that they can learn to perform tasks without explicitly being programmed to do so. They can learn from input-output pairs fed into the network, or they can use reinforcement learning, given some function that defines a good output for a specific input. This way, a neural network can even be trained by another neural network that generates difficult inputs for the other network. As we are building a data-driven model the focus in this chapter will be on the former type of learning.

2.1 Basic concepts of ANNs

An artificial neural network is a computational learning system that uses a network of functions to understand and translate a data input of one form into a desired output. The network is built by several layers of neurons that are connected with each other. Figure 2.1 displays a simple multi-layer feedforward neural network. A feedforward neural network is a type of network in which there are no cycles formed by the connections between neurons. Later in this chapter we will explore a different type of network, in which cycles are allowed.
The nodes in this type of network are organized in layers, depending on their order in the chain. Typically there are three types of layers in a feedforward network:

- **Input layer** This layer simply takes the input values for the network and passes them on. There is no computation done in this layer.

- **Hidden layers** There can be zero or more of these in a feedforward network, they are located between the input and output layers and perform computations.
- **Output layer** In this layer the final computation is done and the data is moved on to the output.

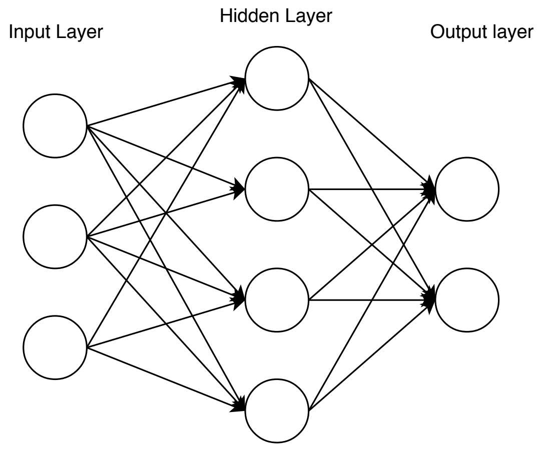


Figure 2.1: A simple multi-layer feed-forward neural network.

When computing output values with a neural network the input values are passed from layer to layer following the network structure until they end up at the output layer. Each time a value passes from neuron to neuron it gets multiplied by a weight that is specific to the connection between the two neurons.
The task of each neuron is to take the weighted sum of its inputs and perform its activation function on that sum. The activation function is usually a simple nonlinear function that squashes the input value between some other values like e.g. [-1, 1]. More details of these function are given further on in this section. The result of this computation is then passed on to the neurons this neuron is connected to, who do the same. The activation functions introduce non-linearity in the computational model and allow for the network to learn complex structures. The computation at a neuron can be formalized by this equation that expresses the output of the j^{th} neuron in the l^{th} layer

$$o_j^l = f\Big(\sum_k w_{jk}^l o_k^{l-1} + b_j^l\Big) \tag{2.1}$$

with the sum being over all neurons k in the $(l-1)^{\text{th}}$ layer, w_{jk}^l the weight from the k^{th} neuron in the $(l-1)^{\text{th}}$ layer to the j^{th} neuron in the l^{th} layer and o_k^{l-1} the output of the k^{th} neuron in the previous layer. The bias b_j^l is introduced to give each neuron a trainable constant value that can shift the output of the function [65]. The bias can be modeled by a special neuron in each input layer that has no inputs and always gives an output of one.
We can make the previous equation more readable by rewriting it in a matrix form. For

this we need a weight matrix $\mathbf{W}^l$ for each layer. The entry in the j^{th} row and the k^{th} column of $\mathbf{W}^l$ is the weight from neuron k in the $(l-1)^{\text{th}}$ layer to neuron j in the l^{th} layer. Similarly, for each layer we define a bias vector, **b** that contains the bias for each neuron. We can now compute an output vector containing the outputs of each neuron in layer l

$$\mathbf{o}^l = f(\mathbf{W}^l \mathbf{o}^{l-1} + \mathbf{b}^l) \tag{2.2}$$

where $f()$ is the element-wise application of the activation function to the input vector. The activation function of a neuron defines its output given a set of inputs. This output is then passed on to the next neurons. It maps the resulting values into the desired range such as between 0 to 1 or -1 to 1 etc., depending upon the choice of activation function. Table 2.1 shows a few commonly used activation functions.

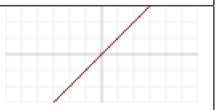
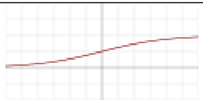
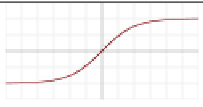
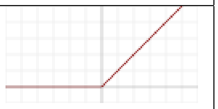
Name	**Formula**	**Range**	**Plot**
Identity	$f(x) = x$	$[-\infty, +\infty]$	
Sigmoid	$f(x) = 1/(1+e^{-x})$	$[0, 1]$	
TanH	$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	$[-1, 1]$	
ReLu	$f(x) = max(0, x)$	$[0, 1]$	

Table 2.1: Activation functions used in neural networks

2.2 Training Neural Networks

So far we have seen how a neural network computes output values from input values. The real usability comes when we change the weights of a network in a way that, given a set of input values, the output values represent values similar to the calculation we are trying to perform with the neural network. The changing of the weights to get better output results, is called training, it's how a neural network is able to learn.

2.2.1 Learning paradigms

Neural networks can be trained in several different ways, depending on whether input data and target data is available during training. There are three major learning paradigms that are all used in different scenarios, they are briefly discussed here. The one we will be focusing on during the rest of this chapter is supervised learning, as we are trying create a crowd simulation model based on data extracted from real crowds.

Supervised learning

In this learning method the neural network is handed an array of input-output pairs $(x, y), x \in X, y \in Y$, with the goal of learning the mapping $f : X \to Y$ between them. This is done by minimizing some loss function between the networks outputs $f(x)$ and the action outputs y in the dataset. The loss function is some quantity that describes how different the output of the network is from the target output, it is further discussed in Section 2.2.2. Common uses of supervised learning are regression and classification.

Unsupervised learning

In some cases we wish to learn the inherent structure of our data without using explicitly-provided labels, as we did in supervised learning. For this case we use unsupervised learning where some data x is given and the cost function to be minimized, that can be any function of the data x and the network's output, f. What this cost function looks like depends on the task, the implicit properties of the model, its parameters and the observed variables. Some examples of supervised learning problems are dimensionality reduction and clustering.

Reinforcement learning

Reinforcement learning is a framework which allows autonomous agents to learn how to perform tasks based on trial-and-error interactions with the environment. The data x are not given, but generated by the agents interactions with the environment. At each point in time, the agent performs an action y and the environment generates an observation y and an instantaneous cost c, according to some dynamics. The aim is to discover a policy for selecting actions that minimizes some measure of a long-term cost.

2.2.2 Loss function

To train a neural network we need a way to quantify how well the network is performing given the expected results. This is achieved via a loss function. The loss function – sometimes called the cost function – is a measure of the amount of error between a neural networks predictions $\hat{\mathbf{y}}$ make and the actual target values in the dataset $\mathbf{y}$. The goal of the training process is essentially to minimize this function with respect to the weights and bias of our network. A few of the most used loss functions are summarized below:

- **Sum of Squared Errors** simply computes the sum of all errors squared.

$$J = \sum_{i=1}^{n} (\mathbf{y}_i - \hat{\mathbf{y}}_i)^2 \tag{2.3}$$

- **Mean Squared Error** is one of the simplest and most effective cost functions that we can use. It can also be called the quadratic cost function or sum of squared errors.

$$J = \frac{1}{n}\sum_{i=1}^{n}(\mathbf{y}_i - \hat{\mathbf{y}}_i)^2 \tag{2.4}$$

- **Mean Squared Logarithmic Error** is a slight variance of MSE. By taking the logarithm of the predictions and actual values the measured variance changes. It is usually used when you do not want to penalize huge differences in the predicted and the actual values when both predicted and true values are huge numbers. MSLE also penalizes underestimates more than overestimates [23].

$$J = \frac{1}{n}\sum_{i=1}^{n}(\log(\mathbf{y}_i + 1) - \log(\hat{\mathbf{y}}_i + 1))^2 \tag{2.5}$$

- **L2** is the square of the L2 norm of the difference between actual value and predicted value. It is mathematically similar to MSE, only do not have division by n.

$$J = \sum_{i=1}^{n}(\mathbf{y}_i - \hat{\mathbf{y}}_i)^2 \tag{2.6}$$

- **Mean Absolute Error** is a quantity used to measure how close forecasts or predictions are to the eventual outcomes. Compared to MSE, MAE is more robust to outliers, as it does not make use of the square. This makes MSE more usefull if large errors should have large consequences for the result.

$$J = \frac{1}{n}\sum_{i=1}^{n}|\mathbf{y}_i - \hat{\mathbf{y}}_i| \tag{2.7}$$

- **Cosine Proximity** is a measure of similarity between two non-zero vectors of an inner product space that measures the cosine of the angle between them. It is thus a judgment of orientation and not magnitude.

$$J = -\frac{\mathbf{y} \cdot \hat{\mathbf{y}}}{||\mathbf{y}|| \cdot ||\hat{\mathbf{y}}||} = -\frac{\sum_{i=1}^{n}\mathbf{y}_i \cdot \hat{\mathbf{y}}_i}{\sqrt{\sum_{i=1}^{n}\mathbf{y}_i^2} \cdot \sqrt{\sum_{i=1}^{n}\hat{\mathbf{y}}_i^2}} \tag{2.8}$$

These are several examples of loss functions used in neural networks. This list is by no means exhaustive as there are many more of these functions used in a lot of different scenarios.

2.2.3 Gradient Descent

The next step in training or network is to find a set of weights and biases that minimize the loss for our training data. This is an optimization problem and can be solved in several different ways. For neural network training, gradient descent and its variants are widely used. In this section we will discuss how the algorithm works and how it can find the best weights for our network.

Gradient Descent Method

The general idea behind gradient descent is to start with a random set of parameters to then figure out in which direction the cost function steeps downward the most. Once that direction is found a small step in that direction is taken, the steepest descent is found again and another step is taken. This gets repeated until a minima is found.

To figure out which direction the loss steeps downward the most, it is necessary to calculate the gradient of the loss function with respect to all of the parameters. A gradient is a multidimensional generalization of a derivative; it is a vector containing each of the partial derivatives of the function with respect to each variable. In other words, it is a vector which contains the slope of the loss function along every axis.

Let's consider the loss function J with respect to the networks weights θ denoted as $J(\theta)$. We denote its gradient vector by ∇J, i.e.:

$$\nabla J(\theta) = \left(\frac{\partial J}{\partial \theta_1}, \frac{\partial J}{\partial \theta_2}, \cdots, \frac{\partial J}{\partial \theta_N} \right) \tag{2.9}$$

How the gradient is computed in neural networks is not straightforward, it is generally done by a method called backpropagation, which will be discussed in the next section. For now it suffices to know that it is an application of the chain rule of derivatives. The next step is to update the parameters in the direction of our gradient. How far we step in each direction is determined by the learning rate η.

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta) \tag{2.10}$$

This is the basic version of gradient descent and it is ideal for linear optimization problems. Due to the non-linearity introduced by the activation functions, the optimization of neural network weights is not a linear problem.

Gradient Descent in Neural Networks

This "vanilla" version of gradient descent is also called batch gradient descent (BGD). It is called this way because it operates on the entire dataset as a batch. But there are some issues with this algorithm when training neural networks:

- The loss function of a neural network is very complex with many hills an valleys. This causes it to have not just one minima, but many local minima. Regular gradient descent could easily get stuck in one of these local minima instead of the global minimum.
- BGD is too slow, as every parameter needs to be evaluated for every sample. In a complicated neural network with many parameters trained by a large dataset this will lead to a very slow training process.

- For a large dataset there will be a lot of redundancies that will not have a large effect on the gradient, making batch gradient descent unnecessarily expensive to estimate the gradient.

We can mostly solve these problems by modifying the gradient descent algorithm. A few of these modified versions are presented in the next paragraphs.

Stochastic Gradient Descent In stochastic gradient descent (SGD) the parameters are updated separately for each sample every epoch (one epoch is when the training algorithm has processed the entire dataset once), shuffling the dataset before each epoch. In doing so it does away with the redundancy of vanilla gradient descent. This method is usually much faster and can be used to learn on-line. SGD is a very noisy method, causing it to sometimes overshoot local minima, whereas BGD will always find the local minima of the basin the parameters are initially placed in. This noisiness, on the other hand, enables SGD to jump to other local minima that BGD would not be able to reach. The term "stochastic" indicates that the one example comprising each batch is chosen at random. The update step is thus looks like this:

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i)}; y^{(i)}) \tag{2.11}$$

for each sample i in the dataset.

Mini-Batch Gradient Descent Mini-batch gradient descent finally takes the best of both worlds and performs an update for every mini-batch of n training examples:

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)}) \tag{2.12}$$

This way, it reduces the variance of the parameter updates, which can lead to more stable convergence. Since it operates on smaller batches the algorithm can be used properly for larger networks. It can also make use of highly optimized matrix optimizations common to state-of-the-art deep learning libraries that make computing the gradient w.r.t. a mini-batch very efficient [50]. This method is usually also referred to as Stochastic gradient descent.

Momentum

Mini-batch gradient descent is particularly slow when there is a long and narrow valley in the loss function surface. In this situation, the direction of the gradient is almost perpendicular to the long axis of the valley. The system thus oscillates back and forth in the direction of the short axis, and only moves very slowly along the long axis of the valley [46]. To combat this issue a momentum term can be implemented that gives the weight update inertia, dampening the oscillations and causing SGD to accelerate faster in the relevant direction. The momentum is implemented by adding a fraction γ of the update vector of the past time step to the current update vector:

$$\begin{aligned} \mathbf{v}_t &= \gamma \mathbf{v}_{t-1} + \eta \nabla_\theta J(\theta) \\ \theta &= \theta - \mathbf{v}_t \end{aligned} \tag{2.13}$$

Nesterov Accelerated Gradient The problem with regular momentum is that it follows the downward slope blindly, it would be better if it knew to slow down before the gradient slopes up again. This is accomplished with Nesterov Accelerated Gradient [42] (NAG), it uses the knowledge that we will use momentum term γv_{t-1} to move parameters θ. We can thus estimate where our parameters will be in the next time step by calculating $\theta - \gamma v_{t-1}$. We can now effectively look ahead by calculating the gradient not w.r.t. to our current parameters θ but w.r.t. the approximate future position of our parameters:

$$\begin{aligned} \mathbf{v}_t &= \gamma \mathbf{v}_{t-1} + \eta \nabla_\theta J(\theta - \gamma \mathbf{v}_{t-1}) \\ \theta &= \theta - \mathbf{v}_t \end{aligned} \tag{2.14}$$

Adaptive Methods

A very important factor in these basic training algorithms is the learning rate η. This value needs to be set correctly for the algorithm to find an optimal solution. Often this learning rate is set at a specific value and left to decay gradually over time to let it converge more precisely to good solution. In this case η is the same for each parameter, which is not a desirable property as it assumes the learning rate schedule is the same for each parameter. A better solution is to have different learning rates for each parameter.

AdaGrad The simplest per-parameter update method is Adaptive Gradient method [28] (AdaGrad). In this algorithm each parameter is updated separately according to its own gradient. To adapt the learning rate for different parameters a term is introduced that attempts to equalize the learning rate between parameters which tend towards large gradients and those that tend to small ones. The update for the i^{th} parameter at the t^{th} timestep/epoch of the training algorithm is given by the following formula:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{\mathbf{G}_{t,ii} + \epsilon}} \cdot \nabla_\theta J(\theta_{t,i}) \tag{2.15}$$

$\mathbf{G}_t \in \mathbb{R}^{d \times d}$ is a diagonal matrix where element i, i corresponds to the sum of the squares of the gradients w.r.t. θ_i up until step t. ϵ is a very small number introduced to avoid division by zero. We can vectorize this formula by calculating the matrix-vector product $\odot$ between $\mathbf{G}_t$ and $\mathbf{g}_t$, the vector of the gradients with respect to the parameters θ:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\mathbf{G}_t + \epsilon}} \odot \mathbf{g}_t \tag{2.16}$$

AdaGrad eliminates the need to manually tune the learning rate, one downside however is the accumulating gradient problem. Since all the values added to the gradient matrix are positive these values will keep on growing, always decreasing the learning rate. Eventually the algorithm will stop learning, even though the cost function might not yet be minimal.

RMSProp To tackle the problem of the aggressively decreasing learning rate in AdaGrad, RMSProp[1] restricts the window of the accumulated past gradients. As opposed to AdaGrad which uses all past squared gradients. The current rolling average $E[g^2]_t$ is calculated as the weighted sum of the previous rolling average and the current gradient.

$$E[\mathbf{g}^2]_t = 0.9E[\mathbf{g}^2]_{t-1} + 0.1\mathbf{g}_t^2 \tag{2.17}$$

The update step is then given by:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[\mathbf{g}^2]_t + \epsilon}}\mathbf{g}_t \tag{2.18}$$

The RMS in RMSProp stems from the root mean squared in the denominator under the learning rate.

ADAM Adaptive moment estimation [31] (ADAM) is a method that combines adaptive methods with momentum based methods. It adapts the learning rate for each parameter separately while using momentum to smooth the path over time steps. In addition to storing an exponentially decaying average of past squared gradients v_t like RMSProp, ADAM also keeps an exponentially decaying average of past gradients m_t, similar to momentum.

$$\begin{aligned} \mathbf{m}_t &= \beta_1\mathbf{m}_{t-1} + (1-\beta_1)\mathbf{g}_t \\ \mathbf{v}_t &= \beta_2\mathbf{v}_{t-1} + (1-\beta_2)\mathbf{g}_t^2 \end{aligned} \tag{2.19}$$

Since $\mathbf{m}_t$ and $\mathbf{v}_t$ are initialized as vectors of zeroes a bias towards zero is introduced. This is corrected using the following formulas:

$$\begin{aligned} \hat{\mathbf{m}}_t &= \frac{m_t}{1-\beta_1^t} \\ \hat{\mathbf{v}}_t &= \frac{\mathbf{v}_t}{1-\beta_2^t} \end{aligned} \tag{2.20}$$

The parameter values are then updated in a fashion similar to RMSProp:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{\mathbf{v}}_t} + \epsilon}\hat{\mathbf{m}}_t \tag{2.21}$$

2.2.4 Backpropagation

So far we have seen several gradient descent methods for updating the weights of our network. An important part of all these methods is the the gradient of the loss function. In this section we will discuss a method to calculate this gradient called backpropagation. The backpropagation algorithm was originally introduced in the 1970s, but its importance was not fully appreciated until a famous 1986 paper by David Rumelhart, Geoffrey

[1]RMSprop is unpublished, it was first introduced by Geoffrey Hinton in the sixth lecture of this Coursera class `https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`

Hinton, and Ronald Williams [51]. That paper describes several neural networks where backpropagation works far faster than earlier approaches to learning, making it possible to use neural nets to solve problems which had previously been insoluble.
The goal of backpropagation is to compute the partial derivatives of the cost function J with respect to all the parameters θ in the neural network. Consider input x, the algorithm starts by executing a feedforward pass. For each layer $l = 1, 2, \dots, L$ the weighted input vector $\mathbf{z}^l = \mathbf{w}^l \mathbf{o}^{l-1} + \mathbf{b}^l$ and the neuron activations $\mathbf{o}^l = \sigma(\mathbf{z}^l)$ are computed. Subsequently the error in the output layer is computed using the following formula:

$$\delta^L = \nabla_{\mathbf{o}} J \odot \sigma'(\mathbf{z}^L) \tag{2.22}$$

here, $\nabla_o J$ is the vector containing the partial derivatives of the cost function J w.r.t. the activations of the neurons in the final layer $\partial J / \partial o_k^L$, $\sigma'(z^L)$ is the inverse activation function performed on the weighted inputs of the final layer and $\odot$ stands for the Hadamard or entrywise product. This error is then back-propagated through the network. For each layer $l = L-1, L-2, \dots, 2$:

$$\delta^l = ((\mathbf{w}^{l+1})^T \delta^{l+1}) \odot \sigma'(\mathbf{z}^l) \tag{2.23}$$

with $(\mathbf{w}^{l+1})^T$ the transpose of the weights of the next layer. Having completed our backpropagation operation we now have a error vector for each layer, with every vector containing the error on input $\mathbf{x}$ for each neuron in that layer. Finally we compute the gradient of the cost function for each weight:

$$\frac{\partial J}{\partial \mathbf{w}_{jk}^l} = \mathbf{o}_k^{l-1} \delta_j^l \tag{2.24}$$

and for each bias:

$$\frac{\partial J}{\partial \mathbf{b}_j^l} = \delta_j^l \tag{2.25}$$

These partial derivatives can then be used in one of the gradient descent methods described earlier to train a neural network, given a set of input data.

2.2.5 Alternative training methods

Gradient descent is widely the most used optimization method for neural networks, this because it it computationally fast and will often lead to a good solution. But optimization is a widely studied problem, with a number solutions, many of which have also been applied to neural networks. Some of these methods are presented here.

Newton's method

Gradient descent is a first order method, as is uses the first order derivative of the cost function, better directions in which to step can be found by making use of the second order derivatives. This is the case for Newton's method, which makes use of the Hessian

matrix of the cost function.
Consider the quadratic approximation of J at θ_0 using Taylor's series expansion:

$$J = J(\theta_0) + \nabla J(\theta_0) \cdot (\theta - \theta_0) + 0.5 \cdot (\theta - \theta_0)^2 \cdot \boldsymbol{H} J(\theta_0) \tag{2.26}$$

with $\nabla J(\theta_0)$ the gradient vector of partial derivatives of the cost function evaluated at (θ_0) and $\boldsymbol{H}J(\theta_0)$ the Hessian matrix with second order partial derivatives of the cost function evaluated at (θ_0). By setting ∇J equal to 0 for the minimum of $J(\theta)$ we obtain the following equation:

$$\nabla J = \nabla J(\theta_0) + \boldsymbol{H}J(\theta_0) \cdot (\theta - \theta_0) = 0 \tag{2.27}$$

Therefore we can now define an update step for Newtons method, starting at parameter vector θ_0:

$$\theta_{i+1} = \theta_i - \eta(\boldsymbol{H}J(\theta_i)^{-1} \cdot \nabla J(\theta_i)) \tag{2.28}$$

with η the training rate. This method generally requires less steps to find a minimum than gradient descent. Exact evaluation of the Hessian and its inverse are however computationally very expensive.

Conjugate Gradient

The conjugate gradient method [47] attempts to combine the fast convergence of Newton's method with the low computational cost of gradient descent. In this training method the search is performed along the conjugate directions which produces generally faster convergence. These training directions are conjugated with respect to the Hessian matrix.
A training vector $\mathbf{d}$ is computed at each step in the training process:

$$\mathbf{d}_{i+1} = \nabla J(\theta_{i+1}) + \mathbf{d}_i \cdot \gamma_i \tag{2.29}$$

where the training direction vector is initialized as $\mathbf{d}_0 = -\nabla J(\theta_0)$. There are multiple versions of the conjugate gradient algorithm and they are distinguished by how γ_i is computed, here we present the Fletcher-Reeves [16] method:

$$\gamma_i = \frac{\nabla J(\theta_i)^T \nabla J(\theta_i)}{\nabla J(\theta_{i-1})^T \nabla J(\theta_{i-1})} \tag{2.30}$$

This is the ratio of the norm squared of the current gradient to the norm squared of the previous gradient. The weights of the network are then updated as such:

$$\theta_{i+1} = \theta_i + \eta_i \mathbf{d}_i \tag{2.31}$$

where η_i is found by performing a line search to determine the optimal distance to move in the current search direction.

