MODELS FOR PEDESTRIAN TRAJECTORY PREDICTION AND NAVIGATION IN DYNAMIC ENVIRONMENTS

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Jeremy Kerfs

May 2017

COMMITTEE MEMBERSHIP

TITLE:	Models for Pedestrian Trajectory Prediction and Navigation in Dynamic Environments
AUTHOR:	Jeremy Kerfs
DATE SUBMITTED:	May 2017
COMMITTEE CHAIR:	Professor John Seng, Ph.D. Department of Computer Science
COMMITTEE MEMBER:	Professor Franz Kurfess, Ph.D. Department of Computer Science
COMMITTEE MEMBER:	Associate Professor Lubomir Stanchev, Ph.D. Department of Computer Science

ABSTRACT

Models for Pedestrian Trajectory Prediction and Navigation in Dynamic Environments

Jeremy Kerfs

Robots are increasingly taking on roles alongside humans. Before robots can accomplish their tasks in dynamic environments, they must be able to navigate while avoiding collisions with pedestrians or other robots. Humans are able to move through crowds by anticipating the movements of other pedestrians and how their actions will influence others; developing a method for predicting pedestrian trajectories is a critical component of a robust robot navigation system. A current state-of-the-art approach for predicting pedestrian trajectories is Social-LSTM, which is a recurrent neural network that incorporates information about neighboring pedestrians to learn how people move cooperatively around each other. This thesis extends that model to output parameters for a multimodal distribution, which better captures the uncertainty inherent in pedestrian movements. Additionally, four novel architectures for representing neighboring pedestrians are proposed; these models are more general than current trajectory prediction systems. In both simulations and real-world datasets, the multimodal extension significantly increases the accuracy of trajectory prediction. One of the new neighbor representation architectures achieves state-of-the-art results while reducing the number of both parameters and hyper-parameters compared to existing solutions. Two techniques for incorporating the trajectory predictions into a planning system are also developed and evaluated on a real-world dataset. Both techniques plan routes that include fewer near-collisions than algorithms that do not use trajectory predictions. Finally, a Python library for Agent-Based-Modeling and crowd simulation is presented to aid in future research.

ACKNOWLEDGMENTS

There are few open-source tools or methods for trajectory prediction, but Anirudh Vemula shared his Tensorflow implementation of Social-LSTM on Github. His well-written code served as a reference for my own implementation, and I appreciate his contribution to the open-source community and aid in developing this thesis. The communities behind Edward, Mesa, and OpenMVG projects were immensely helpful during my research.

The donors of the Loyal Order of Propeller Heads (LOOP) were instrumental in mentoring me and supporting me throughout my time at Cal Poly. Without them, I would not have attended Cal Poly and would not have enjoyed the opportunities that came from my time at Cal Poly. I am especially indebted to Dr. Hartung for his infectious enthusiasm for for Cal Poly Engineering and its students. His unwaivering dedication to the other LOOP scholars and I motivated us to excel inside and outside the classroom.

Dr. Liu advised my Capstone project and introduced me to the exciting world of agricultural robotics. I thoroughly enjoyed our time out in the fields, seeing what robots can offer the argicultural industry. I am grateful for his support and guidance throughout our research projects.

This thesis would not have been possible without my advisor Dr. Seng. I began this project while he was on sabbatical teaching on the other side of the country, but he nonetheless made time to discuss my progress each week. His passion for robotics kept me motivated; I am thankful for his advice and suggestions throughout the project.

My parents were an endless source of encouragement and support. I am immensely grateful for their guidance and patience.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

Robots have become pervasive in a variety of industries and applications. They are able to complete military operations, clean homes, fight forest fires, and transport machinery. Previously, robots were confined to predictable situations like assembly lines where the robots would perform rote tasks. However, now robots navigate complex and dynamic worlds and must adjust their behavior in real-time to changing conditions. Robots must learn from other humans and robots in order to perform well in these situations. Additionally, robots must accurately perceive their surroundings and rapidly process this information to make informed decisions and complete their tasks.

Mobile robots are robots that can move. Roomba vacuum cleaners, self-driving vehicles, and bomb-disposal robots are all mobile robots. Mobile robots are especially challenging to develop and deploy because engineers must carefully specify how the robot should react to all of the conditions that it could encounter while moving. Three core tasks of all mobile robots are mapping, localization, and navigation. Mapping is the process of learning and describing the robot's environment. Localization is the process of determining where the robot currently is located. Navigation combines knowledge of the environment (map) and the robot's position within the environment (location) to plan routes from the current location of the robot to specific waypoints (destinations). Mobile robots should then be able to execute these routes by moving through the environment. The quality of a route may depend on how short it is, the amount of hazards on the route, and the impact that the route will have on other agents. Humans are adept at considering all of the aspects of a route and formulating the best option, so there is potential for robots to learn this behavior from humans,

but it comes with great challenges.

A common approach in modern robotics research is to train robots to behave like humans. If mobile robots can navigate and interact with humans in a pleasant, non-intrusive, and non-threatening way, then the robots will be better able to carry out mundane tasks that humans would otherwise be required to perform. Autonomous cars are an application of robotics where it is especially critical that the robot (car) act in ways that other humans would expect. Even though autonomous cars have the capability to dart in between cars with just inches of space between the bumpers, humans would be frightened with these close encounters and might react poorly, causing accidents. Instead, autonomous cars are designed to drive and respect the space of other cars the way human drivers do. The same principles can be applied to other mobile robots. The focus of this thesis will be on pedestrian environments. In some ways, robot navigation in pedestrian environments involves more uncertainty than autonomous driving due to the more predictable structure of the driving environment. In driving scenarios, the lanes, signs, and right-of-way rules all constrain the possible movements of a car and other cars around it, while pedestrian environments like shopping malls, schools, and airports have limited rules, and the behavior of agents within these environments is much less predictable. Yet, pedestrians still follow implicit rules and respond in predictable ways to outside events.

1.1 Pedestrian Trajectory Prediction

Before attempting to navigate a crowded area, humans predict the state of the environment several steps ahead. They anticipate potential dangers or collisions before they occur. They detect subtle behaviors of others to understand how people react and behave around one another. Once they have assessed the situation, humans use their predictions to formulate a desirable route. This thesis will investigate how

robots can achieve the forecasting ability of humans in pedestrian environments. This forecasting is an essential component for any robot that must cooperatively interact with people.

While humans learn many tasks and behaviors through experiential learning (trial-and-error), moving in crowded environments is learned through watching others' behavior. It would be infeasible and ethically dubious to train robots to move in dynamic pedestrian environments through trial-and-error since the learning process would certainly include many collisions. Rather, the preferred approach is to build a system that can forecast the positions of pedestrians in the future. This prediction system can then be utilized to plan appropriate routes and avoid collisions.

Humans make decisions based on complex rules and intuitions about their environment that they have learned over many years. It is infeasible to fully enumerate and encode the rules that govern human behaviors into an algorithm for predicting human movements. The influences of human movements are varied, and attempts to codify them would likely omit important edge cases (uncommon situations). One way to overcome this challenge is to build a system that can learn these rules through examples. Machine learning is the study of algorithms that learn to make predictions using data. In the case of trajectory prediction, this data could be videos of people walking in public areas.

Since humans are so adept at learning to predict pedestrian movements, a logical machine learning algorithm for estimating trajectories is a system that mimics human learning. While the neuroscience connection is somewhat tenuous, neural networks are a machine learning algorithm that has a basis in the neural pathways of animal brains. Neural networks are a widely successful tool for many domains where systems are taught to recognize and perform tasks at (or above) human-level performance. Recently, neural networks were applied to predicting human trajectories in dynamic

environments. This thesis will extend and improve those results in order to develop a solution for estimating the movements of pedestrians.

1.2 Navigation in Dynamic Environments

After building a model for predicting the trajectories of pedestrians, the next step is to construct a navigation algorithm that can incorporate these predictions. Such a navigation system must be able to effectively plan routes that avoid collisions (or uncomfortably close interactions) between a robot and other robots or pedestrians. Effectively, the trajectory prediction system provides foresight into where people are going, and this foresight can be used to plan better routes. Of course, there are many other factors that go into planning besides the forecasts of pedestrian behavior. A planning algorithm should be able to take into account the kinematic constraints of the robot (what kinds of movements are possible), environmental features (terrain and objects), and the goals of the robot (e.g. minimizing path length or perturbations of other agents).

Planning, and more specifically path planning, is a rich area of research with many algorithms designed for specific use-cases. In this thesis, two of these common approaches are modified and applied to planning robot paths in dynamic environments using the predictions of pedestrian trajectories from the neural network models. The first approach extends A* - a search algorithm for finding shortest paths that uses heuristics to find optimal paths while minimizing the exploration of routes that are unlikely to be optimal. The second approach is a tree search that calculates the sequence of movements that bring the robot closest to its destination. In this work, these two methods are applied to the dynamic navigation task and are evaluated on real-world datasets.

1.3 Contributions

This thesis will focus on the use of neural networks for predicting trajectories and their application to planning in dynamic environments. It makes the following contributions:

- Applies Mixture Density Networks for trajectory prediction,
- Designs four novel neural network architectures for representing pedestrian interactions,
- Constructs two planning approaches for navigating in dynamic environments, and
- Builds a new library, called Argil, for Agent-Based-Modeling and crowd simulation

Mixture Density Networks allow for a neural network to more accurately represent the uncertainty of an agent's position than previous techniques. The previous state-of-the-art approach used a unimodal distribution for trajectory prediction, which is unable to capture the multiple likely paths that pedestrians can take. The novel neural network architectures for pedestrian trajectory prediction achieve comparable results to state-of-the-art models, but they have fewer hyper-parameters to tune than existing solutions. These neural network architectures, to the best of our knowledge, are the first models for trajectory prediction that make no assumptions about which nearby pedestrians are likely to influence the movements of another pedestrian. To demonstrate the practical application of the trajectory prediction models, two path planning algorithms are adapted to dynamic environments by incorporating predictions from the neural network model. Argil, the new Agent-Based-Modeling library, includes built-in support for crowd simulation and has a flexible and succinct API

for model design, visualization, and data output. Argil is the first Python library for pedestrian simulations; it allows developers to more rapidly iterate their simulations than existing alternatives.

This thesis document will first describe the relevant context for this project in Chapter 2. Then the other methods and techniques for trajectory prediction and navigation in dynamic environments will be discussed in Chapter 3. Next, in Chapter 4, the architecture and core attributes of the new neural network solutions will be presented along with the path planning techniques. In Chapter 5, the construction of the system will be explained. The experimentation and testing will be discussed in Chapter 6. Chapter 7 presents the Argil simulation library that was used for building and debugging the models that are presented in the thesis. The opportunities for continued research will be presented in Chapter 8, and the significance of this research will be assessed in Chapter 9.

Chapter 2

BACKGROUND

Autonomous robot navigation has a rich history of research with many advances occurring in the past decade. Work in machine learning has become increasingly relevant for robotic systems; this thesis relies heavily on the machine learning technique called neural networks. This chapter will begin with an explanation of autonomous robots and perception, followed by a survey of relevant neural network concepts. The last section will discuss navigation algorithms.

2.1 Autonomous Mobile Robots

Autonomous robots operate without explicit human control. Semi-autonomous robots receive intermittent instructions from humans but often function independently. There are number of key capabilities that mobile autonomous robots must posses in order to be successful. The first priority is that the robots are safe - they do not damage themselves, objects, people, or animals. Once the robots are deemed safe, they must be able to cooperate and work alongside humans in a predictable and human-friendly manner. If the robot is mobile then it must be able to determine its location, understand its environment, and navigate to its destinations. This section will describe the current state of autonomous robots and how they interact with humans.

2.1.1 Human-Robot Interaction

Human-Robot interaction is the study of how humans and robots exchange information and work together or adversarially. It is often considered a sub-discipline of human-computer interaction, although there are number of key distinctions. The

nature of robots makes the interaction between them and humans especially critical because robots have the capacity to directly inflict physical harm. Humans are also much less familiar with robots than they are with computers, so it is more difficult to predict how humans will react. Another complication with human-robot interaction is the delayed response; since robots typically respond with physical actuation, humans may have to wait longer to receive feedback instead of the immediate feedback that they receive from a computer screen.

A key aspect of Human-Robot Interaction that is important for this thesis is the communication between humans and robots. Since humans rarely direct robots, they are not well-versed in the capabilities and limitations of robots. Most robot systems must be operated by experts who program the robot explicitly by setting navigational waypoints or precise arm movements. These modes of interaction are acceptable for controlled environments, but their performance deteriorates in complex, dynamic settings. Researchers have attempted to build more robust systems for handling human-robot interaction; notably researchers from Carnegie Mellon developed the The Human Robot Interaction Operating System[13], which attempts to rigorously define how humans and robots should effectively communicate. There are a variety of ways that humans can communicate with robots - visually, orally, and cognitively[8] among others.

It is generally desirable for robots to communicate with humans in approximately the same way that humans communicate with each other. Human-to-human communication involves conscious choices (like speech) and subconscious actions (like body language). This paper focuses on the way actions convey information. When humans are walking in a crowded environment, they communicate with other pedestrians by signaling their intent through actions. Moving purposefully forwards indicates that the person wants to continue forwards, while pausing or slowing down indicates that the person is willing to yield or is preparing to stop or turn. If a person chooses to

Table 2.1: Common Sensors for Autonomous Mobile Robots

Sensor	Method
Lidar	laser beam
Sonar	sound waves
Cameras	visible light
Radar	radio waves

take a circuitous route, other pedestrians may conclude that the person was avoiding a dangerous or unpleasant situation. For robots to navigate and interact in these crowded environments, they must be able to understand and respond to these subtle cues and provide the same cues.

2.1.2 Sensing and Perception

In order for autonomous robots to safely operate and complete tasks; robots must be equipped with sensors for perceiving the world. Additionally, robots must have the capacity to analyze the sensor inputs and determine appropriate responses. Sensors and computing resources can be located on the robot platform itself or externally. The quality of the sensors and the computational resources are key limitations for robots. Table 2.1 shows the most common sensors found on autonomous mobile robots. It is critical to consider the available sensors because the information provided by sensors often differs dramatically from the information available to humans from their senses.

The designs produced in this thesis could be implemented on any robot platform that has the capacity to identify and track other agents. Cameras and Lidar would be especially well-suited for navigation in dynamic environments because they can provide frequent updates (more than 30 times per second), and they typically produce dense representations of the environment. The raw data from these sensors must be

processed to estimate the position of other other agents in the scene.

2.2 Neural Networks

Artificial Neural Networks are models for learning complex, often non-linear functions. They can be used to perform classification, regression, and clustering. The neural connections and structure of brains inspired the development of neural networks. The concept was developed in the 1940s, but neural networks lost popularity in the 1990s as other machine learning techniques proved more successful. With massive increases in computing power and vast amounts of data, neural networks achieved renewed popularity in the 2000s. In the late 2000s and 2010s, neural network solutions achieved state-of-the-art results in diverse fields including image classification[29][49], machine translation[57], and speech recognition[3].

Neural networks are composed of neurons; the neurons have weights that are adjusted through training to produce the desired outcomes based on the inputs. These neurons are connected to other neurons; each neuron will apply a function to the inputs that it receives from other neurons to produce a result. Neurons are grouped into layers based on which neurons they are connected to and what function they apply to inputs. There is a tremendous variety of neuron functions and ways of connecting neurons, but a few designs have proved to be the most useful. Generally the level of neurons is not meaningful in large, modern neural networks because the interconnections between neurons is complex. Rather, most neural networks are described by their layers. Layers are groupings of neurons that all perform equivalent operations.

The most common and simplest layer for neural networks is the fully-connected layer. Each neuron in a fully-connected layer receives the output from every neuron in the previous layer. The neurons will then apply a specific weight to the value of

each input and then sum the weighted values. Typically a sigmoid or relu (rectified linear activation) is then applied to the sum in order to add non-linearity to the network. The final value computed in each neuron is the input to neurons in the next layer. The weight that each neuron applies to each input is modified during training. A Feed-Forward Network is a neural network where there are no loops (the outputs only propagate forward through the network). A simple fully-connected layer can be written using matrix multiplication and vector addition as shown in Equation **??** where X is the inputs from the previous layer. X is a column vector with m values where m is the number of neurons in the previous layer. W is the weight matrix with dimensions l by m where l is the number of neurons in next layer. b is a bias column vector with l values. f is an activation function like a relu, sigmoid, or hyperbolic tangent. o is the values of the neurons in the next layer. The first layer in a neural network is generally the input features, and so there are no parameters in this first layer.

$$o = f(WX + b) \tag{2.1}$$

Neural networks are typically trained through a process called Backpropagation. For classification tasks (where the neural network outputs the class of the input), the network is provided with inputs and the correct class of the input. For regression tasks (the output is a continuous value), the network is also provided with input features and the actual value of the features. The network is evaluated with the each input, and the output of the neural network is compared to the correct output. Then the weights are updated based on the error of the network. Backpropagation calculates the errors of each neuron based on the error of the entire network. These errors are calculated by sequentially propagating errors backwards through the network in the reverse order of the forward prediction step. The process of Backpropogation uses the

derivative of the neuron functions to determine the errors of the neurons. The weights are updated based on the contribution to the error using an optimization algorithm; a basic optimizer is Stochastic Gradient Descent (SGD). However, recently more advanced variations of SGD have reached common usage. During training, the error of the network is evaluated on a subset of training data and then the optimization algorithm is applied to make small updates to the weights in order to minimize the loss function (a quantitative assessment of neural network performance). This process is continued for a specified amount of time or once the loss function ceases to decrease.

2.2.1 Recurrent Neural Networks

While Feed-Forward neural networks have no loops, recurrent neural networks (RNNs) allow the output of neurons to be fed back into the same neuron or previous neurons. This is an important property because it enables the neural network to perform predictions based on the current input and the previous state of the network. RNNs are essential in many domains that involve time series data like translation, natural language processing, and video prediction. The additional capabilities of RNNs come at the expense of more difficult training. Backpropogation can still be used, but it must be modified to handle the loops in the network. The common extension to Backpropogation is called Backpropogation-Through-Time (BTT), which involves unrolling the recurrent layers (duplicating the neurons) to create a feed-forward network for a certain number of steps. The gradients can then be computed on this network, and the errors are propagated to the unrolled neurons. The weights are updated to adjust the parameters of the original recurrent neurons. Although BTT is an elegant solution to training RNNs, it often requires significant memory when the network must be unrolled for many steps.

Simple recurrent neural network neurons just feed their output back as input along

with the new input data. The downside of this approach is that it becomes difficult for a neuron to retain information for many steps. One solution to this is the Long Short-Term-Memory (LSTM) cell [23]. The basic version of an LSTM cell is shown in Figure 2.1. There are three gates in an LSTM (forget, input, and output). The forget gate (in the dashed box on the left) concatenates the previous cell state with the input and then multiplies the concatenation by a weight matrix and adds a bias vector; the result is fed into a sigmoid activation function. The output of this process is multiplied by the previous hidden state. All locations where the forget gate outputs a low value are lowered in the hidden state - causing the network to "forget" the values. The input gate has two parts. The first part applies the sigmoid activation to the concatenation of the input and previous cell state after multiplying the concatenated inputs by a weight matrix and adding a bias vector. The second part of the input gate, applies the hyperbolic tangent function same concatenation after multiplying the concatenated inputs by a different weight matrix and adding a different bias vector. The two parts of the input gate are multiplied together (element-wise) and added to the hidden state - essentially incorporating new information in the cell. The output gate also has two compoents. The first component is the hyperbolic tangent applied to the updated hidden state. The next component is sigmoid function applied to the the concatenation of the input and previous hidden state after multiplication by another weight matrix and the addition of a final bias vector. These two components are multiplied together element-wise to produce the next hidden state and output of the cell.

In Figure 2.1, the pink triangles denote the sigmoid activation function, while the red triangles are the hyperbolic tangent activation function. The purple hexagons are multiplications of the input by a weight vector and the addition of a bias vector (similar to a fully-connected layer). The M_i symbol denotes the parameters used in the weight matrix and bias vector. The green X denotes element-wise multiplication,

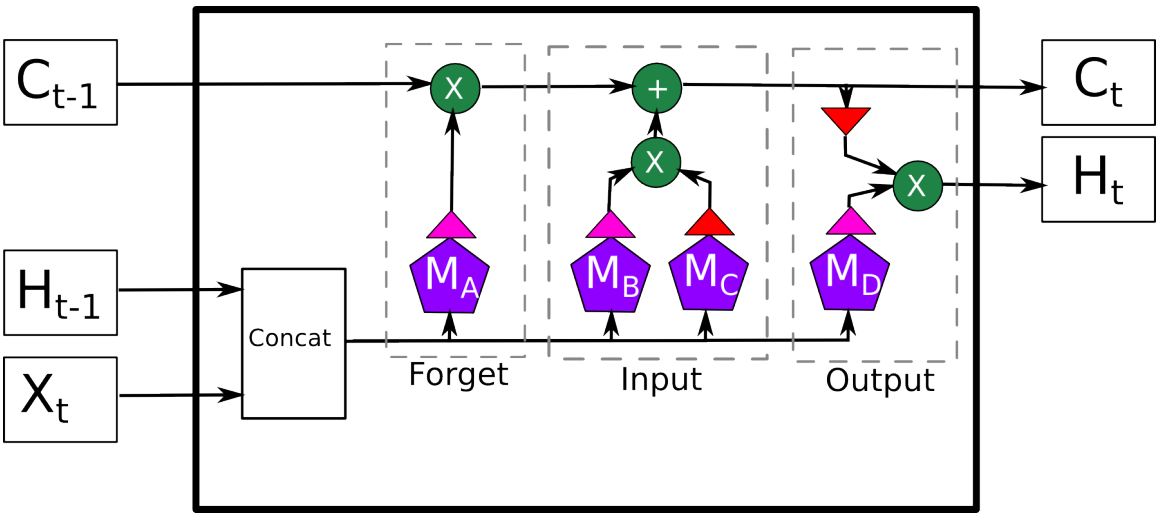


Figure 2.1: LSTM Neuron

and the green $+$ is the addition of the vectors. The same structure is expressed in Equation **??**. κ represents the matrix multiplication with a weight matrix, addition of a bias vector, and the application of the sigmoid activation functions. ρ similarly represents the multiplication with a weight matrix and addition of a bias vector, but the activation function is the hyperbolic tangent.

$$
\begin{aligned}
z_t &= concat(h_{t-1}, x_t) \\
a_t &= \kappa(z_t; M_a) \\
b_t &= \kappa(z_t; M_b) \\
c_t &= \rho(z_t; M_c) \\
d_t &= \kappa(z_t; M_d) \\
c_t &= (c_{t-1} \cdot a_t) + (b_t \cdot c_t) \\
h_t &= \rho(c_t) \cdot d_t
\end{aligned}
\tag{2.2}
$$

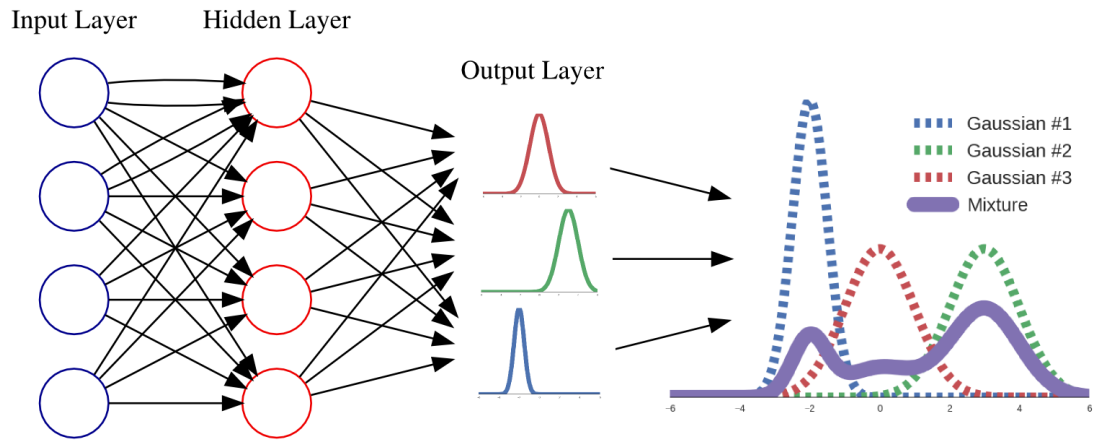


Figure 2.2: Simple Mixture Density Network

2.2.2 Mixture Density Networks

Many of the most successful applications of neural networks are for predicting categorical values like the object in an image or the word that was uttered. However, neural networks have also been used for regression problems where the output is a continuous value. For example, neural networks have been used to predict stock prices [26][31]. There are some situations that arise in regression problems where the output is best approximated with a multi-model distribution. Using stocks as an example, it would be beneficial for a neural network to express the hypothesis that the stock price will go down by 1% with 80% certainty or go up by 2% with 20% confidence, but it is unlikely for it to stay the same. In order for a neural network to produce such a prediction, the model would have to have several output neurons along with weights denoting the certainty of each prediction.

Mixture Density Networks are a general framework for making multimodal predictions. They were introduced by Bishop in 1994 with a demonstration of their application to inverse robot kinematics[6]. A Mixture Density Network outputs the parameters of a mixture of probability distributions along with weights for combining

the component distributions. Typically, the composite distributions are Gaussians, but it is possible to use other distributions. A simple example of a Mixture Density Network is shown in Figure 2.2. The network accepts inputs and transforms the inputs through several layers into the parameters of a mixture distribution.

Mixture Density Networks are closely related to Mixture Models, which are ubiquitous in Bayesian statistics. In probabilistic modeling, Mixture Models are used to infer the latent substructure of a larger group. Both Mixture Models and Mixture Density Networks use a composition of distributions to represent a concept. Mixture Models are used to represent the properties of a population by learning the component distributions directly from the data (typically using Expectation Maximization or Markov Chain Monte Carlo techniques). Mixture Density Networks represent a prediction by learning the component distributions from a neural network applied to inputs.

Mixture Density Networks have be incorporated into models for a variety of applications. Speech and acoustics are good candidates for Mixture Density Networks because the frequencies can be expressed as a mixtures of Gaussians [44][59]. RNADE (real-valued neural autoregressive density-estimator) extends Mixture Density Networks by sharing distribution parameters and making subsequent distributions conditional on previous ones [54]. RNADE is not considered in this work, but it could be a candidate for future work. Alex Graves developed a Mixture Density Network to generate handwriting samples by representing the location of the pen with a mixture of Bivariate Gaussians [17].

2.2.3 Autoencoders

Autoencoders are a type of neural network used for unsupervised learning. The basic autoencoder encodes its inputs into a latent representation and then decodes the la-

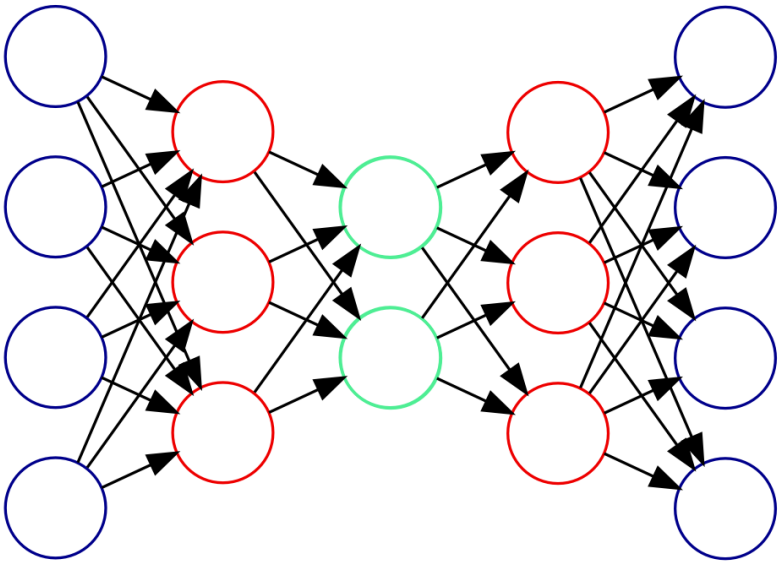


Figure 2.3: Autoencoder

tent representation to approximate the inputs. The network is trained by minimizing the difference between the output of the network and the input. Both the encoding and decoding layers can be deep neural networks. Figure 2.3 illustrates a simple autoencoder where the blue nodes on the left are the original input and the blue nodes on the right are the reconstructed input. The red nodes are hidden layers, and the green nodes are the latent representation.

There are two main considerations when designing autoencoders - the encoding/decoding structure and the latent representation. Typically, the encoding and decoding neural networks are symmetric. The latent representation must be chosen carefully. If the latent representation has a higher dimension than the input dimensions then the autoencoder risks simply memorizing the input data and returning it without learning any interesting structure. To avoid this, researchers have developed several strategies. The most basic solution is to use a small latent layer (like the one shown in Figure 2.3) that has many fewer units than the input dimensions. An alternative is to apply regularization to the activations of the neurons in the latent

representation.

One form of regularization is the sparsity constraint. The sparsity constraint adds a penalty for all of the neurons in the latent layer that are activated (have a high output value). The result of the sparsity constraint is that only a few neurons will be active in the latent layer, which forces the autoencoder to learn the underlying structure of the input data that differentiates each input from the rest. Care must be taken when training an autoencoder with a sparsity constraint to ensure that the sparsity penalty is not too high that no neurons are activated nor too low so that nearly all neurons activate.

Another more complex form of regularization of the latent representation is the Variational Autoencoder (VAE). A VAE regularizes the latent representation by forcing activations of the neurons in the latent layer to not significantly deviate from a Gaussian distribution. VAEs tend to produce dense representations where each neuron in the latent layer is activated, but the values are constrained to a small range. In a VAE, the encoding layer outputs a list of means and standard deviations (the number of means/standard deviations is the latent layer size). Then a sample is taken from each Guassian distribution defined by the means and standard deviations. These samples become the latent representation of the input. The samples are then passed to the decoding network to reconstruct the input. The loss for a VAE is the reconstruction cost (the same as all other autoencoders) and the Kullbach-Leibler Divergence between the Gaussians defined by the encoder and a predefined Gaussian (typically with a mean of 0 and a standard deviation of 1). The Kullbach-Leibler Divergence (KL-D) is a continuous value that describes how different two distributions are from each other. The formula for the continuous version of KL-D is shown below:

$$D_{KL}(P|Q) = \int_{-\infty}^{-\infty} p(x) \cdot log(\frac{p(x)}{q(x)})$$

It is important to note that KL-D is not symmetric because $D_{KL}(P|Q)$ does not necessarily equal $D_{KL}(Q|P)$.

2.3 Navigation

Navigation is the process of determining the steps necessary to get from one place to another. Typically, the goal of navigation is to minimize the cost of movement to a destination. The cost of a route to the destination can include the length of time required, the amount of energy consumed, or the probability of a collision. Prior to performing navigation, the areas of the environment that are safe to traverse must be determined. Additionally, the criteria for selecting a route must be defined. These criteria can vary, but typically in the context of robotics, the shortest route that avoids damage to the robot or the environment is chosen. Navigation can occur at many levels of granularity. For example, a traveler performs navigation by planning to drive to the train station and take a train to the beach. Then the traveler will use maps to plan the optimal sequence of roads to take in order to reach the train station. While driving, the traveler will actively navigate around obstructions in the road, possibly taking detours in order to avoid accidents. This example shows how navigation can be performed at a high-level (drive then ride the train) or at a lower-level (driving wide around an obstruction). Humans are able to seamlessly switch between levels of planning when necessary, but teaching robots to effectively navigate is a non-trivial endeavor. Robots must learn or be programmed to predict human actions, emulate human activity, and respond appropriately to unforeseen events like natural disasters or malicious behavior.

An excellent example of constructing a robot system for navigation is the Navigation Stack in the Robot Operation System (ROS)[43]. ROS is a library and framework that is commonly used for robotics. ROS provides a modular way of defining com-

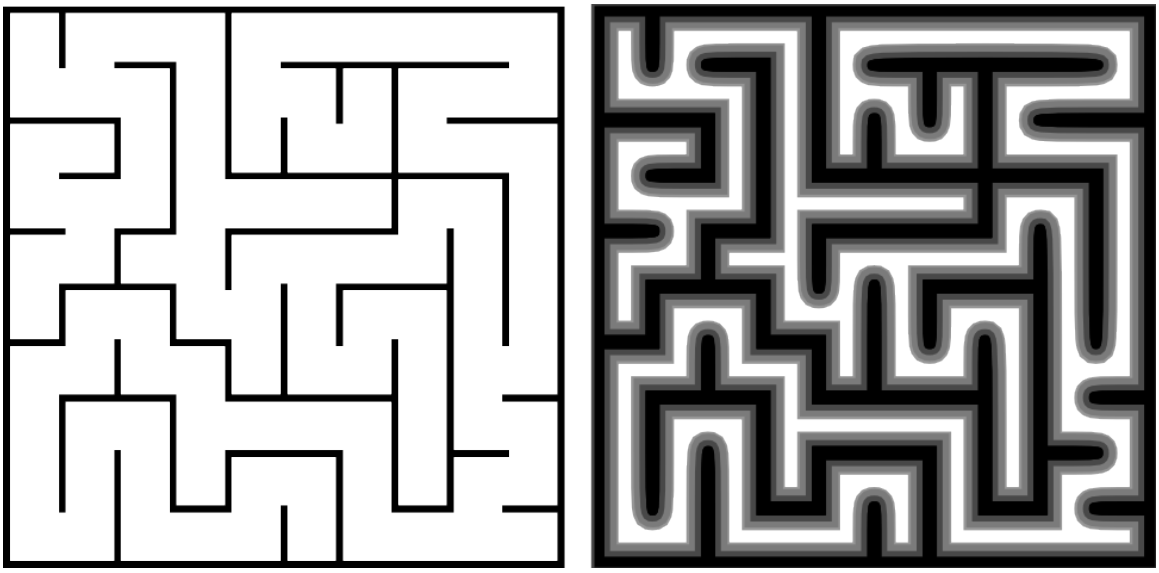


Figure 2.4: Costmap

ponents that accomplish specific tasks for the robot. Researchers, corporations, and hobbyists share their components (called Nodes) with the open source community. The purpose of ROS is to have a unified way of orchestrating all of the various behaviors and functionality of robots. The ROS Navigation Stack is a well-documented and thoughtfully-designed system for route planning that has been implemented for many robot platforms. The following paragraphs consider the ROS Navigation Stack as a case study in how navigation systems can be built and deployed on autonomous robot platforms. There are many other solutions for navigation systems, but they all share many of the same design choices, and it is therefore necessary to only study one of them.

Before a robot can navigate and plan paths, the robot must have a map of the environment and know its location in the environment. In ROS, a map of the environment can be produced by the robot while it explores the environment, or an existing map can be used. The basic map defines what regions are safe and which are not. In ROS, these maps are costmaps. Figure 2.4 is an example of a costmap

constructed from a maze layout. The light regions of the map are navigable while the darker regions are considered dangerous or impossible to move through. Notice that the original map on the left shows the hard walls of the maze, while the map on the right has fuzzy boundaries around the walls. The map on the right is generally preferable because it demonstrates that the robot should not only avoid running into a wall but also avoid close proximity to a wall since the robot may slide or get pushed resulting in a collision. These 2D costmaps can be used for planning.

In ROS, there are global planners and local planners. Global planners use the costmaps and plan a route to the destination that minimizes length of the path while also avoiding dangerous (darker) areas on the map. The basic global planner uses the A* search algorithm. A* is an efficient search method that uses heuristics to focus planning on regions of the map that are more likely to yield optimal routes to the destination. The local planners use the route defined by the global planner and outputs instructions for the robot to ensure that the robot uses its actuators to follow the global plan as closely as possible. A key task for building successful robots is to write a system that correctly balances long-range and short-range planning. In the simplest implementation, neither the local planners nor the global planners consider the movement of other agents. Of course, sensors can detect pedestrians or moving vehicles, but the robot treats these the same as static obstacles. This is a major limitation because it means that robots using these navigation tools cannot effectively move in dynamic environments without colliding or being overly cautious and failing to make forward progress. A navigation system that will succeed in dynamic environments must incorporate knowledge of other agent's trajectories at both the local and global levels.

While there is tremendous variety in navigation algorithms, there is even greater diversity in map building techniques. Navigation systems will always be limited by the accuracy of the available maps. The primary method for building maps is for a

robot to explore an unknown area and construct a map piece-by-piece as it moves around. This requires that the robot estimate its own location and the location of objects around it at the same time. SLAM (Simultaneous Localization and Mapping) is the term that describes this task and the associated solutions for it. One limitation of most SLAM algorithms is that they are not robust to moving agents in an environment. For example, if a person walks in front of a robot while it is exploring, it may denote the person as an obstacle in the map even though the person is not a permanent fixture of the environment. While maps should be free of transitory objects and agents, the position and velocity of mobile agents is an essential component of navigation. The task of fusing static maps with information about mobile agents is still an active area of research. This thesis will focus exclusively on the effect of mobile agents on trajectory prediction and applications for navigation, but it is important to consider that real-world navigation must incorporate static and dynamic features.

Chapter 3

RELATED WORK

3.1 Navigation and Trajectory Prediction

Navigation and obstacle-avoidance in dynamic environments is a challenging task for robots. Recently researchers have developed many methods for robot navigation that enable robots to actively avoid collisions and minimize the discomfort of humans while still allowing the robot to reach its destination. Complimentary to navigating in dynamic environments is predicting how other agents will move. Often the tasks of trajectory prediction and navigation are approached together, so this section discusses methods for navigation and trajectory prediction.

3.1.1 Goals of Navigation

The most obvious objective for any navigation technique is to plan a route from one location to another. A navigation algorithm cannot be considered successful if it does not produce a viable path to the destination. However, there are many other considerations when the navigation occurs in a crowded environment. Kruse et al. produced a thorough survey of the criteria for successful robot navigation in the presence of humans [30]. Kruse et al. identified three major categories of evaluation metrics that can be used to determine how well a robot navigates - comfort, naturalness, and sociability. A robot that does not invoke fear, stress, or unease in humans would be comfortable for humans to be around. Robots that navigate, move, and interact like humans are considered natural, while robots that respect cultural/social rules like right-of-way would have a high degree of sociability. These categories are abstract and not amenable to specifying in an algorithm; however, other

researchers have considered quantitative metrics that can measure the success of a robot at navigating a human environment. Many researchers have devised penalties for robots that get too close to a person.

3.1.2 Reinforcement Learning

Reinforcement Learning (RL) is the process of learning from repeated trials. The problem of Reinforcement Learning is typically described a Markov Decision Process (although there are several other variations). A Markov Decision Process (MDP) is defined as a state space, an action space, a probability of going from one state to another conditioned on the action taken, and the rewards that are received for transitioning from each state to another state. For robot navigation, the state space is typically all physical locations that a robot can occupy; the action space is all physical actuations that the robot can perform. The probability of transitioning from one state to another is dependent on the features of the environment and the other agents. Rewards are problem-specific, but typically robots would be rewarded for getting close to their destination and not colliding or interfering with other agents. Inverse Reinforcement Learning (IRL) is the task of learning the reward function for a Markov Decision Process. Several researchers have applied IRL to cooperative navigation. The following paragraphs describe three illustrative examples of reinforcement learning in robot navigation.

Chen et al. used Deep Reinforcement Learning for producing short collision-free paths for multiple agents[9]. Deep Reinforcement Learning is a way of solving RL problems using neural networks to estimate the value of going from one state to another or the value of an action and state pair. Chen et al. only considered simulations, but their results showed that neural networks can learn to incorporate cooperative collision avoidance and constraints to produce efficient and safe paths.

Ziebart et al. used IRL and Bayesian methods to output the log-likelihood of a pedestrian's next locations [60]. They conducted the experiment using real-world data collected from observing a kitchen. The environment was discretized into a grid, and the cost of visiting each cell was learned conditioned on the distribution over possible destinations. They then planned the path of a robot based on the current map. After planning a step, they simulated the trajectories of other pedestrians and updated the cost of cells based on the likelihood that pedestrians would be hindered by a robot's presence in that cell. The algorithm lets the developer define the trade-off between efficiency of the robot reaching its destination and the amount of hindrance to pedestrians. The main limitation of the approach is that it does not model the responses of pedestrians to the robot (see the Frozen Robot Problem in the next section). In a realistic scenario, humans move out of each other's way and are thus likely to also yield to a robot; however, the approach taken by Ziebart et al. does not model this possibility.

Kretzschmar et al. designed a probabilistic framework for robot navigation where the importances of various features on the paths are learned from examples of human movements [28]. Some of the features considered are proximity to obstacles, proximity to other pedestrians, and changes in velocity/acceleration. The authors use Mixture distributions to represent choices between several options (such as passing an obstacle on the right or the left). The work is experimentally validated using a contrived environment where humans were observed for four hours. A strong advantage of the engineered features is that it is possible to interpret the impact of other agents and obstacles in a straightforward, probabilistic way. The downside of the engineered features is that it may not generalize to complex environments.

3.1.3 Probabilistic Models

3.1.3.1 Interacting Gaussian Processes

Trauntman and Krause developed the Interacting Gaussian Process (IGP) model to solve the *frozen robot problem*[52]. When a robot is given a destination, but there are people in the way, the robot may be unable to make forward progress and therefore appear "frozen". Humans do not experience this problem because they engage in cooperative navigation where humans make room for other humans when they notice that someone intends to move through the crowd.

The IGP model represents the trajectories of agents (humans and robots) as a Gaussian Process. A Gaussian Process is a distribution of functions, where the distribution of values at each step is a Gaussian random variable. In the context of trajectory prediction, each step is a random variable that defines where the agent is likely to be; the Gaussian Process is the distribution over these individual random variables and thus represents a trajectory. Trauntman and Krause developed the theory for IGP using one-dimensional trajectories, but additional Gaussian Processes could be used to model the three dimensions of the Euclidean coordinate space. A Gaussian process is defined by a mean and a covariance function. The distribution for any time step is a function of all previous time steps. A standard Gaussian Process has no way of directly incorporating information from other Gaussian Processes (trajectories), so Trautman and Krause introduced a potential function. They multiply the Gaussian Process' probability density for each timestep of an agent by a potential that shifts the distribution away from the other agents. The resulting distribution can be multimodal, unlike the original Gaussian Process random variables. Trauntman and Krause parameterized the potential based on the minimum distance that is seen between pedestrians.

When predicting the next position of a pedestrian, the Maximum A-Posteriori (MAP) estimate of the position is chosen. The MAP estimate is the mode of the posterior distribution, where the posterior distribution is the result of multiplying the distribution of the step of the Gaussian Process with the potential function. The mode is the single most likely position. Trauntman and Krause advocate the use of importance sampling, which approximates the MAP estimate through multiple samples from the posterior distribution.

IGP is a promising technique because it can represent multimodal distributions and has relatively few parameters. Trauntman and Krause report that reasonable estimates can be computed in less than .1 seconds, so the approach seems to be fast enough for real-world applications. The potential function allows the robot to infer how people will move in order to accommodate it, thus resolving the frozen robot problem. IGP was also validated on a robot in a crowded cafeteria [53]. The major limitation of IGP is that it assumes that robots can move just like humans, and the custom potential function may not be applicable to heterogeneous agents (cars, bikes, etc.) or be able to incorporate long-range influences.

3.1.3.2 Obstacle Maps

Vemula et al. constructed a variation of the IGP model that represented the neighbors of an agent in an occupancy grid [56]. The occupancy grid encodes the locations of other agents relative to the agent whose trajectory is being predicted. The squared exponential automatic relevance determination (SE-ARD) kernel is used to learn the affect of neighbors on the trajectory. Unlike IGP, there is no need to provide the true final destination of each agent. The major advantage of this work is that the potential function is learned rather than specified with user-defined parameters. One limitation of the model is that there is no obvious way for the model to incorporate

the velocities or previous states of the neighbors.

3.1.4 Social Forces

One of the earliest and most famous methods for modeling pedestrian trajectories in crowded spaces is Social Forces - introduced and popularized by Helbing et al. [22]. The Social Forces model represents pedestrians as particles that experience and exert forces. Repulsive forces push pedestrians from obstacles and other pedestrians. Attractive forces can pull pedestrians towards their destination and towards their group. Helbing introduced equations that define these forces along with empirically validated parameters to produce realistic simulations. At each timestep of a simulation, the forces applied to each pedestrian are combined, and the velocity (direction and magnitude) of the agent is then computed based on the forces.

The main limitation of Social Forces models is the large number of parameters that must be specified. The default parameters can yield reasonable trajectories, but the parameters must be carefully fine-tuned to fit a particular scenario. Different cultures, environments, and events all affect the desired velocity, distance to others, and preferred side of a pathway among many other factors. Another significant limitation is that generally all pedestrians are assumed to share the same parameters. Therefore, each pedestrian responds the same way to other pedestrians and obstacles. Of course, it is possible to choose parameters for each pedestrian, but this makes the task of defining the model much more difficult.

Despite its relative simplicity and the challenge of choosing good parameters, the Social Forces model has been used in several practical applications. Helbing et al. described the use of Social Forces to model pedestrians during evacuations and used the resulting models to critique the design of buildings and walkways [21]. Mehran et al. used the Social Force model to detect abnormal behavior (such as people running

from a dangerous situation) [36]. Social Force models and other physics-inspired models have been used to model traffic patterns and produce recommendations for the construction of roads and paths that can accommodate large crowds [20]. The Social Forces model has also been used for robot navigation by Mehta et al. [37] where the robot learned to switch between several policies (follow, go-solo, and stop) partially influenced by the Social Forces exerted by other agents.

3.1.4.1 Learning Social Etiquette

Robicquet et al. recorded a large collection of videos of people walking, riding, and biking using drones over crowded areas of Stanford University's campus[45]. The dataset is unique in its scale and the inclusion of multiple agent types (cars, bikes, pedestrians, etc.). The authors used the data to evaluate a novel technique for predicting trajectories. Their work is closely related to the Social Forces model; however, several of the parameters are learned rather than specified. Specifically, they learn an energy potential that represents how sensitive each agent is to other agents. The potential is defined as the multiplication of Gaussians where the standard deviations of the Gaussians roughly correspond to how much space each agent attempts to maintain between themselves and others. The parameters for this energy potential are learned for each agent trajectory, and the resulting parameter values are clustered to identify groups of agents who move in similar navigation styles. The parameters are learned based on an equation that models the displacement of an agent from a linear trajectory. The most significant result of the paper is the idea that different agents will have different navigation styles, which means that the Social Forces model (even with well-chosen parameters) will likely not accurately model heterogeneous scenarios where individuals have distinct speeds and methods of movement.

3.1.5 Neural Networks

3.1.5.1 Social-LSTM

Alahi et al. proposed a neural network model to predict pedestrian trajectories[2]. They modeled the trajectory using LSTMs. This was the first major application of neural networks to pedestrian trajectory prediction. The main insight of the paper was the creation of a grid of the hidden states of the neighboring agents. The grid is centered at the agent whose next position is going to be predicted. Each cell contains a sum of the latent representations (hidden states) of all agents who are inside that cell relative to the position of the current agent. By incorporating the hidden states of the neighbors, the network predicts the next position of each agent based on that agent and all of the neighboring agents. The latent representations of the neighbors are the output of the LSTM for that agent at the previous timestep. The authors chose hyper-parameters that defined the grid size and the distance of relevant neighbors using simulations. Their final model used an 8 by 8 grid that was 32 pixels wide. They trained the network to output a Bivariate Gaussian that minimized the negative log likelihood of the actual next position of the agent. During testing, they sampled from the Bivariate Gaussian and inputted the samples back into the network to compute a complete path. The network was provided 8 timesteps of positions and then inference was performed for 12 timesteps in their validation tests. The model was evaluated on the ETH[41] and UCY[32] pedestrian datasets.

The architectures presented in this thesis are extensions and modifications of the core model described in the Social-LSTM paper. Here, we will describe the specifics of the Social-LSTM model. The state of each agent at each timestep is represented by the output of an LSTM. This Agent-LSTM encodes the coordinates and influence of neighbors at each timestep. To illustrate the architecture of Social-LSTM (and later the novel designs presented in this thesis), a sample scenario will be considered. The

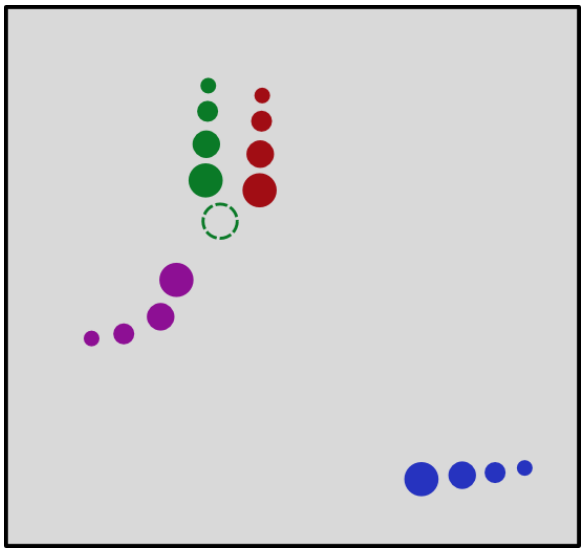


Figure 3.1: Sample Scenario

sample scenario is shown in Figure 3.1. The example objective that will be considered is the task of predicting the next position of the green agent (whose agent id is 4).