Quasi-Newton method

Quasi-Newton methods are an alternative of the Newton's methods that attempt to reduce the computational requirements by building up an approximation of the inverse Hessian matrix at each timestep using only information on the first derivatives of the loss function. The update step is similar to Newton's method:

$$\theta_{i+1} = \theta_i - \eta(\boldsymbol{B}J(\theta_i) \cdot \nabla J(\theta_i)) \tag{2.32}$$

where $\boldsymbol{B}J(\theta_i)$ is an incremetal approximation of the Hessian matrix of the cost function J at i.
There are different versions of the Quasi-Newton method and they are distinguished by how $\mathbf{B}$ is computed. Here we present the Broyden–Fletcher–Goldfarb–Shanno (BFGS) method, named after the researchers who independently developed it [6][15][18][56]. Initially $\mathbf{B}$ is set at an initial guess. At each step of the algorithm the step to be taken is calculated as:

$$\mathbf{s}_i = \eta(\mathbf{B}_i^{-1} \cdot \nabla J(\theta_i)) \tag{2.33}$$

where η can be static or optimized via line search. This is followed by parameter update: $\theta_{i+1} = \theta_i + \mathbf{s}_i$ and computation of temporary value $\mathbf{y}_i$ that holds the difference of the gradient:

$$\mathbf{y}_i = \nabla J(\theta_{i+1}) - \nabla J(\theta_i) \tag{2.34}$$

using the values of $\mathbf{y}$ and $\mathbf{s}$ at i, $\mathbf{B}$ is updated using the following formula:

$$\mathbf{B}_{i+1} = \mathbf{B}_i - \frac{\mathbf{B}_i \mathbf{s}_i \mathbf{s}_i^T \mathbf{B}_i}{\mathbf{s}_i^T \mathbf{B}_i \mathbf{s}_i} + \frac{\mathbf{y}_i \mathbf{y}_i^T}{\mathbf{y}_i^T \mathbf{s}_i} \tag{2.35}$$

At each timestep we actually need the inverse of matrix $\mathbf{B}_i$, instead of inverting it at each timestep we can modify Equation 2.35 using the Sherman-Morrison formula [57] for computing the inverse of the sum of an invertible matrix and the outer product of two vectors.

$$\mathbf{B}_{i+1}^{-1} = \mathbf{B}_i^{-1} + \frac{(\mathbf{s}_i^T \mathbf{y}_i + \mathbf{y}_i^T \mathbf{B}_i^{-1} \mathbf{y}_i)(\mathbf{s}_i \mathbf{s}_i^T)}{(\mathbf{s}_i^T \mathbf{y}_i)^2} + \frac{\mathbf{B}_i^{-1} \mathbf{y}_i \mathbf{s}_i^T + \mathbf{s}_i \mathbf{y}_i^T \mathbf{B}_i^{-1}}{\mathbf{s}_i^T \mathbf{y}_i} \tag{2.36}$$

Allowing us to more efficiently compute the inverse of the approximation of the hessian at each timestep.

Gauss-Newton algorithm

The Gauss-Newton algorithm [17] is another training method that approximates the Hessian using only first order derivatives. To do so this method requires the loss function to be the sum of squared errors (Equation 2.3). It makes use of the Jacobian matrix $\mathbf{J}$ that contains all first-order partial derivatives of a vector-valued function, which in the case of neural network training are the partial derivatives of the error function of each output of each sample with respect to each network weight. The gradient at step i

can be computed by multiplying the Jacobian by error vector e containing the error the network made on each output of each sample, where $e_{p,m}$ is the error the network made on the p-th output of the m-th sample.

$$\nabla J(\theta_i) = \mathbf{J}_i\mathbf{e}_i \tag{2.37}$$

Using the definition of the Hessian matrix and the fact that the loss function is the sum of squared errors we can define the relationship between the Hessian and the Jacobian:

$$\mathbf{H} \approx \mathbf{J}^T\mathbf{J} \tag{2.38}$$

for a good explanation of why this holds the reader should refer to [71]. Using these new methods of computing the Gradient and Hessian we can define our parameter update step:

$$\theta_{i+1} = \theta_i - \eta(\mathbf{J}_i^T\mathbf{J}_i)^{-1} \cdot \mathbf{J}_i\mathbf{e}_i) \tag{2.39}$$

This algorithm is an improvement over the Newton method in the sense that it does not require the second-order derivatives of the loss function to be computed. It does however suffer from the fact that $(\mathbf{J}^T\mathbf{J})$ might not be invertible.

Levenberg-Marquardt algorithm

The Levenberg-Marquardt algorithm [36][38], also known as the damped least-squares method, is another algorithm for solving non-linear minimization problems that especially use the sum of squared errors loss function (Equation 2.3). It is often used due to its stable and fast convergence which stems from the fact that it blends stability of the gradient descent algorithm with the convergence speed of the Gauss-Newton method. The basic idea is that the algorithm uses Gauss-Newton and that it switches to gradient descent when the error surface is more complex than the quadratic situation, only switching back when the curvature is proper to make a quadratic approximation.
The Levenberg-Marquardt algorithm ensures the the approximation of the Hessian matrix is invertible by modifying it:

$$\mathbf{H} \approx \mathbf{J}^T\mathbf{J} + \lambda\mathbf{I} \tag{2.40}$$

with λ the positive dampening factor that ensures positiveness of the Hessian and $\mathbf{I}$ the identity matrix. The update rule can now be formulated as:

$$\theta_{i+1} = \theta_i - \eta(\mathbf{J}_i^T\mathbf{J}_i + \lambda\mathbf{I})^{-1} \cdot \mathbf{J}_i\mathbf{e}_i) \tag{2.41}$$

When λ is zero this is just the Newton-Gauss method, when λ is large, this is becomes gradient descent. The λ value is initialized large at first in order to start with small gradient descent steps. When the algorithm is performing well λ is decreased, switching to the Gauss-Newton method for faster convergence. When the algorithm is performing badly, this is an indication that the surface is too complex and λ is decreased to switch back to the more stable gradient descent.

Generally this is a very stable and fast algorithm but it has its drawbacks as it cannot be used for loss functions other than the sum of squared errors, neither does it support regularization terms. For large neural networks or large datasets the Jacobian matrix can become very large, giving the algorithm large memory costs.

2.2.6 Overfitting

We now have the tools to train our networks parameters given a dataset, a naive way of approaching this would be to feed our entire dataset into the gradient descent algorithm and proceed to test the performance of our trained network by calculating the loss between the predictions made for that dataset and the actual labels. Following this method could introduce the overfitting phenomenon. Overfitting occurs when the model is over-optimized to accurately predict the training set, at the expense of generalizing to unknown data, which is the objective of learning in the first place. This happens when the network is trained too well causing it to simply mimic the training set.
One solution to the overfitting problem is to split our training data in a training and test set. When training the network the gradient descent optimization is calculated on the training data each iteration, but the performance is measured on the test set. Since the test set is withheld from the training set it is a good way to measure how well the model is learning to generalize the data. The longer we train, the more likely our training accuracy is to go higher and higher, but at some point, it is likely the test set will stop improving. This is a cue to stop training at that point. We should generally expect that training accuracy is higher than test accuracy, but if it is much higher, that is a clue that we have overfit. This two way split is however not enough, as we also need an unbiased metric to compare between different versions of our model, with for example, different layouts. To do this a third partition of the data is introduced: the validation set. Generally the dataset is split in two parts of 80% and 20% training and test data respectively. The training data is then again split in 80% and 20% to obtain the training and validation sets.

Regularization

Regularization is a technique which makes slight modifications to the learning algorithm such that the model generalizes better. One way overfitting occurs is when the magnitude of the weights grows too large. Regularization attempts to combat this by penalizing large parameters in the network. This is achieved by adding a regularization term based on the weights in the network to the loss function. Due to the addition of this regularization term, the values of weight matrices decrease because it assumes that a neural network with smaller weight matrices leads to better models. Consider the mean squared error loss function, with $\mathbf{y}$ being the targets from the dataset and $\hat{\mathbf{y}}$ the predictions by the neural network

$$J = \frac{1}{n}\sum_{i=1}^{n}(\mathbf{y}_i - \hat{\mathbf{y}}_i)^2 \tag{2.42}$$

We implement regularization in this function by adding a regularization term that depends on the weights of our network:

$$J = \frac{1}{n}\sum_{i=1}^{n}(\mathbf{y}_i - \hat{\mathbf{y}}_i)^2 + R(\theta) \tag{2.43}$$

The regularization term can take different forms but the most widely used are L_1 and L_2:

- $R_{L1}(\theta) = \lambda \sum_{w}^{\theta} |\mathbf{w}|$
- $R_{L2}(\theta) = \lambda \sum_{w}^{\theta} \mathbf{w}^2$

with the sum being over all the weights and biasses w of the network.

Dropout

Recently a new regularization method called “dropou” has been introduced to combat overfitting [61]. The key idea is to randomly drop neurons, along with their connections, from the neural network during training. This ensures that the network does not become overdependent on specific neurons. Before every epoch a random number of neurons is picked to be dropped, this amount depends on a parameter usually set in the 20% to 50% range. Figure 2.2 is an illustration of the effect of adding dropout to a neural network, it is taken from the original publication.

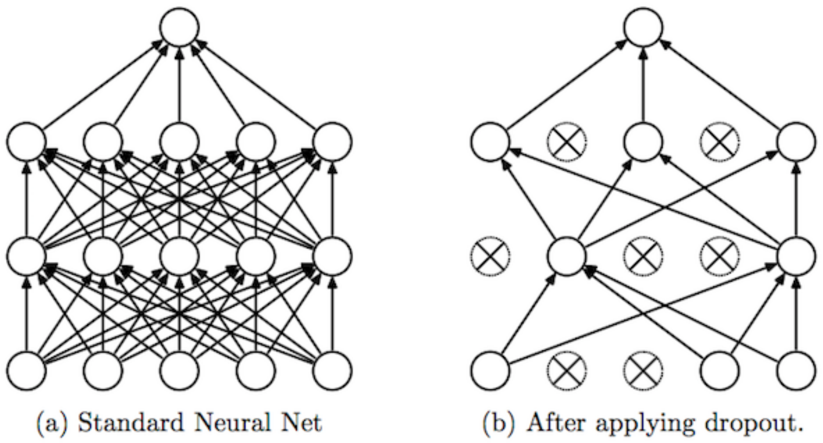


Figure 2.2: Two neural networks, showing the effect of applying dropout during the training process.

Early Stopping

Normally neural networks are trained within a fixed amount of epochs, the training process is only halted if the training algorithm has done that amount of iterations through the dataset. At a certain point the loss value of the test dataset will stop decreasing and in some cases will even start to increase, whereas the loss value of the training set could still be decreasing because the network is learning to just mimic the data from the training set. It is desirable that the network would stop training as soon as the performance of the network on the test dataset stops improving. This is illustrated in Figure 2.3. Early stopping can be triggered by setting a maximum number of epochs over which no improvement can occur after which training will be stopped.

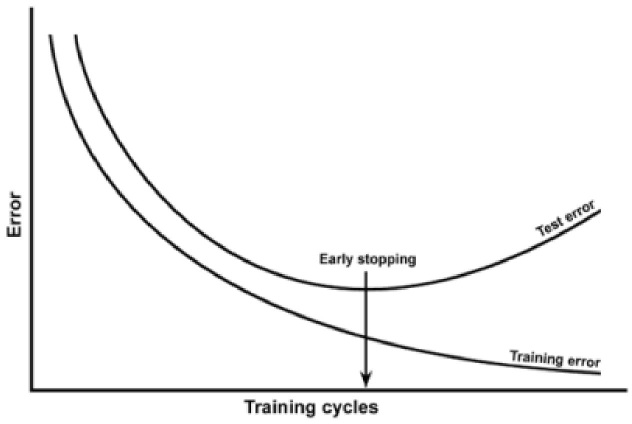


Figure 2.3: Illustration of the training process of a neural network which suffers from overfitting. The indication shows when the training process should stop.

2.3 Hyperparameter optimization

Up until now we have focused on how we change the parameters of the neural network to make it generate desirable outputs. The only parameters we have thus far considered where the weights and biases, but there is another set of parameters that we can adapt to improve the performance of the network. These parameters are called the hyper-parameters and include: the amount of layers, the learning algorithm, the activation function for each layer, the amount of neurons in each layer, the dropout percentage per layer, the learning rate etc. In this section we present a few hyperparameter selection methods.

2.3.1 Grid Search

In grid search the set of possible values for each parameter is defined and the model is trained for each possible combination. The combination for which the model performed

the best is then selected. Performance is measured by some metric of choice, e.g. mean squared error.
Intuitively this is a very simple and exhaustive method of selecting the parameters, but it's running time can quickly become very long as it tries every possible permutation of the parameter settings. Grid search however, is embarrassingly parallel, as typically the hyperparameter settings it evaluates are independent of each other.

2.3.2 Random Search

The idea behind random search is to evaluate a randomly selected subset of parameter values to decrease the running time compared to grid search. The problem with fully random search is that it will often select parameter combinations that are very similar leading to redundant evaluations of parameter combinations. Further more, if there are not enough data points it will not fully cover the parameter space. This problem can be solved by using low-discrepancy sequences (also refered to as quasi-random sequences). These are sequences of numbers that are better equidistributed in a given volume than pseudo-random, or random numbers [11].

2.3.3 Gaussian Process Approach

This approach uses Gaussian Processes in combination with an Acquisition Function [5]. The Gaussian process uses previously evaluated hyperparameter setups and their resulting performance to make an assumption about unobserved hyperparameters. The Acquisition Function uses this information to suggest the next set of parameters.
A Gaussian process is a probability distribution over possible functions, defined by a mean function and a covariance function. Bayesian inference is used to update the distribution of function by observing new training data. Adding more observations narrows the distribution of functions.
Expected improvement is used as acquisition function to select the next set of parameters. If we are attempting to minimize our performance metric, the EI is calculated via this formula:

$$g_{min}(x) = max(0, y_{min} - y_{lowestexpected}) \tag{2.44}$$

where y_{min} is the minimum observed value for y and $u_{lowestexpected}$ is the lowest possible value from the confidence interval associated with each possible value of x. The EI formula can easily be modified to maximise a performance metric.
Some problems with this approach include the fact that it does not work well for categorical values, the best score can just be a lucky output as neural network optimization often includes randomness, the Gaussian process itself has hyperparameters which can be difficult to set and it works slower when the amount of hyperparameters increase [58].

2.3.4 Tree-structured Parzen Estimator Approach (TPE)

The Tree-structured Parzen Estimator Approach [5] attempts to fix some of the disadvantages of regular Gaussian process approach. This algorithm works by first performing

a few iterations of random search to collect some data. The collected data is divided in two parts: a group of observations with good performance scores and the rest. The goal of TPE is to now find a set of parameters that is more likely to be in the first group. The fraction of the groups is determined by the user, along with the amount of initial random search iterations.
The next step is to model the likelihood probability for the two groups. Using the likelihood probability of the group with the good samples we sample a number of candidates. For each of these candidates we can compute the Expected Improvement:

$$EI(x) = \frac{l(x)}{g(x)} \tag{2.45}$$

with $l(x)$ the probability of the candidate being in the first group and $g(x)$ that of the second group. The sample with the highest EI is selected. The likelihood probability distributions are calculated using Parzen-windows density estimators [45], a non-parametric method for estimating a continuous density function from data. Tree-structured in the name stems from the parameter space, which is defined in the from of a tree.

2.4 Recurrent Neural Networks

So far we have only considered the regular feed forward neural network, but there exist a lot of different types of neural networks that perform better for certain tasks. The same concepts of neurons, weights and activation functions still apply, but they are used in more advanced ways to tackle more complex problems. In this section we present the Recurrent Neural Network.
Recurrent neural networks (RNNs) are a type of network where at least one of the connections between units forms a directed cycle. In practice this means that at each time step the output of the previous time step is also an input to that neuron. This makes RNNs capable of learning features and long term dependencies from sequential and time-series data since they have a form of memory [52]. This removes one of the main limitations of vanilla neural networks: the fixed input and output size. These networks allow us to operate over sequences of vectors, both as input or output. Because of their effectiveness when working with sequences RNNs are often used in Natural Language Processing. We address this type of network because crowd simulation is a problem that has a temporal structure that can possibly be exploited by RNNs.

2.4.1 Architecture

As with feed forward networks a RNN also has an input layer, a number of hidden layers and an output layer. Assuming our input layer has N inputs the input is a sequence of vectors through time t such as:

$$\{\ldots, \mathbf{x}_{t-1}, \mathbf{x}_t, \mathbf{x}_{t+1}, \ldots\} \tag{2.46}$$

where $\mathbf{x}_t = \{x_1, x_2, \ldots, x_N\}$. Consider a RNN with one hidden layer, the input units are connected to the hidden units in this layer weighted by matrix $\mathbf{W}^{IH}$. This hidden layer has M units that are connected to each other through time with recurrent connections and to the output layer. The output of this hidden layer is defined as:

$$\mathbf{h}_t = f_H(\mathbf{o}_t) \tag{2.47}$$

where

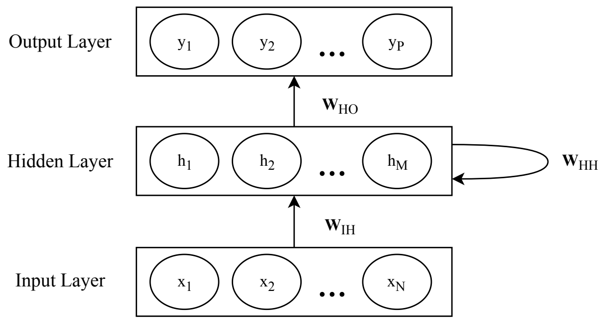
$$\mathbf{o}_t = \mathbf{W}^{IH}\mathbf{x}_t + \mathbf{W}^{HH}\mathbf{h}_{t-1} + \mathbf{b}^H \tag{2.48}$$

with $f_h()$ the activation function in the hidden layer (these are the same as the functions described in **??**), $\mathbf{b}_h$ the bias of the hidden units and $\mathbf{W}^{HH}$ the weights matrix for the recurrent connections between the hidden units themselves. This hidden layer is in turn connected to the output layer with weighted connections $\mathbf{W}^{HO}$. The output layer has P units $\mathbf{y}_t = \{y_1, y_2, \ldots, y_P\}$ that are computed as

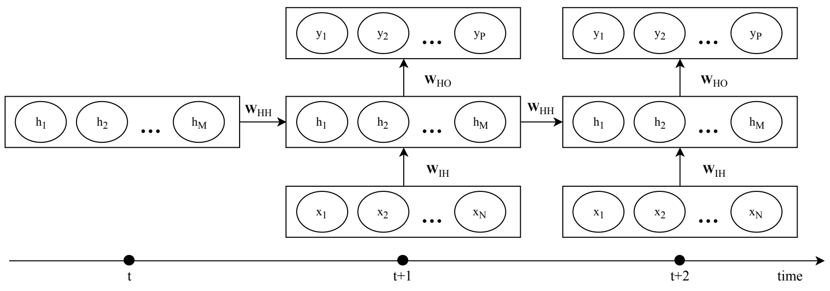
$$\mathbf{y}_t = f_O(\mathbf{W}^{HO}\mathbf{h}_t + \mathbf{b}^O) \tag{2.49}$$

Figure 2.4 contains a visual representation of how data flows through a RNN and how outputs are computed. Due to their great flexibility RNNs allow us to use vectors as both input and outputs, this gives several different ways to use them. The different possibilities are shown in Figure 2.5 and summarized below.

- **One to one** does not make use of recurrence and is the same as a feedforward network as it takes a fixed size input and provides a fixed size output. An example of this is a network that uses a patients blood values to compute the probability this person has diabetes.
- **One to many** takes a fixed size input and produces a sequence of outputs. This can be used for image classification where the input is a fixed size image and the output is a sequence of keywords describing the image.
- **Many to one** has a sequence as input and generates a single fixed size output. This type of RNN can be useful when, for example, trying to predict the weather tomorrow based on the weather conditions of the past week.
- **Many to many** receives a sequence of inputs and outputs a sequence of vectors after it has taken the entire input sequence. This is useful for problems where the input has a different length then the output. An example of this is machine translation, where the input is a Dutch sentence that gets translated to German.
- **Synced many to many** is roughly the same as the previous method except for the fact that an output is produced at every input. This results in the inputs and outputs having the same length. An example of this would be a video classification problem where a label needs to be generated for each frame.



(a) Folded RNN



(b) Unrolled RNN

Figure 2.4: Diagrams showing the design of a RNN with one hidden layer. The upper image shows the folded RNN whereas the bottom image shows the RNN unrolled through time.

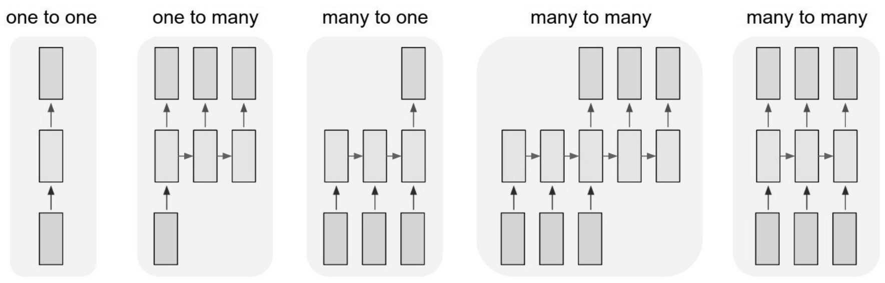


Figure 2.5: Different ways in which recurrent neural networks can be used.

2.4.2 Training

As with feed forward neural networks we train the network by minimizing the loss function with respect to the parameters of the network. The loss function for a RNN is the sum of the losses in each time step.

$$J = \sum_{t=1}^{T} J_t(\mathbf{y}_t, \hat{\mathbf{y}}_t) \tag{2.50}$$

This loss function can be minimized using gradient descent as described in Section 2.2.3. The backpropagation method however needs to be adapted to be able to compute the error-derivatives through time, this is done by a method called backpropagation through time (BPTT).

Backpropagation Trough Time

BPTT is one of the commonly used techniques to train recurrent networks. BPTT “unfolds” the neural network in time by creating several copies of the recurrent units which can then be treated like a (deep) feed-forward network with tied weights. Once this is done, a standard forward-propagation technique can be used to evaluate network fitness over the whole sequence of inputs, while a standard backpropagation algorithm can be used to evaluate partial derivatives of the loss criteria with respect to all network parameters [20].

2.4.3 Long Short Term Memory

It has been shown that recurrent neural networks are not effective at learning long term dependencies [4]. The layers and time steps of deep neural networks relate to each other through multiplication. When there are many time steps, this causes derivatives to become susceptible to vanishing or exploding. When a gradient is exploding, it it presumed too powerful and it can cause a small error on one end of the network to lead to a large value on the other end. This problem can be solved by truncating or squashing gradients. Vanishing gradients can become too small for computers to work with or for networks to learn, this is a harder problem to solve.
Hochreiter et al. [25] proposed Long Short-Term Memory Units (LSTMs) as as a solution to the vanishing gradient problem. Just like RNNs LSTM networks have a chain like structure, but every repeating module consists of four separate layers. Such an LSTM chain is illustrated in Figure 2.6.