The overall sequence of operations for predicting the location of the green agent is shown in Figure 3.2. Each green box is a vector or tensor that is calculated for the green agent. Each box with a dashed gray border is an operation on vectors or matrices. The s_t^4 vector is the representation of the influence of neighboring agents on the fourth agent (green agent) at timestep t. The e_t^4 vector is the representation of the current coordinates of the fourth agent at timestep t. The coordinate and neighbor representations are used as inputs to the Agent-LSTM ($LSTM_a$) along with the previous hidden state (h_{t-1}^4) and cell state (c_{t-1}^4), where the hidden and cell states are specific to the green agent. The previous states will represent the previous trajectory of the green agent. The Agent-LSTM then outputs the next hidden and cell states along with an output vector. A fully-connected layer is applied to the output vector to output parameters of a Bivariate Gaussian that specifies the likely position of the green agent in the next timestep. The fully-connected layer is represented by Φ with weight matrix W_θ and bias vector b_θ. There is no activation function

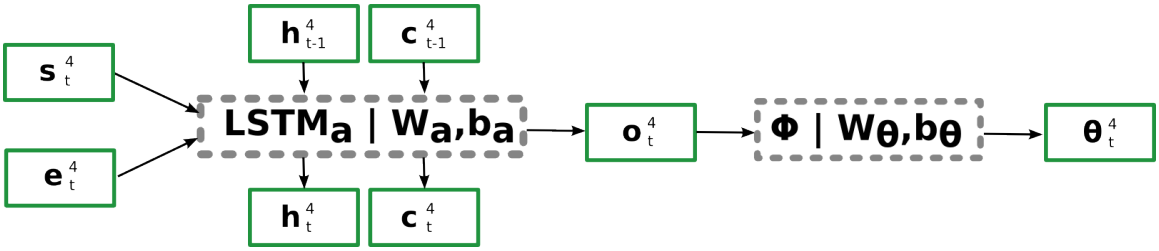


Figure 3.2: Core Social-LSTM Architecture

applied in the Φ operation. The details of how the output parameters are used to define a probability distribution are discussed in *Chapter 4* Design. For Social-LSTM, the probability distribution is specified by just 5 parameters. Equation **??** is the mathematical equivalent of Figure 3.2 for predicting the position of agent i at timestep t.

$$\begin{aligned} o_t^i, h_t^i, c_t^i &= LSTM_a(s_t^i, e_t^i; h_{t-1}^i, c_{t-1}^i; W_a, b_a) \\ \Theta_t^4 &= \Phi(o_t^i; W_\Theta, b_\Theta) \end{aligned} \tag{3.1}$$

Figure 3.3 and Equation **??** show how the coordinate (e_t^i) and neighbor (s_t^i) embedding vectors are computed. The coordinate vector e_t^i is just a fully connected embedding of the raw coordinate values with the weight matrix W_e and bias vector b_e. Ψ denotes the embedding using a relu activation. A relu activation is a function that is 0 when the input value is less than 0 and is equal to the input value when the input is greater than 0. The construction of the neighbor vector s_t^i is much more involved. First, a tensor P_t^i is created as a grid with dimensions (k, k, l) where k is the number of cells in the width and height and l is the size of the hidden state from the Agent-LSTM. The indicator function $\mathbf{1}_{mn}$ is 1 when the input values are within the m$^{\text{th}}$ row and n$^{\text{th}}$ column of the grid. The inputs to the indicator function are the differences between each neighboring agent's coordinates and the current agent's coordinates. The value of every grid cell is the sum of the latent representations of all

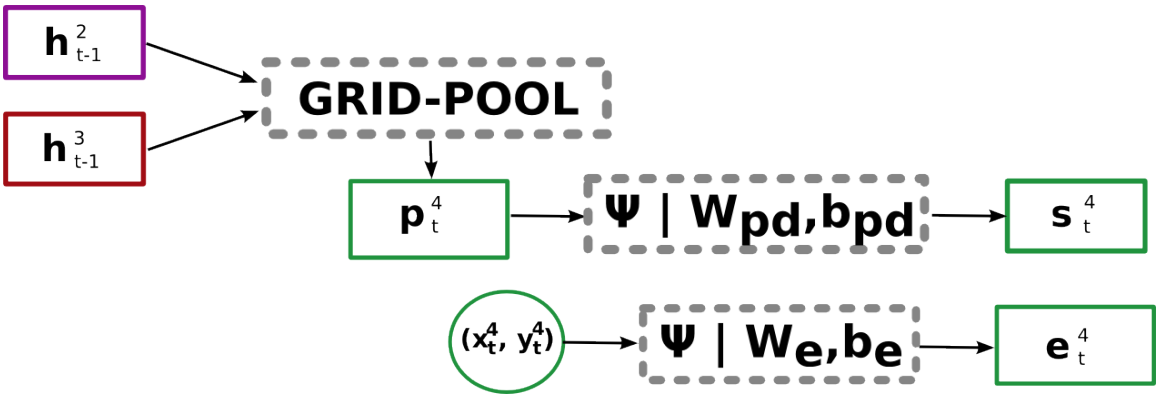


Figure 3.3: Neighbor and Coordinate Representations for Social-LSTM

neighbors that exist in that cell (when the indicator function is 1). The grid is then embedded into a tensor by applying a fully-connected layer Ψ with weight matrix W_{ps} and bias vector b_{ps}.

Figure 3.3 does not incorporate the hidden state of the blue agent in the grid because the blue agent's current position is outside of the grid.

$$
\begin{aligned}
e_t^i &= \Psi(x_t^i, y_t^i; W_e, b_e) \\
P_t^i(m, n, :) &= \sum_{j \in N^i} \mathbf{1}_{mn}[x_t^j - x_t^i, y_t^j - y_t^i] h_{t-1}^j \\
s_t^i &= \Psi(P_t^i; W_{ps}, b_{ps})
\end{aligned}
\tag{3.2}
$$

Like most neural networks, there are a number of hyper-parameters that must be set in order to fully specify the model. For Social-LSTM, the important hyper-parameters are the size of the grid (how much of the environment is included) and the granularity of the grid (number of cells). Selecting a grid size that is too small will mean that relevant neighbors that would affect the movement of the agent will not be considered. On the other hand, a large grid size may sacrifice the precision of the spatial position of each neighbor since the grid cells will each cover a large portion of the environment that may include many agents. It may seem desirable to have a

large grid size (cover much of the environment) and a large number of cells to achieve high coverage and high spatial granularity, but there are several disadvantages to this. First, the matrix P and the weight matrix W_{pd} will become enormous and will occupy considerable memory during training and inference. Additionally, the network will have to learn more parameters, which generally requires more training data in order to encounter many training samples that include the agents in all of the cells.

3.1.5.2 Soft + Hardwired Attention

Fernando et al.[12] designed a novel sequence-to-sequence neural network architecture based on Social-LSTM. They postulate that a weakness of Social-LSTM is that the next position of the current agent is predicted using only a function of the hidden representation of the current agent and the social tensor of the hidden states of the neighbors at the current timestep. In their research, Fernando et al. use an attention mechanism to incorporate all previous hidden states of both the current agent and the neighboring agents when making predictions. This has the benefit of allowing the neural network to model how the agent reacts to different scenarios and including that information in subsequent predictions. The attention mechanism uses learned parameters to weight the previous states of the current agent and hardwired weights based on Euclidean distance to weight the neighbors' previous hidden states. Additionally, the neural network is trained as a sequence-to-sequence model, so the model never represents the next position with a probability distribution like Social-LSTM. Rather, the first part of the network encodes a sequence of 20 time steps, which is then fed into the second part of the network that decodes the next 20 time steps.

Fernando et al. evaluated their architecture against Social-LSTM on the crowded Grand Central Station Dataset[58] and Edinburgh Informatics Forum Dataset[34].

Prior to fitting their model, Fernando et al. first clustered the trajectories based on destination and then trained separate models for each cluster. Their attention-based model was superior to Social-LSTM in each evaluation category and dataset. Since their architecture involves several departures from Social-LSTM, it is not obvious whether the attention mechanism in particular was the most important modification. Even the reduced model that omits information about neighbors performs nearly as well as Social-LSTM.

3.2 Deep Representations

Relatively straightforward convolutional neural networks and recurrent neural networks have produced significant results in a variety of domains. However, deeper and more powerful neural network architectures have begun to show increased potential in recent years. This thesis relies on these modern designs; this section summarizes the relevant advances in neural network construction.

3.2.1 Attention in Neural Networks

Attention is the general concept of focusing on specific details. Humans have a well-developed capacity for attention. When someone is speaking, humans generally concentrate on the speaker rather than the plain wall behind the speaker. When driving, humans focus on the road in front of them and nearby cars rather than the color of the sky. Of course, humans can also attend to things that are not relevant. Sometimes, human drivers look closely at accidents, causing them to subconsciously slow down. During uninteresting speeches, humans may decide to focus on the people they are sitting next to or the chores that need to be completed later.

Without an ability to attend to specific concepts, humans would likely be overwhelmed with the amount of information that they can perceive through their senses.

In a similar way, it can be beneficial to endow neural networks with the ability to attend to certain aspects of the input rather than others. The key components of attention mechanisms in neural networks are the selection and representation of the information to attend to. Many attention mechanisms have been proposed, but the key distinctions for this thesis are hard versus soft attention and spatial versus temporal attention.

3.2.1.1 Soft and Hard Attention

Attention mechanisms have the potential to improve the accuracy and speed of neural networks since expensive processing can be applied only to the relevant parts of the inputs, and extraneous noise in the inputs can be ignored. However, training these powerful networks can be challenging. The design of the attention mechanism determines how the model can be trained. If the attention mechanism is fully-differentiable (called "soft") then the network can be trained using Backpropagation similar to most other neural networks. However, if the attention mechanism is not differentiable (called "hard"), then it is no longer possible to use simple Backpropagation. Therefore for "hard" attention mechanisms, Reinforcement Learning techniques are usually the method of choice for training.

An example of a "soft" attention mechanism is a weighted combination of all inputs. One part of the neural network will output a weight for each input, and the inputs times their weights is then fed into another part of the neural network for further processing. An example of a "hard" attention mechanism is the selection of a strict subset of the input. By only choosing some of the input to use, it is no longer possible to compute the gradients of the attention mechanism.

3.2.1.2 Spatial and Temporal Attention

A neural network can use attention mechanisms for attending on any type of data; however, there are two typical applications of attention - spatial and temporal. Some of the first attention mechanisms were applied in recurrent neural networks to attend to the state of the network at previous time steps. One of the best examples of this temporal attention is the seminal paper by Bahdanau et al. about Neural Machine Translation [4]. They used LSTMs to encode a sentence in one language and then more LSTMs to decode the sentence in a different language. The decoding LSTMs received input from the output of the encoding LSTM (as usual) but also from a weighted summation of the states of the encoding LSTM cells. The weights were computed using an alignment network (a type of attention). Even earlier, Graves used a similar methodology for handwriting generation[18]. Graves constructed a network that could attend to a windows of the input while generating handwritten characters.

Recently, researchers have considered spatial attention where a neural network is trained to attend to certain regions in a spatial domain (most often the attention focuses on a part of an image). An excellent example of spatial attention is the famous DRAW neural network created by Gregor et al. [19]. The DRAW model is a Variational Autoencoder that uses a recurrent neural network to read sections of an input image and then another set of recurrent neurons to reconstruct the input image by selecting regions of the output canvas and drawing to them. The DRAW network outputs images much like humans - by copying individual parts of the images at each timestep. The regions that DRAW attends to are based on the hidden state of the recurrent neural network from the previous timestep.

3.2.2 Hierarchical Recurrent Neural Networks

Humans are remarkably adept at understanding relationships in complex systems. For example, people can look at a building and recognize the materials used to build the structure as well as the physical interactions among the components that allow the building to remain upright. Deep convolutional neural networks have been used to learn the spatial relationships among the components of a face (e.g. noses are usually located between the eyes, which are above the mouth). However, there has been much less success in the task of learning complex temporal relationships. Recently, researchers have advanced the state-of-the-art in speech recognition, text understanding, and translation using recurrent neural networks. Combining convolutional neural networks and recurrent neural networks has demonstrated promising results on video understanding and prediction.

One promising direction for learning representations of complex spatial and temporal relationships is hierarchical recurrent neural networks. These models use multiple layers of recurrent cells to encode multiple dimensions. In a hierarchical recurrent neural network, the outputs of one set of recurrent neurons is used as the input sequence to another set of recurrent neurons. The first layer could encode the temporal patterns of discrete entities, and the second layer could encode the temporal representations across multiple entities. The following two paragraphs discuss some applications of hierarchical recurrent neural networks.

3.2.2.1 Spatial Temporal Learning

Ibrahim et al. used hierarchical LSTMs to learn representations of group activities [25]. In their paper, an LSTM was used to encode the activity of individuals by operating on the output of a convolutional neural network at each timestep. The activity of the group was encoded using another set of recurrent neurons that encoded

the outputs of each individual in the scene. Du et al. employed hierarchical LSTMs to recognize actions of individuals based on skeletal movement over time [11]. An LSTM was applied to the temporal patterns of each skeletal component through the timesteps. Then further LSTMs encoded these outputs into larger and larger groupings of skeletal components. Peng et al. used hierarchical LSTMs to understand geometric scenes [42] with sub-networks called P-LSTM for semantic concepts and MS-LSTM for structural concepts.

3.2.2.2 Natural Language Processing

Hierarchical LSTMs have also been employed successfully in a variety of Natural Language Processing tasks. Li et al. constructed a paragraph autoencoder that used two levels of LSTMs - one for the word sequences and one for the sentence sequences[33]. This two-level hierarchy outputted text that properly maintained much of the semantic meaning of the original text. Tan et al. used a Hierarchical LSTM model for image caption prediction [50]. They modeled the captions using one level of LSTM that encoded small sequences of words, and the next level of LSTM encoded these sequences of word groups.

Chapter 4

DESIGN

To safely navigate a dynamic environment, a robot must be able to predict the future positions of other agents. Most humans are able to deftly move through large crowds since they have a keen sense of where people will go. Predicting the trajectory of other agents is also important for following tasks where a robot will be more likely to maintain close proximity to a target if it can anticipate future movements. This section describes the motivation and structure of four novel neural network architectures for incorporating the influence of neighboring pedestrians into predicting the pedestrian trajectories. The goal of the four novel neural network architectures is to learn the relevance of neighboring agents to the prediction of the next position of an agent. An improved way of representing the intrinsic uncertainty involved in trajectory prediction will also be presented. These models can be used to directly estimate the most probable subsequent positions of agents, or the models could be incorporated into full navigation systems. Two examples of such full navigation systems will be presented that utilize trajectory predictions to plan routes that are unlikely to interfere with other pedestrians.

4.1 Multimodal Predictions

Regardless of how well a model learns about the common factors that affect pedestrian movement, it will never be possible to perfectly predict the path of pedestrians. There is inherent stochasticity in human movement. A pedestrian may suddenly remember that they forgot something and turn around, they may see a friend and run over to meet them, or they may choose to walk around an obstacle on either the right or left

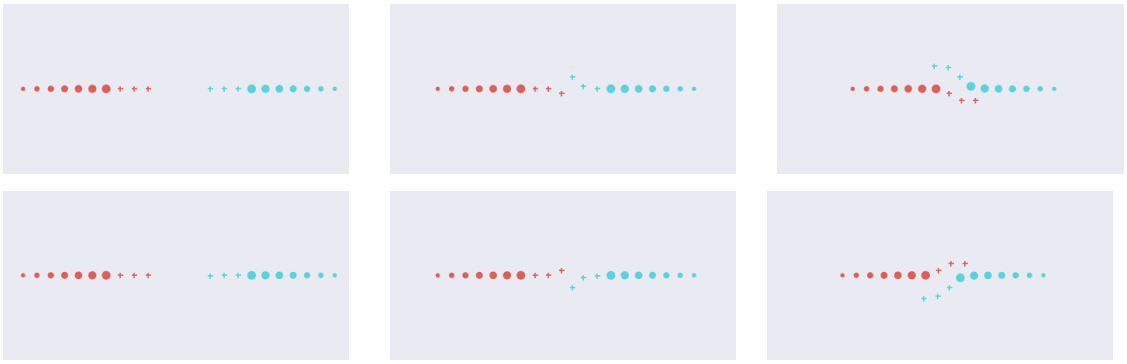


Figure 4.1: Viable collision avoidance routes

side. Figure 4.1 shows two scenarios where the red agent goes around the blue agent on either the right or the left side. Both of these routes are possible, so a prediction of the future locations of the red and blue agents should be multimodal since there is no way to know which route will be taken.

Since we never have complete certainty about the paths of pedestrians, it does not make sense to use point-estimates. Instead, the algorithms should express the likely subsequent position of the agent as a probability distribution. Social Forces (SF) outputs point-estimates, and there is no obvious extension that would enable it to output probability distributions. Interacting Gaussian Processes (IGP) outputs a probability distribution, which can be multimodal. Unfortunately, the distributions predicted by IGP are mathematically complex, and thus sampling procedures are fairly involved. Social-LSTM outputs a Bivariate Gaussian distribution, which is unimodal.

Social-LSTM was proven to be superior to IGP in several benchmarks, but Social-LSTM is limited because it cannot represent a multimodal trajectory since it outputs a single Gaussian. IGP, however, is capable of multimodal outputs. Therefore, it is sensible to develop a neural network like Social-LSTM that has the additional capacity for multimodal outputs like IGP. We propose to construct a Mixture Density

Network to output a mixture of Bivariate Gaussians for trajectory prediction. The parameters of the mixture of Bivariate Gaussians can be produced by a network like Social-LSTM or one of the alternative architectures proposed in the subsequent section. By outputting the parameters of a mixture distribution, the model can produce multimodal predictions. Additionally, sampling from the mixture distribution is trivial since it has a known formula.

There are two major advantages of a Mixture Density Network for pedestrian trajectory prediction over a single Bivariate Gaussian. The first advantage is representing multiple likely paths. If two pedestrians approach directly toward each other, it is possible for them to pass on either side of each other. It would be beneficial for a network to be able to predict those two distinct possible outcomes, which is not achievable with a single Bivariate Gaussian. Additionally, a Mixture Density Network can predict the location of pedestrians many timesteps ahead more readily than a network that outputs a single Bivariate Gaussian. When the predictions are made for multiple timesteps away there are more possibilities for the location of the pedestrian, and the ability to specify a mixture of Bivariate Gaussians would likely contribute to greater accuracy.

The Social-LSTM model outputted five parameters to define a single Bivariate Gaussian - $u_x, u_y, \sigma_x, \sigma_y, p$. Then the likelihood of the subsequent position of the agent was given by the Bivariate Gaussian distribution. The Probability Density Function (PDF) for a Bivariate Gaussian is shown in Equation **??**. The higher the value of the PDF at the ground-truth position, the better the network is performing. u_x is simply the expected x position of the agent in the next time step, while u_y is the expected y coordinate of the agent in the next time step. σ_x and σ_y are the standard deviation of the x and y coordinates respectively; they represent the uncertainty of the prediction. p is the covariance of the x and y distributions. Note that the PDF is not the probability of the agent being located at a given x and y position. The Bivariate

Gaussian distribution is continuous, so the probability of any specific location is 0. For convenience, the log of the PDF is often used since the range of values is smaller, and the formula can be written as additions rather than multiplications.

$$
\begin{aligned}
f(x,y) &= \frac{1}{2\pi\sigma_x\sigma_y\sqrt{1-p^2}} exp(-\frac{z}{1-p^2}) \\
z &= \frac{(x-u_x)^2}{\sigma_x} + \frac{(y-u_y)^2}{\sigma_y} + \frac{2p(x-u_x)(y-u_y)}{\sigma_x\sigma_y}
\end{aligned}
\tag{4.1}
$$

The alternative formulation proposed in this thesis uses a mixture of Bivariate Gaussians. To keep the parameters simple, the Bivariate Gaussians are diagonal. A diagonal Bivariate Gaussian means that the covariance is set to 0. Diagonal Bivariate Gaussians are ellipses that are oriented vertically or horizontally - there is no correlation between the x and y distributions. In small-scale experiments, diagonal Bivariate Gaussians did not reduce the performance of the networks, and it makes the models less complex. Since IGP uses two independent Gaussian Processes (GP), the joint stationary distribution of the two GPs was also a diagonal Bivariate Gaussian before considering the impact of neighbors. Equation **??** shows the single Bivariate Gaussian PDF, which is just Equation **??** when p is 0.

$$
\begin{aligned}
f(x,y) &= \frac{1}{2\pi\sigma_x\sigma_y} e^{-z} \\
z &= \frac{(x-u_x)^2}{\sigma_x} + \frac{(y-u_y)^2}{\sigma_y}
\end{aligned}
\tag{4.2}
$$

The PDF in Equation **??** can easily be expanded into a mixture distribution by outputting parameters to multiple Bivariate Gaussian and outputting a weight for each Bivariate Gaussian component. The weights must sum to zero. The mixture model PDF is the weighted sum of the PDFs of the individual Bivariate Gaussians as shown in Equation **??**.

$$\mathcal{F}(x, y) = \sum_{j=1}^{n} \alpha_j f_j(x, y) \tag{4.3}$$

For the Mixture Density Network formulation with n component Bivariate Gaussians, there are $5n$ parameters since there are n of u_x, u_y, σ_x, σ_y, and α_i. The α_is are the weights of each Bivariate Gaussian. These parameters are produced using the final fully-connected layer of the network. Equation **??** shows how the parameters of the mixture model are calculated from the output of the network. o_i^t is the final representation of agent i at timestep t. The network is trained to produce o_i^ts that represent all information about an agent necessary to predict its next location. This information could include the agent's previous locations, velocity, and the influence of neighbors. Φ denotes a fully-connected layer with weight matrix W_Θ, biases b_Θ and no activation function. The size of W_Θ is $(n * 5,\ m)$ where m is the number of components in the mixture and m is the output dimensions of o_i^t. Similarly, b_Θ has dimensions (n * 5). The mixture parameters are then extracted from the result of the fully-connected layer that produced Θ (with dimensions n*5). It is important to note that the standard deviations are element-wise exponentials of raw values since standard deviations must be greater than zero. The softmax applied to the α values ensures that sum of the αs is 1.0, which is essential for the mixture distribution to be a valid probability distribution.

$$
\begin{aligned}
\Theta &= \Phi(O_i^t; W_\Theta, b_\Theta) \\
u_x &= \Theta[1...n] \\
u_y &= \Theta[(n+1)...2n] \\
\sigma_x &= e^{\Theta[(2n+1)...3n]} \\
\sigma_y &= e^{\Theta[(3n+1)...4n]} \\
\alpha &= softmax(\Theta[(4n+1)...5n])
\end{aligned}
\tag{4.4}
$$

Equation **??** shows how a softmax is computed over a vector. The constant e is raised to the power of each element of the original vector. These exponentiated values are then scaled by dividing by the summation of e to each of the entries in the vector.

$$
softmax(v) = \frac{e^{v_i}}{\sum_{j=0}^{k} e^{v_j}} \tag{4.5}
$$

4.1.1 Prediction Examples

Figure 4.2 illustrates how predictions are made over a series of timesteps. In this diagram, the current pedestrian whose next location is being predicted is the black dots (the largest dot is the current position and the smaller dots are previous locations). The cross is the ground-truth position that is being predicted. The background color is a representation of the predicted distribution. The predicted likelihood for the next position is greatest in the dark red zones and lowest in the purple regions. The colors are on a logarithmic scale of the PDF of the predicted distribution. This example was taken from a simulation that is similar to the Hallway scenario (discussed in Chapter 5), but with more pedestrian-pedestrian interactions. The predictions were produced using the Neighbor-Attention model (presented in the next section), which outputs the parameters for a mixture of 20 Bivariate Gaussians. In this example, it is not pos-

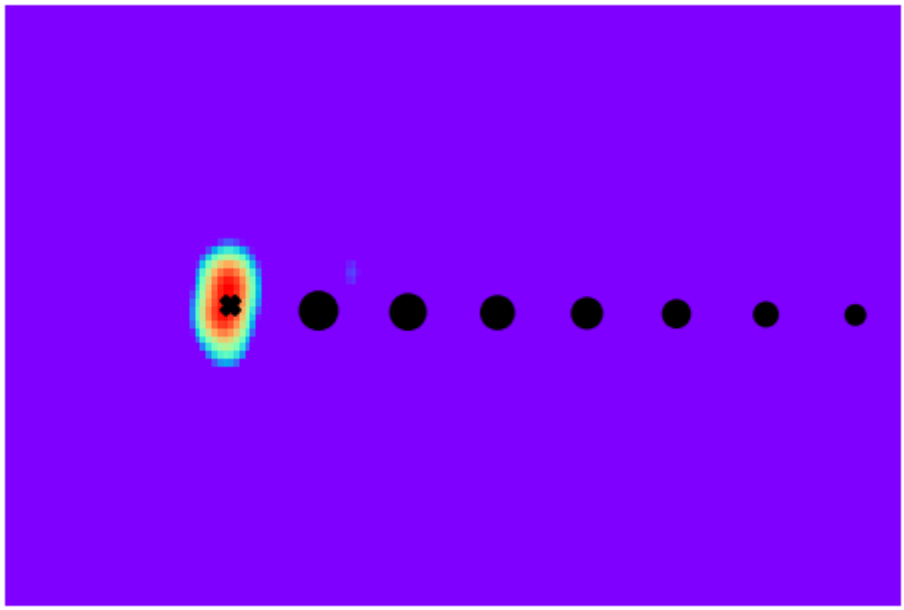


Figure 4.2: Basic Trajectory Prediction

sible to identify the component Bivariate Gaussians because the distribution appears to be unimodal. However, for this situation, a unimodal distribution appears to fit the data well. Although mixture outputs have the capacity to represent multimodal distributions, they can also represent unimodal distributions.