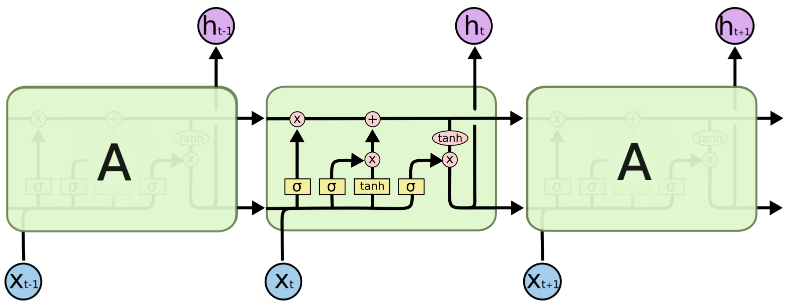


Figure 2.6: LSMT unit in a chain.

Besides passing on the output of the previous timestep to the next timestep the cell state **c** is now also passed on, this is the core idea behind the LSTM unit. The cell has the ability to add and remove information from this cell state, this is done by the four new layers that were added.

The first layer is the **forget gate layer**, this layer determines what information we throw away from our cell state. It does does so by adding the previous output $\mathbf{h}_{t-1}$ and current input $\mathbf{x}_t$, multiplying them by $\mathbf{U}_f$ and $\mathbf{W}_f$ respectively, the weights of the recurrent connections and input, adding the bias $\mathbf{b}_f$ an finally taking the sigmoid of this to output a number between zero and one for each number in the cell state. Zero meaning completely forget this and one means completely keep this. This operation is given by:

$$\mathbf{f}_t = \sigma(\mathbf{W}_f\mathbf{x}_t + \mathbf{U}_f\mathbf{h}_{t-1} + \mathbf{b}_f) \tag{2.51}$$

Having decided what information we will throw away, the next step is to select new information to store in the cell state. This is done in two parts. Firstly we'll decide on what values we want to update. This is done in the **input gate layer**. The calculation is similar to that of the forget gate layer, only with a different set of weights and bias:

$$\mathbf{i}_t = \sigma(\mathbf{W}_i\mathbf{x}_t + \mathbf{U}_i\mathbf{h}_{t-1} + \mathbf{b}_i) \tag{2.52}$$

The next step in determining what new information will be included is creating a vector of candidate values $\tilde{c}_t$ that could be added to the cell state. This is done with a *tanh* layer, that also has a separate set of weights:

$$\tilde{\mathbf{c}}_t = tanh(\mathbf{W}_i\mathbf{x}_t + \mathbf{U}_i\mathbf{h}_{t-1} + \mathbf{b}_i) \tag{2.53}$$

Having decided what information to forget and what information to add we can move on to updating the previous cell state $\mathbf{c}_{t-1}$ into new state $\mathbf{c}_t$. We forget the information deemed not necessary by multiplying the old state $\mathbf{c}_{t-1}$ with vector $\mathbf{f}_t$ that indicates what needs to be forgotten. Subsequently new information is added by adding the candidate values $\tilde{\mathbf{c}}_t$ scaled by how much we decided to update each state value. This leads to the following equation for the value of the next cell state:

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t \tag{2.54}$$

Finally, our output $\mathbf{h}_t$ needs to be calculated. To compute this we will filter current cell state $\mathbf{c}_t$. This is achieved by pushing the cell state through a tanh operation. Next, we determine what parts of the cell state will be kept by, again applying a sigmoid to the concatenation of the previous output and current input, similar to the forget gate layer. To compute the final output we multiply these two:

$$\begin{aligned} \mathbf{o}_t &= \sigma(\mathbf{W}_o\mathbf{x}_t + \mathbf{U}_o\mathbf{h}_{t-1} + \mathbf{b}_o) \\ \mathbf{h}_t &= \mathbf{o}_t \odot tanh(\mathbf{c}_t) \end{aligned} \tag{2.55}$$

Due to their cell state LSTMs are far better at handling long term dependencies then standard RNNs, thus solving the vanishing gradient problem.

2.4.4 Gated Recurrent Unit

Another variation of the standard Recurrent unit is the Gated Recurrent Unit (GRU) [10]. The GRU uses a structure similar to the LSTM but it lacks an output gate and thus does not make use of the cell state c. Due to the missing output gate the GRU uses less parameters making them easier to train, but it has also been shown that the GRU is strictly less powerful than the LSTM [67]. An illustration of a GRU cell is shown in Figure 2.7.

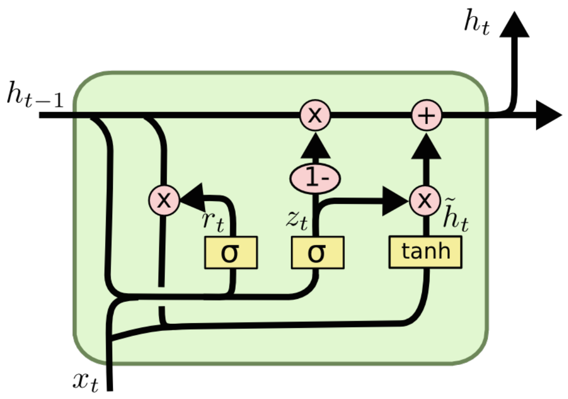


Figure 2.7: Illustration of the Gated Recurrent Unit.

The first part of the GRU is the **update gate**, which determines what information needs to be updated and passed to the future and which information needs to be kept. This is done by applying the sigmoid function to the weighted sum of the current input and the output of the previous timestep:

$$\mathbf{z}_t = \sigma(\mathbf{W}_z\mathbf{x}_t + \mathbf{U}_z\mathbf{h}_{t-1} + \mathbf{b}_z) \tag{2.56}$$

The second part of the GRU is the **forget gate**, this layer determines what information from the previous timestep will be reset.

$$\mathbf{r}_t = \sigma(\mathbf{W}_r\mathbf{x}_t + \mathbf{U}_r\mathbf{h}_{t-1} + \mathbf{b}_r) \tag{2.57}$$

Having determined what information from our previous timestep we want to use in the computation of the output at this timestep we can compute the preliminary output:

$$\tilde{\mathbf{h}}_t = tanh(\mathbf{W}_h\mathbf{x}_t + \mathbf{U}_h(\mathbf{r}_t \odot \mathbf{h}_{t-1}) + \mathbf{b}_h) \tag{2.58}$$

Finally we mix our preliminary output with the output of the previous timestep, according to the ratio determined by the output of the forget gate layer:

$$\mathbf{h}_t = (1 - \mathbf{z}_t) \odot \mathbf{h}_{t-1} + \mathbf{z}_t \odot \tilde{\mathbf{h}}_t \tag{2.59}$$

2.4.5 Attention in Recurrent Neural Networks

Recent attempts in trying to improve the performance of RNNs has seen researchers augmenting them with new properties. There are four directions this research has taken that stand out as exciting [43]: Neural Turing Machines, Attentional Interfaces, Adaptive Computation time and Neural Programmers. All these methods are based on the same technique: attention, a mechanism for shifting the focus of the neural network. The first two methods are briefly discussed in this section.

Neural Turing Machines

Neural Turing Machines (NTM) [19] extend the capabilities of standard RNNs by coupling them to external memory resources, which they can interact with by attentional processes. These networks consist of a controller that receives inputs and generates outputs like a regular neural network, but this controller also interacts with read and write heads that control the external memory. It is important that the read and write processes are differentiable in order to be able to train the network using a method like gradient descent. This is a difficulty since memory addresses seem to be discrete. To solve this "blurry" read an write operations are used, which interact with each element in the memory to a certain extend. The blurriness is determined by an attentional focus mechanism that constrains the operations to a small portion of the memory. To focus the operations, the read and write heads emit normalized weighting vectors over each element in the memory, that determine how much will be read or written to that element. The reading operation is thus a weighted sum over all memory locations:

$$r \leftarrow \sum_i w_i M_i \tag{2.60}$$

When writing a value to memory, the new memory value is a convex combination of the old value M_i and the write value p:

$$M_i \leftarrow w_i p + (1 - w_i) M_i \tag{2.61}$$

To determine what memory positions the attention gets focused on, a combination of two addressing mechanisms is used. Firstly, content-based addressing focuses attention on locations based on the similarity between their current values and the values emitted by the controller. This type of addressing is not suitable for all problems, for this reason location-based addressing is added which allows for relative movement in memory, enabling the NTM to loop. This is achieved by first performing the content-based addressing by querying the memory for a specific vector. The result of this is interpolated with the weighting from the previous time step. Locational addressing is then performed by convolving the weighting with a shift filter. Since this convolution can cause leakage of weightings the focusing vector is sharpened. Experiments demonstrate that this type of network is able to learn simple algorithms from example data and to use these algorithms to generalise well outside its training regime.

Attentional Interfaces

Attentional Interfaces are a technique used to make networks focus on specific parts of their input that hold more importance to the current task. A clear use case for this is in the field of machine translation, where sentences are translated to a different language by a sequence-to-sequence recurrent neural network. Since both sentences could have different lengths, the RNN first reads all of the input sentence while constructing a dense vector that represents the sentence. When the sentence is read, this dense vector is passed on, and the next part of the RNN reads this vector, using it to output a translated word at each timestep. This is sometimes referred to as the encoder-decoder structure.
Bahdanau et al. [2] improved on this model by using a distinct context vector for each word in the output sequence generated by an attention model. This distinct context vector c_i to generate output y_i depends on a sequence of annotations $(h_1, h_2, \ldots, h_{T_x})$ generated by the encoder:

$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j \tag{2.62}$$

with T_x the length of the input sequence. The weight of each annotation h_j is computed by

$$\alpha_{ij} = \frac{exp(e_{ij})}{\sum_{k=1}^{T_x} exp(e_{ik})} \tag{2.63}$$

where

$$e_{ij} = a(s_{i-1}, h_j) \tag{2.64}$$

is an alignment model which scores how well the inputs around position j and the output at position i match, based on the hidden state and the j-th output of the input sequence. a is a feedforward neural network that is trained along with the rest of the network.
Since it is required that the annotations h_i contain information about the preceding and following words the encoder employs a Bidirectional RNN [54] which can pass information in both directions of time. This is achieved by a forward RNN that reads inputs from $(x_1, \ldots, x_T)$ computing $(\overrightarrow{h_1}, \ldots, \overrightarrow{h_T})$. This is followed by a backwards RNN that reads states $(x_T, \ldots, x_1)$ computing the backwards hidden states: $(\overleftarrow{h_1}, \ldots, \overleftarrow{h_T})$. The annotation for each word x_j is obtained by concatenating the forward and backwards states at j. The full process is illustrated in Figure 2.8.

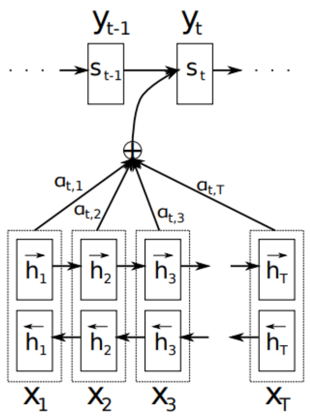


Figure 2.8: Illustration of the attentional translation model trying to generate the t-th target word y_t given a source sentence $(x_1, x_2, \ldots, x_T)$.

Chapter 3

Using neural networks for crowd simulation

This chapter uses the knowledge that was gained in the previous chapter to solve the crowd simulation problem. Firstly the exact nature of the problem is defined in order to properly construct a solution. Following this, two neural network methods are proposed to tackle the problem of crowd simulation. The first one is a simple feedforward neural network whereas the second one is a recurrent neural network. For both these methods it is described how we transform our dataset into training data for the network, as well as how the networks are trained. Once the networks are defined and trained a section is dedicated to explaining the crowd simulation process and how the trained networks are used in that process.

3.1 Problem definition

Since crowd simulation is a very broad term that can refer to any type of computational method pertaining to the simulation of crowds it is necessary to strictly define the problem we will be trying to solve. As mentioned earlier this thesis will purely focus on the movements of the agents in the simulation. The factors that will influence an agents movement are the direction to the goal and the movement of its neighbours. Social forces like panic and group behaviour are thus not taken into consideration.
A simulation at a specific point in time consists of a number of agents in a 2D environment:

$$agents = [A_1, A_2, ..., A_n] \tag{3.1}$$

Each agent has a position p, a speed v and a goal g:

$$A_n = (p_x, p_y, v_x, v_y, g_x, g_y) \tag{3.2}$$

Besides agents, a simulation also includes a list of obstacles:

$$obstacles = [O_1, O_2, ..., O_n] \tag{3.3}$$

For simplicity sake we define obstacles as rectangles with a position, height and width. Where agents are the digital representation of humans, obstacles are walls, buildings and other inanimate objects. We define a method that changes the velocity at each time step for each agent so that collisions with other agents and objects are avoided and the agent eventually reaches his goal in a natural way. Formally the simulation can be expressed as a function f that is executed at each update step to change the velocity for each agent.

$$(v_x, v_y) = f(A_n, obstacles, agents) \tag{3.4}$$

More specifically we attempt to train a neural network so that it can generate new velocities for the agents given the current state of the simulation. The goal is to create a crowd simulation method that can efficiently generate movement that closely resembles that of a real crowd. This is attempted by training our network on real-life data of the movement of people in crowds. There are many approaches to the design and training of neural networks. Simple Feed-Forward neural networks are an option, however the temporal structure of the data could be leveraged by using Recurrent Neural Networks which are often used for time series prediction. There are a lot of data sets of tracked crowds available so using these for supervised learning can allow the neural network to learn human like behaviours.

3.2 Agent-based feedforward method

In this section an agent-based crowd simulation method is proposed that uses a basic feed-froward neural network that is trained using data obtained from real life crowd videos in which the individuals are tracked. This approach is agent-based, meaning that at each time step each agents environment is considered separately to compute a new velocity vector for that agent. This is opposed to crowd-level simulations, in which the entire crowd is the input to the algorithm.

3.2.1 Data pre-processing

The first step is to prepare our dataset in order to be used as input for our neural network. The goal is to subtract state-action pairs that encapsulate the perception of the current environment of a specific agent at that time step and the action taken as a reaction to that perception. It is important to find the right amount of information to represent the input state and to format in a correct manner. When we use too little information the neural network will not be able to learn why a specific state leads to its corresponding action, but when we use too much information it will take too long to learn as the network will need to learn which information is unnecessary and the network complexity will need to increase.

Dataset

The dataset with tracked crowd data used in this project is the Arial Images Sequences dataset by Karlsruhe Institute of Technology (KIT) [1]. It contains multiple tracked videos, the data for each video comprises of an aerial image sequence and a XML file with manually labeled trajectories of all visible persons.
The available image sequence has been created by cutting out a predefined region of interest in a sequence of aerial images. The sequence was captured from a plane flying at an altitude of about 1500 m. The camera system consists of three cameras which are looking in front, nadir and back direction. Every Image has been ortho-rectified with a digital elevation model and directly geo-referenced with GPS/IMU system. Afterwards all images taken at the same time are fused in a mosaic to form a single image. These pre-processing steps contain small errors which become visible in the frame alignment. The number of frames in a sequence depends on the overlap in flight direction and the configuration of the camera.
The labeling of the position of individuals has been done manually by researches and students. Therefore the reference data contains a small amount of errors. The center position of the person's head has been marked in all frames, in which the person is present. In frames and crowded situations in which a single person is hardly visible they still tried to estimate the track as good as possible. A person's track does not have to be labeled continuously. A specific person can therefore vanish and reappear.
The XML files use the syntax of the CVML, an XML-based computer vision markup language for use in cognitive vision, to enable separate research groups to collaborate with each other as well as making their research results more available to other areas of science and industry [37]. The XML file for each video contains a list of frames, each frame contains a list of objects which correspond to the tracked peoples. The relevant data included for each frame is the coordinated universal time (UTC) the frame was captured and the ground sample distance (GSD). GSD in a digital photo of the ground from air or space is the distance between pixel centers measured on the ground and is used to convert pixel locations to meters. Each object has an ID that is unique to every tracked object in the sequence so that it can be used to differentiate people in different frames. Besides the ID every object contains an x and y position in pixels for the current frame.

Input format

We now need to extract a list of state-action pairs from this dataset to train our model with. These states need to contain enough information so that our network can learn how to generate the action. The action an agent takes in a certain situation is influenced by the movement and positions of his neighbours and the objects in the scene as he is trying to avoid collision with those, as well as by the agents ultimate goal. Since a neural network works with a fixed amount of inputs it is not possible to encode an arbitrary amount of information about the scene in the input. We do however, need to somehow

[1] `http://www.ipf.kit.edu/downloads_People_Tracking.php`

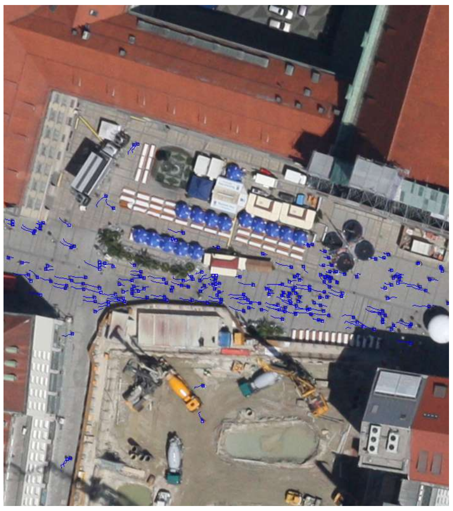
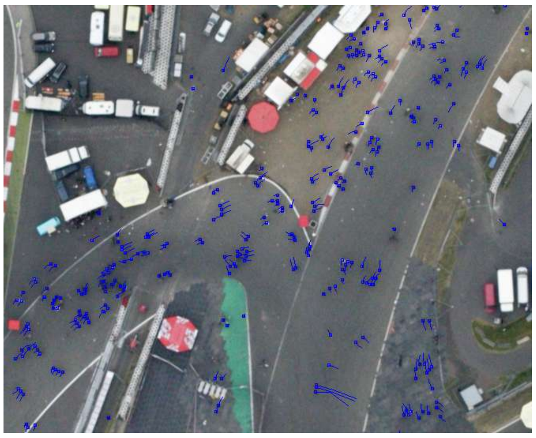



Figure 3.1: Two example frames taken from the dataset with the agents positions indicated by blue squares.

enable our agents to navigate around our scene and avoid objects. Simply encoding the nearest objects as input for the neural network would cause agents to always try and follow a straight line to their target while avoiding objects as they come leading to unwanted behaviour like for example wall hugging. For this reason the input to our neural network is a predefined path navigates the agent around the obstacles, this is inspired by the work of Song et al.[60]. The path could be automatically generated by an algorithm like A* or could be set manually which might be desirable in entertainment applications where the agents don't need to follow the shortest but pass certain objectives. The information about the path is combined with information about the closest neighbours, essentially combining long term path planning with short term collision avoidance. This is analogous to how humans navigate a crowded space: they have some general sense of which path they have to take to get to their destination and follow this path while avoiding other people. The inputs to the neural network are the following:

- The current speed of the agent in consideration, expressed as a vector $\mathbf{v_t} = (v_{t,x}, v_{t,y})$
- The desired moving direction $\mathbf{a}$ which is the vector between the current goal $\mathbf{g}$ in the predefined path and the current position $\mathbf{p}$: $\mathbf{a} = \mathbf{g} - \mathbf{p}$.
- The relative location $(\Delta x, \Delta y)$ and speed $(\Delta v_x, \Delta v_y)$ of the agents neighbours. The amount of neighbours chosen as input is a parameter that needs to be tweaked to assure trade-off between complexity and fidelity of the model. An FOV value is also chosen that determines the maximum distance between the agent and a neighbouring agent for them to be considered a neighbour. This can lead to the agent having less than the predefined number of neighbours in which case their relative speeds and positions will be set to zero. To ensure these zero values are

interpreted as no neighbours by the network a special method of normalization is used that will be explained later.

The output consist of a vector $\mathbf{v}_{t+1} = (v_{t+1,x}, v_{t+1,y})$ that represents the agents speed at the next time step. The samples are extracted from the dataset by reconstructing the agents trajectories for each scene using their unique identifiers. The state-action pairs $(\mathbf{s}, \mathbf{a})$ are generated by going through each trajectory, calculating the speed, desired direction, neighbour positions and speeds at each step. Data samples are spaced by roughly half a second, we opted to use every other sample, leaving about a second between samples. This makes the input data more smooth. The distances and speeds are converted to meters and meters per second respectively using the GSD measure provided by the dataset.

Parameter rotation

The walking or running directions of pedestrians are continuous. This incurs a problem that it is impossible for the network to learn all the possible directions because it is continuous. To generalize the problem and ensure that the neural network can be applied in each of the continuous directions a special rotation step is further applied to the input. This also makes it easier for the network to learn from the data leading to fewer epochs being needed to train. The first step is to rotate the agent's speed $\mathbf{v}_t$ counterclockwise to the x-axis (1,0). This leaves us with angle α by which we rotated $\mathbf{v}_t$ and a scalar speed value v'_t as the speed has lost its y value. Next the desired direction vector $\mathbf{a}$ is rotated by α giving us $\mathbf{a}'$. We encode this in the input using the directed angle between this vector and the (1,0) axis as the magnitude of the vector not of importance. This value is called the target angle ψ. We use the directed angle so that a distinction can be made between values of $\mathbf{a}'$ symmetrical around the x-axis. The directed angle is computed by the difference of `atan2` between the vectors. Finally all of the neighbour positions and speeds are also rotated by α as well as the velocity at the next step to obtain $\mathbf{v}'_{t+1}$. The process is illustrated in Figure 3.2, the original parameters are drawn in black and the parameters after rotation by α are shown in red. The circle with the i in it represents the i-th neighbour relative position to the agent.

Normalization

For most machine learning problems it is generally best practice to normalize the inputs. This is also the case for neural network training as it ensures that all the inputs are in a comparable range and get treated in a fair manner. In this case we only normalize the neighbouring agents positions, as mentioned earlier this needs to be done in a way that we can represent non-existent neighbours as zeroes. This is achieved by making the x and y values of neighbours closer to the agent closer to one and the ones further away closer to zero. The following formulas are used for agent position (x, y) and neighbour

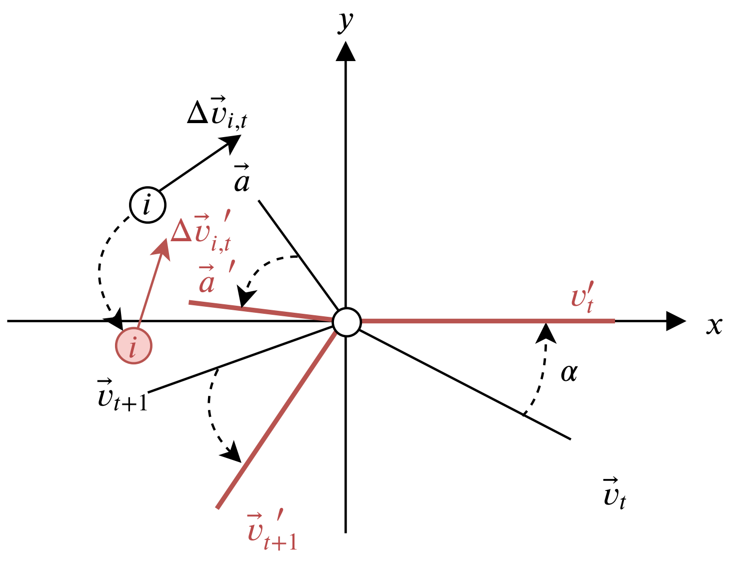


Figure 3.2: Illustration of the parameter rotation process for the feeforward neural network for a specific sample with one neighbour. The original vectors are shown in black and the rotated vectors are shown in red.

position (x_i, y_i):

$$x_i' = \begin{cases} 1 - (x_i - x)/FOV, & \text{if } x \leq x_i. \\ -(1 + (x_i - x)/FOV), & \text{otherwise.} \end{cases} \tag{3.5}$$

$$y_i' = \begin{cases} 1 - (y_i - y)/FOV, & \text{if } y \leq y_i. \\ -(1 + (y_i - y)/FOV), & \text{otherwise.} \end{cases} \tag{3.6}$$

Similarity search

Different humans in the same situation could make entirely different decisions, meaning that very similar states in our data could be coupled with drastically different actions. This leads to contradictions in our dataset, making it more difficult to train the neural network. To reflect the intricate nature of human decision making each state is given an array of actions that correspond to similar states as apposed to just the one action. The neural network will thus output multiple actions from which we can choose at simulation time. The following steps are taken to prepare the training samples for this:

1. For each of our samples a k nearest-neighbour search is performed to find similar states. Two samples similarity is given by the cosine similarity between their states. With A and B being the vectors representing the states of the two samples.

$$sim(sample_1, sample_2) = \frac{\mathbf{a} \cdot \mathbf{b}}{||\mathbf{a}|| \cdot ||\mathbf{b}||} = -\frac{\sum_{i=1}^{n} a_i \cdot b_i}{\sqrt{\sum_{i=1}^{n} a_i^2} \cdot \sqrt{\sum_{i=1}^{n} b_i^2}} \tag{3.7}$$

this is a value between -1 and 1, with 1 being exactly the same and -1 completely different.