Figure 4.3 highlights the importance of the Mixture Density Network formulation. This example was constructed using the same modified Hallway scenario with the same model as the previous example. The white dots are another agent in the scene that is initially heading directly towards the black agent. Unlike, the unimodal distribution in Figure 4.2, there are two obvious modes of the predicted distribution in this situation. The reason for the two modes is that the network is not sure whether the black agent will go forward with the white agent to his left or right. This is a contrived example, but it is still a useful depiction of how the models can predict future positions. Examples from real-world scenarios are similar, but the multimodal distributions are not as distinct (and therefore more difficult to visualize).

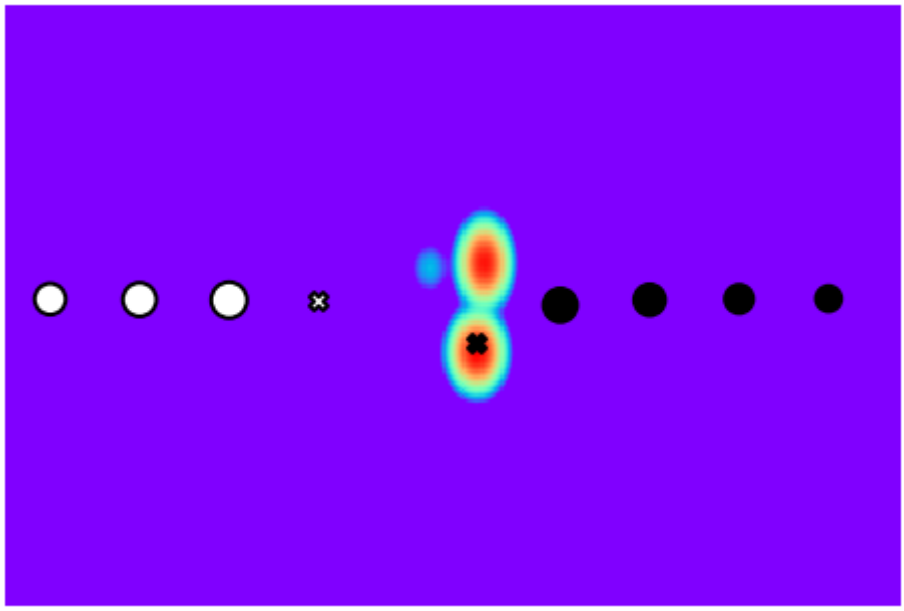


Figure 4.3: Multimodal Trajectory Prediction

4.2 Neighbor Representations

A critical component of predicting the next position of the agent is the environment in which the agent is moving. For example, pedestrians do not typically collide with walls or other pedestrians. This work focuses on the representation of neighboring agents and how these neighbors can influence the trajectory of an agent. Social Forces (SF), Interacting Gaussian Processes (IGP), and Social-LSTM (S-LSTM) are all capable of modeling the following aspects that influence pedestrian trajectories:

- Nearby Pedestrian Avoidance,
- Group Cohesion, and
- Continuity.

Nearby pedestrian avoidance means that pedestrians tend to walk in a way that minimizes the chances of getting too close to another person. Group cohesion de-

scribes the tendency of people to walk with people that they know. Pedestrians also prefer to continue walking at a constant velocity and avoid sudden shifts in direction, which means that people select actions based on continuity with previous actions. Social Forces and Interacting Gaussian Processes also account for the goal oriented behavior of pedestrians who are trying to get to their destination.

Features that may affect pedestrian movement but cannot readily be accounted for in Social-LSTM, Social Forces, or Interacting Gaussian Processes are the following:

- Long-range mimicry behavior,
- Long-range planning, and
- Environmental features.

When a person is walking and they see people walking in a circuitous route in the distance, the person may assume that the best route is the longer way rather than the direct way. In this way, people incorporate the behavior of distant pedestrians into their own walking choices; this is called mimicry. Additionally, pedestrians often consider the potential for collisions with fast-moving agents (quick pedestrians, bicyclists, cars, etc.) even though the agents are not close by. The SF, IGP, and S-LSTM models limit the impact of distant agents, so these models would not readily incorporate long-range features. Finally, environment features like the quality of the walking surface are not accounted for in any of the models, although Social Forces explicitly incorporates aversion to collisions with solid objects.

One goal of this thesis is to develop models that are capable of representing the influence of mimicry behavior and long-range planning. The addition of environmental features could be readily incorporated by inputting a dense semantic map into the network, but this will be left for future work. To produce a general model, the selection of valid neighbors must be learned by the neural network instead of specified

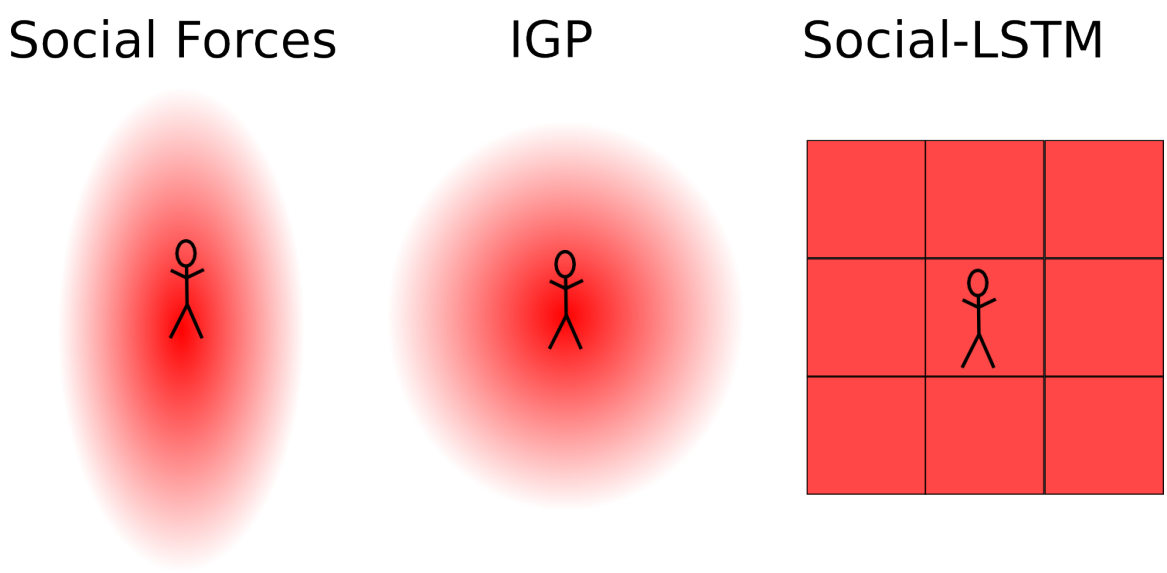


Figure 4.4: Structure of Personal Space

as hyper-parameters or engineered potentials. Figure 4.4 shows how SF, IGP, and Social-LSTM represent the personal space of pedestrians. For SF, the influence of a neighboring pedestrian is related to their position on equipotential lines of an ellipse where influence decays exponentially from the agent. Pedestrians in front of the agent have greater influence than ones on the sides. For IGP, the influence of neighbors also decays exponentially by Euclidean distance from the agent, but there is no difference between neighbors in front or on the sides of the agent. Social-LSTM uses a grid centered at the agent, and the representations of agents within each grid cell are summed to represent the immediate neighbors. Social-LSTM allows the model to selectively weight the influence of each grid cell. However, neighbors that are outside all of the grid cells will not be considered by the model.

Four novel architectures were constructed to consider all neighbors when predicting trajectories. These architectures were inspired by Social-LSTM, but they all differ in how they represent neighboring pedestrians. The differentiating feature of each design is the calculation of the neighbor representation (referred to as the social tensor in the Social-LSTM paper). The diagrams presented in this section reference the

sample scenario shown in Figure 3.1. All of the diagrams demonstrate how the next position of the green agent is calculated. To differentiate between the architectures, the models will be referred to using an abbreviation that describes how the model represents the influence of neighboring agents. Social-LSTM will be referred to using the abbreviation *S-LSTM*.

All four novel architectures share the same overall structure of Social-LSTM. This basic model structure was presented in *Chapter 3* in Figure 3.2 and Equation **??** where the diagram refers to the scenario of predicting the subsequent location of the green agent (agent id of 4) in the next time step. The coordinate embedding e is calculated exactly the same as in the Social-LSTM model for all architectures. The neighbor representation s, however, is calculated in a different way for each model. These ways of representing the neighbors are discussed next. There are two underlying goals that unify the approach for each architecture:

- Reduce spatial hyper-parameters and
- Consider influence of all agents.

As stated above, the existing state-of-the-art solutions for trajectory prediction involve specific hyper-parameters that define the influence of neighbors. These hyper-parameters are a major roadblock for developing new robot navigation systems that incorporate agent trajectory prediction. These hyper-parameters must be tuned to each environment (based on the speed and preferences of other agents). If a system can learn to detect which pedestrians are relevant and likely to influence the behavior of other agents, then the engineers do not have to spend time testing and tuning parameters. Additionally, even with well-tuned parameters, there are likely cases where distant agents may influence the trajectories of other agents. A model should be capable of incorporating both close-range and long-range interactions in order to output the best possible estimates of the trajectories of other agents. The

architectures presented in this thesis do not assume which pedestrians are relevant and do not require any special hyper-parameters for adjusting the importance of other agents. In this way, these designs can save engineers time and have the potential to more accurately estimate trajectories since the influence of other agents is learned by the model.

4.2.1 Occupancy Grid (OCC)

Tracking multiple targets simultaneously while also moving is not an easy feat. In order for robots to utilize pedestrian trajectory prediction for planning, the robot must first be able to track all of the neighboring agents over time. Since this may not always be feasible, the Occupancy Grid (OCC) architecture is presented as a way of predicting trajectories without historical knowledge of neighboring agents. The additional advantage of representing neighbor coordinates in a large grid is that it is trivial to encode uncertainty in the measurements of other pedestrians. Typically, the sensors on robots have limited accuracy and therefore measurements are accompanied by a degree of uncertainty. A robot may detect a person, but represent the knowledge of that person as a probability distribution that describes the likely position of the person rather than the precise coordinates of the person.

The Occupancy Grid architecture creates an agent-centric grid that has a width of twice the scene width and a height of twice the scene height. The grid is partitioned into cells where the number of cells is a hyper-parameter although in testing the granularity (cell count) of the grid had a minimal effect on performance once the cell count exceeded about 100. Each cell is initially set to zero. Then the PDF of the coordinate for each neighboring agent in the scene is evaluated at each grid cell and the result is added to the value of the grid cell. The PDF of the coordinates can be provided by a robot based on the accuracy of its sensors. In the experiments

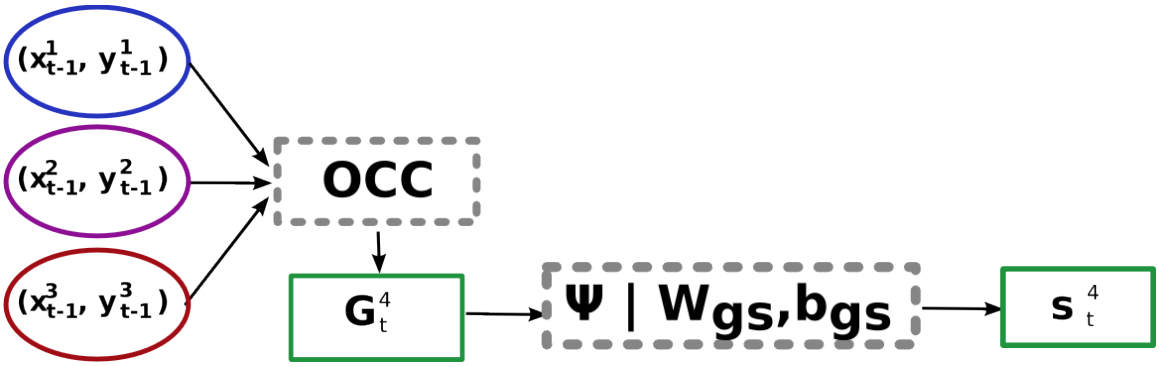


Figure 4.5: Occupancy Grid Architecture (OCC)

conducted for this thesis, the PDF was specified as a diagonal Bivariate Gaussian centered at the actual position of the agent, and variance in the x and y directions set to approximately the minimum observed distance between agents.

Equation **??** shows how the values of each grid cell are calculated. f_t^j is the PDF of the distribution of the neighbor j's position at time t. For the experiments presented in this thesis, f was defined as the PDF of $\mathcal{N}((x_t^j, y_t^j), (\sigma, \sigma))$. Where (x_t^j, y_t^j) is the true position of neighbor j at time t and σ is the square root of the minimum distance between two agents. s_t^i is the embedding of the grid using weights W_{gs} and biases b_{gs}.

$$\begin{aligned} G_t^i(k,l) &= \sum_j f_t^j(k,l) \\ s_t^i &= \Psi(G_t^i; W_{gs}, b_{gs}) \end{aligned} \tag{4.6}$$

Figure 4.5 shows how the Grid is computed and used in the context of the entire network. The gray *OCC* box outputs an n by n matrix where n is the number of cells in the width and length of the grid. The inputs to the *OCC* operation are the coordinates of the other agents and not the latent representations from the Agent-LSTM. Of course, for real applications, the inputs would be the parameters of a distribution specifying the uncertainty about the location of the neighboring agent.

The Social-LSTM paper also presented an Occupancy model, but this was using the same small grid as used in the full Social-LSTM approach. Additionally, the Occupancy model in the Social-LSTM paper was binary where each grid cell was 1 if there was one or more agents in it or 0 otherwise.

The major limitation of the Occupancy Grid architecture is that there is no way to represent the velocities or previous trajectories of neighboring agents. This is a severe limitation because strategies for avoiding collisions depend on knowledge of velocity. For example, if there is one agent in front of another; if the agents are heading towards each other, then at least one of them should take evasive action and step to one side. However, if both agents are going in the same direction, there is no need to alter their course. In the Occupancy Grid model the network cannot learn how to respond differently in those two situations. Nonetheless, the Occupancy Grid model is less complex than the other ones and has the added benefit of allowing for uncertain measurements of the location of neighbors.

4.2.2 Hierarchical LSTM (HIER)

It is common practice for pedestrians to scan to the left and right before crossing a street. During this scan, the pedestrian is assessing the location and velocity of other pedestrians and vehicles before determining whether it is safe to cross the street. This also happens as pedestrians navigate dense crowds; they tend to search the environment for paths that are most free of obstructions. The way pedestrians scan their surrounding was the motivation for the Hierarchical LSTM (*HIER*) model. It is intuitive for humans to construct a mental map of the environment by considering each agent one at a time and storing the relevant information about that agent in their memory. The Hierarchical LSTM model seeks to emulate this behavior.

The Hierarchical LSTM model replaces Social-LSTM's grid-pooling of the neigh-

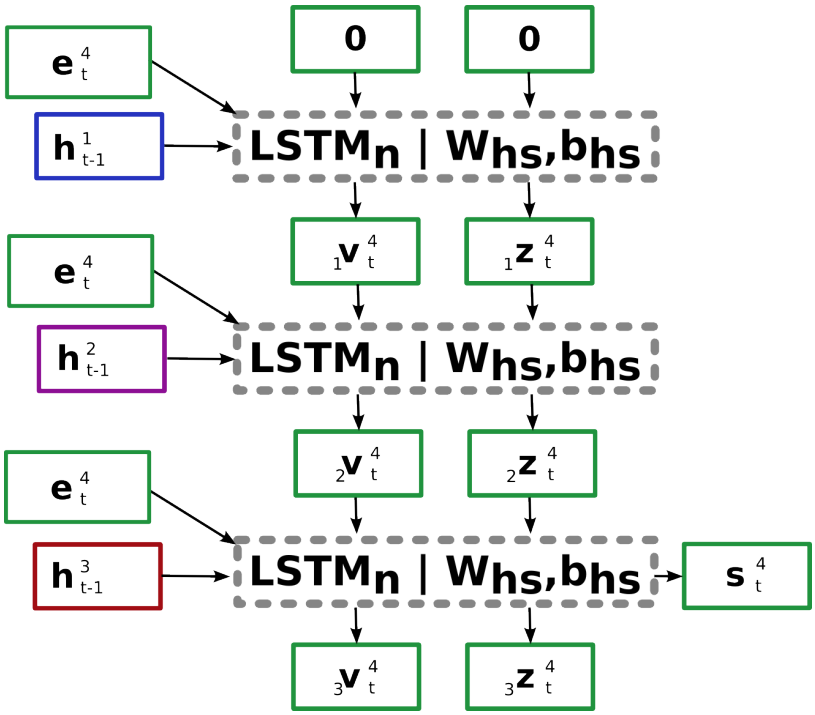


Figure 4.6: Hierarchical LSTM Architecture

boring agents with an additional LSTM. To differentiate the two LSTMs, the LSTM for the agent will be referred to as Agent-LSTM, while the LSTM that processes the neighbors will be referred to as Neighbor-LSTM. The Neighbor-LSTM encodes all of the latent representations of the neighbor trajectories into a single tensor. This tensor is the neighbor representation that is then inputted into the Agent-LSTM along with embedding of the current agent's coordinates. Figure 4.6 shows how the Neighbor-LSTM is applied to the trajectory representations of the neighbors to produce the neighbor tensor s. The initial cell state and hidden states are 0 vectors, but then the hidden states and cell states are populated with the important information from each latent state of the neigbors.

Equation **??** defines how the neighbor representation tensor is calculated for the Hierarchical LSTM architecture. The input to the Neighbor-LSTM is a concatenation of the latent representations of a neighbor's trajectory along with the embedding

of the coordinates of the current agent. The reason that the current agent's coordinate embedding is included is to enable the network to learn the importance of each neighbor relative to the current agent rather than the importance of each neighbor in the general sense. Essentially, the gates of the LSTM can calculate values that depend on the distance of the neighbor to the current agent and use that information to determine the impact of the neighbor on the current agent's path. Equation **??** shows how the Neighbor-LSTM is applied sequentially to all of the neighbors in the scene. ${}_jv_t^i$ represents the hidden state of the Neighbor-LSTM and ${}_jz_t^i$ represents the cell state of the Neighbor-LSTM after encoding the j$^{\text{th}}$ neighbor at timestep t when the current agent is i. The _ is the output of the LSTM at the intermediate timesteps. The intermediate outputs of the Neighbor-LSTM are discarded because only the output after processing the final agent includes all of the relevant information about each neighbor in the scene.

$$
\begin{aligned}
{}_{-,1}v_t^i, {}_1z_t^i &= LSTM_n(h_{t-1}^1, e_t^i; \mathbf{0}, \mathbf{0}; W_{hs}, b_{hs}) \\
{}_{-,2}v_t^i, {}_2z_t^i &= LSTM_n(h_{t-1}^2, e_t^i; {}_1v_t^i, {}_1z_t^i; W_{hs}.b_{hs}) \\
&\dots \\
s_t^i, {}_{k-1}v_t^i, {}_{k-1}v_t^i &= LSTM_n(h_{t-1}^k, e_t^i; {}_{k-1}v_t^i, {}_{k-1}z_t^i; W_{hs}, b_{hs})
\end{aligned}
\tag{4.7}
$$

Since LSTMs are adept at capturing information over long time horizons, the order of the processing of each neighbor should not be significant, although the model might be more robust if during training the order of the neighbors is shuffled. Another possible choice would be to sort the neighbor representations by their distance to the current agent in descending order (farthest away first). This may potentially help the model capture the information from the nearby agents, which are more likely to be important, but it was not implemented in this version. A huge number of neighbors in the scene may make the calculation of the neighbor tensor very slow since the LSTM

is applied to each one, but it is unlikely that a robot's sensors could detect more than a few dozen agents since other agents would be occluded from view.

4.2.3 Spatial Attention (SATTN)

Humans are inundated with information in crowded scenarios like shopping areas, schools, and airports. Each of a human's senses provide a different perception of the environment. Some percepts provide critical information to navigating - seeing an imminent collision or hearing a train approaching - while other senses may be superfluous - smelling exhaust or seeing an advertisement. Humans have a fine-tuned mechanism for attending to only the relevant percepts that can be used to aid in decision-making. Attention is especially important for navigation; humans cannot process all available information so they must choose what is the most relevant. When crossing a street, the most salient information would be about the oncoming traffic, while knowledge about other pedestrians across the street is less significant. The goal of the Spatial Attention architecture is to train a network to attend to the neighboring agents that are present in the regions that are most likely to influence the movement of the current agent.

The Social-LSTM architecture can be considered a type of attention model where the regions that deserve attention are predefined using hyper-parameters. The *SATTN* architecture lets the model learn these regions. The primary motivation for the Spatial Attention architecture is the DRAW network that was created by Gregor et al. [19]. The attention mechanism in the DRAW network is a set of parameters that align a collection of Gaussian filters against a patch of the input image. The DRAW network uses Soft-Attention, which means that the entire network is differentiable and can be trained using Backpropagation. For the DRAW network, only five parameters are needed to specify the grid used to apply the Gaussian filters. Unlike the DRAW

network, the Spatial Attention architecture for trajectory prediction employs one or more distinct Gaussians for attention. The number of attentions is a hyper-parameter of the model. In handwriting generation, it is acceptable to focus on a single region of the input image, but in trajectory prediction it may be important to attend to multiple distinct regions.

The *SATTN* architecture is not a traditional attention mechanism because the model learns global attention regions while most attention networks (including DRAW) select the region to attend to based on some function of the inputs. While it would be straightforward to compute the attentions based on the inputs to the network (current trajectory of the agent), there is far less data available in trajectory prediction than other domains (like handwriting recognition), so the decision was made to learn global attentions rather than ones specific to the state of the inputs. The next section will present an alternative attention architecture that does use the state of the current agent to attend to different agents.

$$
\begin{aligned}
R_t^i(m) &= \sum_{j \in N^i} g_m(x_t^j - x_t^i, y_t^j - y_t^i) \cdot q_{t-1}^j \\
s_t^i &= \Psi(R_t^i; W_{rs}, b_{rs})
\end{aligned}
\tag{4.8}
$$

Equation **??** describes the way that the neighbor tensor is computed. m is the attention index where R_t^i has the shape (n, l) where n is the number of attentions and l is the dimension of the latent representation of the trajectories of each agent. m goes from 1 to n. g_m is the PDF of a diagonal Bivariate Gaussian $N(\mu_x^m, \mu_y^m, \sigma_x^m, \sigma_y^m)$ where each parameter is learned by the network. The total number of parameters learned by the model for the attention mechanism is n * 4 since each attention require 4 parameters. The total value of each attention component of the neighbor tensor is the sum of each neighbor agent's hidden representation weighted by g_m evaluated on the difference between the neighbors coordinates and the current agent's coordinates.

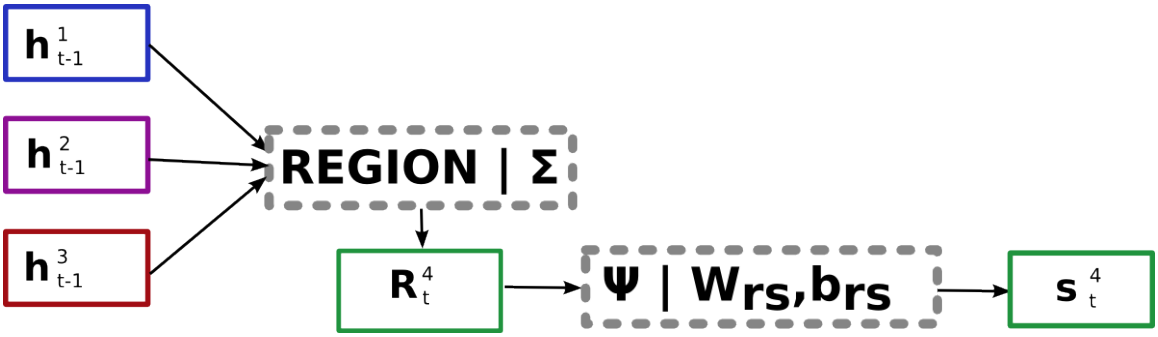


Figure 4.7: Spatial Attention Architecture

The attention matrix R is embedded into a vector s using a fully-connected layer with relu activation and weight matrix W_{rs} and bias vector b_{rs}.