2. For every sample the actions of the k most similar states(including its own) as calculated in the previous step are combined in an output vector of size $1 \times 2k$.
3. Each states action is replaced by the action vector constructed in step 2.

This action results in an array of samples with size $2 + 4n + 2k$, with n the amount of neighbours and k the number of actions generated trough similarity searching. These samples are now ready to be used in the training of the network. The input and outputs of the neural network are summarized in table 3.1.

Layer	Name	Meaning	
Input	v'_t	Scalar that stores speed magnitude	
	ψ	Directed angle between $\mathbf{a}'$ and (1,0)	
	Δx_i	The relative x coordinate of the i^{th} neighbour	for every n neighbours
	Δy_i	The relative y coordinate of the i^{th} neighbour	
	$\Delta v_{i,x}$	The relative speed in the x-direction of the i^{th} neighbour	
	$\Delta v_{i,y}$	The relative speed in the y-direction of the i^{th} neighbour	
Output	$\mathbf{v}'_{t+1,x}$	The speed in the x-direction in the next time step	for every k action
	$\mathbf{v}'_{t+1,y}$	The speed in the y-direction in the next time step	

Table 3.1: A summary of the inputs and outputs of the feed-forward neural network

3.2.2 Network training

Now we have created the input samples we can start designing and training our network. For this method we opted to use a feedforward neural network with dense/fully-connected layers and dropout on each hidden layer. The network is set to stop training after ten epochs with no improvement. The size of the input layer is $2 + 4n$ and the size of the output layer is $2k$. To find the best possible network model we used Tree-structured Parzen Estimator (TPE) hyperparameter optimization as described in Section 2.3.4. Given a search space for the hyperparameters this algorithm will select a configuration for which the model performs best. The data is split in a 90% and a 10% part, with the 10% being the data withheld from the training process to evaluate the performance of a specific hyperparameter setup. For each training iteration the 90% is again split in 90% and 10% with the 10% now being the test set on which the training performance of the model is tested. We extracted a total of 10352 samples from the dataset, 1036 of these will be used in the validation set, 932 will be used in the testing set an finally 8384 will be used to perform the actual training. Table 3.2 shows an overview of which parameters will be optimized by the algorithm and the possible values they can take.

Hyperparameter	**Type**	**Options**
Number of neurons in hidden layers	Ordinal	[10, 20, 50, 100]
Number of hidden layers	Ordinal	[0, 1, 2, 3]
Activation function	Categorical	[tanh, sigmoid, ReLu]
Dropout percentage	Continuous	[0.0, 0.5]
Batch size	Continuous	[16, 64]
Learning algorithm	Categorical	[RMSProp, ADAM, Mini-batch gradient descent]

Table 3.2: An overview of the parameters that are optimized by the hyperparameter optimization and their possible values

Besides the hyperparemeters there are a few other parameters concerning the crowd simulation we have to set. These are the amount of neighbours n for each agent we consider as the input data, the number of actions k that are generated for each state, and the value for the FOV around each agent we consider for the input. Because these parameters affect the shape of the input data they must be chosen before we can start the hyperparameter optimization process. Once we have selected our model we can retest the model with different values for the k, n and FOV. For now we selected five neighbours, five actions and an FOV of three meters.
Using the TPE hyperparameter optimization algorithm we determined the best performing network to use the ReLu activation function, have a dropout of 7%, a batch size of 58, use two hidden layers with 100 neurons in each while being trained by the RMSProp optimizer. This network achieved a Mean Squared Error of 0.0743 for the validation dataset. Knowing that the network outputs five actions this is an average of 0.0149 error per action. The results of 30 runs of the TPE algorithm are shown in the parallel-coordinates plot in Figure 3.3. To better show the impact that the separate parameters have on the performance of the model the distribution of loss values with respect to each parameter are shown in Figure 3.4. The same random seed was used for each experiment to ensure reproducibility and consistency between experiments.

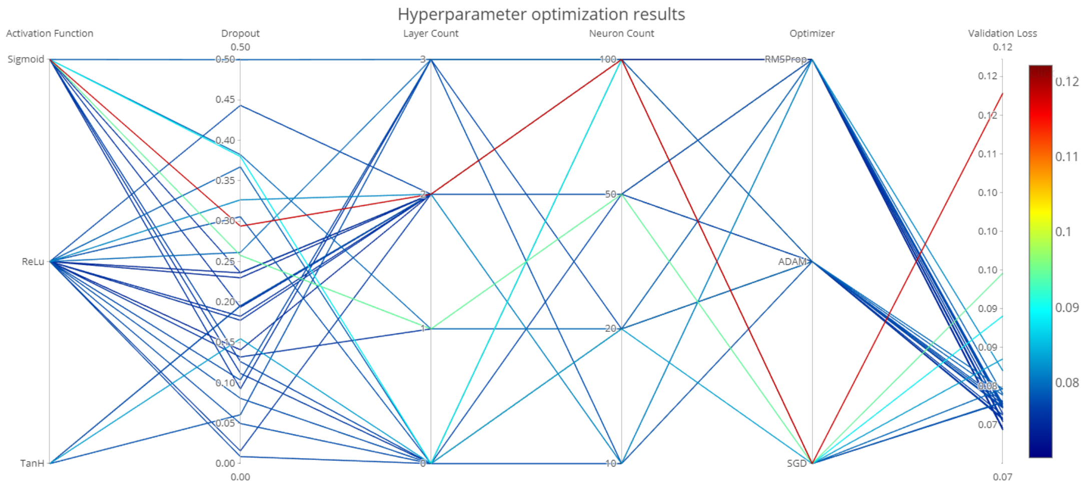


Figure 3.3: Results of the TPE hyperparameter optimization algorithm for the feedforward network

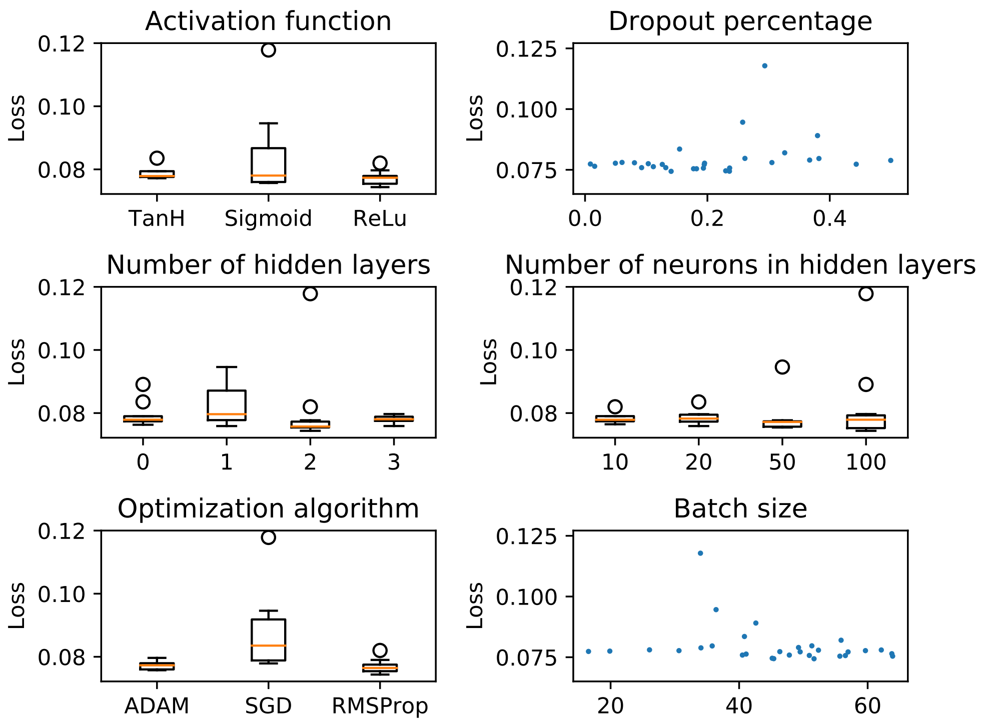


Figure 3.4: Results of the TPE hyperparameter optimization algorithm for the feedforward network

From this illustration we can conclude that the difference in performance between the best couple of parameter setups is rather small. The best ten experiments are all within 0.001255 of each other. This gives us the luxury to consider the execution time of the forward propagation. In crowd simulation scenes with many agents this is of importance since inputs need to be forward propagated through the network for each agent every few time steps. We should thus asses whether to pick a less accurate model that is able to compute outputs faster. Computation time is affected by the amount of computations that need to be done to calculate the output, which is determined by the amount of layers and number of neurons in these layers. The activation function used in the network also has an impact on the computation speed as the Sigmoid and TanH functions are more expensive compared to the ReLu function since they make use of the exponent operation whereas ReLu simply takes the maximum which is easier to compute. To pick a model that is both fast and accurate enough for our purposes the forward propagation calculation times for the best ten models were measured. This was done by predicting the output values for 7500 samples from the test set for each of the models. Computations were all done on the 2.40 GHz Intel Core i7-5500U CPU in an Acer Aspire laptop. In Figure 3.5 the results of these experiments are depicted. We conclude that our best performing model is also the fastest model with an execution time of 0.157 seconds for 7500 input samples.

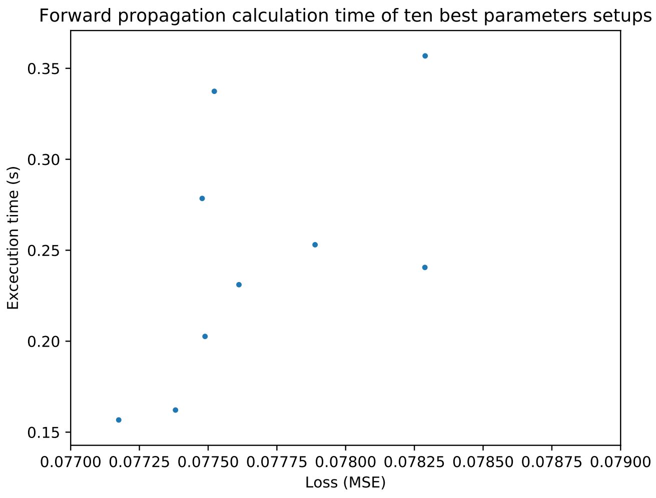


Figure 3.5: The feedforward propagation time for 7500 samples and validation loss for the ten best hyperparameter setups.

Figure 3.6 shows the evolution of the loss over the training set and validation set during the training process of the best model as chosen by TPE. We can see the network stopped training after 26 epochs as no further improvement had been made over the previous ten epochs. Having identified and trained a neural network that can generate agent velocities in a crowd simulation context so that agents do not collide with each other and obstacles, we can now move on to describing how this network is used to simulate a crowd.

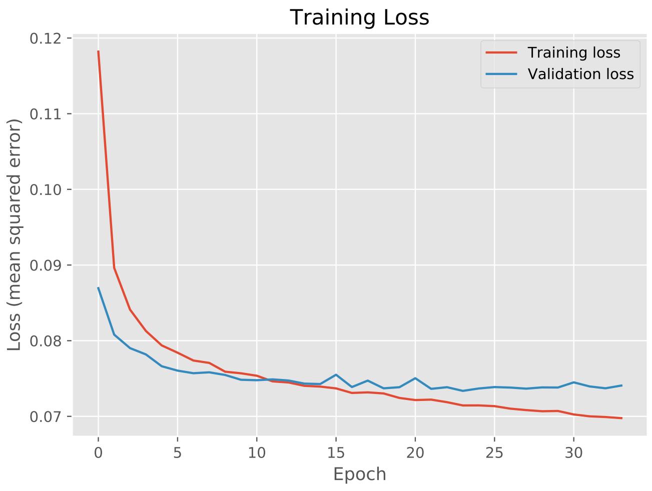


Figure 3.6: Evolution of the loss over the training set and validation set during the training process of the feedforward network

3.3 Agent-based recurrent method

The neural network described in the previous section only makes use of a snapshot of the current state of the agent and its surroundings to decide on what action to take. Humans on the other hand have a short term memory that stores information about the previous few time steps. In a crowd simulation context this information can be used to better estimate nearby peoples movement. In this section a new neural network is proposed that can also use information of the previous time steps to generate better actions for the agents in our simulation. This is achieved by using a recurrent neural network with long-short term memory units that is trained via supervised learning.

3.3.1 Data pre-processing

In the previous method the neural network used the relative positions and speeds of its neighbours in combination with its own speed and goal direction to make a decision on which action to take. The method proposed in this section uses a series of time-frames as input to the neural network. If the training samples for the previous method where an array of state action pairs $(\mathbf{s}, \mathbf{a})$ this method has a state array for each action $((\mathbf{s}_1, \ldots, \mathbf{s}_t), \mathbf{a})$.

Input format

To properly define the input for this method we need to realize that our input data consists of trajectories. Consider $\mathbf{t}$, the array that contains all the trajectories our training data belong to. A trajectory is a collection of all the state-action pairs $(\mathbf{s}, \mathbf{a})$ that are extracted from the dataset that belong to the same agent in chronological order, with $(\mathbf{s}_{i,t}, \mathbf{a}_{i,t})$ being the state action pair of the i^{th} trajectory at the t^{th} timestep. The content of each state look like this:

- Vector $\Delta\mathbf{d} = (\Delta d_x, \Delta d_y)$ containing the distance between the current position and the previous position of the agent on this trajectory. This value replaces the speed vector as it is no longer needed since the network can now derive the speed because it knows the difference in location between time steps. Furthermore the acceleration of agent can also be derived if enough timeframes are used as input. In the feedforward method the neural network only knew the speed of the agent along a straight line, but it can now use the successive $\mathbf{d}$ vectors to construct the exact path the agent has taken over the input timeframes. All this extra information should enable the neural network to generate better action vectors.
- The desired moving direction computed by the A* algorithm $\mathbf{a}$. This is the same as in the feedforward method.
- The time passed since the previous sample Δt. This value is needed to be able to calculate the speeds from the distance vectors.
- The relative locations $(\Delta x, \Delta y)$ of the n nearest neighbours. This stays the same as in the feedforward method, except for the fact that the relative speeds are omitted from this model as these can be derived using the relative positions and difference in time.

So just as in the previous method we are left with an array of state-action pairs, but since these are now considered as temporal data the states need to be structured in a different way to serve as input for our neural network. We are attempting to generate an action based on the previous couple of time-frames, the type of RNN most suited for this would be the many-to-one network. To generate an action we now use an array of states as opposed to the the one state used in the previous section. This leads to the question of how many previous frames we use as input for our move generation. For our purposes we decided to use a five second window, knowing that the samples we took from our dataset are separated by roughly a second this translate to five time-frames for each action. The data is transformed in the format suitable for our RNN by generating a new array of state-action pairs where the states are an array of the previous 5 states in the trajectory for that action. The training data for our network is an array of samples that consists of all the samples in trajectories transformed as described above $samples = [t'_1, t'_2, \ldots, t'_n]$ with

$$t'_i = [((s_1, \ldots, s_t), a_t), ((s_2, \ldots, s_{t+1}), a_{t+1}), \ldots, ((s_{j-t}, s_j), a_j)] \tag{3.8}$$

with j the number of samples/timesteps in trajectory i and t the amount of timesteps used as input for the network.

Parameter rotation

Just like in the feedforward method we rotate our samples to better generalize the input and as a result train the network faster and better. But since we now have multiple input samples for one output it is important to rotate all the input samples that belong to the same state vector $(s_1, \ldots, s_t)$ by the same angle. The rotation is thus executed after the states are grouped per time slice as described in the previous section. The angle by which to rotate $((s_1, \ldots, s_t), a)$ is equal to the angle between the $\Delta\mathbf{d}$ of the chronologically last sample s_t of the state vector and the x-axis (1,0). All of the states in the state vector are then rotated by this angle α in a manner similar to the parameter rotation of the feedforward method. The biggest difference is that in the feedforward method the speed vector was converted to a scalar as the y-value of the vector was lost. This is not the case for this method since only the $\Delta\mathbf{d}$ of s_t loses its y-value, the other distance vectors are not necessarily rotated to the x-axis and could thus still have a y-value.

Similarity search

In the feedforward method we combined the actions of similar states to more closely model the human decision process and to allow for more options to avoid collision when simulating a crowd. This is also done for this method, the combining is done after the states of the previous timesteps were added to the state space. To determine the most similar states to a certain state the cosine similarity is thus calculated between the two vectors that are the concatenation of the five states that make up the state space for the two state-action pairs.

Layer	**Name**	**Meaning**		
Input	$\Delta x'$	Distance travelled in x direction since last timeframe		
	$\Delta y'$	Distance travelled in y direction since last timeframe		
	ψ	Directed angle between $\mathbf{a}'$ and (1,0)		for every t
	Δt	Time passed since previous timeframe in seconds		timesteps
	Δx_i	The relative x coordinate of the i^{th} neighbour	for every n	
	Δy_i	The relative y coordinate of the i^{th} neighbour	neighbours	
Output	$\mathbf{v}'_{t+1,x}$	The speed in the x-direction in the next time step	for every k	
	$\mathbf{v}'_{t+1,y}$	The speed in the y-direction in the next time step	action	

Table 3.3: A summary of the inputs and outputs of the recurrent neural network

3.3.2 Network training

Having built the training samples in the previous section we can move on the the training process. For this method we used fully connected recurrent neural network. The network makes use of LSTM cells, has dropout after each layer and is set to stop training after ten epochs with no improvement. The network is set up as a many-to-one network. The networks expects inputs of the form $(4 + 2n) \times t$, with n the number of neighbours selected and t the number of considered time frames for each output. After inputting t states the network will be trained produce an action vector of the form $2k$, with k the

amount of actions that we want the network to generate. These parameters allong with the FOV, need to be set before training can begin. In this section we will illustrate the training process with an example that uses $k = 5$, $t = 5$, $n = 5$ and $FOV = 3$.
Once we have selected these values and generated the specific training data for these values we move on to hyperparameter optimization. This is done with the Tree-structured Parzen Estimator which is described in Section 2.3.4. The data is split in a 90% and a 10% part, with the 10% being the data withheld from the training process to evaluate the performance of a specific hyperparameter setup. For each training iteration the 90% is again split in 90% and 10% with the 10% now being the test set on which the training performance of the model is tested. We extracted a total of 4752 samples from the dataset, 475 of these will be used in the validation set, 428 will be used in the testing set an finally 4277 will be used to perform the actual training. The parameters to by optimized by the TPE algorithm are the same as for the feedforward network and are shown in Table 3.2.
The hyperparameter optimization method selected the Sigmoid activation function, a dropout percentage of 23,3, a batch size of 32, one hidden layer, 50 neurons for each layer and the ADAM optimizer. This specific parameter setup had a validation loss of 0,09175, which is calculated with the MSE.

3.4 Simulation

So far we have described a method for generating agent velocities given their states, this section describes the entire crowd simulation process and how the trained neural networks tie into this. How we define a crowd simulation problem and what the eventual goal is is described in Section 3.1.

3.4.1 Scene initialization

The crowd simulation scenarios are given in an XML format defined in SteerBench [59]. The format describes a number of different types of agents an objects, but the ones we used are the following:

- Obstacle: a rectangular obstacle defined by a minimum and maximum x value and z value. The format also specifies y, but these are ignored as we are creating a 2D simulation.
- Obstacle Region: a rectangular region in which a number of square objects are placed on random locations. The region is defined by its bounds, a number of obstacles to be spawned and the size of these obstacles.
- Agent: a human agent that is attempting to reach a goal (or sequence of goals). The agent is defined by a radius, initial position and speed and a goal sequence. This goal sequence is a series of static locations that the agent is trying to visit, for each goal a desired speed is also given.

- Agent region: a rectangular region in which a number of agents are placed on random locations. The region is defined by its bounds, a number of agents to be spawned and the goals for these agents. These goals can be the same static goal for each agent or a random position within the region. This is useful for large simulations since this way we do not need to hardcode each agent separately.

The XML file also contains a header that specifies the name of the scenario and its bounds. Some features included in the format that are not implemented in our work include different types of targets such as: seeking dynamic targets and fleeing targets and different type of obstacles like round obstacles.
All obstacles and agents are read from the XML file by a parser, subsequently all coordinates of the scene are shifted to ensure that no negative coordinates exist this makes rendering the simulation easier later on. Additionally the bounds of the simulation scenario are clipped to just include all agents and obstacles since these are often too large. Finally a check is performed whether agents are spawned on a obstacle, this is possible due to the random agents- and obstacle-regions. When an agent is in on an obstacle he is moved to the top right corner of this obstacle. This is also done for each goal of the agent, since these could also be randomly put on an obstacle. An example of a scene is shown in Figure 3.7, here 400 agents are spawned in four rooms of an office, they are all trying to reach the exit.

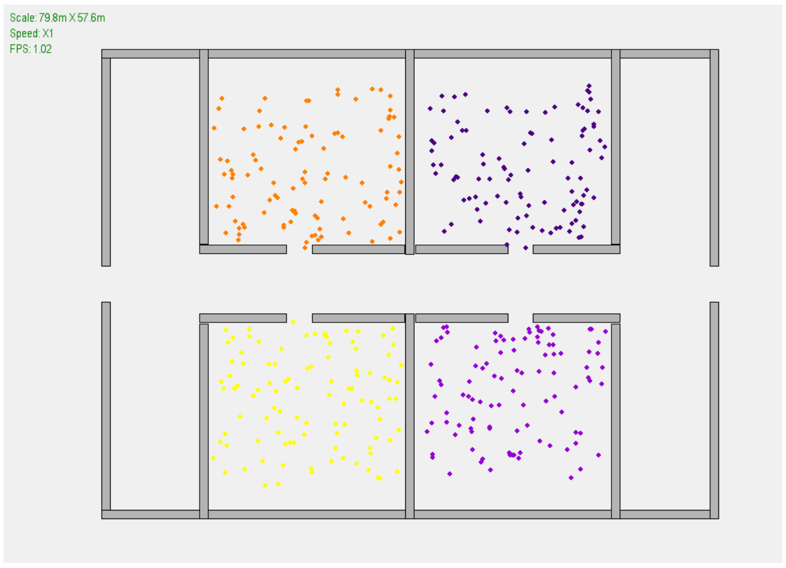


Figure 3.7: An example scenario with 400 agents in a simple office.