Figure 4.7 is a visual representation of the attention architecture. In the diagram Σ refers to all learned parameters, which is each μ_x^m, μ_y^m, σ_x^m, and σ_y^m.

4.2.4 Neighbor Attention (NATTN)

Like the Spatial Attention architecture, the Neighbor Attention model learns to attend to parts of the input that are most likely to be relevant in predicting the trajectories. Unlike, *SATTN*, the Neighbor Attention architecture selects the importance of other pedestrians individually rather than through weights given by Gaussians. *NATTN* is a more traditional attention model because the inputs are used to determine what parts of the input should be attended to. The intuition behind the Neighbor Attention model is that humans may perform a cursory glance around the environment then focus on the aspects of the environment that are most meaningful. The *NATTN* model is the only architecture that essentially performs two layers of analysis of other pedestrians. Informally, the first layer considers the positions of the other pedestrians in relation to the current pedestrian and outputs a measure of how influential each other pedestrian is likely to be to the current pedestrian. These measures of importance are used to create a weighted vector of the latent representations of the other

pedestrians. Like the *SATTN* model, the Neighbor Attention model is Soft-Attention and is therefore fully differentiable and can be trained using Backpropagation.

Concurrently with our effort to build attention mechanisms for trajectory prediction, Fernando et al.[12] also devised an attention architecture that shares some similarities with the Neighbor Attention model (but almost not overlap with *SATTN*). However, Fernando et al. only learned temporal attention where their model learned to attend to the latent state of the current pedestrian at previous timesteps. They used hard-wired attention that weights the latent representations of other neighbors using the inverse of Euclidean distance. Therefore, the only learned attention in the work of Fernando et al. is in the temporal domain, while the model presented here only learns attention in the spatial domain. The models presented by Fernando et al. still face the same limitations as Social-LSTM, IGP, and Social Forces because the importance of neighbors is predefined rather than learned.

$$
\begin{aligned}
q_t^i &= \frac{e^{\Psi(e_t^i, e_t^m; W_u, b_u)}}{\sum_l e^{\Psi(e_t^i, e_t^l; W_u, b_u)}} \\
A_t^i(m) &= q(m) \cdot h_{t-1}^m \\
s_t^i &= \Psi(A_t^i; W_{as}, b_{as})
\end{aligned}
\tag{4.9}
$$

Equation **??** describes the way that the neighbor tensor is computed. As before, Ψ is a fully-connected layer with a relu activation function. q_t^i is a vector with length equal to the number of neighbors that holds the importance weights for each neighbor. These importance weights are calculated using a softmax over the fully-connected layer that has as inputs the embedding vector of the coordinates of the current pedestrian and the coordinate embedding of each other pedestrian. When the Ψ function with parameters W_u and b_u outputs a large value for a neighbor, then that corresponding entry in the q_t^i vector will be large (and thus the neighbor is considered important). The softmax is used to force the sum of the entries in the q_t^i

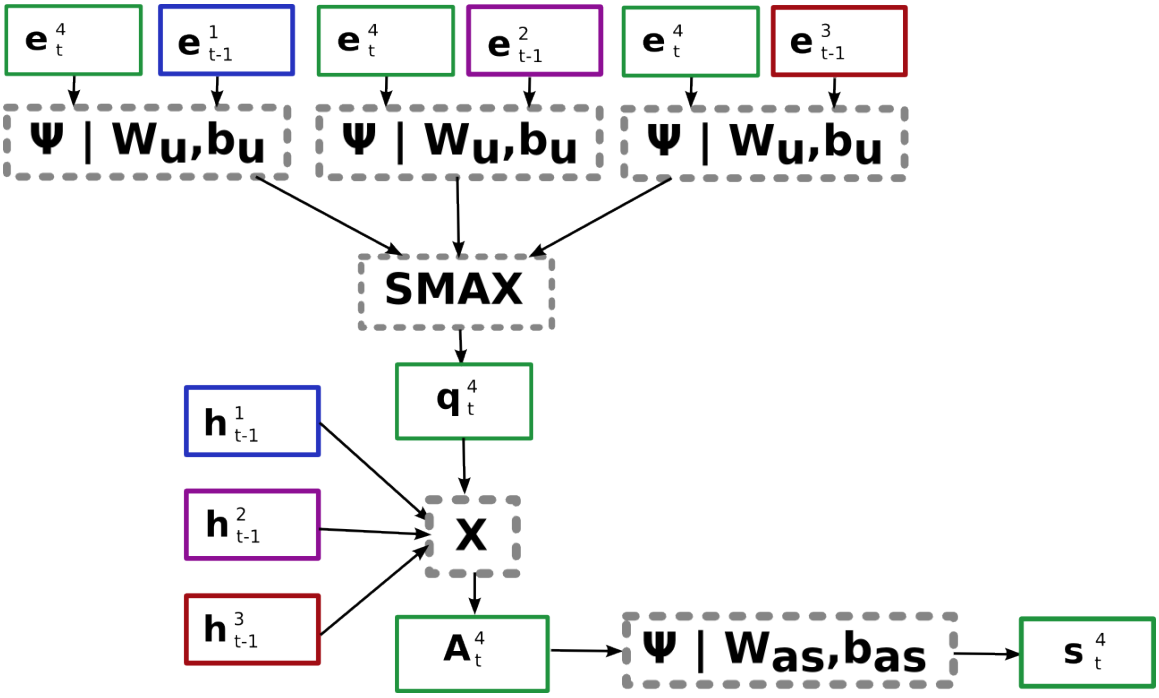
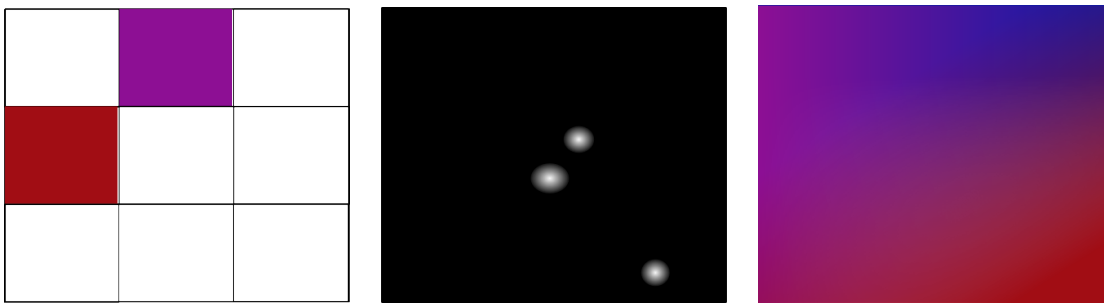


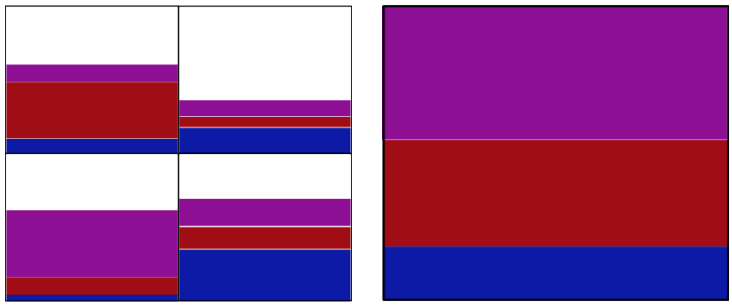
Figure 4.8: Spatial Attention Architecture

vector to equal one, which ensures that no single latent representation of a neighbor can be massively inflated. Each entry of the matrix A_t^i is calculated as the weight of the neighbor m times the latent representation of neighbor m (h_{t-1}^m) at the previous timestep. The weights are broadcast to perform element-wise multiplication. Finally, the A_t^i matrix is embedded into a social tensor using the relu activation function, weights W_{as}, and biases b_{as}.

Figure 4.8 is a visual representation of the attention architecture. The X in the gray box represents element-wise multiplication of the entries of q_t^i against the corresponding latent representations. The diagram clearly highlights the fact that this model incorporates two representations of the neighbors (the coordinate embedding and the latent trajectory representation), while all other architectures only used one representation.



(a) S-LSTM (b) OCC (c) HIER



(d) SATTN (e) NATTN

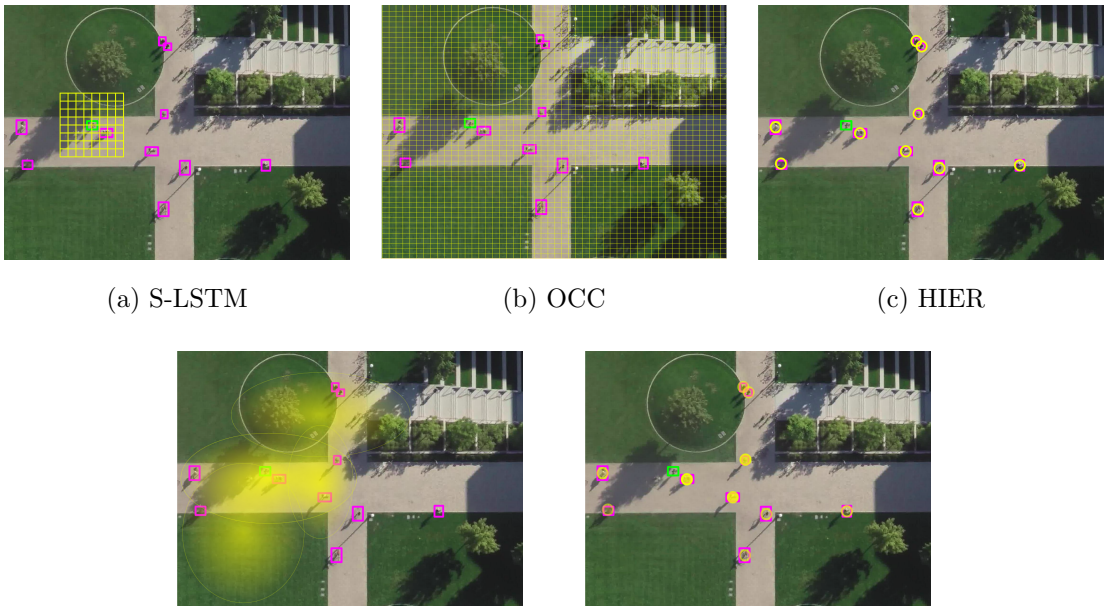
Figure 4.9: Structure of Social Tensor

4.2.5 Representation Summary

Figure 4.9 illustrates the conceptual differences between the various ways of representing the social tensor that models the influence of neighbors on the path taken by the current agent. The colors correspond to the latent representations of the other pedestrians in the example scenario where the goal is to predict the next position of the green agent. The Social-LSTM model (*S-LSTM*) uses a grid where the hidden representation of each nearby agent is placed in the grid cell according to each neighbor's location relative to the current agent. The total size of the grid restricts which agents are relevant to the current agent. In Figure 4.9, the blue agent is not represented in the Social-LSTM social tensor since it is outside the grid around the green agent (and is thus not considered relevant). In all three architectures proposed in this thesis, all agents are considered. The Occupancy Grid (*OCC*) encodes the

coordinates of other pedestrians into a large grid that encompasses the entire scene. The white dots in the Occupancy Grid example are the large values in the grid (representing regions where pedestrians are located). The Hierarchical LSTM (*HIER*) remembers information about each neighboring agent by sequentially examining each agent's latent representation. The mixing of the colors in the Hierarchical LSTM social tensor indicate how the Neighbor-LSTM incorporates components from the latent representations of each neighboring agent. The Spatial Attention (*SATTN*) model learns to weight the influence of neighbors based on regions that are defined by learned parameters. In Figure 4.9 the SATTN model is learning four regions that each weight the influence of the neighbors differently (where the number of regions is a hyper-parameter). Finally, the Neighbor Attention (*NATTN*) architecture selects the importance of neighbors by comparing their coordinates to the current agent's coordinates and applying those weights to the latent representations of the neighbors. In the figure, the box for the *NATTN* model illustrates how the social representation is a weighted combination of the latent representations of all neighbors (similar to the *SATTN* model). All of the depictions in Figure 4.9 are representations prior to embedding with the exception of the *HIER* model, which does not require an additional embedding layer.

Figure 4.10 shows the relevant regions in a scene that are considered by each architecture. The scene is taken from the Stanford Drone Dataset[45]. The regions shown in the diagram are for illustration purposes only and do not represent the true regions learned by each architecture. The green agent is the target of the prediction and all other agents are highlighted in pink. The Social-LSTM model uses grid pooling, so only those agents within the yellow grid are considered. The Occupancy Grid uses a massive grid with many tiny cells that capture the coordinates of all agents. The Hierarchical LSTM considers all other agents, which is why a yellow circle is placed on all agents. The Spatial Attention model selects the regions of neighbors



(a) S-LSTM (b) OCC (c) HIER

(d) SATTN (e) NATTN

Figure 4.10: Relevant Neighbor Regions for Architectures

to consider by modifying the parameters of one or more Bivariate Gaussians. In the figure, the Gaussians are represented by yellow ellipses. The darker yellow regions are given more weight. The figure depicts a boundary around the ellipses to highlight their shape, but in the real model every neighboring agent is included in every attention component. The weight approaches zero for agents that are far away from the center of the Bivariate Gaussian. The Neighbor Attention model incorporates information from all agents, but it selectively weights each agent, which is why some of the yellow circles on agents are shown darker than others.

The grid of latent representations used by the Social-LSTM model requires a much larger number of parameters than the architectures that are proposed in this research. Table 4.1 shows that S-LSTM learns nearly double the parameters of the next largest model (Hierarchical LSTM) and learns more than four times as many parameters as the smallest model (Neighbor Attention). Architectures with fewer parameters are generally advantageous because less memory is required to store the values. However,

Table 4.1: Number of Learned Parameters for Architectures

Model	Parameters (thousands)
S-LSTM	669
OCC	247
HIER	342
SATTN	161
NATTN	153

fewer parameters does not necessarily translate into faster inference or less overall memory consumption during inference. The memory and speed requirements of a model are highly specific to the software used to implement them. Nonetheless, reducing the number of parameters in a model can achieve a variety of desirable results including reducing the likelihood of over-fitting and producing more compact representations that can be more readily used in other systems.

4.3 Planning in Dynamic Environments

4.3.1 Dynamic Horizon A*

The A* search algorithm is nearly ubiquitous in path planning applications due to its simplicity and impressive performance. Before planning a route using A*, the map is usually discretized into cells where each has a cost associated with it (see *Chatper 3* for a detailed treatment of costmaps). The cost of each cell can be used to constrain the search space or alternatively to penalize paths. In the constraint formulation, cells with costs above a threshold are marked as unnavigable and excluded from the planning. For the penalty formulation, the cost of each cell is a component of the overall cost of a route; this means that longer paths may have an overall lower cost if they traverse lower cost cells.

A* requires a heuristic cost function that is admissible, which means that the heuristic function must always produce an optimistic estimate of the expected cost from the current location to the goal. When the cost of each cell is used as a penalty, creating an admissible and useful heuristic cost function is much more difficult than the constraint case. In this work, A* will be used on a costmap where the robot is constrained to some cells while others cells are deemed impassable. The A* search performed for this project departs from the standard use-case because it must consider dynamic elements in the environment.

The proposed extension to A* is Dynamic Horizon A* where dynamic objects are incorporated into the costmap only when they are near the robot. The rationale for this choice is that agents that are far away will likely move before the robot is able to reach their current location, yet it is important for the robot to plan a route that avoids nearby agents. The horizon must be specified according the specifications of the robot. The requirement of defining a horizon may appear antithetical to the previous work on eliminating hyper-parameters in the trajectory prediction models, but the horizon value is much different. The horizon is based on the kinematic constraints of the robot - how far it can move in a single timestep. For Social-LSTM, the hyper-parameters are chosen to best predict human trajectories where it is not known a-priori which neighbors may affect the trajectories. For a robot, the influence of neighboring pedestrians is known since the robot is merely planning a route that is free of obstructions. The horizon value is specified as a radius from the current location of the robot. All cells within this horizon circle (defined by the radius) are checked to ensure that they are not likely to be occupied in the next timestep by another agent and that there are no static obstacles in that cell. Cells outside of the circle are considered valid if there is no static object in the cell.

The likelihood of a cell being occupied is calculated using the neural network models for pedestrian trajectory prediction. For each agent in the scene, the parameters

of the distribution of their likely next location is specified by the output of the neural network. Samples are drawn from these distributions and the distance between the sampled points and the coordinates of the cell are compared. If the distance between the cell and the sample is less than two times the radius of a pedestrian, then the sample indicates a potential collision. By taking many samples (in the experiments, 1000 samples were taken), the proportion of collisions with the samples indicates the overall probability of a collision if the robot moves to that cell in the next timestep. The developer of the robot can then define a threshold where a probability greater than the threshold means that the cell is not safe since the possibility of collision is too high, while probabilities lower than the threshold mark valid cells.

The Dynamic Horizon A* approach is convenient because it can be readily incorporated in path planning systems that only handle static obstacles. Additionally, it is fast and will converge to the shortest possible path for the current costmap. However, there are two drawbacks. The first issue is that it only considers one timestep into the future and thus it cannot consider the long-range influences of other agents potentially walking into its path in the future. The second issue is that it has an inflexible cost function. A* planners are used to calculate the shortest safe path to the destination, but there are other criteria that may affect the choice of a path. For example, the robot may wish to minimize the amount of times that it forces pedestrians to walk around it. There is no clear way to incorporate such a goal into the A* search procedure.

While not immediately apparent, a well-chosen horizon value should enable the Dynamic Horizon A* method to minimize the effects of the frozen robot problem. The frozen robot problem occurs when the robot cannot plan a viable route - no safe path exists. If static objects block all paths to the destination, then there is obviously no planning algorithm that can overcome the issue, but there are other cases where dynamic agents (pedestrians) may currently block all paths to the destination. Yet,

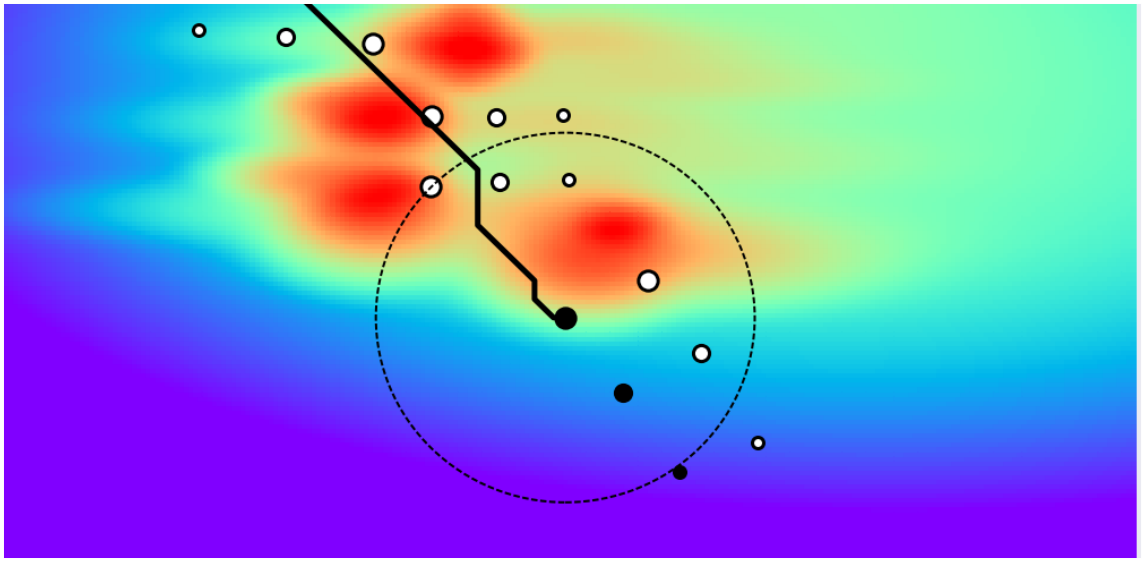


Figure 4.11: Dynamic Horizon A* at Each Timestep

people are likely to make room for a robot or other pedestrian trying to go forward. Since the robot only considers cells invalid when a neighbor is likely to be in the cell and the cell is within the horizon, the robot can still plan a path towards the destination that is blocked by people as long as the people are outside of the horizon. A horizon that is too large may cause the robot to plan circuitous routes or fail to find any route (frozen robot problem). A horizon that is too small may allow the robot to collide with pedestrians.

Figure 4.11 is an example of the plan created by the Dynamic Horizon A* algorithm at each timestep. After constructing the path (shown in black), the subsequent position of the robot (at the next timestep) is the location on the planned path that is closest to the destination but still possible for the robot to reach within a single timestep. The dashed line in the figure indicates the dynamic horizon. The white dots are the positions of other pedestrians, and the background is a heatmap showing the probability of agents occupying the space in the next timestep. The path clearly avoids the red regions (areas that are highly likely to be occupied) within the circle.

However, the path goes directly through the red regions outside of the horizon because it is expected that the pedestrians will have moved past these areas by the time the robot reaches them.

4.3.2 Tree Search

An alternative to the A* approach described above is a local tree search. A tree search constructs a tree of nodes (representing states or positions) and links between the nodes (representing actions or movements). The goal of the search is to find a path that maximizes a reward or minimizes a cost. For navigation, the nodes are positions of the robot, and each layer of the tree (nodes with the same depth) are positions of the robot at a specific timestep. The root of the tree is the current position of the robot. Unlike the A* method, the state-space can remain continuous, but the action-space (set of all actions) will be made into a discrete set. The discrete set of actions essentially limits the state-space to also be a countable set since only a finite number of actions can be taken. For the purposes of navigation in two-dimensional space, the actions will be defined as tuples of distance and angle pairs. As a concrete example, consider that the robot can move any number of feet in $\{0, 1, 2\}$ at any angle in $\{0, \frac{\pi}{4}, \frac{\pi}{2} ... \frac{7\pi}{4}\}$ for each timestep. That makes the total action-space contain 24 pairs although 7 of those are redundant because the angle does not matter if the robot does not move forward.

Even with a discrete set of actions, it may be intractable to explore a sequence of all possible actions from each reachable state. Therefore, it is desirable to explore paths (action sequences) that are more likely to lead to optimal (or near optimal) routes. However, it is also reasonable to explore paths that may not at first appear as optimal as the others, but may in the end be the most successful. The trade-off between searching around the paths that are known to be good and searching for other

paths whose payoffs are unknown is a famous issue usually described as exploration versus exploitation. A class of problems called Multi-Armed bandits are often used to analyze solutions that balance exploiting known good solutions while exploring for potentially better solutions. One technique that is useful for these situations in Upper Confidence Bound (UCB), which is explained thoroughly in Sutton's book on Reinforcement Learning [48]. Equation **??** shows the details for UCB where $Q(s, a)$ is the value of a certain action a taken in state s. $N(s, \cdot)$ is the number of times the state s has been visited (explored), while $N(s, a)$ is the number of times the action a has been chosen in state s. The c variable is a user-defined weight for how to balance the first and second terms.

$$Q(s, a) + c \cdot \sqrt{\frac{ln(N(s, \cdot))}{N(s, a)}} \tag{4.10}$$

Using UCB, the search should explore (choose the action) that maximizes the UCB equation. If the $Q(s, a)$ for an action is high then it will be more likely to be chosen since it has performed well. However, the second part of the equation will skew the algorithm to choose actions that have not been explored as often as the other actions from the current state. One method of tree search is called Monte Carlo Tree Search where nodes are typically chosen using the UCB criteria (although other methods can also be used) when statistics of the states and actions are known (have already been visited). For unvisited states, a random action is generally chosen. Since good heuristics for estimating the value of being in a state are available for path planning (namely the distance of the current state to the goal) it is not necessary to choose actions randomly. Rather, the UCB equation is used at each state to choose actions during the search process, which means that the approach taken in this thesis is not Monte Carlo Tree Search since there is no randomization.

The initial quality of a state and action ($Q_o(s, a)$) is set according to the Equation

?? where *dist* is a function that calculates the Euclidean distance between two points. v is the maximum velocity of the robot and n is the maximum number of timesteps into the future that will be considered. g is the destination coordinates and s_o is the starting coordinates of the search (current position of the robot). $V(s)$ is the value of a state, which is one minus the scaled distance from the state s to the goal. The worst possible result of the robot's movement from s_o after n steps is being w away from the goal, while the best possible result is being only b away from the goal. Therefore, w and b are used to scale the value of a state $V(s)$ to be between 0 and 1. The initial quality of a state and action pair ($Q_o(s,a)$) is the value of the state that results from taking action a from the state s to reach s'.

$$\begin{aligned} w &= dist(s_o, g) - v * n \\ b &= dist(s_o, g) + v * n \\ V(s) &= 1 - (dist(s, g) - w)/(b - g) \\ Q_o(s,a) &= V(s') \end{aligned} \tag{4.11}$$

During the search process, the planning starts at the root node (current state of the robot) and selects the action that maximizes the UCB equation. Then the same procedure is applied at the subsequent state and so on until the maximum depth (n) is reached. The value of the last node in the exploration is then propagated backwards to all of the previous states to update each $Q(s,a)$ where the new $Q(s,a)$ is just the average value of all sequences that go through that pair. The number of visits to each state and each state and action pair are also updated. This procedure is repeated many times. Once the search is complete, the action from the root node that was visited most often is chosen for the next movement of the robot.