3.4.2 Path planning

Our neural network models rely on a global path planner to give it a general direction, hence a path needs to be generated for each agent before the simulation can start. For

this task the A*[14] algorithm is used, a path-planning algorithm that can find the shortest path in a weighted graph while using a heuristic to guide itself.
To use the A* algorithm the 2D environment needs to be represented as a search space, an underlying data structure that is used to plan a path to any given destination. A scene (Figure 3.8a) can be represented in many different ways such as rectangular grid (Figure 3.8b), quadtree (Figure 3.8c), convex polygons (Figure 3.8d), points of visibility (Figure 3.8e), and generalized cylinders (Figure 3.8f). For our implementation we use the rectangular grid approach.

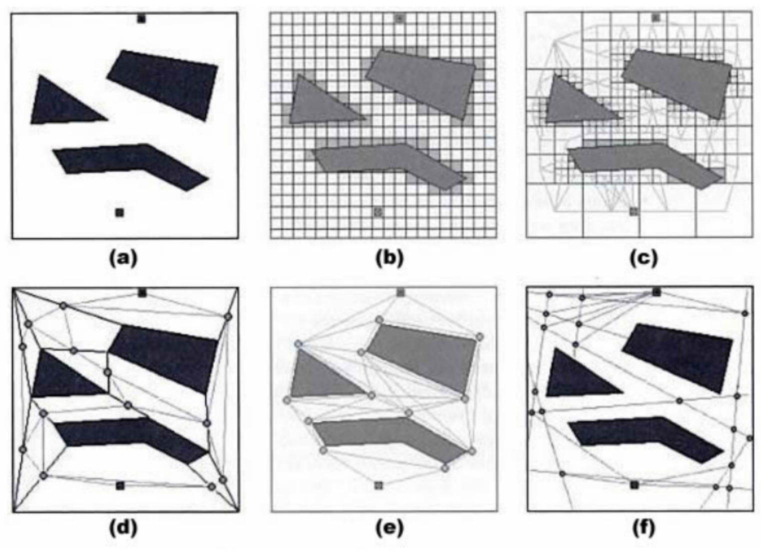


Figure 3.8: Five ways to represent search space [39].

The grid is constructed by dividing the scene into a grid of one meter by one meter tiles, subsequently every tile that touches an obstacle is removed from the grid. A graph is constructed from this grid by creating a node for each tile in the grid and creating an edge between all neighbouring tiles, that is if this tile exists in the grid. We have now constructed a graph of which each node represents a reachable geographical location in the scene and of which each edge represents the ability to walk between the two nodes that it links. The rectangular grid search space for the office example is shown in Figure 3.9. This illustration shows a few drawbacks of this method. Firstly not all the space in the scene is accessible: when a tile in the grid contains an obstacle the entire tile is considered not accessible even though part of the tile might not be an obstacle. Downsides of this include the fact that not all accessible space is considered for the path planning which might cause the algorithm to not find the actual shortest path, but more severely: no path can be found that passes an opening smaller than the tile size leading to the algorithm not finding a path for the agent, even though there could be one. An example of such scenario is shown in Figure 3.10. Secondly this method requires a large graph which introduces two problems: bad performance of the algorithm as it has a larger graph to traverse and a larger memory requirement, which could be a problem for larger scenes. These problems could all be solved using one of the alternative search space representations mentioned earlier, but the rectangular grid was chosen due to ease of implementation.

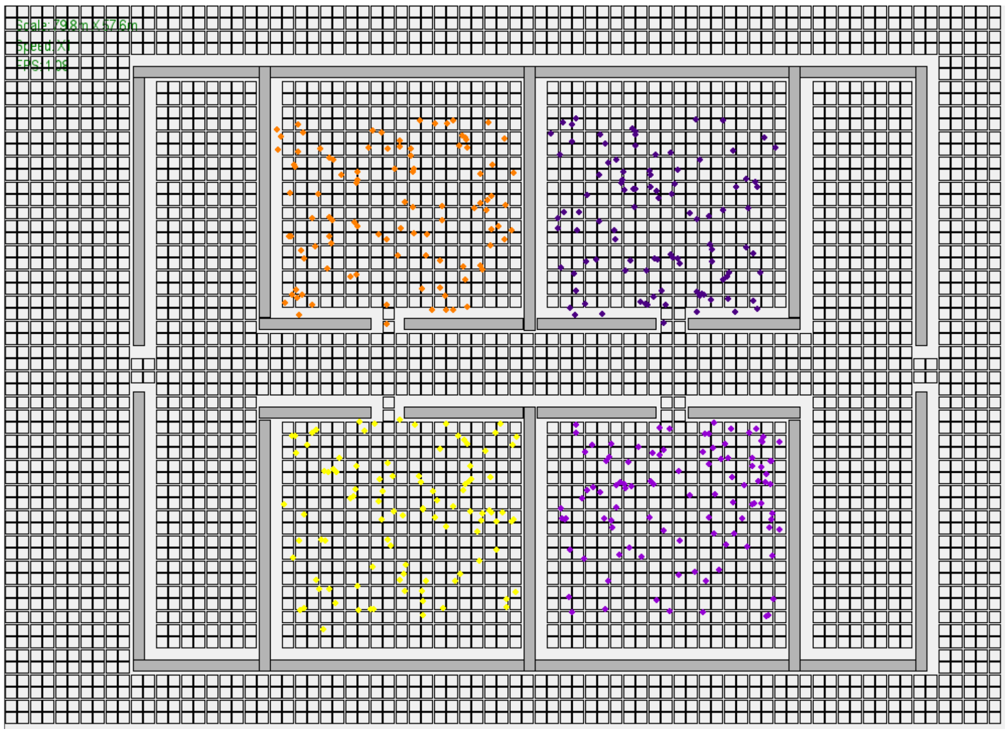


Figure 3.9: Illustration of the search space representation for the office example.

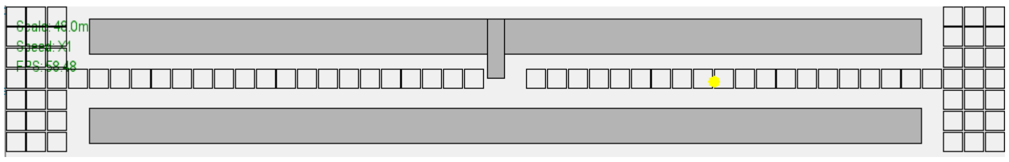


Figure 3.10: Example of how the path planner can fail due to the shortcomings of the rectangular grid method.

Having constructed our search space we can now move on to constructing the paths for our agents. This is achieved by using A* to construct the shortest path between each of the agents goals in their goal sequence using the search space graph. This will result in a series of nodes, corresponding to tiles in the 2D space of our scene. The array of locations the agent should visit is computed by taking a random point in each of the tiles in their path. The random point is taken from a normal distribution around the center of the tile. The randomness is introduced to reflect the randomness of real crowds. The normal distribution around the center is chosen as this leads to the most natural looking trajectories: real humans do not zigzag when walking straight ahead, this is possible when choosing an uniform random point on tiles. The paths generated for each agent are shown in Figure 3.11. It should be noted that these paths are mere guidelines that are inputted in the neural network, which is entirely in charge of generating velocities for the agents. This leads the agents to follow trajectories that diverge from these paths, depending on their neighbours as they are trying to avoid these.

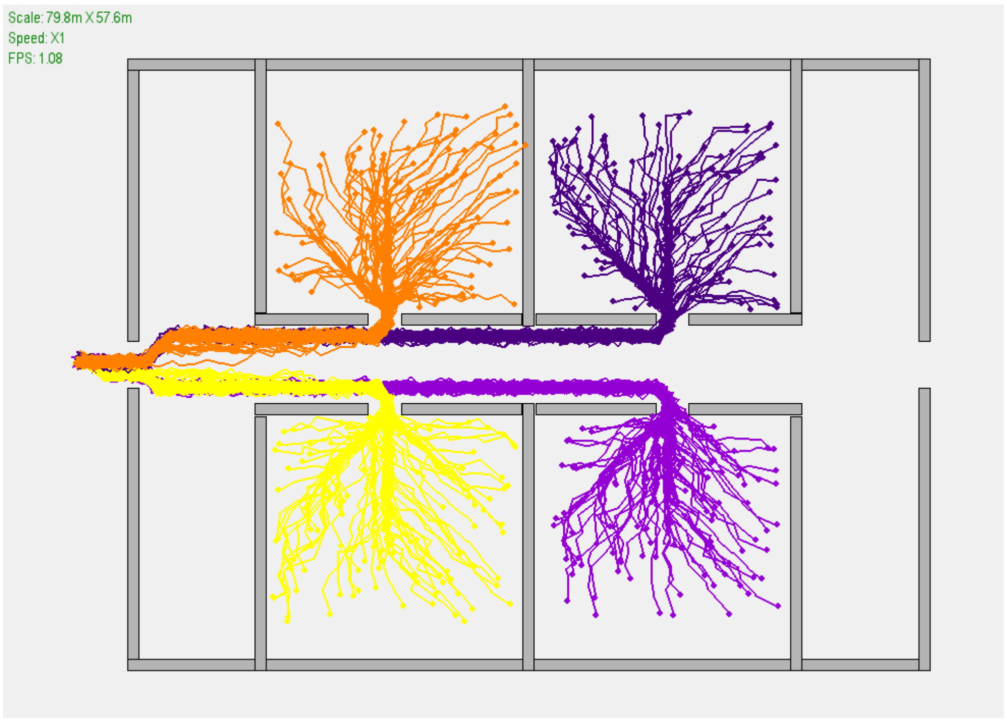


Figure 3.11: Illustration of the paths generated for the agents for the office example.

This method of global path planning is not ideal as a lot of improvements can be made in both the quality of the paths and the performance of the algorithm. Furthermore, extending this method to three dimension could create issues as the complexity increases drastically. The paths generated in this way do however suffice for our purposes as we focus on the neural network training aspect. Having initialized our scene and generated the paths necessary for our neural network we can move on to the actual simulation loop. Here the movements of the agents are constantly computed and updated as they attempt to reach their goals.

3.4.3 Simulation Loop

The simulation loop is a process that gets called each frame of the simulation, here the agents positions get updated, states are constructed to input into the neural network and agents get assigned new velocities in order to reach their goal and evade collisions. The agents initial positions and velocities vectors are set according to the values described in the XML scenario, their current goal is set to the first goal in their path, which was computed in the previous section. The simulation loop has two main parts: agents position update and agents velocity update. The position update happens every timestep whereas the velocities get only updated once a specific amount of time has elapsed. This is controlled by the *update time* parameter, setting this to a smaller value will update velocities more often, leading to agents being able to better react to their environment. As we will see later on, the velocity update step is rather expensive in scenarios with many agents, so lowering this parameter causes worse performance. This could be taken into consideration when using this method for real time simulation.

Position update

The first step of the loop is to update each agents position. This is done by adding their velocity to their position. The velocity is scaled by the amount of seconds that has passed since the last frame. After the positions are updated it is checked whether the agent has reached the the next position in its path. We consider the next position reached if the agent is within one meter of the point. If this is the case the agents current goal is set to the next position in the agents path.

Velocity update

This part of the simulation is where the neural networks come into play, as we use them to calculate new velocity vectors for our agents. Having trained our networks earlier using state-action pairs extracted from video data, we now need to construct the state for each agent in our simulation in order to generate an action for them that will lead them to find their goal while avoiding collisions with other agents and objects. Since we don't generate a single action, but an array of possible actions, we need to also carefully select one of the actions that gets suggested by the network. This is also done in the velocity update step.

State construction Firstly we need to collect the necessary data and construct a state for all of our agents. To recap: for the feedforward neural network this is the speed scalar, angle to the current goal and the positions and velocities for each neighbour. For the recurrent neural this is the distance travelled in the x and y directions since the last timeframe, the angle to the goal, the time passed since the previous timeframe and the positions of the neighbours. To get consistent results from our network, the same pre-processing steps as described in Section 3.2.1 and Section 3.3.1 for the feedforward an recurrent method respectively, are repeated on this data.
An important and costly step that is required for both methods is the finding of each agents k-nearest neighbours within his FOV as we need their positions and velocities. In our we get an agents k-nearest neighbours in a brute force manner: we loop over all other agents an select all agents within the FOV, subsequently the k closest neighbours are selected. This is a very expensive operation as it has a complexity of $O(n^2)$ with n the amount of agents. For simulations with lots of agents this can take a long time, hampering our ability to simulate real time. Work has been done to improve the speed

of k-nearest neighbour search in continuously moving environments, such as Continuous K-Nearest Neighbor Search for Moving Objects [70]. Implementation of one of these methods is however out of the scope of this project.
Having found our k-nearest neighbours all of the information to be included in the state can be computed for a specific agent as follows:

- **Feedforward method:** The current speed vector v_t is simply the current speed of the agent, angle ψ can be computed by finding the directed angle between the agents current goal vector and (1, 0) and the relative positions (Δx_i and Δy_i) and speeds ($\Delta v_{i,x}$ and $\Delta v_{i,y}$) of the neighbours can be found by subtracting the positions and speeds of the k-nearest neighbours from those of the agent. This is followed by the rotation and normalization steps from Section 3.2.1.
- **Recurrent method:** The distance travelled since the last timeframe (Δx and Δy) can simply be computed by subtracting the agents current position from the agents previous position, angle ψ can be computed by finding the directed angle between the agents current goal vector and (1, 0), Δt is simply the time passed since the previous frame and the relative positions (Δx_i and Δy_i) of the neighbours can be found by subtracting the positions of neighbours from those of the agent. In the recurrent method the last t states are kept in an array as the neural networks requires t-timesteps as input. Parameter rotation and normalization are done like in Section 3.3.1.

For each agent in the simulation we compute their state as described above, these states are then fed into the neural network. The network responds with an array of k actions for each agent. To complete the velocity update step we need to pick the actions which best suits the needs of the agent.

Action selection Given k action vectors for an agent we need to select the best one, given the agents neighbours and the obstacles in the scene. We do so by constructing a list of usable actions from the k actions. This is done by attempting to predict collisions with the agents neighbours and the scene's obstacles for the given actions.
For each action a_i we consider point $p_{t_1,i} = p_t + \Delta t a_i$, with p_t the current position of the agent. So this is the point the agent would be at the next update time if it would have taken this action. If the line segment between p_t and $p_{t_1,i}$ intersects with any line of any object in the scene, action i is removed from consideration. We can consider the obstacles in our simulation as a collection of line segments. Since we only work with static objects we can optimize this process by sorting all line segments of our obstacles by the x-coordinate of their rightmost endpoint. If we need to check whether a line segment intersects any obstacle, we can loop through the array of line segments until we find a line whose rightmost x-coordinate is larger than the leftmost x-coordinate of the line segment for which we want to check intersection. For the part of the array we just looped through we do not need to check intersection, as this is impossible. For simulations with a lot of obstacles this can lead to a significant gain in performance.

Having removed all the actions that would lead to a collision with a wall we now select an action for our agent by defining a reward function that should be evaluated and maximised for each agent. The reward function $r(\mathbf{a})$ attempts to create as much distance between other agents while moving the fastest towards the goal:

$$r(a_i) = \frac{1}{n}\sum_{n} d(\mathbf{p}_t + \Delta t \cdot \mathbf{a}_i, \mathbf{x}_n + \Delta t \cdot \mathbf{v}_n) + (d(\mathbf{p}_t, \mathbf{g}) - d(\mathbf{p}_t + \Delta t \cdot \mathbf{a}_i, \mathbf{g})) \tag{3.9}$$

with n the number of neighbours of the agent, $d(\mathbf{a}, \mathbf{b})$ the distance between point $\mathbf{a}$ and $\mathbf{b}$, $\mathbf{x}_n$ and $\mathbf{v}_n$ the position and velocity of neighbour n and $\mathbf{g}$ the current goal of the agent. The first part of this formula is the average distance to each neighbour, to reward collision avoidance behaviour. The second part is the amount of distance this action would take the agent closer to their goal, this is included to ensure agents do not simply wander off to avoid collisions without caring about their goal. This function is evaluated for each action and the action with the largest reward value is selected. This function is a reflection of peoples desire to avoid dense sections of crowds while still attempting to reach their goal. To change the behaviour of the model the two terms could be weighted. For example if we want agents to care less about colliding with other agents, we weight the goal reaching parameter more.
Having selected the usable actions from the k actions we now select the action that will be the agents new velocity by sorting the actions that are left by their angle to the desired goal vector. The action with the smallest angle is eventually chosen. When no suitable action is found the speed vector is set to (0,0) so the agent can wait until neighbours blocking his path have moved. Please note that this not perfect, as agents can easily change direction over the one second period that is considered. This action selection method is just a quick way to remove any bad actions as we assume that the neural network generated actions that where mostly already good.
To summarize: our simulation method follows these steps:

1. Initialize simulation by reading agents and obstacles from an XML file
2. Apply global path planning to generate a path for each agent to their goal
3. Repeat until all agents have reached their goal or a fixed timeout time has passed:
    (a) Update agents positions
    (b) Update agents velocities:
        i. Construct each agents state and input it into the neural network to generate possible actions
        ii. Select an appropriate action from the actions generated in the previous step

Chapter 4

Validation and comparison

In this chapter we attempt to show that we accomplished the goal we set out to achieve in this thesis, which was to train a neural network to simulate a realistic crowd, using data collected from real crowds.

4.1 Methodology

For the evaluation of our method we based ourselves on SteerBench [59], a benchmark framework for objectively evaluating steering behaviors for virtual agents. Their work describes a diverse set of test cases, metrics of evaluation, and a scoring method that can be used to compare different steering algorithms. To prove their scoring method is an effective quantity for scoring simulation methods, they conducted a user study. Participants were shown the simulation generated by two algorithms for a number of test cases and were asked to give their opinion about which algorithm was more intelligent in each test case. The algorithm deemed best by their scoring method always coincided with the algorithm chosen by users.

4.1.1 Metrics

SteerBench details a number of metrics that can be computed for an agent over the course of a run, a complete list can be found in the paper along with a description of each. The metrics are divided in three primary metrics, metrics that that apply to each scenario, and detailed metrics, metrics that apply to specific scenarios. An example of a detailed metric is 'degrees turned', which could be analyzed in a scenario where an agent is expected to go to his goal in a straight path. SteerBench computes the metrics for each agent, whereas we compute them for all agents in one simulation run for a given scenario. These metric can only be used to compare the performance of simulation methods on the same scenario. For our purposes we selected the three primary metrics:

1. **Collisions:** This is the number of unique collision events that happen in total in the simulation. In general, fewer collisions indicate more realistic steering behavior.

2. **Time Efficiency:** Time efficiency measures how quickly the agents are able to reach their goal destinations. The quicker the agents reach their goals, the more time efficient the agent is. It is well accepted that efficient behaviors are natural. We do however need to combine efficiency metrics, since an agent teleporting to its goal would be very time efficient, but not very realistic. Time Efficiency is measured as the average time it takes an agent to reach their goal.

3. **Effort Efficiency:** Effort efficiency measures how much effort an agent spends to reach its goal destinations. The less effort an agent spends, the more effort efficient the agent is. We compute the effort efficiency as the average of the total kinetic energy spent by each agent over each sample for that agent. The formula for kinetic energy is: $E_k = \frac{mv^2}{2}$, for the mass m we pick a value of 70, as this is not given in our simulation.

Furthermore we select an additional three metrics from the detailed metrics to get a better insight in how our model is performing. We feel that the selected metrics apply to all scenarios, just like the three metrics above. These extra metrics are:

4. **Agents Finished:** It is imperative that all agents (that can reach their goal) reach their goal in a good crowd simulation algorithm. Agents getting stuck is an indication of unwanted behaviour. We measure this as the percentage of agents that has reached their final destination.

5. **Collision Depth:** Slight collisions are more natural then very severe collisions, just as people grazing shoulders is more likely to occur in a crowd then people full on bumping into each other. This is not reflected in the number of collisions metric, for this reason we introduce the collision depth metric. It is computed as the average maximum depth of all unique collisions.

6. **Second-order effort efficiency** The second order effort efficiency is another effort efficiency metric that is calculated by the average of the total acceleration each agent has. This stems from the fact that it is unnatural for an agent to constantly accelerate and decelerate. Agents should, when possible, attempt to keep a constant velocity.

In SteerBench the three primary metrics are combined to create a benchmark score, we opt to analyze each metric separately to get better insight in how our models behave.

4.1.2 Scenarios

The metrics described above need to be calculated over a simulation run for a given scenario. In this section we present five varied scenarios that we feel, capture most of the crowd simulation tasks. These scenarios are also taken from the SteerBench paper. For each scenario we show the global paths computed for each agent by the A* algorithm, without randomness. The shown paths are thus not the paths actually followed by the agents, but simply the paths they attempt to follow given their goal.

Three agent confusion

The first scenario we check is a simple scenario where three agents cross each others paths in the middle of the scene. This scene looks simple, but it forces the model to avoid collision with multiple agents. The scenario is illustrated in Figure 4.1.

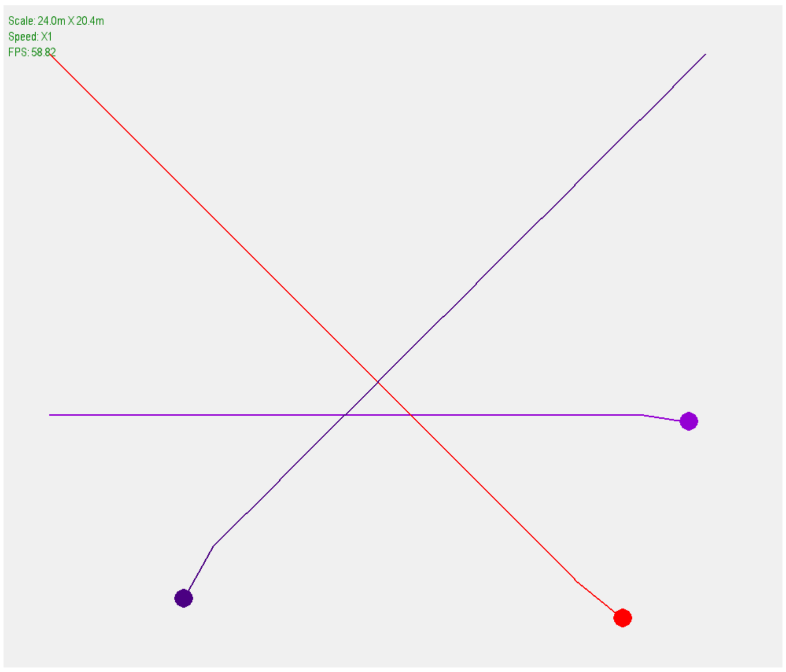


Figure 4.1: Three agent confusion scenario

Two agent squeeze

In this scenario there are two agents, both agents start and goal positions are on the same horizontal axis. The agents are in a narrow hallway and thus needs to pass each other, while not colliding. This scenario is illustrated in Figure 4.2

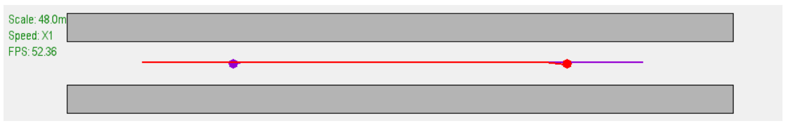


Figure 4.2: Two agent squeeze scenario

Urban environment

The third scenario we consider contains an urban environment with buildings distributed evenly over the space. 50 agents are randomly spawned in the area with all random goals.