Of course, this discussion has omitted the key details of what happens when a state is not valid. The validity of a state is calculated using the same logic as was used for

the A* approach where many samples are taken from the predicted distributions of the next location of each agent. A state is invalid if the probability of a collision is greater than a threshold. The major advantage of the tree search is that there is an explicit notion of time since each level of the tree occurs as the same timestep. Therefore at the root node, the probability distributions based on the current positions of the nearby agents are used. At the second depth, a position for each neighboring agent is sampled from the first set of predicted distributions. The previous locations of the agents along with these estimated positions at the next timestep are used as inputs to the neural network to output new parameters for probability distributions that represents where the agents will likely be two timesteps into the future. A similar procedure is used at greater depths of the tree. In this way, the algorithm is able to take into account the movement of other agents across many timesteps, which was not possible in the Dynamic Horizon A* approach.

When the chosen action yields a state that is invalid, the iteration down the current sequence is terminated early. The value that is propagated up the tree to root is the value associated with the state before the invalid state. This captures the intuitive notion that if last valid state were reached then no progress could be made by choosing that action that led to the invalid state. Cutting the search short in these cases can dramatically decrease the average quality of the states and actions that led to the invalid state, so the search will prioritize actions that do not lead to invalid states.

One feature of this tree search method is that it will often visit the same sequence of states and actions many times. Although that appears undesirable, it is actually useful because each time a node deep in the search tree is reached, the estimated current positions of other agents will be different due to the stochastic sampling of the distributions. By visiting these states multiple times, the algorithm verifies that these desirable sequences of actions are robust to all the possible movements of the

pedestrians. Even just a few times when a seemingly optimal sequence results in an early termination due to one of the states becoming invalid will reduce the value of that sequence and force the search to consider paths that are more likely to be successful. Sometimes the search finds an optimal path early (for example, when there are no nearby pedestrians), and repeated traversal of the path does not yield new information. In these cases, the search is stopped when the number of visits to one action is more than double all visits to the other actions in that state and the number of total visits to the state is greater than a user-defined number (in the experiments, this number was 10).

In the formulation of the tree search here, the quality of the action and state pairs is a function of the distance to the goal after taking the action in that state; however, this same tree search approach could handle alternative metrics like how much the robot disturbs other agents. In Monte Carlo Tree Search, the value of a leaf node (last node in a search sequence) is typically estimated by applying a rollout policy from that state to completion (reaching the goal). The current approach just uses the heuristic of distance to the goal from the last state to define this value of the leaf node, but a rollout policy may be needed if there are no good heuristics for approximating the chosen metric of a good path.

Overall, tree search is a straightforward way of finding paths in a dynamic environment that minimize a cost or maximize a reward. It is robust at handling highly stochastic environments because it can repeatedly explore a sequence of actions. Since the tree search uses the neural network models for predicting the locations of pedestrians, theoretically it should be highly effective at solving the frozen robot problem because it can detect how pedestrians will respond when it plans a route directly towards the blocking pedestrians.

Figure 4.12 showcases how the tree search picks the next position of the robot. All

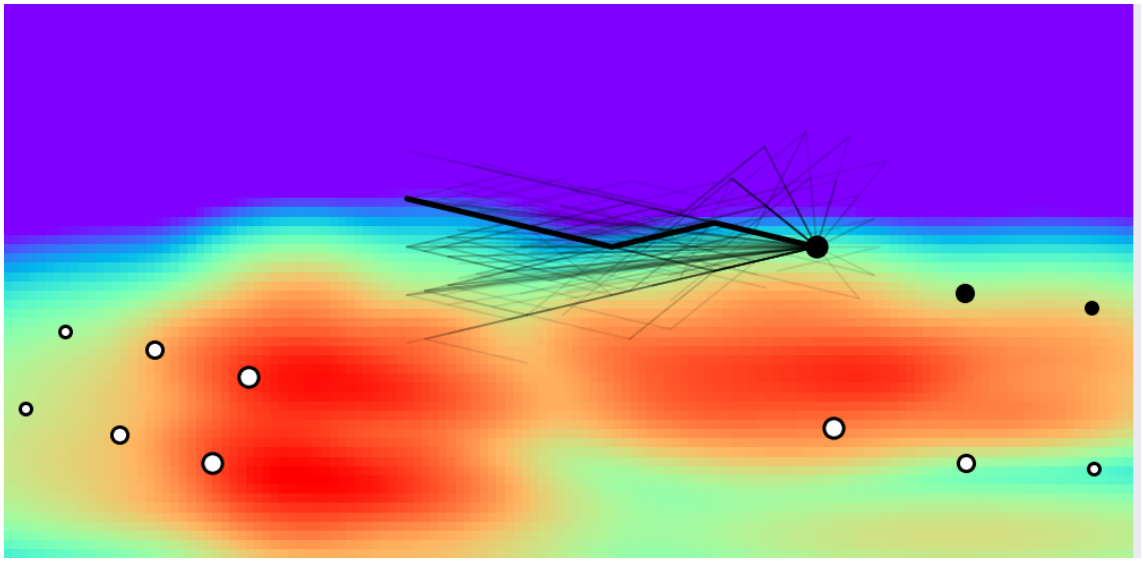


Figure 4.12: Tree Search at Each Timestep

of the thin black lines are movement sequences that were evaluated by the algorithm. The thick black line is the final sequence. The robot will move take the first action in the final sequence. The figure depicts the large variety of paths that were tested. The darker lines are generally all pointing towards the left, which indicates that these paths were the most promising (result in a position that is closest to the destination). The final sequence appears to bend upwards (even though the destination is slightly lower than the current position). The reason for this deflection is to make room for the pedestrian directly below the robot. By simulating the movement of the other pedestrians, the algorithm detects that at future timesteps, the robot may collide with the agent directly below it if the robot moves lower. After moving forward at this timestep, the entire tree search is repeated at the next timestep.

Chapter 5

IMPLEMENTATION

There is no exact process for training and evaluating neural networks. There are many techniques that can be applied to solve issues like slow convergence, local minima, and overfitting. Many of these techniques and procedures have one or more hyper-parameters that must be appropriately set to ensure the desired results. The techniques and methods outlined in this section were evaluated on small simulation scenarios before running on the final datasets. Using simulations allowed for the rapid testing of hyper-parameters and techniques since the networks converge much faster on simple scenarios with few agents. The simulations were also used during the construction of the architectures to ensure correctness of the implementation.

5.1 Simulations

All of the simulations were were created using the Social Forces model. Argil - a library designed for pedestrian simulation - was used to model, visualize, and process the scenarios. More information about Argil is presented in *Chapter 7*. Argil includes an implementation of the Social Forces model that was inspired by the implementation in Menge[10].

For debugging, two environments were constructed - a Hallway scene and a Fork scene. Sample frames from the Hallway are shown in Figure 5.1, and sample frames from the Fork scene are shown in 5.2. In each figure, the current location of the agent is the largest circle while the trail of smaller circles shows the previous locations of the agent. In all scenarios, there are ten agents moving from one *sink* towards one of the other *sinks*. A *sink* is an entrance or exit from the environment (the Hallway has

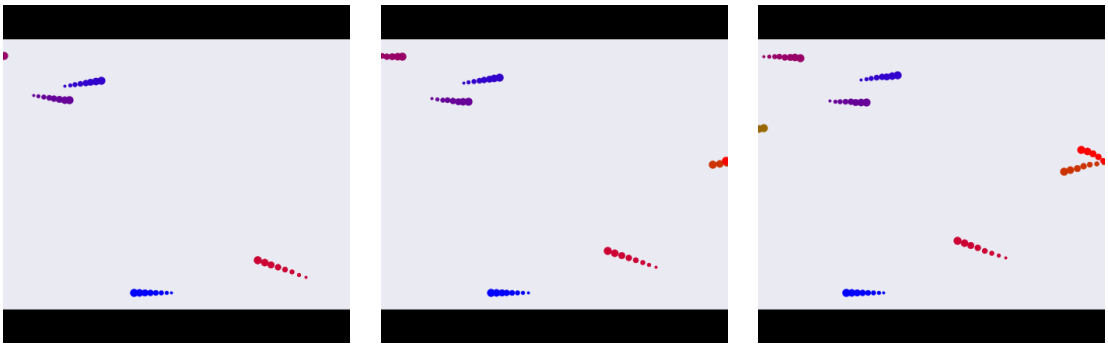


Figure 5.1: Representative Sequence from Hallway Simulation

two *sinks* and the Fork has three *sinks*.

Data was obtained by simulating the progress of agents for 2000 timesteps, and each agent began its path after a random delay where the delays are calculated from the uniform distribution with lower bound of 0 and upper bound of 400 timesteps. Each scene corresponds roughly to a 10 meter by 10 meter region in the real-world. For every tenth timestep, the position of all agents was recorded.

For the Hallway scenario, agents were randomly assigned to a starting side and each agent then moved towards the other side horizontally. The vertical start and end positions of each agent were calculated using a Normal distribution with a standard deviation of 2.0 centered at a y-coordinate of 7.5 for the agents traveling right and a y-coordinate of 2.5 for agents traveling left. This choice of y-coordinate ensures that the majority of the agents traveling right will be on the upper half of the hallway while those traveling left will be on the lower half. Figure 5.1 shows how the tendency of agents to travel together in the same direction results in relatively few head-on collision-avoidance situations.

The Fork scenario (Figure 5.2) includes a much greater number of pedestrian interactions. Agents in the Fork scenario were randomly assigned to one of the three start sides and then a goal destination was also selected at random. To make the

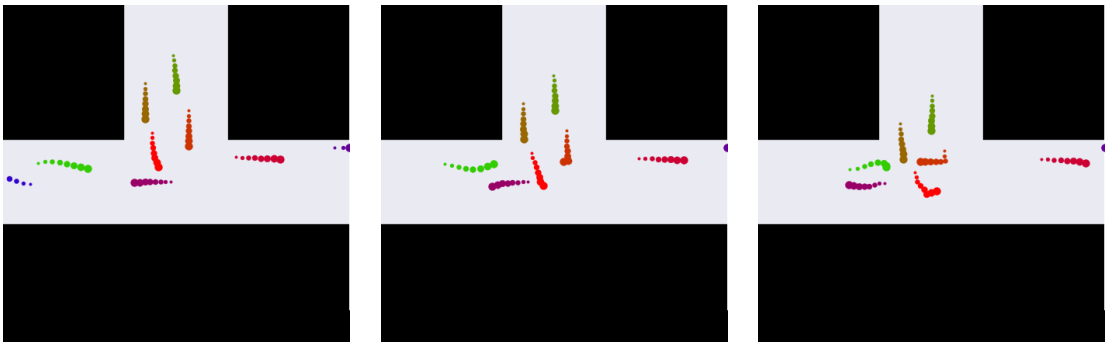


Figure 5.2: Representative Sequence from Fork Simulation

scene even more interesting, each agent was also given a waypoint in the center where all three pathways converge. These center waypoints must be reached by the agent before it can continue on to its final destination. The center waypoints were assigned using the Uniform distribution where every position in the center is equally likely to be chosen.

Evaluating architectures on multiple types of interactions provides a better estimate of how well the model may perform on real-world situations where there is a mix of complex and simple interactions. Since the Hallway simulation involves few pedestrian interactions, it was used to validate that all neural network architectures were capable of learning the linear patterns of movement. The models converged to reasonable solutions within 30 minutes to one hour on the Hallway simulations, so these simulations were used to evaluate general hyper-parameters. The Fork scenario involves many interactions between agents; the Fork simulations were used to validate the neighbor representations to ensure that the models were able to detect nearby agents.

5.1.1 Hardware and Software

All neural network models were trained on Amazon EC2 with the memory optimized instances. r4.large instances (15.25 Gigabytes RAM, 2 vCPUs) were used for small simulations with fewer than 10 agents; the r4.xlarge instances (30.5 Gigabytes of RAM, 4 vCPUs) and the r4.2xlarge instances(61 Gigabytes of RAM, 8 vCPUs) were used for scenarios with more than 10 agents. Training on GPUs was only marginally faster than CPUs, so the less expensive servers without GPUs were used. The bottleneck that prevented faster training on GPUs was the high memory requirements of the models during training. When the maximum number of agents in a scene was ten, most models used 4 to 8 Gigabytes of RAM. For models trained on environments where up to 40 agents were present in an individual scene, 10 to 35 Gigabytes of RAM were required. The SATTN model typically had the highest memory requirements since each attention Gaussian would need to process all other agents in a loop.

All networks were implemented in TensorFlow[1] v1.0, and the networks were trained with a learning rate of .003 using the RMSProp[15] optimizer. Every epoch (training using all samples) the learning rate was annealed by 10%. Decaying the learning rate has been shown to help networks converge to better solutions. Gradient clipping was employed to prevent the error gradients for becoming too large or too small. When the absolute value of the gradient becomes large, the parameters can quickly approach infinity or negative infinity. To avoid this, each individual gradient was clipped to be between positive five and negative five.

In general the training of the neural network models was chaotic. Small changes to the hyper-parameters like learning rate or Dropout probability could dramatically affect the convergence. Using the Social Forces simulations was essential to quickly debug the architectures. A logistical challenge of building the models was the high memory requirements; a custom system for queuing training jobs on AWS servers was

developed to reduce time spent configuring and setting up training jobs.

5.2 Reducing Overfitting

Overfitting is the term used to describe models that effectively memorize the training data, and then fail to generalize well to new datasets. Overfitting is an especially common problem in deep neural networks because the networks have a high capacity to learn relationships in the data that may be just noise in the training data rather than true patterns. In this work, four techniques were employed to reduce overfitting and improve the generalization performance of the networks: data augmentation, Dropout, weight regularization, and early-stopping.

5.2.1 Data Augmentation

The first method was to artificially expand the training data through data augmentation. Each frame of the original datasets was flipped in the x-direction, y-direction, and both directions to produce three equivalent representations of the trajectories. A visualization of the data augmentation strategy is shown in 5.3. Using flipped coordinates enables to the network to learn the relative movement of the pedestrians rather than the movement in just one direction.

Neural networks are most successful when they are presented with many unique samples. Many successful image classification networks use tens of thousands of images, and translation networks are often trained with hundreds of thousands of words or sentences. Data augmentation is a simple technique for increasing the number of training samples. The data augmentation is this thesis is relatively naive and only helps the network learn orientation agnostic relationships. More involved methods like jittering the coordinates could be beneficial, but these will be left for future work.

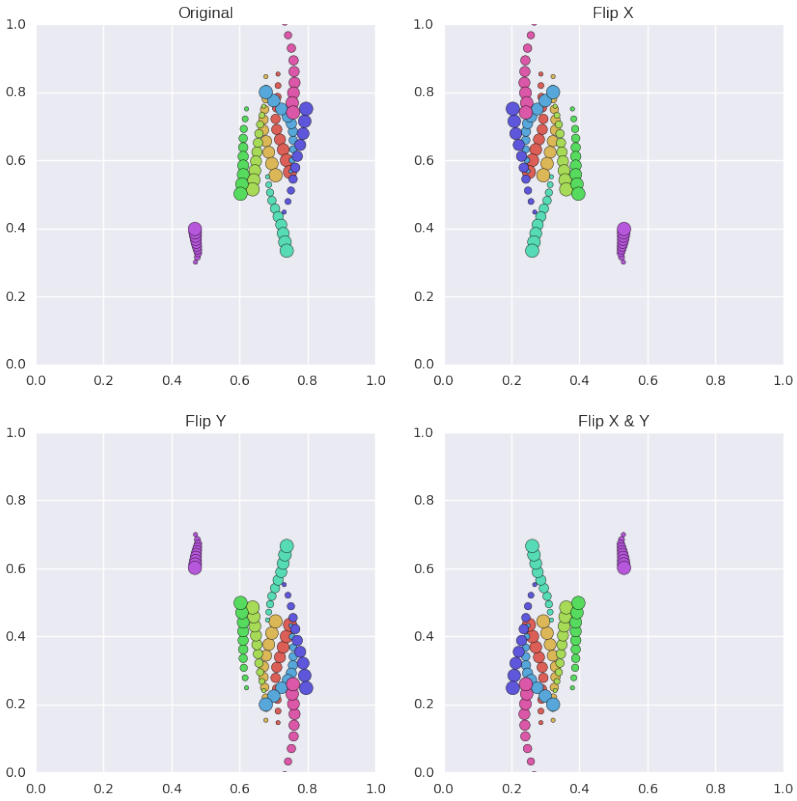


Figure 5.3: Data Augmentation

5.2.2 Dropout

Dropout[47] was the second method that was implemented to avoid overfitting. Dropout is a technique that randomly sets some of the activations in layers of the network to 0 during training. By setting some activations to 0, the model is forced to more fully utilize all parameters of the network. An alternative view of Dropout is that it causes the network to learn an ensemble of smaller (reduced) networks. Recently, the concept of Dropout has been used to quantify the uncertainty in a neural network[14]. Dropout has been applied with success to a diverse array of neural network architectures. Dropout is most readily applied to Feed-Forward networks that include fully-connected layers or convolutional layers. Applying Dropout to recurrent neural networks requires a careful approach. If Dropout is applied to the memory (hidden

or cell state) of a recurrent neuron, then the performance of the model may degrade drastically, as information is no longer able to propagate through time properly. Thus it is typical to use Dropout only on the inputs or outputs of recurrent neurons.

The major source of overfitting in the models that were tested was the neighbor representation. During initial experiments, the networks would memorize configurations of neighbors to predict subsequent locations of an agent. To prevent this, Dropout was applied only to the neighbor tensor. Dropout was applied to the neighbor representation with a keep probability of 50%. Therefore, 50% of the values in the neighbor representation were set to zero during each batch of training. After the validation loss stopped decreasing, the Dropout was removed, and the models were trained further. In this way, the networks are forced to rely on the sequence of previous positions of the agent first before learning how neighbors might influence that trajectory.

5.2.3 Weight Regularization

Weight regularization is a well-known method that is used in linear regression and neural networks. Weight regularization adds an additional penalty (based on the magnitude of the weights) to the loss function. In L1 regularization, the penalty is computed using the absolute value of the weights, and in L2 regularization, the penalty is computed using the squared value of the weights. In both cases, the model then learns a set of weights that simultaneously maximize accuracy and minimize the absolute value or squared value of the weights. L1 regularization tends to produce sparse weight matrices where only some of the weights are significant and the others are close to zero. L2 regularization tends to produce weight matrices where all weights are relatively small, with few or no large weights. Weight regularization can be applied to any of the parameter matrices in a network, but it is typically not applied to the

bias vectors.

For the neural networks considered in this thesis, weight regularization was used to avoid nan (not a number) values. Generally a nan value occurs when a weight matrix has massive values (or tiny values) that cause a value to reach positive or negative infinity. Nan values cannot be used as parameters of a probability distribution, so it is essential that the neural network never introduce nans. By regularizing the weight matrices, no nan values were produced. For all trained models, L2 regularization with a scale factor of .001 was applied to the weight matrices of the final probability parameters (not the bias vectors).

5.2.4 Early-Stopping

All models were trained for approximately 100 epochs, although some were trained for less if the loss failed to decrease after 400 batches. Additionally, early-stopping was implemented to reduce the chances of overfitting. After each epoch, the loss on a validation set was computed. If the loss on the validation set increased by more than a set threshold from the last evaluation, then training was halted. If a network is allowed to train forever, it may continue to improve its fit on the training data, but it may perform worse on new data. The early-stopping technique is one way of detecting the point at which the network begins to no longer learning generalizable rules but rather begins memorizing the randomness inherent in the training data.

While effective, early-stopping requires careful planning in order to ensure that the training is not stopped too soon. For the training of the models in this thesis, a slack of .3 was chosen. As long as the network never produced a validation error that was .3 worse than the best validation error, then it was allowed to continue to train. This slack is especially important in the beginning of training where there is still large changes to the weights. After the training was halted, the weights associated with

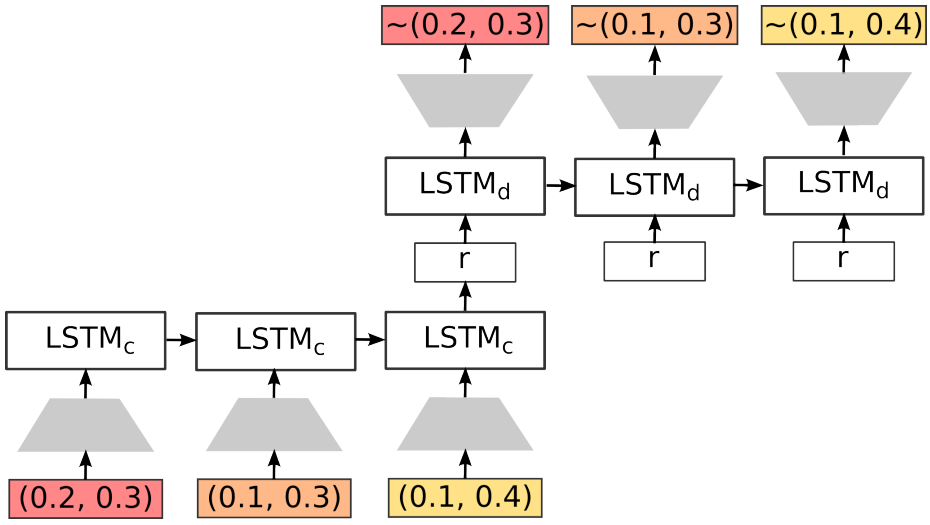


Figure 5.4: Autoencoding Architecture for Trajectories

the smallest validation error were used in the evaluation on the test data.

5.3 Pre-training with Autoencoders

Autoencoding networks were designed and evaluated to test various ways of encoding the neighbor information into vectors and represent previous trajectories. It is straightforward to create a trajectory autoencoder like the one shown in Figure 5.4. This autoencoder works by first embedding each coordinate in the trajectory of an agent (the gray trapezoids) then feeding those embedded coordinates into the LSTM. In the diagram, there are three coordinates that constitute the trajectory. After the final coordinate is processed, the output of the LSTM is used as an input to a decoding LSTM. The decoding LSTM outputs a dense vector for the same number of timesteps as the encoding LSTM was applied. Each of these outputs is decoded using a fully-connected layer. The autoencoder is then trained to minimize the squared Euclidean distance between the output of the decoder and the actual coordinates at each timestep. Both variational regularization and activity regularization were tried. However, neither regularization technique improved the performance of the network,

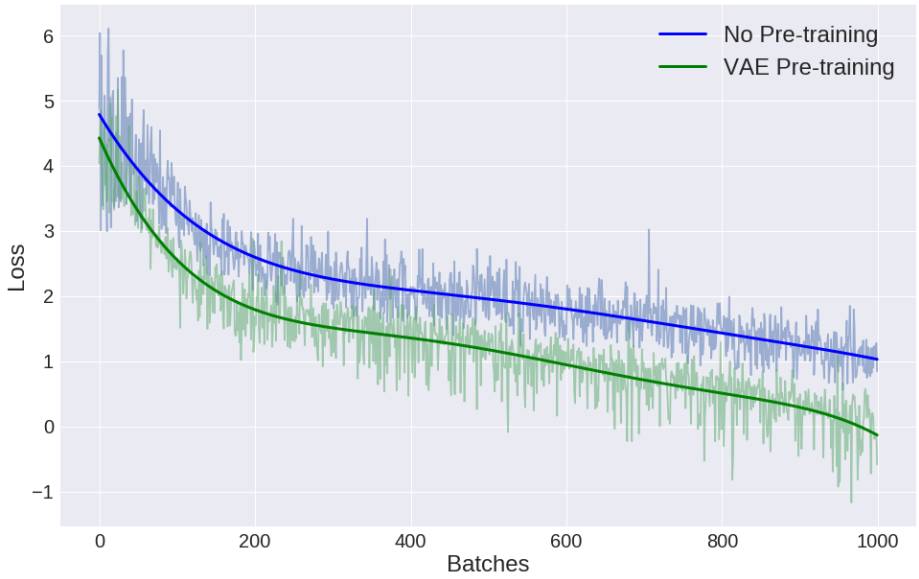


Figure 5.5: Effect of Pretraining on Learning

so in the final version no regularization was applied.