This is a realistic simulation of a city, with sparsely distributed agents that should be able to avoid collisions. This scenario is illustrated in Figure 4.3.

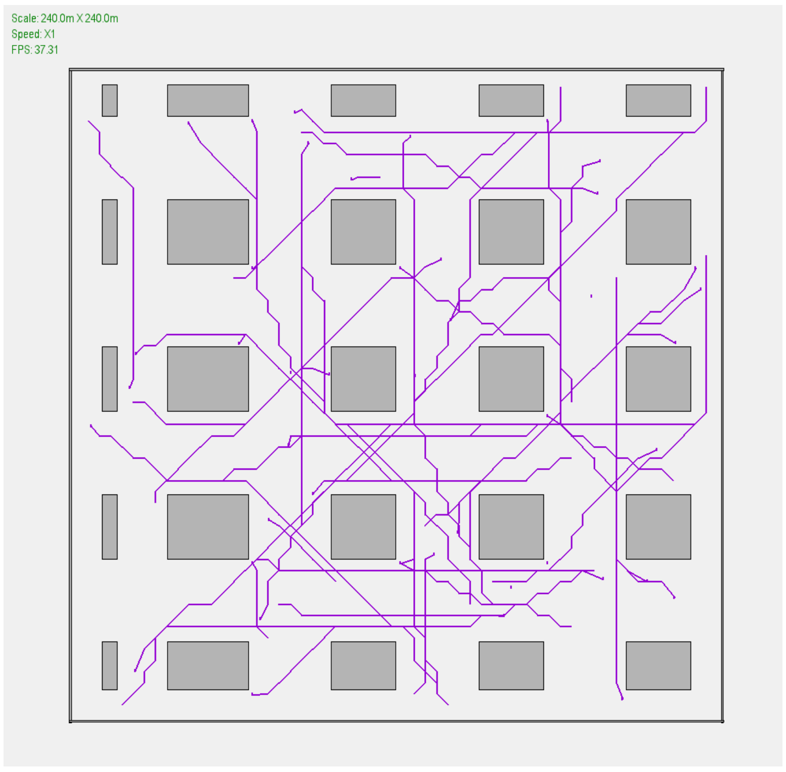


Figure 4.3: Urban environment

Office evacuation

Next we analyze a more dense scenario. Here we have an office with four rooms, in each room there are 100 agents. The agents in the scenario are all going to the same position outside of the office. This scenario is illustrated in Figure 4.4.

Four way oncoming obstacle

The final scenario we analyze is a less realistic one, but it could offer insight on how models perform under very difficult situations. The consists of four hallways that form a cross and meet each other in the middle, in the middle there is an obstacle leading to a very narrow passage. In each hallway there are 100 agents, all agents are trying to reach to the hallway opposite of their starting position.

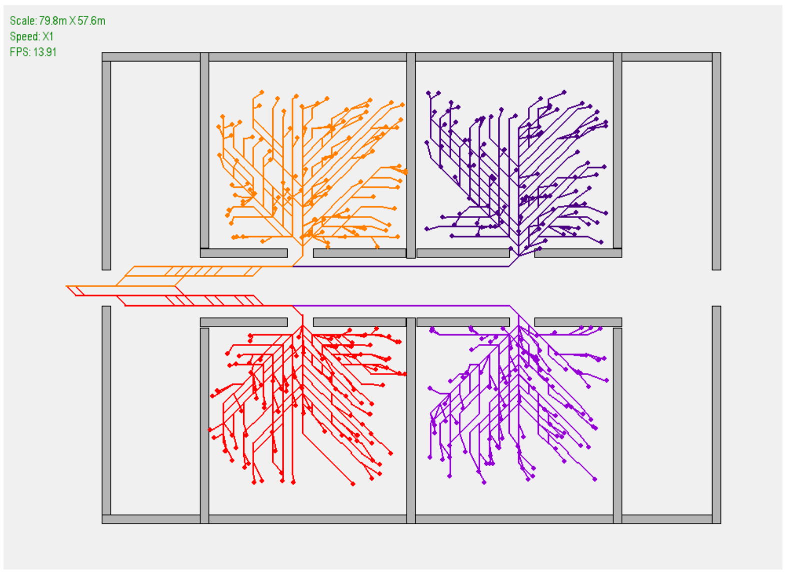


Figure 4.4: Office evacuation scenario

4.2 Experiments

Since we have described the metrics we will measure and on which scenarios we will measure these, we can move on to gathering statistics on our model. For this it is important to note that there is some randomness in our method, such as the randomness in the path generation and more importantly, some scenarios spawn agents on random locations. To achieve consistency and reproducability in our experiments we used a fixed random seed for each experiment. We first describe the different models we will analyze followed by the results for these models.

4.2.1 Models

We run the metric test on multiple different version of both our feedforward and recurrent models. These versions all have different external parameter values. The external parameters being: k the amount of actions to be generated, n the amount of neighbours considered, FOV the field of view for which neighbours are considered and t, the amount of samples in time that are considered (only for the recurrent model). In this way we hope to gain insight in how, for example, increasing the number of considered neighbours affects the behaviour of the model. In an attempt to verify our work in comparison to existing models we also compare the results to that of the Social Forces model as described in Section 1.1.1. We based our implementation of the Social Forces model on the work of Moussaid et al. [40] who observed pedestrian behaviour to quantify the effects of interactions. The laws describing the interactions were then formalized in mathematical terms and implemented in the social force model.
The different models we analyze are shown in Table 4.1, each of the models is given a

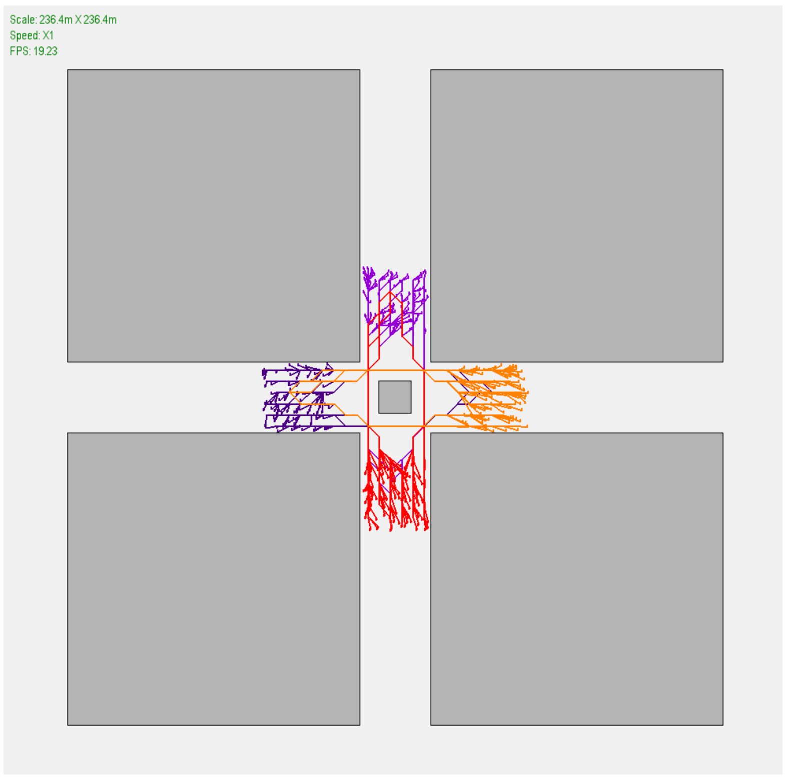


Figure 4.5: Four way oncoming obstacle

unique name so they can easily be referenced.

4.2.2 Results

Here we present the results recorded in all of the experiments. We first discuss the results for each scenario to gain an insight in what type of model performs well in what type of situation. This is supplemented by a discussion on a per-metric basis.

Per Scenario Discussion

In this section we present the recorded metrics for each model per scenario. For each scenario the results are presented by six barcharts. The bars in the chart are colour coded depending on the type of network the model they represent use: blue stands for the feedforward networks and green stands for the recurrent networks (red is the Social Forces model for comparison).

Three agent confusion The results for the three way scenario are shown in Figure 4.6. All models succeed in steering their agents towards their goal destinations, they all do this in times comparable to the social forces model. Three of our models do however

Model	**Neighbours**	**FOV**	**Actions**	**Lookback**	**Name**
Feedforward	5	3	5	/	ff_n_5_f_3
Feedforward	10	3	5	/	ff_n_10_f_3
Feedforward	5	6	5	/	ff_n_5_f_6
Feedforward	10	6	5	/	ff_n_10_f_6
Recurrent	5	3	5	5	rec_n_5_f_3
Recurrent	10	3	5	5	rec_n_10_f_3
Recurrent	5	6	5	5	rec_n_5_f_6
Recurrent	10	6	5	5	rec_n_10_f_6

Table 4.1: Table summarizing the different models we analyze.

cause a collision between agents, whereas the social forces model does not. For two of these models the collisions were not that severe as the depth was only small. The three models with a collision all considered ten neighbouring agents in their input. Even though there are only two other agents in this scenario this could be an early indication that presenting too much information that is not necessary to the network could lead to it not being able to learn behaviours properly. Finally the amount of kinetic energy spent is comparable to the social forces model for each of our own model, this in contrast to the average total acceleration which is much higher for our models in comparison to the social forces model.

Two agent squeeze The second scenario we analyzed was the two agent squeeze scenario, the results of which are shown in Figure 4.7. Again all agents finished for each model, all of our feedforward models managed to do so in a faster time than the social forces model, although by a small margin. The recurrent models on the other hand where all slower. Three of our feedforward models and one recurrent model managed to outperform the social forces model since they had no collisions whereas the social forces model did not manage to steer the agents away from each other. The collision depth for the social forces model was however only very small as with the one feedforward model that had a collision. The collisions of the two recurrent models were more severe. For the effort efficiency we notice that the social forces model again massively outperforms our models when it comes to average total acceleration. For the kinetic energy the results are about the same across the board with the recurrent models doing slightly better.

Urban scene The next considered scenario is a bit more expansive with a number of agents navigating a urban scenario. This time not every model managed to steer every agent to its goal, only the social forces and three of the feedforward models managed this. For the other models a few agents did not reach their goal, possibly getting stuck on an object. A few of the feedforward models had a lower average time spent per agent compared to the social forces model the other models where slower but only by small margins. All of the feedforward models got a number of collisions comparable to that of the social forces model. The recurrent two recurrent models with the smaller

FOV experienced a significantly lower number of collisions than the social forces model, whereas the two models with the higher FOV had a significantly higher number. We note that the two models with the small number of collisions are the two models that spent the most time on average for each agent. For the total average acceleration the trend of the two previous scenarios is continued, as our models again have many fluctuations in velocity leading to a high total average acceleration. For the average total kinetic energy most models score on par with the social forces algorithm, with the exception of the feedforward network with the high fov and high neighbour count.

Office evacuation This is the first scenario with an extremely dense crowd, the results are shown in Figure 4.9. We see that, when it comes to number of collisions, the social forces model performs better than all of our models, although some of the recurrent models manage to achieve a number of collisions that is only slightly higher. When we look at the chart for amount of time spent per agent we notice that the feedforward models all reach their goal faster on average, but apparently at the cost of more collisions. The recurrent models again score similar to the social forces model. No model managed to steer all agents towards the goal, but all of the recurrent models did slightly worse than the others. There is a noticeable difference in the average collision depth between the social forces model and the rest, with the former performing better that the others. For the effort efficiency the trend of the previous three scenarios is broken when it comes to acceleration, as now several of our models outperform the social forces model. There are however two models that do significantly worse, these being the models that use ten neighbours. For the average kinetic energy all models score roughly the same except for the five neighbours-six fov model, as it massively outperforms, this model also did very good on the acceleration score.

Four way oncoming obstacle The results of the final scenario we analyzed are shown in Figure 4.10. Firstly we notice that the social forces model and all feedforward models manage to steer all agents towards their goal whereas the recurrent models all manage to do so for most agents. The number of collisions around the same for all models with the two recurrent models with smaller FOV and the feedforward model with higher FOV and higher amount of neighbours performing better than the social forces model. Like in previous scenarios the feedforward models spent less time on average per agent. Just as in the previous example the social forces model managed to have less severe collisions than the other models. When it comes to average acceleration most models perform around the same as the social forces model except for ten neighbour three FOV model which has a huge spike. For the average kinetic energy the feedforward models perform around the same as the social forces model and the recurrent models all do way worse.

Per Metric Discussion

In the remainder of this section we take a look at the results for each metric separately in an attempt to see if there is a connection between the external parameters of the crowd simulation models and the characteristics of the scenarios the were deployed in.

Collision count and depth and time spent When it comes to number of collisions we notice that the best performing models are the two recurrent models with an FOV of three, indicating that quality of the simulation is positively impacted by extra temporal information supplied in the recurrent models and that only considering neighbours in a smaller FOV helps the networks performance as it is given less irrelevant information to process. This last point is made clear by the fact that the recurrent models with an FOV of six on average perform way worse on most scenarios. This is not the case for the feedforward models however, as there does not seem to be a connection between their external parameters and number of collisions.
We can clearly see the results of the recurrent models with smaller FOV avoiding collisions better in the charts for average time spent, as they almost always spend more time on average. This indicates that these models prefer taking a longer path if it means avoiding collisions better.
For the average collision depth it is again difficult to see a connection between the external parameters and the depth. We can only remark that when there are a lot of collisions as is the case in the last two scenarios, the average collision depth for all models seems to even out at a number that is slightly worse than that of the Social Forces model.

Percentage of agents that finished Managing to steer as much agents as possible towards their intended location is an important trait of a good crowd simulation method. For the two less dens scenarios with not a lot or no of obstacles all models manage to steer all agents towards their goal. For the urban scenario all recurrent models perform slightly worse than the Social Forces model and feedforward models with the exception of the feedforward model with the FOV of six and ten neighbours, which performs significantly worse than the rest. The trend of the recurrent models performing worse than the feedforward and Social Forces model continues for the more dense scenarios, albeit by a small margin.

Effort Efficiency metrics The effort efficiency metrics, average total kinetic energy spent and average total acceleration are good indications of erratic behaviour and unnatural movement.
When it comes to average total acceleration we notice that for the not so dense crowds the Social Forces model massively outperforms our models with a much lower value. We hypothesise this is due to the fact that the Social Forces model generates an acceleration to be added to the current velocity, whereas our model generates a new velocity to replace the previous velocity leading to larger fluctuations in subsequent velocity values. Potential solutions for this include: redefining our model to generate acceleration at each step as opposed to generating new velocities or adding a term that rewards a lower difference in velocity to Equation 3.9 that is used to select an appropriate new velocity. Remarkably, for higher density crowds, our models perform around the same as the Social Forces model, with some even performing better. This is not the case for

the models with a small FOV and high neighbour count, as they still have a way larger average total acceleration.
For average total kinetic energy there is a clear difference between the two types of models and the Social Forces model. For the three non-dense scenarios the recurrent models manage to outperform the other two, especially the two models with the smaller FOV. For the dense office scenario most models perform about the same with the exception of the feedforward model with the large FOV and small neighbour count. In the dense four way obstacle scenario the trend is reversed as the recurrent model is outperformed by the Social Forces and feedforward models.

4.3 Conclusion

The goal of this work was to explore the possibility of applying the power of neural networks to data-driven crowd simulation, and more specifically to the behavioural aspect of the simulation. This way we hoped to achieve more realistic, more versatile and at run-time computationally less expensive models. We first examined a number of existing crowd simulation methods, both using neural networks and other methods. We do this to see what work already exists to use as a starting point for our method and to be able to situate our work in the broad spectrum of crowd simulation approaches. In order to apply neural networks to the problem of crowd simulation, we then thoroughly investigated the inner workings of neural networks, how they are trained and what variations exist on them.
We then moved on to proposing two neural network models that can generate new velocities for agents in a crowd in order to steer them towards their goal while evading collisions with other agents. The first of which was a feedforward neural network that was essentially a combination of the work of Song et al. [60] and Wei et al. [66] taking, what we feel is, the best of both approaches. In an attempt to exploit the temporal nature of the data collected on the crowds we modified this model to create a recurrent neural network, providing the model with more data hopefully resulting in better actions generated by the network. Furthermore an entire crowd simulation process was developed in which these two methods tie in.
To confirm that we managed to achieve the goals that we set out we set up a validation method based on the work of Sign et al. [59], SteerBench. From this we selected a number of metrics regarding the movement and interactions of the agents and a number of simulation scenarios over which these metrics could be computed. These metrics are based off the behaviour of real humans in crowds and could thus be used to evaluate the realism of our method as well as to compare the methods with each other and with already existing work. And this is exactly what we did, comparing four of our feedforward and four of our recurrent models, each trained and executed with different settings with the already established Social Forces model described in Section 1.1.1.
From these experiments we conclude that there is a large variety in how our different models perform for the different scenarios. But we do conclude that most of our models manage to perform on par with the established Social Forces model, even in some cases outperforming it. Especially the recurrent models with the smaller FOV manage to perform well for most metrics, leading us to conclude that these have the best potential. This is not to say that all tested models managed to perfectly accomplished the targets set out at the beginning of this thesis. A good indication of a realistic crowd is collision avoidance, this should be easy in a very low density situation, yet three of our models fail to do so in the three agent confusion scenario. Furthermore the average total acceleration metric indicated that in low density situations the agents did not manage to generate smooth movement, even though the situation should be easier to navigate.
In conclusion, we managed to use the knowledge gained in the fields of crowd simulation and artificial neural networks to develop two crowd simulation models that can gener-

ate realistic crowds and rival the performance of an established method such as Social Forces. We confirmed this by setting up an easy to use and extend validation framework based off quantitative metrics computed over simulation runs of specific scenarios.

4.4 Future work

There is still a lot more work to be done towards realising the full potential of neural networks in crowd simulation. We should start at our own model, where the effects of the different external parameters (number of actions, FOV and number of neighbours) should be studied more thoroughly to better understand their effects in different situations. This could be complemented by a more complete study of what factors influence a persons behavioural choices in a crowd to potentially develop an alternative state representation of a humans perception in a crowd.
Something that completely lacks in our model are higher level behaviours. By this we mean the model does not take emotions such as fear into account when generating movement. Neither is behaviour such as group formation or leadership considered in the crowd simulation model. Emotions could be included by training separate neural networks on datasets captured of people experiencing those emotions, this way you would have a *fear* model and a *relaxed* model for example that could both be used in the simulation depending on the state of the agent they are used for. Another option to quantify emotions on a zero to one scale and input these along with the rest of the state. To accomplish this we would also need to provide these values to the neural neural network when training it, meaning that these *emotional values* need to be known for our training data, which is not a simple task. To model grouping behaviour we could apply a similar tactic, encoding for example, the center of the group this agent belongs to in their state space. But this again would mean that this information should somehow be available when training the neural network, which is not trivial unless we work with data recorded from experiments we set up ourselves, since we would be able to assign groups to participants. And the same holds for leadership, where we could easily encode *influence* values for each of an agents neighbour, with the requirement that we can somehow derive this information from the training data.
In this work we presented two microscopic models, meaning that the model considers one agent and its environment, this incurs copious amount of pre-processing as each agents state needs to be constructed. A potential solution to this problem is to create a neural network model that can process all agents at once, generating an array of new velocities for these agents. This would mean that we are working with an array of inputs and outputs of undefined length. This is what recurrent networks excel at, they do however assume the data is sequential (like for example an English sentence), which is not the case for an array of agents. For example, the information on the seventh agent needs to propagated to the third agent just as much as the other way around. In order to deal with this bidirectional recurrent neural networks [54] can be used, which propagate information both backwards and forwards in time.
Furthermore we feel that our method could benefit from more training data as it would

allow it to learn more about the interactions of humans in crowds. If we were to implement a method for tracking pedestrians from video images this would supply us with practically unlimited data to train our model on, as video data from crowds is plentiful. Crowd simulation is more than just generating 2D movement for agents, among others it also includes animating these agents. We could extend our model by generating not only new velocities, but also rotations for each of the agents joints, considering the agent now is a 3D character skeleton. This way the model could generate hyper-realistic crowds that show behaviour such as an agent rotating his body sideways to get through a narrow part of a crowd. Training such a network would require detailed input information on the agents skeletons, this could be achieved by recent works on pose estimation [7]. It should be noted that in order to generate complex data like this a larger network has to be used that will need more training data to understand the intricate movements of the human body.

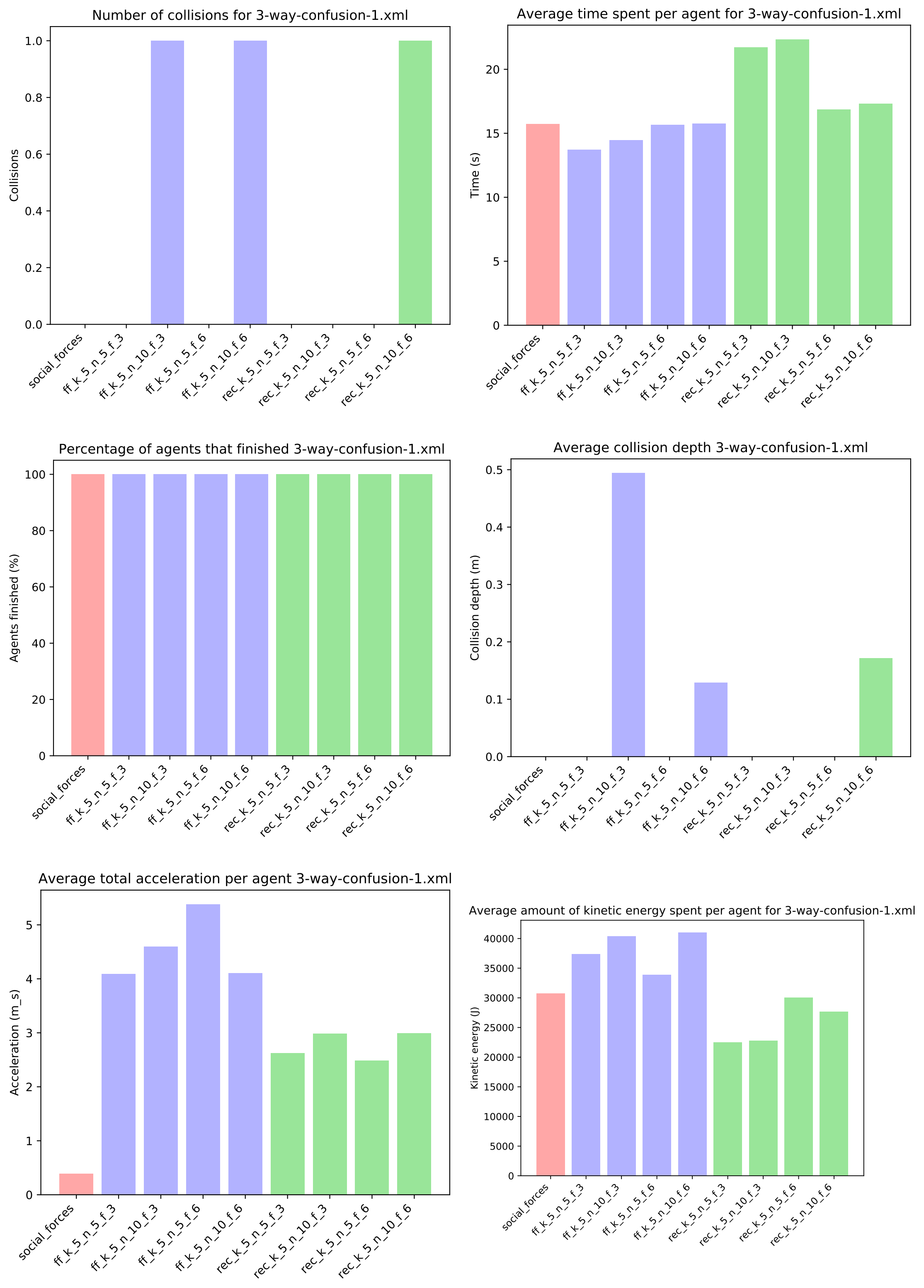


Figure 4.6: Metrics computed for the different models in the three agent confusion scenario.

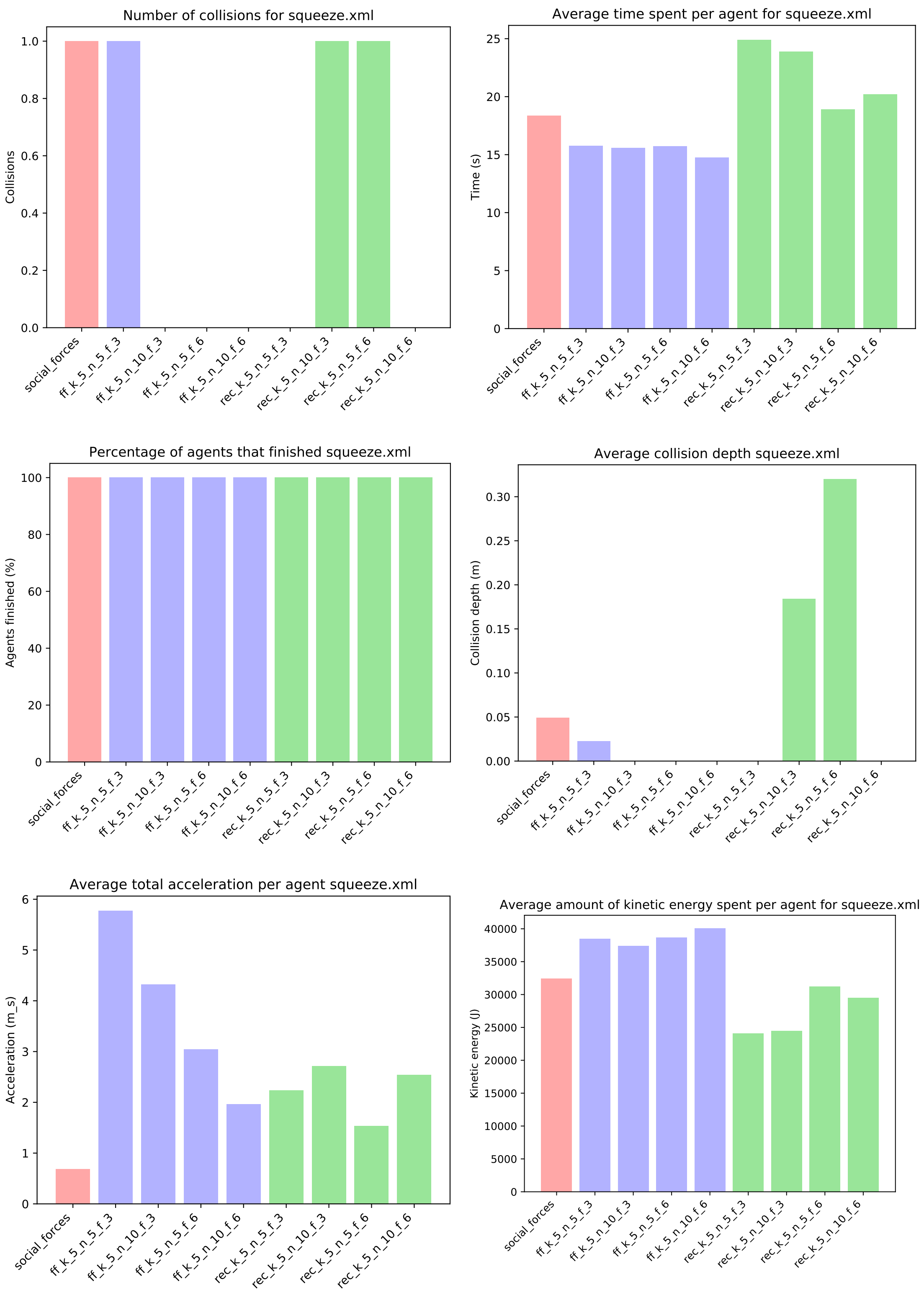


Figure 4.7: Metrics computed for the different models in the two agent squeeze scenario.

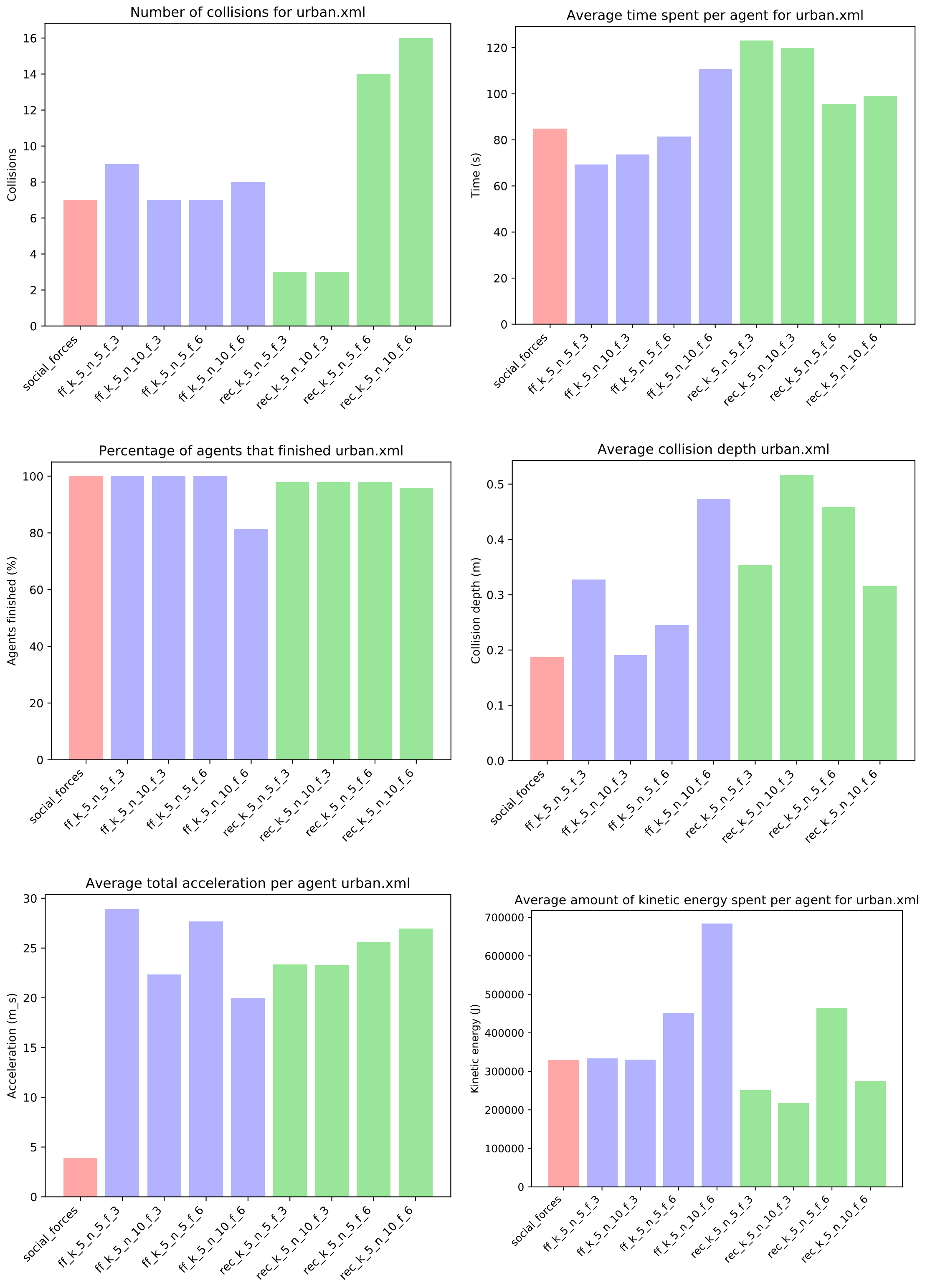


Figure 4.8: Metrics computed for the different models in the urban scenario.

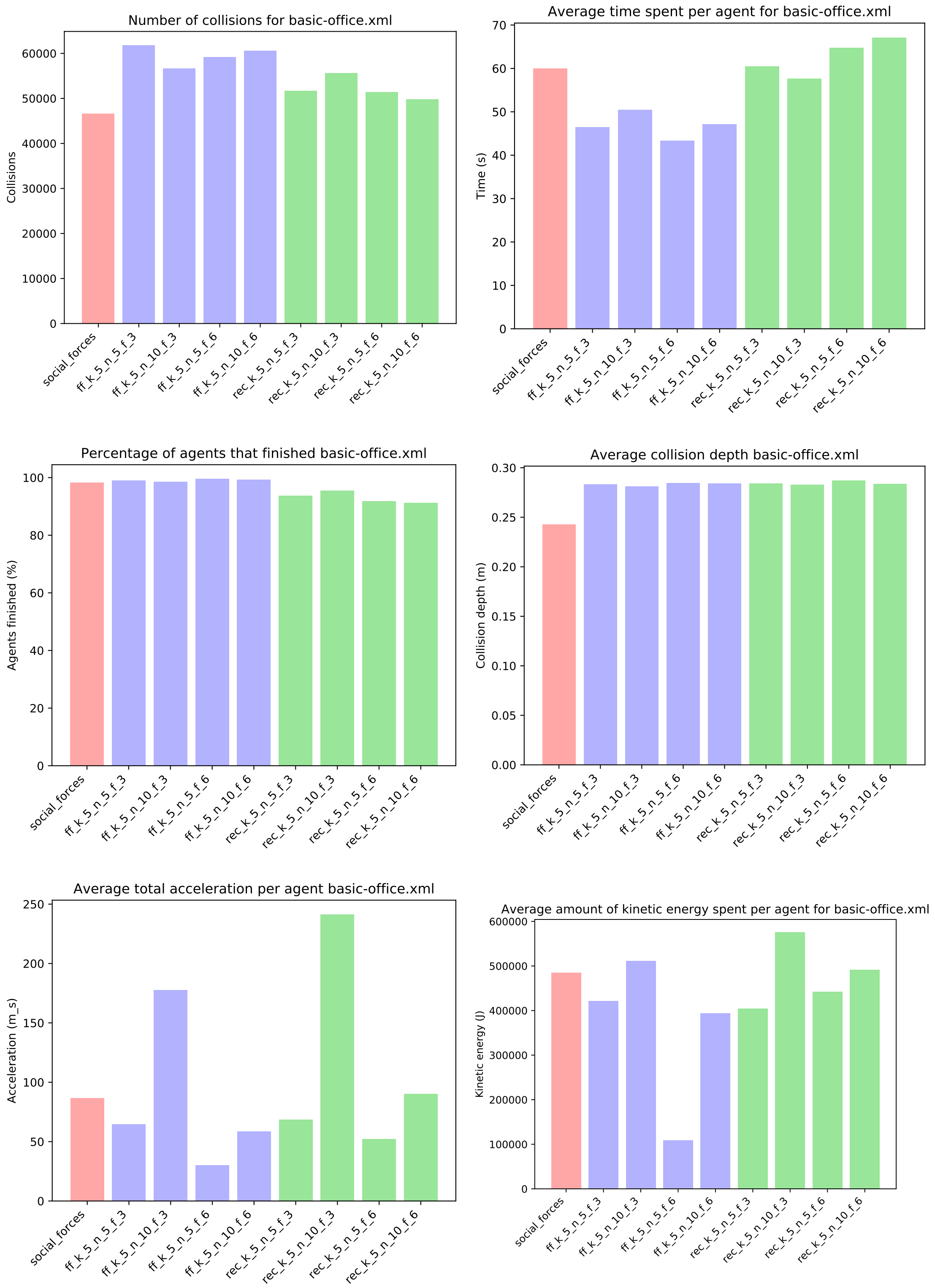


Figure 4.9: Metrics computed for the different models in the office evacuation scenario.

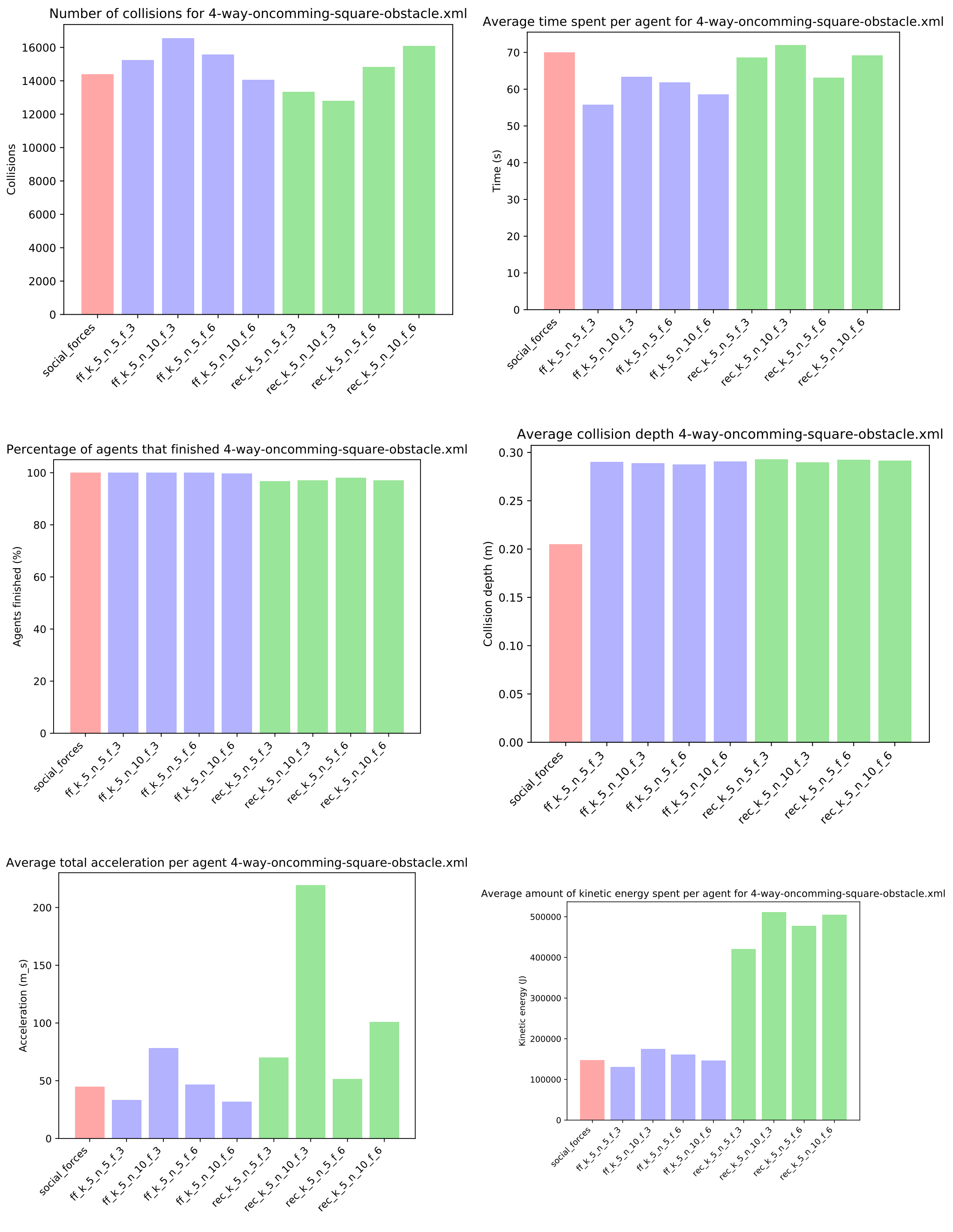


Figure 4.10: Metrics computed for the different models in the four way oncoming obstacle scenario.

Appendix

A Data Gedreven Simulatie van Menigtes door middel van Neurale Netwerken: Nederlandse Samenvatting

De mensheid heeft al lang interesse in het gedrag van mensen in een grote menigte, hierdoor is er ook al een veel onderzoek gedaan naar welke factoren hier effect op hebben, en welke regels het gedrag van een mens in een menigte bepalen. Met de opkomst van computers hebben we de mogelijkheid om deze regels te gebruiken om de bewegingen van mensen in menigtes te simuleren, deze tak van de computer wetenschappen heet crowd simulation. Crowd simulation heeft tal van toepassingen zoals het genereren van menigtes om te gebruiken in films of games, of om het gedrag van mensen in een gebouw te testen, wat nuttig kan zijn voor architecten. Naast het simuleren van bewegingen omvat crowd simulation nog andere belangrijke taken: onder meer het genereren van geloofwaardige animaties voor de *agents*, gevarieerde modellen en texturen creëren en het efficient renderen van alle *agents*.
De focus van deze thesis ligt op het genereren van de bewegingen van de mensen in de menigte. Specifiek willen we gebruik maken van data die geregistreerd is over echte menigtes, gemotiveerd door een overvloed aan beschikbare luchtbeelden door de opkomst van drones, en recente verbeteringen in voetganger traceermethoden; wat er voor zorgt dat positionele data over de bewegingen van mensen in menigtes makkelijk te verkrijgen is. Data gedreven crowd simulation is niets niews, maar de meeste methodes die binnen deze tak van crowd simulation gemaakt zijn maken gebruik van computationeel dure opzoek operaties tijdens de simulatie. Daarbij, herhalen deze methoden enkel data uit hun database, zonder te leren over de onderliggende regels die het gedrag van de menigte bepalen. In deze thesis proberen we machine learning toe te passen op het probleem van data gedreven crowd simulatie, zo hoeven we het computationeel dure gedeelte van het simulatie proces slechts één maal uit te voeren tijdens het trainen van het model, en kunnen we de onderliggende regels van het menselijke gedrag afleiden om zo betere simulaties te verkrijgen voor situaties niet aanwezig in de training data.
Een van de meest krachtige modellen binnen de machine learning zijn de neurale netwerken, om deze reden focussen we in deze thesis ons op het trainen van neurale netwerken teneinde het genereren van realistische bewegingen voor mensen in een menigte. We stellen twee modellen voor: een standaard feedforward netwerk en een recurrent netwerk. Beide genereren acties voor een bepaald persoon in de menigte gebaseerd op hun omge-

ving, ze werken dus op het agent-niveau. Hiernaast stellen we ook een framework voor waarbinnen deze netwerken kunnen opereren om een volledig crowd simulatie process te voorzien. Hiervoor overlopen we de bestaande literatuur omtrend neurale netwerken en crowd simulatie.

A.1 Crowd Simulation

De simulatie van menigtes is een breed veld, om hier onderzoek binnen te doen is het belangrijk om te kijken naar welke methodes er al bestaan. We doen dit om zo te bekijken op welke werk we kunnen verder bouwen en om onze eigen methode te plaatsen binnen de correcte context. Uit onderzoek blijkt dat we crowd simulaties kunnen classificeren gebaseerd op twee criteria: de lengte van de simulatie en het aantal personen dat we simuleren. Verder kunnen we de methodes gebruikt voor deze simulatie classificeren volgens de granulariteit waarmee ze de menigte modelleren. Hierin ondescheiden we *agent*-, *entity*- en *flow*-gebaseerde methodes, in volgorde van aflopende autonomie die de agents hebben over hun bewegingen in de simulatie. Deze verschillende types van simulatie methodes worden gebruikt voor verschillende combinaties van de classificatie schalen voor crowd simulations, dit is geïllustreerd op Figuur 1.1.

Agent-based

In *agent*-gebaseerde simulaties worden de agenten gemodelleerd als autonome individuen met een bepaalde intelligentie graad. Binnen dit soort simulaties herkennen we: eerste orde en tweede orde methodes. Eerste orde methodes bepalen interacties aan de hand van de posities van de agenten. Zo is er het Boids model van Reynolds [48], dat aan de hand van drie regels het gedrag van een zwerm vogelachtige wezens bepaald. Verder is er het Social Forces model [24] dat *agents* in de menigte als particles beschouwd waar krachten op inwerken die gebaseerd zijn op factoren zoals het bereiken van de vooropgestelde doelen en het ontwijken van andere mensen en obstakels. Tweede orde methodes maken gebruik van de posities en snelheden van *agents* om zo beter botsingen te vermeiden door middel van lineare extrapolatie van hun paden. Methodes binnen dit veld genereren ofwel afstotende krachten gebaseerd op toekomstige botsingen om deze zo te vermeiden of zoeken botsing-vrije nieuwe snelheden aan de hand van mogelijke toekomstige botsingen.

Entity-based

Entity-gebaseerde methodes zien de *agents* als homogene entiteiten wiens bewegingen worden beïnvloed door globale en locale regels. Veel van deze methodes maken gebruik van cellulare automata in combinatie met kansdistrubuties die bepalen wanneer personen van cel kunnen wisselen.

Flow-based

Flow-gebaseerde methodes simuleren alleen abstacte gedragingen van de personen in een menigte. Deze methodes zijn ideaal voor menigtes met een groot aantal *agents* doordat ze computationeel veel goedkopen zijn, dit is echter ten koste van het detail waarmee we de menigte kunnen bekijken, dit gaat ook vaak gepaard met onrealistische bewegingen. Dit type methode kan bijvoorbeeld gebruik maken van grafen, tegels met potentiaal-velden of een fysieke simulatie gebaseerd op de verplaatsing van vloeistof.

Data-driven

Verder bestaat er zoiets als data-gedreven crowd simulation, omdat het uitdrukken van het menselijk gedrag door middel van vaste regels tot een gebrek aan complexiteit kan leiden, proberen onderzoekers binnen deze tak van crowd simulation gebruik te maken van data die verzameld is door voetgangers in een menigte te traceren. Methodes binnen did veld stellen een database op van voorbeelden verzameld uit de data. Tijdens de simulatie worden voorbeelden uit deze database opgevraagd die gelijkaardig zijn aan die in de simulatie, diens acties worden dan overgenomen. Complexere methodes maken gebruik van intelligentere opvragingen en locale regressies om betere acties te genereren, maar er wordt bijna altijd gebruik gemaakt van dure opvragingen uit de database.

Neurale netwerk methoden

Er zijn reeds een aantal methoden die data gedreven crowd simulatie verwezelijken aan de hand van neurale netwerken, getraind door middel van supervised-learning. Voornamelijk onderscheiden we hier methodes die aan de hand van de staat van de omgeving van de *agent* een gepaste actie genereren door middel van lineare regressie en methodes die classificatie toepassen op de staat van de agents om zo een gepaste actie te kiezen uit een overeenkomende categorie uit de voorbeelden database.
Het is belangrijk om een voorgestelde crowd simulatie methode te kunnen evalueren. Dit is niet altijd eenvoudig aangezien het gedeeltelijk subjectief is. Men kan bijvoorbeeld bepaalde situaties simuleren waarvan we weten hoe een echte menselijke menigte zou reageren om dan de resultaten kwalitatief te vergelijken. Zo een scenario is bijvoorbeeld een aantal mensen dat door dezelfde deur wil gaan: we weten dat er een circelvormige ophoping zal vormen rond de deur. Een andere mogelijkheid is om gebruik te maken van gegevens verzameld over video data van echte crowds, hiervoor kunnen we een aantal metrieken voor bepalen en afwijkingen tussen de echte data en simulatie zoeken. Ten slotte is het ook mogelijk om een aantal metrieken die eigen zijn aan een echte menigte te berekenen voor de gesimuleerde menigte. Zo zal een echte menigte altijd zo weinig mogelijk botsingen proberen veroorzaken en zo weinig mogelijk versnellen en vertragen.