The weights and biases of the coordinate embedding layer and the encoding LSTM that were learned during autoencoding were used to initialize the weights in the base model (an architecture that uses a single LSTM and incorporates no information about neighbors). The decoding LSTM parameters and the decoding fully-connected layer parameters were not used since there is no analogue of them in the real architectures. The use of weights from another model is called pre-training. Pre-training has been shown to not only increase the rate of convergence of a network but also improve overall performance. Figure 5.5 is a representative example of how the use of pre-training increased the rate at which the network was able to learn.

While the autoencoded weights for the LSTM benefited the baseline model that omits information about neighbors, these LSTM weights could not be restored into the other models since more than just the coordinates are inputs to the LSTM in the other models. Nonetheless, the success of autoencoding is what motivated the use of Dropout and then removing Dropout. By using Dropout on the neighbor rep-

resentations and then eliminating Dropout, the networks rely more heavily on the previous trajectory than neighbors at the onset of training. This is akin to learning the representation of the trajectories using autoencoding before applying those representations in the full prediction model. A Hierarchical Autoencoder was developed to encode coordinates into trajectory tensors with one LSTM and then use another LSTM to encode all of trajectory tensors into a single tensor that was then decoded into trajectory representations and then individual coordinates. Unfortunately, the Hierarchical Autoencoder failed to converge to meaningful values and was thus not incorporated into final designs.

Chapter 6

RESULTS

There are three aspects of this thesis that can be evaluated using both qualitative and quantitative techniques. The first is the ability of the proposed neural network architectures to accurately predict the subsequent locations of pedestrians in a crowded scene. The second is how well a Mixture Density Network can estimate the final destination of a pedestrian (where they will exit the scene), which cannot readily be accomplished with models that specify only a single Bivariate Gaussian. Finally, the path planning algorithms are compared to the routes chosen by real-world pedestrians.

6.1 Trajectory Prediction

Two experiments were conducted to examine the effectiveness of each trajectory prediction network - one on simulated pedestrians and one on a real-world dataset. In addition to the Social-LSTM model and the four models presented in this thesis, another baseline model was trained and evaluated. The baseline does not include neighbor information and is just a simple LSTM over the coordinates of individual agents. The metric used to compare models is the mean log likelihood of the ground truth positions given the parameters outputted by the neural networks.

$$Score = \frac{\sum_{i=1}^{n} log(P((x_i, y_i)|\Theta_i))}{n} \tag{6.1}$$

Equation **??** shows how the mean log likelihood measure is calculated where n is the number of instances to predict and i is the index of the instance. An instance

is a case where the agent was observed for the certain number of timesteps, and the coordinates x_i and y_i are the ground-truth positions of that agent in the next timestep (not known to the neural network.) Θ_i are the parameters of the probability distribution produced by the neural network for instance i.

The authors of Social-LSTM (and the authors of IGP) evaluated their models on predicting the actual trajectories of agents over several timesteps into the future. This approach was not taken in this research because outputting a single point-estimate for where a pedestrian is likely to be disregards the inherit uncertainty in where that agent may go. In the Social-LSTM paper, the authors show an example where Social-LSTM predicted on valid path while the actual agent chose a different path. In their analysis, this valid path was heavily penalized since it deviated significantly from the ground-truth. However, acknowledging that path as a possible choice for the agent is important for navigation applications where all possible paths of an agent should be considered. Therefore, evaluating the models using the probability of the ground-truth locations better captures the ability of a network to specify its certainty in the predictions. Of course all models proposed in this thesis could be used to output a single path by sampling from the distributions specified by the parameters outputted by the neural network, but these exact paths are not as important to the task of navigation.

6.1.1 Simulations

Using the same procedure that was used to build simulations for debugging, a final scenario was constructed and run for evaluation of the models. The evaluation environment is the Intersection (Figure 6.1) where agents move between four *sinks*. Each agent was assigned a random starting location and a random end location. Agents were also assigned a midpoint location deterministically by calculating the average of

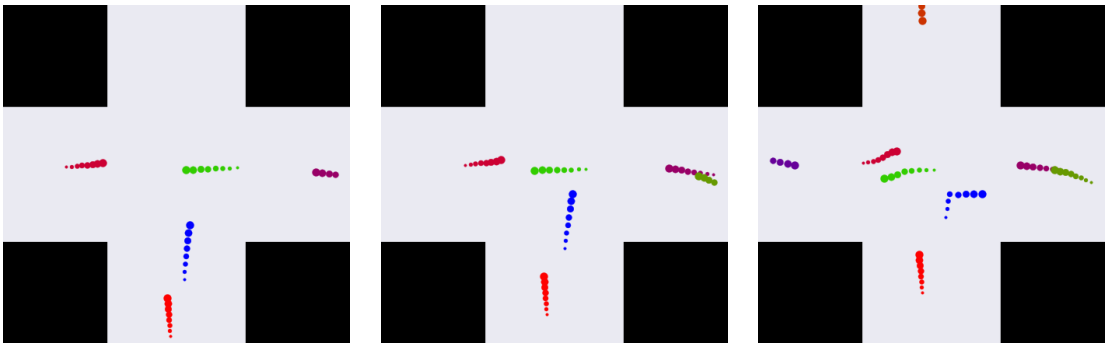


Figure 6.1: Representative Sequence from Intersection Simulation

the starting and ending x and y coordinates. The Intersection scenario involves many interactions between nearby agents.

Like the original Social-LSTM paper, coordinates were embedded in a vector of size 64, and the Agent-LSTM contains 128 values in the hidden and cell states. The S-LSTM model was trained using an 8 by 8 grid and a neighborhood size of .05, which is roughly equivalent to the original 32 pixel neighborhood size used in the Social-LSTM paper when converted from pixel-space to the scaled simulation data. For the S-ATTN model, the number of attentions was set to 4. All models were trained over 8 timesteps where at each timestep the model predicted the position of each agent at the next timestep. The loss function is the negative log of the PDF of the next position using the parameters outputted by the network.

100 iterations of the Intersection simulation were created for training, 20 for validation, and 20 for testing. Training was halted when the error on the validation set started to increase according to the method described in *Chapter 5*.

Table 6.1 shows the complete results from the Intersection scenario. The columns labeled with Single refer to models that outputted only one Gaussian while the columns labeled Mixture predicted using 20 Gaussians. **Bold** entries indicate the highest value in each column. The entries in the table are calculated using Equation

Table 6.1: Results for Intersection Simulation

	Single	Mixture
BASE	5.88	6.49
S-LSTM	6.73	**7.58**
OCC	6.48	7.21
HIER	**7.29**	7.50
SATTN	6.17	7.04
NATTN	6.72	7.49

??. The higher values correspond to a better model that has higher accuracy at predicting where the agent will move next. While the model is trained with a loss function that is applied at each timestep, the results are only based on predicting the position of an agent after observing a full 8 timesteps of positions of that agent.

The results are also summarized in Figure 6.2. The dashed red line indicates the score of the BASE model (does not incorporate neighbor information) with only a single Bivariate Gaussian probability distribution. The dashed yellow line is the score of the BASE model using 20 Bivariate Gaussians.

When using only a single Bivariate Gaussian, the Hierarchical LSTM achieved the best performance, but the original Social-LSTM model barely outperformed the Hierarchical LSTM when using a mixture of Gaussians. For all models, using a Mixture Density formulation significantly improved performance. Notably, the Neighbor-Attention architecture also outperformed the Social-LSTM model when the output was a single Bivariate Gaussian. Due to the randomness of the training process, it is likely that the differences among Social-LSTM, Hierarchical LSTM, and Neighbor Attention models are not significant for practical purposes.

All models that incorporated information about neighboring agents outperformed

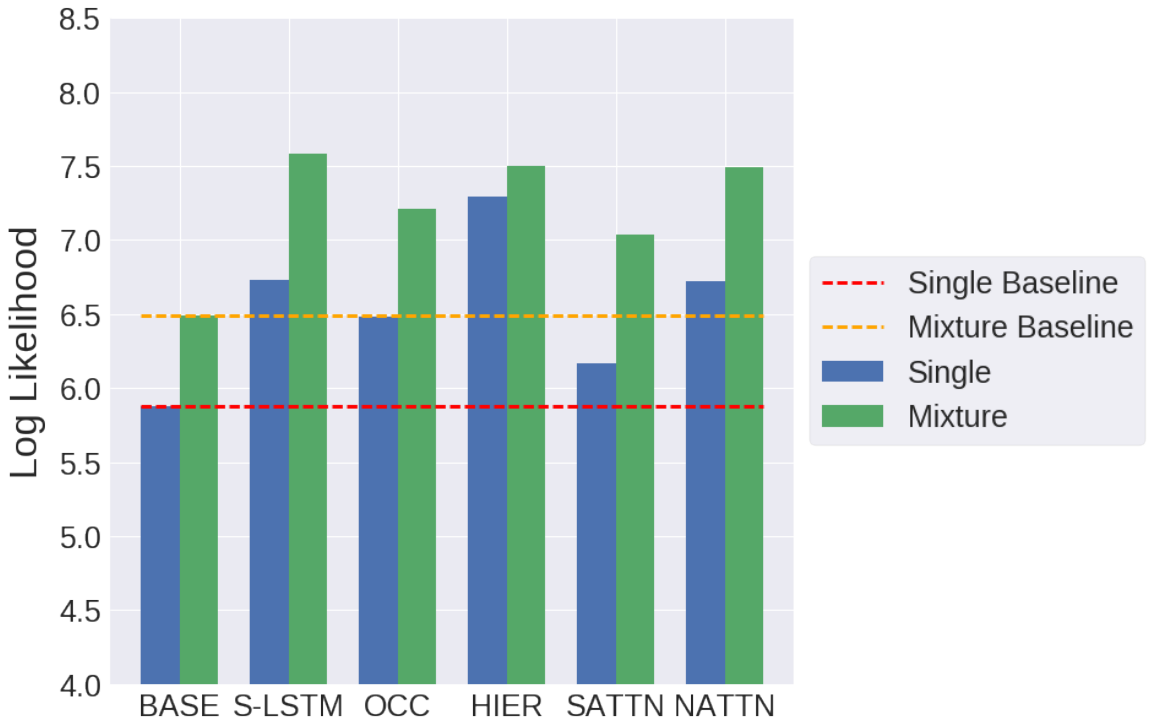


Figure 6.2: Intersection Simulation Results

the baseline. For the single Bivariate Gaussian ouptut, the worst performing model (besides the baseline) was the Spatial Attention architecture. The Spatial Attention model learned to attend to regions specified by a mean of zero and relatively large standard deviations of approximately 1.5. This indicates that the Spatial Attention model was only incorporating vague information about neighbors. The reason that it outperformed the baseline is likely partly attributable to the additional entropy induced by the neighbor vectors, which allowed it to train longer before overfitting.

The Occupancy Grid (OCC) model only outperformed the baseline and Spatial Attention model. This is not surprising because the OCC model does not have access to the velocities of the other agents, so it may not be able to adequately anticipate some evasion behavior that depends on which direction the neighboring agents are heading.

Figure 6.3: Sample Frames of UCY Dataset[32]

6.1.2 UCY Pedestrian Dataset

While simulations are useful for choosing hyper-parameters and selecting potential architectures, real-world datasets provide the strongest validation of neural network models. The authors of the Social-LSTM paper used data from the ETH[41] and UCY[32] datasets. Both of these datasets were constructed from an overhead camera looking at an area of land where many people were walking. The position of the agents at at each timestep was recorded and logged. The duration of the timestep between measurements for both datasets was .4 seconds. The UCY datasets have significantly higher concentrations of people and therefore the interactions between pedestrians have a larger impact on the trajectories of individuals. Additionally, the height of the video recorded for the ETH and UCY datasets is different, so it is not easy to train using both datasets unless accommodations are made for the difference in height. Due to the difficultly of normalizing the scale of the ETH and UCY datasets caused by the disparity in camera heights, only the UCY datasets were used for the validation of the models presented in this thesis. UCY was chosen over ETH because there are more trajectories and the crowds are denser. Specifically, the three Zara datasets were used. A sample sequence from the first video is shown in Figure 6.3.

The Zara sequences of the UCY dataset include both videos and annotations. For this project, only the annotations were used. The annotation files are listed as a spline for each agent where each timestep when the agent was present is an entry in spline

for that agent. The entries include the x and y position of the agent (in pixels with the center of the video being 0,0) along with the frame number and the gaze direction (not used). Before training, the coordinates were shifted and scaled to be between 0 and 1 to make it easier to compare the results to those from the simulations where the scale was also 0 to 1. The first two videos were used for training and the third was used for testing and validation. Together, the first two videos have 19527 frames. The first 2000 frames of the third video was used for validation and to perform early stopping. The remaining 5524 frames from the third video were used to evaluate the performance of each architecture.

The setup of the parameters and systems were the same as for the Intersection simulation, although this time the Social-LSTM used a scaled equivalent of a 32 pixel-wide grid since that was the recommendation of the authors. Like before, the models were trained to compute the parameters of a probability distribution. The mean log probability of all predictions on the last timestep (8) for the test data was collected and summarized in Table 6.2. When only a single Gaussian was used, the Social-LSTM model was the best, while the Hierarchical LSTM exceeded the performance of all other architectures when outputting parameters for 20 Gaussians (Mixture). Surprisingly, the best models are different from the Intersection Simulation where the Hierarchical LSTM was the best when using a single Gaussian, but the Social-LSTM architecture was the best for the mixture of Gaussians. These slightly different outcomes further confirm the idea that the Social-LSTM and Hierarchical LSTM appear to achieve nearly equivalent results.

Figure 6.4 is a visual summary of the data. One notable occurrence in the results is that the mixture of Gaussians actually hurt the performance of the two attention models even though it provided a major boost to all of the other architectures. One explanation for this behavior is that the mixture of Gaussians makes the models more prone to overfitting, and thus the generalization error suffered. For future

Table 6.2: Results for UCY Dataset

	Single	Mixture
BASE	6.01	6.41
S-LSTM	**7.98**	8.07
OCC	6.94	7.53
HIER	7.79	**8.14**
SATTN	6.99	6.93
NATTN	7.90	7.45

work, more rigorous regularization could be incorporated to test this hypothesis. The reason that more Gaussians makes the model prone to overfitting is that the Mixture Density Networks have more capacity to learn specific training examples. In the worst case, for each set of similar trajectories, the mixture versions may output parameters that specify a mode for the next position of each trajectory in the set of similar trajectories. However, this approach would not learn the true underlying motivations for how pedestrians pick their next movements and thus would perform poorly on new examples. Since all other architectures performed better using many Gaussians, it does not seem that overfitting was a major issue overall.

The pattern of performance in the UCY data is similar to the results from the Intersection scenario, which means that the simulation was a useful analogue to real-world interactions. Although never achieving the best result, the Neighbor Attention model also achieved results that are only slightly lower than the Hierarchical LSTM and Social-LSTM models (especially in the single Gaussian case). Once again, the Spatial Attention and Occupancy Grid architectures were the worst performing of the proposed models. For the single Gaussian case, the Occupancy Grid was the worst while for the mixture case, the Spatial Attention was the worst. Remarkably the Occupancy Grid achieved impressive results (7.53) when allowed to output parameters

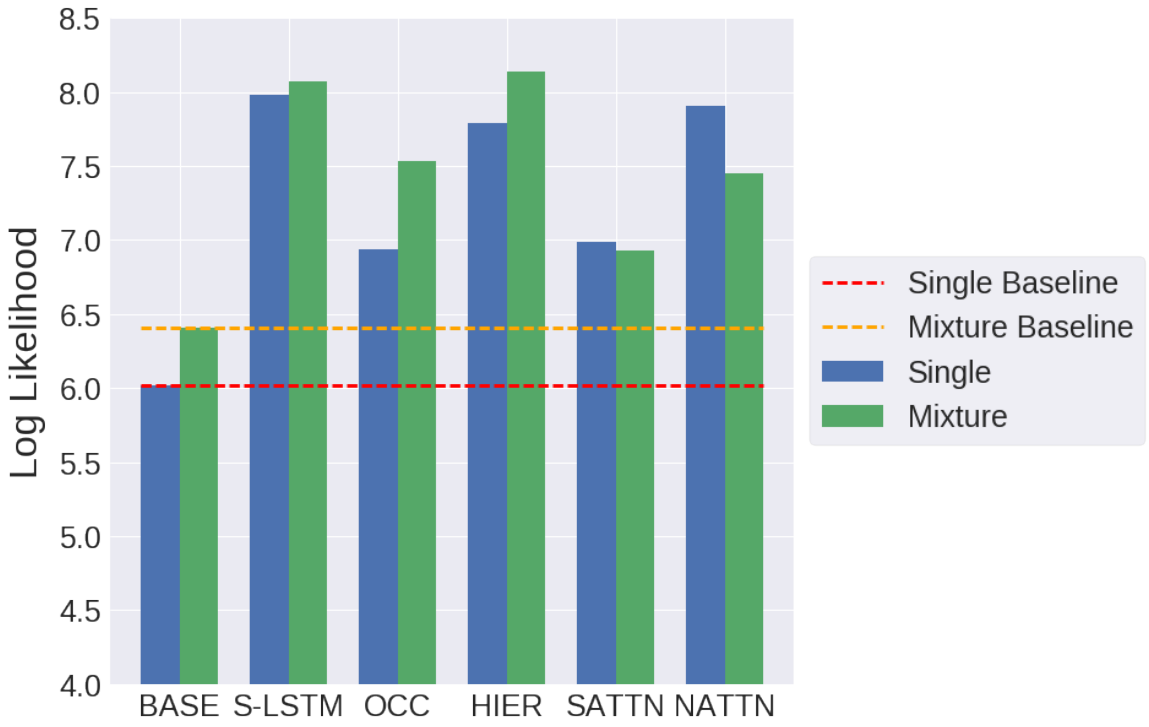


Figure 6.4: UCY Results

for a mixture distribution.

6.1.2.1 Limitations

A major limitation of this dataset is its size. Using less than 20,000 frames of video for training is meager in comparison to the hundreds of thousands of training samples that are available in other domains. Compounding the small dataset size is the fact that many frames do not even include pedestrian-pedestrian interactions since sometimes there are only one or two pedestrians in the scene (although there are still many more interactions that other datasets like ETH). The results certainly indicate that the Hierarchical LSTM and the Neighbor Attention models are comparable or nearly comparable to the well-configured Social-LSTM model, but larger datasets should be able to show the differences between them.

The scene is also fairly narrow in the geographic area that it covers. People can cross the environment in just over 15 timesteps, so the dataset does not adequately test whether the models can learn from long-range interactions. Since there are only two exists from the scene (on the left and on the right), there are few interactions between agents coming at each other from perpendicular directions. There is no noticeable mimicry behavior, and the scenes are composed solely of pedestrians who are moving at relatively the same speed. These limitations, however, are also beneficial to training because the consistent destinations (left or right) mean that goal-oriented behavior is not a major contributing factor to the movements of people since it is obvious what the end destinations will likely be. Thus the movements of pedestrians are more attributable to navigating around other people and obstacles, which better tests the models proposed in this thesis. The authors of Social-LSTM noted that the ETH datasets often had few pedestrians in the scenes, so much of the choices made by agents were related to their end-goal rather than their interactions with others. This made it difficult for the authors to demonstrate the advantage of the Social-LSTM model.

6.2 Destination Prediction

A major advantage of outputting a mixture of Gaussians is that long-range positions can be estimated despite the fact that there may be multiple likely results. The Mixture Density Network formulation of each architecture discussed in this thesis is capable of robustly estimating the destination of agents in the environment. The ability to predict destinations (or more aptly the probability distribution of the destination) is an important function for several applications. Interacting Gaussian Processes (IGP[52]) and the extension to IGP proposed by Vemula et al. [56] rely on estimates of the pedestrian destinations to predict trajectories and navigate cooper-

atively. The paper describing IGP assumes that destinations are known in order to simplify the evaluation, while Vemula et al. learn a prior distribution over discrete destinations using a Bayesian approach. Both of these methods could benefit from a system that can accurately estimate destinations. Moreover, predicting destinations could have applications in analyzing the impact of a robot on pedestrians. If a probability distribution can describe the likely destinations of other pedestrians, then the robot can seek a route that minimizes its interference with the optimal routes that the pedestrians could take.

To validate the effectiveness of Mixture Density outputs for long-range prediction, the baseline LSTM model with no neighbor influences was trained to output the destinations of pedestrians in the Intersection simulation. The baseline architecture was used because nearby pedestrians do not affect pedestrian destinations in the Intersection scenario. In a real-world situation, other architectures could easily replace the baseline LSTM. For the tests, the positions of pedestrians for four timesteps were inputted into the neural network, which then predicted the parameters for a probability distribution specifying where the pedestrian would likely leave the environment.

For comparison, a traditional Gaussian Mixture Model was fit on the training data using the Scikit-Learn implementation [40]. The traditional Gaussian Mixture Model learns Gaussians that best represent the destinations of the agents in the training set. No location or velocity information is available to the Gaussian Mixture Model. There are a number of more advanced techniques for constructing models that incorporate position and velocity in predictions. These complex models are often represented as graphs of dependencies between variables. Unfortunately, designing effective probability models of this nature is time-consuming and error-prone, so the naive Gaussian Mixture Model was retained as the baseline.

The results are summarized in Table 6.3 using varying numbers of Bivariate Gaus-

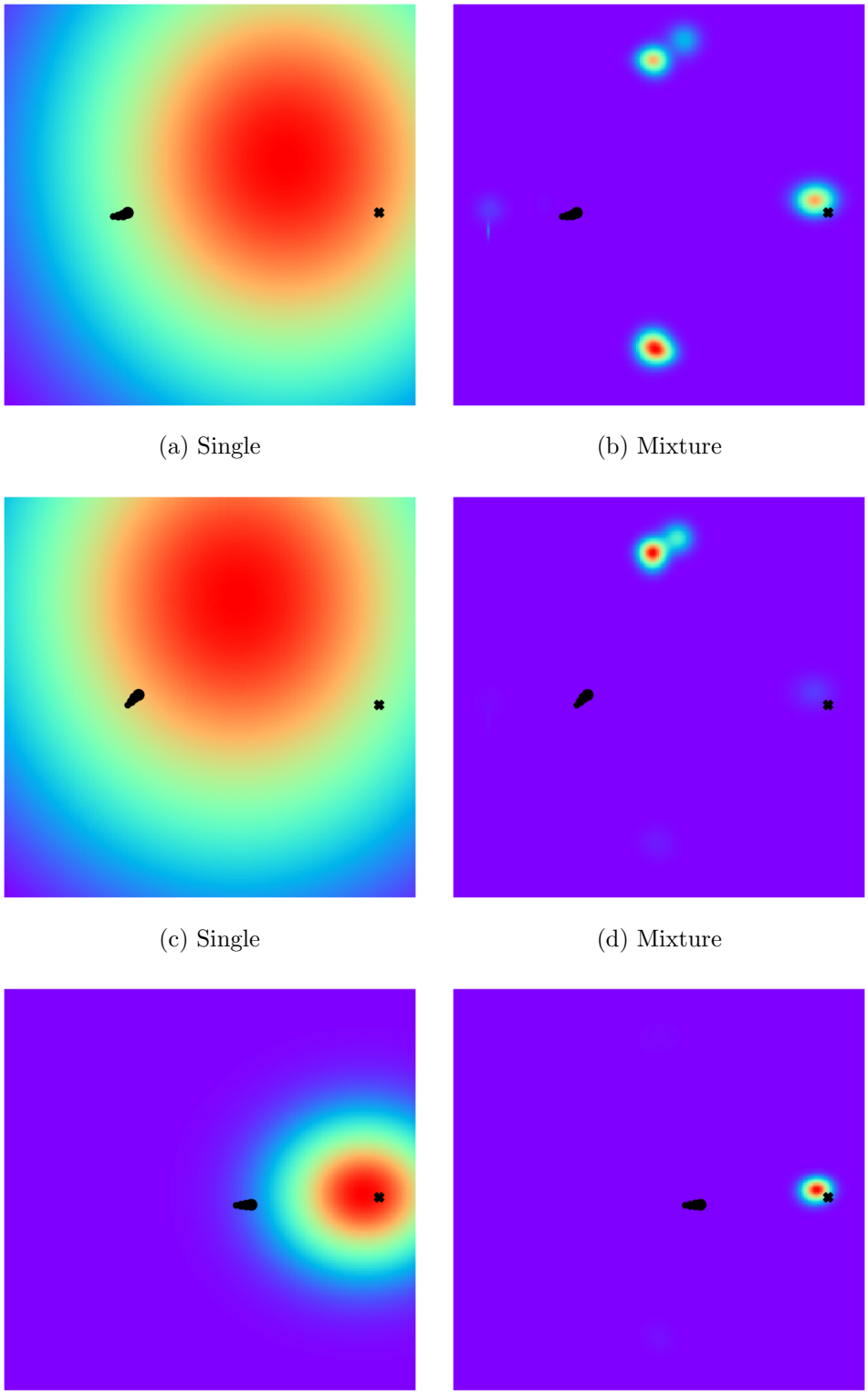
Table 6.3: Destination Prediction Results

Number of Gaussians	Mixture Model	Neural Network
1	-0.634	1.638
2	0.461	2.279
3	1.398	2.903
4	1.518	2.409
5	2.943	3.860
10	2.478	3.908
20	2.783	4.396

sians. The models were trained using the same data as the trajectory prediction task, and the models were evaluated on the held-out test data.