A.2 Artificiële Neurale Netwerken

Een artificieel neural netwerk is een wiskundige constructie die, gegeven een input vector, hier complexe niet-lineaire berekeningen op kan doen om vervolgens een output vector te genereren. Deze berekeningen worden gedaan door middel van een netwerk van neuronen, deze neuronen verwerken waardes en geven deze door aan de neuronen waarmee hun output verbonden is. De neuronen zijn geordend in lagen, volgens de volgorde waarmee ze in aanraking komen met de input. Elk netwerk heeft een input en output laag, hiertussen kunnen eventueel verborgen lagen liggen. De verwerking die een neuron doet is de gewogen som van al zijn inputs, op deze som wordt de niet-lineaire activatie functie toegepast. Een neuraal netwerk heeft een specifiek gewicht voor elke verbinding tussen twee neuronen. Het netwerkt berekend zijn output door waardes vanuit de inputlaag door te propageren naar de volgende lagen tot er ze bij de output zijn.

Neurale netwerk training

De echte bruikbaarheid van neurale netwerken komt wanneer we ze gaan trainen om specifieke taken uit te voeren. Dit trainin gebeurt door de gewichten van de verbindingen tussen de neuronen aan te passen om zo een gepaste omzetting tussen input en output waardes te creëren. In deze thesis ligt de focus op supervised learning, dit is wanneer we een machine learning model trainen aan de hand van een dataset bestaande uit input-output paren waartussen we de relatie willen aanleren. Hiervoor is het nodig dat we een maat hebben voor de performantie van het netwerk, hoe goed het netwerk er in is geslaagd om de relatie tussen de input-output paren te leren. Zo een maat heet de *loss function* en het trainingsproces is gebaseerd op het kiezen van een set gewichten die ervoor zorgen dat deze functie zo klein mogelijk is. Hiervoor wordt gebruik gemaakt van een proces genaamd *gradient descent*, een wiskundige optimalisatie methode die aan de hand van de partiële afgeleide van de loss function de parameters aanpast zodat deze functie het hardste daalt. De partiële afgeleide van de loss functie tussen de verwachte en voorspelde outputs van het neurale netwerk ten opzichte van de parameters van het netwerk wordt berekend aan de hand van *backpropagation*, een process waarbij de fout gemaakt door het netwerk achterwaarts door het netwerk wordt gepropageerd. In de praktijk wordt standaard *gradient descent* amper gebruikt aangezien er betere versies zijn die gebruik maken van adaptieve leersnelheden voor de verschillende parameters. Bij het trainen van een neuraal netwerk is het belangrijk om ervoor te zorgen dat het netwerk niet leert om data te herhalen, maar dat het leert over de onderliggende regels die de verbanden tussen input en output bepalen, dit heet *generalization.* Om er voor te controlleren dat een netwerk omkan met nieuwe data wordt er een deel van de dataset onthouden bij het trainingsproces, dit deel kan dan later gebruikt worden om te verifiëren dat het netwerk kan generaliseren. Verder zijn er nog een aantal methodes die kunnen gebruikt worden om dit te voorkomen, zoals dropout, early stopping en regularization.

Hyperparameter optimalisatie

Tot dusver zijn enkel de interne parameters aan bod gekomen, die bepalen hoe een netwerk berekeningen maakt. Er zijn echter nog een aantal hyperparameters die de topologie van een netwerk bepalen en hoe het getrained wordt. Enkele voorbeelden hiervan zijn het aantal verborgen lagen, het aantal neuronen in iedere laag en de gebruikte activatie functie. Deze kunnen manueel gekozen worden, maar er betaan ook echter een aantal methoden die dit automatisch doen. Voor deze methodes wordt er een derde deel van de dataset onthouden, en wordt het netwerk telkens met een andere configuratie van hyperparameters getrained, dit getrainde netwerk wordt vervolgens geëvalueerd op deze derde dataset. De parameter configuratie met de beste performantie wordt dan geselecteerd. Dit heet hyperparameter optimalisatie en er zijn verschillende varianten van. Verschillende methodes worden getypeerd aan de hand van hoe ze de volgende parameter configuratie selecteren. Dit kan willekeurig of aan de hand van een schatting van de functie van de parameters ten op zichte van de performantie.

Recurrente Neurale Netwerken

Vooralsnog hebben we enkel het *feedforward* neurale netwerk beschouwd, er zijn echter een heel aantal varianten hierop. Een populaire daarvan is het recurrente neurale netwerk, een netwerk structuur die gedefinieerd is over een aantal tijdstappen. Op elke tijdstap wordt er een input opgenomen samen met de output van de vorige tijdstap, wat het in staat stelt om te redeneren in de tijd. Deze structuur is zeer interessant voor ons werk omdat trajecten van personen in een menigte gedefinieerd zijn over een tijdsdimensie. Deze netwerken worden getrained door het netwerk uit te rollen in de tijd, waardoor we een gewoon *feedforward* netwerk krijgen, wat via *backpropagation* en *gradient descent* getrained kan worden. Deze netwerken zijn vaak niet goed in informatie onthouden over meer dan een paar tijdstappen , hierdoor zijn er een aantal speciale versies van de RNN ontwikkeld zoals het *Long-Short Term Memory* [25] en de *Gated Recurrent Unit* [10].

A.3 Toepassing van neurale netwerken op crowd simulation

In dit hoofdstuk ontwikkelen we twee neurale netwerk modellen voor crowd simulatie, gebaseerd op de kennis uit de twee vorige hoofdstukken. Vooraleer we overgaan tot het definiëren van onze methodes is het belangrijk om het probleem naukeurig af te bakenen. Zo stellen we dat onze simulatie plaats vind in een tweedimensionele omgeving en bestaat uit een aantal *agents*, dewelke allemaal een positie, een doel en een snelheid hebben (bestaande uit een tweedimensionele vector die de deelsnelheid in de x en y richting aangeeft). Verder omvat een simulatie scenario een aantal obstakels, deze worden gemodelleerd als een rechthoek. Het doel van de crowd simulation methode is om tijdens elke update stap een nieuwe tweedimensionele snelheid te genereren voor elke agent. Bij het genereren van deze snelheid moet er rekening gehouden worden met de drie belangrijkste doelen van een persoon in een menigte: het bereiken van het doel, het ontwijken van andere voetgangers en het vermeiden van botsingen met obstakels. Hiervoor presenteren wij twee neurale netwerk modellen die dit trachten te verwezelijken. Deze modellen werken op het *agent*-niveau, wat betekend dat de input van het netwerk een representatie is van de omgeving en interne doelen van de agent en de output de nieuwe snelheid van deze agent.

Feedforward methode

De eerst voorgestelde methode maakt gebruik van een volledig geconnecteerd *feedforward* neuraal netwerk, getrained op state-action paren die uit een een echte menigte zijn verzameld. Belanrijk is hier dat we een juiste voorstelling kiezen van de staat van een bepaalde *agent*, er moet genoeg informatie aanwezig zijn om het verband te leggen tussen de bijhorende actie, evenals mag er niet teveel informatie zijn aangezien dit ervoor kan zorgen dat het netwerk te traag of niet leert. De gekozen data in de agent staat is: zijn eigen snelheidsvector, de hoek tussen gewenste bewegingsrichting en de relatieve locaties en snelheden van zijn dichtbijzijnste buren binnen een bepaalde straal. Het aantal beschouwde buren en de straal zijn twee parameters die ingesteld kunnen worden en invloed hebben op het gedrag van het model. De actie is simpelweg de snelheid tijdens de volgende tijdsstap. Om onze data beter te generaliseren roteren we de snelheidsvector van elke sample naar de x-as, de andere parameters worden ook volgens deze hoek geroteerd. Verder passen we normalisatie toe op de relatieve afstanden van de buren zodat we een afwezige buur kunnen encoderen als nul waarde. Tenslotte worden er voor elke state-action pair een aanal gelijkaardige states gezocht, van deze states worden de acties aan de actie van de state toegevoegd. Dit reflecteerd het niet deterministische menselijke keuze proces en geeft ons meer opties tijdens de simulatie. Het netwerk wordt getrained door het Tree-structured Parzen Estimator hyperparmeter optimalisatie algoritme.

Recurrente methode

Het recurrente model is gelijkaardig aan het vorige model, nu wordt een actie echter niet bepaald door één state maar door een reeks van vorige states van de agent. Dit zorgt ervoor dat de states ook anders moeten worden opgesteld. In plaats van de snelheidsvector bevat de state nu het verschil in positie met de vorige stap. Om inschattingen over snelheden te maken wordt de verlopen tijd sinds de vorige state meegegeven. Voor de buren wordt ook enkel nog het verschil in positie meegegeven, zo beschikt het netwerk over informatie van het volledige pad van elke buur gedurende de beschouwde tijdsperiode. De gewenste bewegingsrichting blijft hetzelfde. Weer passen we de rotatie en conbinatie van gelijkaardige states toe, ditmaal echter niet over de state-action pairs maar over de gehele reeks van beschouwde states. Het netwerk wordt op dezelfde manier getrained.

Simulatie

Simulaties worden geïnitialiseerd door een scene uit een XML bestand te lezen. Deze scene bevat een aantal agents en obstakels. Er kunnen ook agent- en ostakel regios gedefinieerd worden die dan bevolkt worden door een aantal willekeurige entiteiten.Om complexe scenarios te navigeren wordt er voor elke agent een reeks van doelen opgesteld aan de hand van het A* algoritme. Hiervoor wordt de scene opgedeeld in een grid, met een tegel voor elk deel van de scene dat geen obstakel bevat.
De simulatie loop voert elke iteratie een positie update uit, hier worden de agents simpelweg volgens hun huidige snelheid verplaatst. Er kan ook een snelheid update gebeuren,

dit is niet noodzakelijk elke iteratie en hangt af van een vaste parameter. Tijdens deze update stap wordt er voor elke agent een state opgesteld die vervolgens als input dient voor het neurale netwerk. Het opstellen van deze state is een dure operatie omdat voor elke agent zijn dichtste buren gevonden moeten worden. Eens de state voor elke agent is opgesteld kunnen we het neurale netwerk gebruiken om voor elke agent een lijst van mogelijke acties te genereren. Een actie wordt gekozen door elke actie die voor een botsing met een obstakel zorgt te verwijderen uit de lijst. De overgebleven acties krijgen elk een score toegewezen die acties beloont die ver naar het doel bewegen terwijl ze zo veel mogelijk afstand van buren houden.

A.4 Validatie en vergelijking

Tenslotte valideren we ons model door een aantal verschillende versies te vergelijken met het Social Forces model.

Methode

Als vergelijking en validatie methode definieren we een aantal metrieken gebaseerd op SteerBench [59]. Deze metrieken zijn: aantal botsingen, tijd gespendeerd, geleverde inspanning, aantal agents dat hun doel bereikt, diepte van botsingen en totale versnelling. Deze metrieken quantificeren de eigenschappen van een menselijke menigte: botsingen worden vermeden, mensen willen zo snel mogelijk met zo min mogelijk moeite hun doel bereiken. Deze metrieken werden gemeter over een aantal uiteenlopende scenarios voor vier verschillende versies van het feedforward- en recurrente model en voor het Social Forces model.

A.5 Resultaten

Uit de resultaten blijkt dat er veel variatie is tussen hoe de verschillende modellen in verschillende situaties presteren. In de meeste gevallen slagen de modellen er in om gelijkaardige reslutaten als het Social Forces model te behalen, en in sommige gevallen zelfs betere resultaten. Specifiek zijn het de recurrente modellen die gebruik maken van een kleiner gezichtsveld die het beste scoren. Een metriek waar onze modellen slecht op scoren is de totale versnelling, hierop doen ze het vaak aanzienelijk slechter dan het Social Forces model. Dit duid erop dat de agents vaak grote verschillen in snelheid ervaren, wat duid op niet efficiënte bewegingen.

Bibliografie

[1] C. G. Atkeson, A. W. Moore, and S. Schaal. Locally weighted learning. *Artif. Intell. Rev.*, 11(1-5):11–73, Feb. 1997.

[2] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate, 2014.

[3] C. A. L. Bailer-Jones, R. Gupta, and H. P. Singh. An introduction to artificial neural networks. 2001.

[4] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, March 1994.

[5] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl. Algorithms for hyper-parameter optimization. In *Proceedings of the 24th International Conference on Neural Information Processing Systems*, NIPS'11, pages 2546–2554, USA, 2011. Curran Associates Inc.

[6] C. G. BROYDEN. The Convergence of a Class of Double-rank Minimization Algorithms 1. General Considerations. *IMA Journal of Applied Mathematics*, 6(1):76–90, 03 1970.

[7] Z. Cao, T. Simon, S.-E. Wei, and Y. Sheikh. Realtime multi-person 2d pose estimation using part affinity fields, 2016.

[8] U. Chattaraj, A. Seyfried, and P. Chakroborty. Comparison of pedestrian fundamental diagram across cultures, 2009.

[9] S. Chenney. Flow tiles. In *Proceedings of the 2004 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '04, pages 233–242, Goslar Germany, Germany, 2004. Eurographics Association.

[10] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, Doha, Qatar, Oct. 2014. Association for Computational Linguistics.

[11] I. L. Dalal, D. Stefan, and J. Harwayne-Gidansky. Low discrepancy sequences for monte carlo simulations on reconfigurable platforms. In *2008 International Conference on Application-Specific Systems, Architectures and Processors*, pages 108–113, July 2008.

[12] W. H. E. Day and H. Edelsbrunner. Efficient algorithms for agglomerative hierarchical clustering methods. *Journal of Classification*, 1(1):7–24, Dec 1984.

[13] C. Delgado-Mata, J. I. Martinez, S. Bee, R. Ruiz-Rodarte, and R. Aylett. On the use of virtual animals with artificial fear in virtual environments. *New Generation Computing*, 25(2):145–169, Feb 2007.

[14] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *Intelligence/sigart Bulletin - SIGART*, 37:28–29, 12 1972.

[15] R. Fletcher. A new approach to variable metric algorithms. *The Computer Journal*, 13(3):317–322, 01 1970.

[16] R. Fletcher and C. M. Reeves. Function minimization by conjugate gradients. *The Computer Journal*, 7(2):149–154, 01 1964.

[17] C. Gauss. *Theoria motus corporum coelestium in sectionibus conicis solem ambientium.* Carl Friedrich Gauss Werke. sumtibus F. Perthes et I. H. Besser, 1809.

[18] D. Goldfarb. A family of variable-metric methods derived by variational means. *Mathematics of Computation*, 24(109):23–26, 1970.

[19] A. Graves, G. Wayne, and I. Danihelka. Neural turing machines, 2014.

[20] A. Gruslys, R. Munos, I. Danihelka, M. Lanctot, and A. Graves. Memory-efficient backpropagation through time. *CoRR*, abs/1606.03401, 2016.

[21] L. B. Gustave. *Psychologie des foules.* Ancienne Librairie Germer Baillière et Cie, 1895.

[22] S. J. Guy, J. van den Berg, W. Liu, R. Lau, M. C. Lin, and D. Manocha. A statistical similarity measure for aggregate crowd dynamics. *ACM Trans. Graph.*, 31(6):190:1–190:11, Nov. 2012.

[23] Z. Hao. Loss functions in neural networks. `https://isaacchanghau.github.io/post/loss_functions/`, Jun 2017.

[24] D. Helbing and P. Molnar. Social force model for pedestrian dynamics. *Physical Review E*, 51, 05 1998.

[25] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, 12 1997.

[26] R. L. Hughes. A continuum theory for the flow of pedestrians. *Transportation Research Part B: Methodological*, 36(6):507 – 535, 2002.

[27] R. L. Hughes. The flow of human crowds. *Annual Review of Fluid Mechanics*, 35(1):169–182, 2003.

[28] E. H. John Duchi and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12:2121 – 2159, 2011.

[29] I. Jolliffe. *Principal Component Analysis*. American Cancer Society, 2005.

[30] I. Karamouzas, P. Heil, P. van Beek, and M. H. Overmars. A predictive collision avoidance model for pedestrian simulation. In A. Egges, R. Geraerts, and M. Overmars, editors, *Motion in Games*, pages 41–52, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[31] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[32] T. M. Kisko, R. L. Francis, and C. R. Nobel. EVACNET4 USER'S GUIDE . `http://tomkisko.com/ise/files/evacnet/EVAC4UG.HTM`, 10 1998. [Online; accessed 27-April-2019].

[33] J. Lee, J. Won, and J. Lee. Crowd simulation by deep reinforcement learning. pages 1–7, 11 2018.

[34] K. H. Lee, M. Geol Choi, Q. Hong, and J. Lee. Group behavior from video: A data-driven approach to crowd simulation. volume 2007, pages 109–118, 01 2007.

[35] A. Lerner, Y. Chrysanthou, and D. Lischinski. Crowds by example. *Comput. Graph. Forum*, 26:655–664, 2007.

[36] K. Levenberg. A method for the solution of certain non-linear problems in least squares. *Quarterly of Applied Mathematics*, 2(2):164–168, 1944.

[37] T. List and R. Fisher. Cvml - an xml-based computer vision markup language. volume 1, pages 789–792, 01 2004.

[38] D. Marquardt. An algorithm for least-squares estimation of nonlinear parameters. *Journal of the Society for Industrial and Applied Mathematics*, 11(2):431–441, 1963.

[39] P. Mehta, H. Shah, S. Shukla, and S. Verma. A review on algorithms for pathfinding in computer games. 03 2015.

[40] M. Moussaïd, D. Helbing, S. Garnier, A. Johansson, M. Combe, and G. Theraulaz. Experimental study of the behavioural mechanisms underlying self-organization in human crowds. *Proceedings of the Royal Society B: Biological Sciences*, 276(1668):2755–2762, 2009.

[41] S. R. Musse and D. Thalmann. A model of human crowd behavior : Group interrelationship and collision detection analysis. In D. Thalmann and M. van de Panne, editors, *Computer Animation and Simulation '97*, pages 39–51, Vienna, 1997. Springer Vienna.

[42] Y. E. Nesterov. A method for solving the convex programming problem with convergence rate $o(1/k^2)$. *Soviet Mathematics Doklady*, 27:372 – 376, 1983.

[43] C. Olah and S. Carter. Attention and augmented recurrent neural networks. *Distill*, 2016.

[44] S. Paris, J. Pettré, and S. Donikian. Pedestrian reactive navigation for crowd simulation: a predictive approach. *Comput. Graph. Forum*, 26:665–674, 2007.

[45] E. Parzen. On estimation of a probability density function and mode. *Ann. Math. Statist.*, 33(3):1065–1076, 09 1962.

[46] N. Qian. On the momentum term in gradient descent learning algorithms. *Neural Networks*, 12(1):145 – 151, 1999.

[47] M. R Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *Journal cf Research of the National Bureau of Standards Vol*, 49, 12 1952.

[48] C. W. Reynolds. Flocks, herds and schools: A distributed behavioral model. *SIGGRAPH Comput. Graph.*, 21(4):25–34, Aug. 1987.

[49] P. J. Rousseeuw. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics*, 20:53 – 65, 1987.

[50] S. Ruder. An overview of gradient descent optimization algorithms. *CoRR*, abs/1609.04747, 2016.

[51] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533 EP –, Oct 1986.

[52] H. Salehinejad, J. Baarbe, S. Sankar, J. Barfett, E. Colak, and S. Valaee. Recent advances in recurrent neural networks. *CoRR*, abs/1801.01078, 2018.

[53] A. Schadschneider. Cellular automaton approach to pedestrian dynamics - theory. 2001.

[54] M. Schuster and K. K. Paliwal. Bidirectional recurrent neural networks. *IEEE Trans. Signal Processing*, 45:2673–2681, 1997.

[55] J. A. Sethian. A fast marching level set method for monotonically advancing fronts. *Proceedings of the National Academy of Sciences*, 93(4):1591–1595, 1996.

[56] D. F. Shanno. Conditioning of quasi-newton methods for function minimization. *Mathematics of Computation*, 24(111):647–656, 1970.

[57] J. Sherman and W. J. Morrison. Adjustment of an inverse matrix corresponding to a change in one element of a given matrix. *The Annals of Mathematical Statistics*, 21(1):124–127, 1950.

[58] Y. Shevchuk. Hyperparameter optimization for neural networks.

[59] S. Singh, M. Kapadia, P. Faloutsos, and G. Reinman. Steerbench: a benchmark suite for evaluating steering behaviors. *Journal of Visualization and Computer Animation*, 20:533–548, 09 2009.

[60] X. Song, D. Han, J. Sun, and Z. Zhang. A data-driven neural network approach to simulate pedestrian movement. *Physica A: Statistical Mechanics and its Applications*, 509:827 – 844, 2018.

[61] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.

[62] K. O. Stanley and R. Miikkulainen. Efficient reinforcement learning through evolving neural network topologies. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2002)*, page 9, San Francisco, 2002. Morgan Kaufmann.

[63] D. Thalmann and S. R. Musse. *Crowd Simulation.* Springer-Verlag London, 2 edition, 2013.

[64] A. Treuille, S. Cooper, and Z. Popović. Continuum crowds. *ACM Trans. Graph.*, 25(3):1160–1168, July 2006.

[65] ujjwalkarn. A Quick Introduction to Neural Networks. `https://ujjwalkarn.me/2016/08/09/quick-intro-neural-networks/`, August 2016. [Online; accessed 26-November-2018].

[66] X. Wei, W. Lu, L. Zhu, and W. Xing. Learning motion rules from real data: Neural network for crowd simulation. *Neurocomputing*, 310:125 – 134, 2018.

[67] G. Weiss, Y. Goldberg, and E. Yahav. On the practical computational power of finite precision rnns for language recognition, 2018.

[68] P. Widmer. Learning crowd behaviour with neuroevolution. 2017.

[69] D. Wolinski. *Microscopic crowd simulation : evaluation and development of algorithms.* PhD thesis, Université Rennes, Data Structures and Algorithms [cs.DS]., 12 2016. English. NNT : 2016REN1S036. tel-01420105.

[70] Yifan Li, Jiong Yang, and Jiawei Han. Continuous k-nearest neighbor search for moving objects. In *Proceedings. 16th International Conference on Scientific and Statistical Database Management, 2004.*, pages 123–126, June 2004.

[71] H. Yu and B. Wilamowski. *Hao Yu and B. M. Wilamowski, LevenbergMarquardt Training Industrial Electronics Handbook, vol. 5 Intelligent Systems, 2nd Edition, chapter 12, pp. 12-1 to 12-15, CRC Press 2011.*, pages 12–1 to 12. 01 2011.

[72] M. Zhao, S. J. Turner, and W. Cai. A data-driven crowd simulation model based on clustering and classification. In *2013 IEEE/ACM 17th International Symposium on Distributed Simulation and Real Time Applications*, pages 125–134, Oct 2013.

[73] S. Zhou, D. Chen, W. Cai, L. Luo, M. Low, F. Tian, V. Tay, D. Wee Sze Ong, and B. D. Hamilton. Crowd modeling and simulation technologies. *ACM Transactions on Modeling and Computer Simulation*, 20:20, 10 2010.