In all cases, the neural network outperformed the traditional Gaussian Mixture Model, which does not consider the velocity or position of the agent. Increasing the number of component Gaussians generally improved the Mixture Model and the neural network. There is no reliable way to directly compare these results to the solution proposed by Vemula et al. in their paper because they chose to discretize the possible destinations. The ability of this model to learn to predict destinations in the continuous domain makes it more accessible for new environments since there is no need for a human to manually specify what areas are considered potential destinations.

A qualitative comparison of the predictions is show in Figure 6.5. The Single label refers to the neural network trained to output the parameters of a single Bivariate Guassian, while the Mixture label refers to the neural network trained to output the parameters of 20 Bivariate Gaussians. The red regions are the most likely, followed by orange, yellow, green, blue, and finally purple, which is the least likely areas. The black dots show the positions of the agent overtime while the black cross is the



(a) Single (b) Mixture

(c) Single (d) Mixture

(e) Single (f) Mixture

Figure 6.5: Qualitative Destination Prediction Results

actual destination. It is clear that the Single Gaussian cannot adequately represent the distinct possibilities, while the multiple Gaussians align closely to the potential exits from the Intersection scenario. Parts c and d shows a partial failure case where the agent appears to be moving towards the top exit, when in reality the agent's end destination is on the right. Notice that the Mixture version still has a light blue area at the actual destination, indicating that the network predicts there is still a minute chance that the agent will end up at the right exit. The Single version cannot represent this small chance, so instead it outputs high variance. Note that each figure has an individual color-scale, so in reality the peak (red region) of the Single versions are much lower than the peaks of the Mixture version.

6.3 Planning in Dynamic Environments

Evaluating planning algorithms on dynamic environments is a difficult problem. These algorithms cannot be applied to real-world robots before they have demonstrated their safety and effectiveness in simulations. The major hurdle that prevents robust evaluation is the fact that other agents in the environment will react to the movements of the robot. These planning algorithms should be able to consider how these pedestrians will react to the robot when it follows the path. Unfortunately, there is no accurate way to simulate exactly how pedesrians will respond. Therefore, the approach taken here mimics how IGP tested its route planning solution. In scenes from the UCY dataset, one pedestrian is selected to be replaced by a robot. The planning algorithm is then used to develop a route for the robot to reach the original pedestrian's destination. People in the scene react and avoid the original pedestrian, so ideally, the planning algorithm can exploit the room made by the original pedestrian in the scene. Although this is not a perfect evaluation since the pedestrians in the scene are reacting to the original pedestrian's route and not the one taken by the robot, it still

presents a meaningful way of comparing these algorithms.

There are two criteria that are used to evaluate the performance of the algorithms. The first is the average displacement of the robot's proposed location from the actual location of the original pedestrian at each timestep. This measure assess how closely the robot follows the path taken by the pedestrian. The underlying assumption for this metric is that the pedestrians are choosing an optimal path that is the shortest safe path. Although this assumption is unlikely to be completely valid, manual inspection of the scene indicates that humans generally follow a straightforward and short path that is free of collisions. The second measure of success is the number of near collisions where the robot got too close to a pedestrian in a way that would make that person uncomfortable.

For these experiments, the Social-LSTM model was trained on the first and third Zara datasets. Routes were planned for 20 pedestrians in the second Zara dataset. Most paths were approximately 15 timesteps long. Four different planning approaches were evaluated. The A* algorithm with no obstacles (A* NO) just plans the shortest path to the destination without considering any impediments. The A* algorithm with static obstacles (A* SO) only considers the current location of pedestrians as obstacles - people are treated as static objects. The last A* approach uses the Dynamic Horizon modification (A* DH) where regions that the neural network predicts a pedestrian may enter in the next timestep are considered unnavigable. Finally, the tree search approach (TS) uses the neural network to predict multiple timesteps into the future in order to plan routes. The horizon for the Dynamic Horizon A* was chosen as a circle with radius equal to .2 where the entire scene has a width and height of 1.0. The tree search plans 3 timesteps ahead, and for both TS and A* DH, the threshold for determining whether a location is valid is a 1% probability of collision. The area of each person is a circle with radius equal to .015. A near collision is when the distance between any two agents is less than 2.5 times the radius of a pedestrian. At

Table 6.4: Results of Path Planning

	Mean Displacement	Number of Near Collisions
A* NO	0.10	13
A* SO	0.15	13
A* DH	0.12	6
TS	0.09	1

each timestep the A* algorithms are able to move to the cell that is farthest along the path as long as the distance from the cell to the start is less than or equal to 1.2 times the average velocity of an agent. For the tree search, the robot can move 0, $.6 \cdot v$, or $1.0 \cdot v$ (where v is the average velocity of an agent) and in any direction where 12 angles are evenly spaced around the unit circle. The A* algorithms discretizes the environment into a grid that is 100 by 100.

For each timestep in the scene, all planning algorithms searched for the best path and outputted the best subsequent location of the robot. These positions were recorded, and then the neural network was applied to the ground-truth values of all agents in the scene for that timestep and predictions were outputted for their next locations. Each time the neural network predicted a position, it included the coordinates of the original pedestrian that was replaced by the robot. These predictions were used at the next timestep to determine whether cells were valid or not. Each algorithm was rerun at each timestep starting with the last position that the algorithm outputted. Variations of A* like D* are optimized for situations where there is significant re-planning, but these were not considered since the speed of the algorithms was not a factor in the analysis.

The results from the 20 tests are shown in Table 6.4. Lower mean displacement means that the path followed more closely what the pedestrian actually took, which is desirable because it is assumed that pedestrians choose the shortest reasonable

path. The tree search approach achieves the smallest mean displacement and the fewest number of near collisions among all of the approaches. The A* algorithm with no obstacles (A* NO) and with only static obstacles (A* SO) depart drastically from the path taken by the pedestrian, and collides or nearly collides with someone in more than half of the tests. When the Dynamic Horizon A* is used, the robot experiences near collisions six times. Upon manual inspection of the routes, it appears that these near collisions happen because the algorithm plans a route where it gets stuck between agents and cannot get to a safe state. In the real-world people would likely move around the robot and it is unlikely that the collisions would occur, but it still demonstrates how the limited ability of the algorithm to incorporate long-term forecasts can lead to behavior that is unlike human behavior. Finally, the tree search method (TS) performs remarkably well, achieving the closest proximity to the original pedestrian's path and the fewest number of collisions.

The following figures illustrate examples of the planned paths - both successful and not successful. The translucent circles in the background are the positions of the other pedestrians over time where the smaller dots are farther back in time than the largest dots. In many cases it is difficult to determine where exactly each pedestrian was at each step of the plans, but the goal of the figures is to highlight the differences between the plans rather than the exact interactions with the pedestrians. The black lines are the actual path taken by the original agent that was replaced by the robot, and the black X marks the final destination of the pedestrian. In nearly all cases, the planned routes did not end up reaching the goal because the algorithms sometimes took steps smaller than the average velocity of the agent. This is not a major detriment to the results because proximity to the destination is still a good indication of the success of the plan.

Figure 6.6 showcases several common themes in the planned routes. The first path to notice is the brown one for the A* that did not consider any obstacles (all states

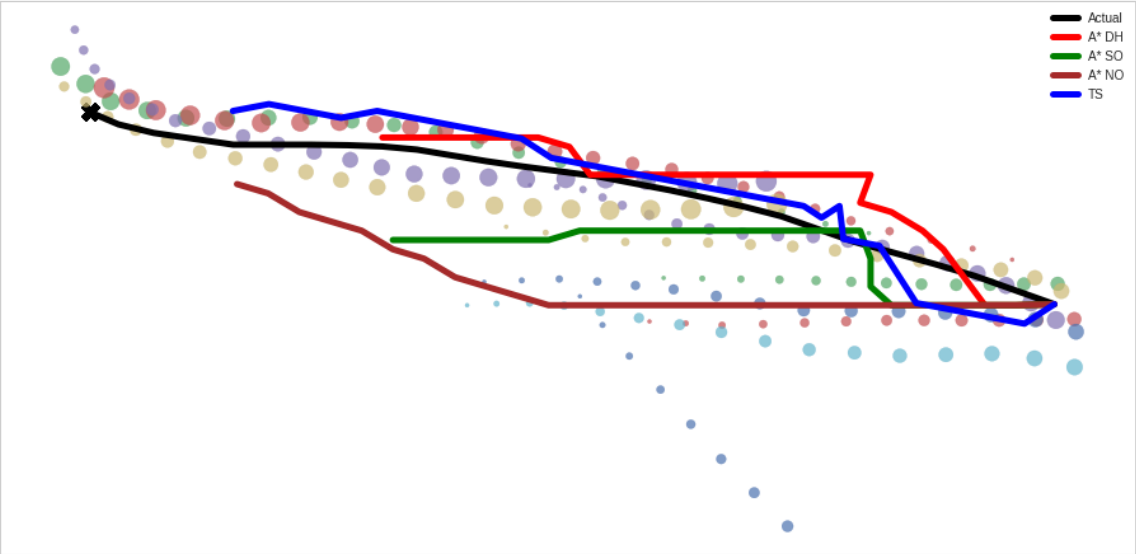


Figure 6.6: Sample Routes 1

were valid). Since the destination involves much more change in the x coordinate than in the y coordinate, the basic A* algorithm planned a route that moved directly in the x direction before moving diagonally toward the goal. This is not a natural movement for people since humans generally prefer smooth arcs; however, it would be a decent path for a robot that can often change directions much more quickly than a human. However, both the A* without obstacles and with only static obstacles chose to go directly towards the destination when that way was largely blocked by pedestrians. The Dynamic Horizon A* took a relatively jagged path, but it followed the contours of the actual path quite well. Besides a slight deviation at the outset when the tree search chose to move lower in the y direction, the tree search path very closely aligns with the actual path. In all of these paths, the planned routes have many more abrupt turns than the smooth ground-truth route. In order for a robot to plan routes that are closer to those of humans, it may be beneficial to add a criteria that the angle between subsequent positions is below some value.

Figure 6.7 also shows the preference for the A* algorithm to first move horizontally

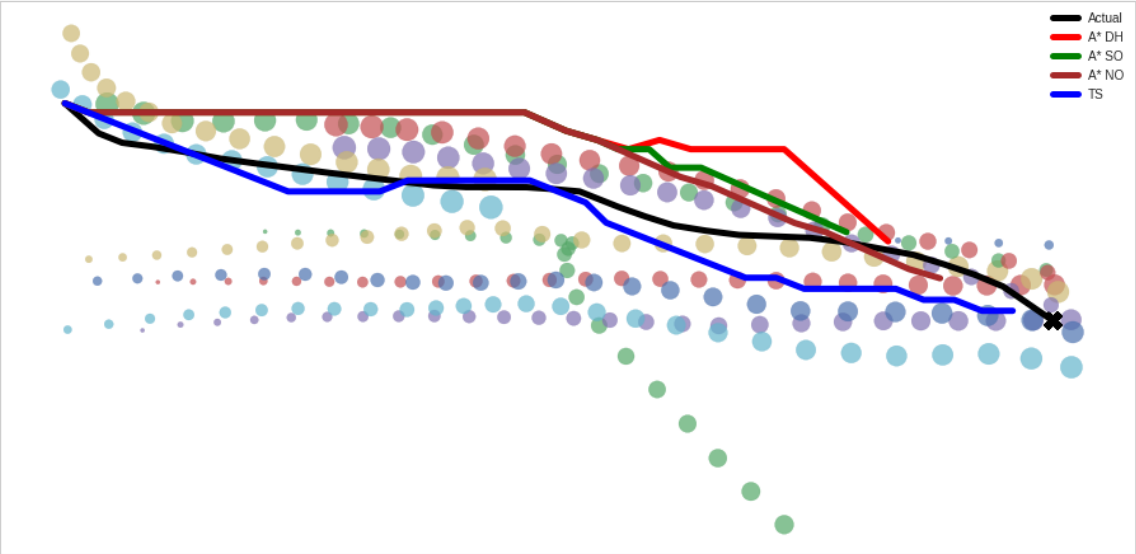


Figure 6.7: Sample Routes 2

and then move diagonally towards the goal. Using a finer granularity of cells may induce the A* algorithms to plan routes that do not involve as much flat horizontal movement. In this example, the Dynamic Horizon A* plans a route that seems to exercise excessive caution in moving away from the pedestrians when no agent actually approached it. While it is desirable to independently validate planning algorithms and neural networks for trajectory prediction, this example shows how intertwined the two are. In this case, it appears that the network outputted a high probability that one of the agents would move upwards even though none of them did. Improving the models for trajectory prediction may make both the Dynamic Horizon A* and tree search methods perform even better. The tree search plan in this scenario follows the path of the original pedestrian closely in the beginning, including going from a diagonal path to a horizontal one, but it then departs to take a route that is much lower than the actual route. By inspecting the pedestrians in the scene, it looks as though the tree search chose to go beneath (lower y coordinates) than the gold-colored agent, while the actual path went above and behind the gold agent. Although the true path and the tree search plan are not close throughout the sequence, it appears

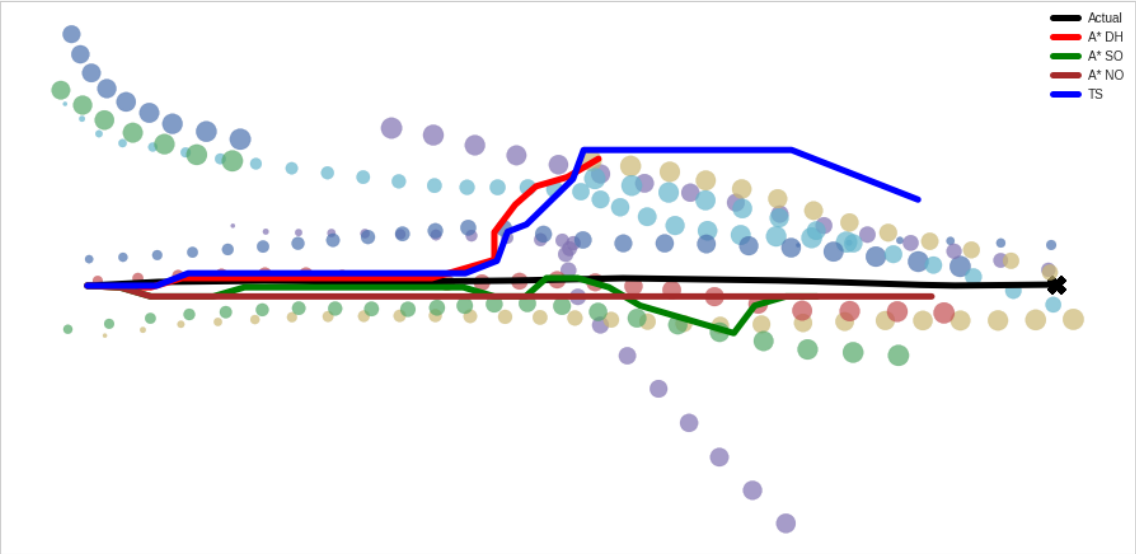


Figure 6.8: Overly Cautious Behavior

that the tree search plan is a reasonable approach.

While many of the paths appeared quite close to the actual path, there were some routes that made massive deviations from the real pedestrian's movements. Figure 6.8 shows the Dynamic Horizon A* and tree search methods move higher around the pedestrians rather than taking the more straightforward route used by the real pedestrian. This massive departure may have been caused by the large uncertainty in the neural network prediction that made the probability of the other pedestrians colliding with the robot non-negligible. Careful inspection of the pedestrian dots shows that the red pedestrian is walking very close to the actual pedestrian, so the neural network could have outputted predictions showing that the red pedestrian might be in the way of the robot in future timesteps. Nonetheless, the massive evasive action seems unnecessary and might be ameliorated by reducing the threshold for making a state invalid. In this situation, the direct routes of the A* without obstacles or with only static obstacles actually more closely resembles the real pedestrian's route.

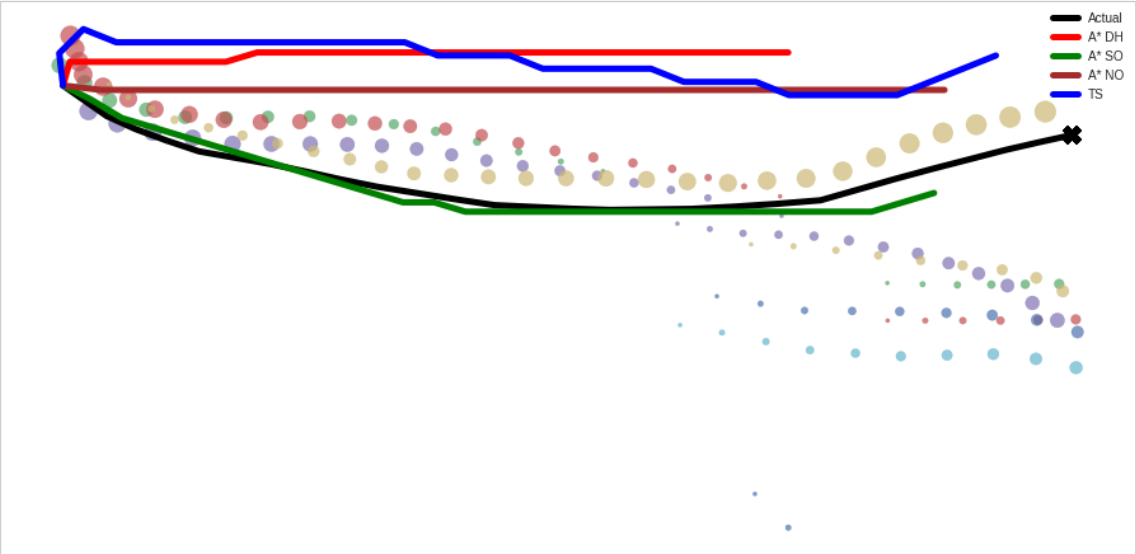


Figure 6.9: Failure to Walk Side-by-Side

Another failure case is shown in Figure 6.9. In the original scene, two pedestrians were walking together from the left side to the right side (the gold dots and the pedestrian replaced by the robot - black line). However, none of the planning algorithms take into account the propensity of pedestrians to walk side-by-side, so the A* without obstacles, Dynamic Horizon A*, and tree search choose direct paths that take a higher path towards the destination. Only the A* with static obstacles stays on the correct side of the gold agent. Another factor not shown in the diagram is that the upper routes are actually going right in front of buildings - something that humans prefer not to do in order to avoid collisions with opening doors or people exiting the buildings. If walking side-by-side with a pedestrian is desired behavior for a robot, then the planning algorithms could be replaced with a method for mirroring the movements of the pedestrian.

Chapter 7

ARGIL - CROWD SIMULATION

Argil[1] is an Agent-Based-Modeling framework with a focus on robot, crowd, and swarm simulations. Agent-Based-Modeling (ABM) is the process of simulating scenarios where one or more agents interact with each other and the environment. ABM is used in a variety of disciplines. Biologists have used ABM for modeling evolutionary behavior or ecological situations. Sociologists have used ABM when simulating the interactions individuals and groups. Generally, ABM is conducted with the goal of analyzing the composite behavior of all agents together, although the same software can be used for assessing the behavior of individual agents.

Prior to Argil, there were limited options for light-weight crowd simulations. Additionally, existing crowd simulation tools could not effectively handle custom agents and customized behavior without significant developer effort. Argil combines the benefits of dedicated crowd simulation software with the flexibility of ABM solutions.

7.1 Design

Argil was designed to be intuitive and easy-to-use, yet powerful enough for serious simulations. One of the major differentiating features of Argil is its robust support for visualization and data collection. Argil is a Python library, so all models and experiments are defined and invoked by writing Python code.

[1]The word "argil" is a type of pottery clay. It was chosen as the name of the library since it connotes creativity and flexibility - two guiding principles for the framework.

7.1.1 Model Definition

The most important component of Argil is the agent models. An *Agent* model must supply a *step* and *reset* method. The *step* method is called during each timestep of the simulation to update the agent, while the *reset* method is called prior to the start of the simulation to initialize the agent. Argil includes a built-in agent called *SocialForceAgent* that moves according the Social Forces model. While it is easy to build agents with custom behavior, the focus of this thesis is on crowd simulations, so only the *SocialForceAgent* will be used. An example of how a *SocialForceAgent* can be defined is shown in Listing 1. The first line creates the agent and defines its start location and radius. The second line adds a color property to the agent, and the third line adds a waypoint (a goal for the agent to reach).

```
1 agent1 = SocialForceAgent(x=4, y=5, radius=.3)
2 agent1.color = "blue"
3 agent1.add_waypoint((10, 5))
```

Listing 1: Definition of an Agent

Argil also supports *Objects*, which are just like *Agents*, but they do not change during the a simulation. An *Environment* is created to contain all of the agents and objects for a simulation. A sample definition of objects and an environment is shown in Listing 2.

```
1 upper_wall = Object(x=0., y=0., width=10., height=1.)
2 lower_wall = Object(x=0., y=9., width=10., height=1.)
3
4 agents = [agent1]
5 objects = [upper_wall, lower_wall]
6 env = Environment(agents, objects, width=10., height=10.)
```

Listing 2: Definition of Objects and an Environment

Once the environment is defined, simulations can be run using the environment.

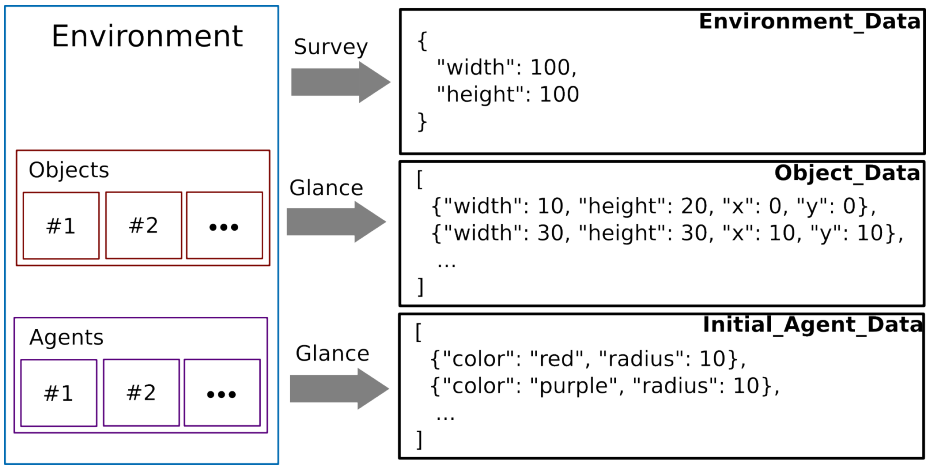


Figure 7.1: Initialization of Simulation

During a simulation, the *step* methods of each agent are called at each timestep, and the relevant state of the agents is stored. Developers have complete control over what information is recorded during the simulations. Three functions can be written by developers to access the state of the environment before and during the simulation. Figure 7.1 shows the pre-simulation initialization. The *glance* and *survey* functions are user-defined. The *glance* function is applied to each object and each agent, and it returns the time-invariant properties of each entity (properties that will be constant throughout the simulation). The *survey* function is applied to the whole environment and returns information about the environment as a whole such as the height, width, and resolution. The results of applying the *survey* and *glance* functions are stored and returned after the simulation.

During execution of the simulation, another user-defined function is invoked at each timestep on each agent. The process taken by the simulator at each timstep is shown in Figure 7.2. The simulation will store the results returned by the invocations of *observe* and return it at the end of the simulation. A sample of how these *glance*, *survey*, and *observe* are defined is shown in Listing 3. Once the simulation is completed, the record can be used to produce visualizations or datasets.

```
def glance(entity):
        if isinstance(entity, SocialForceAgent):
                return {"_shape": "circle",
                                "radius": entity.radius,
                                "color": entity.color}
        else:
                return {"_shape": "rectangle",
                                "x": entity.x,
                                "y": entity.y,
                                "width": entity.width,
                                "height": entity.height,
                                "color": "black"}

def observe(agent):
    return {"x": agent.x, "y": agent.y}

def survey(env):
        return {"width": env.width, "height": env.height}

sim = RecordSimulation(glance, observe, survey, num_steps=100)
record = sim.run(env)
```

Listing 3: Simulating an Environment

7.1.2 Visualization and Data Output

The records produced by running a *Simulation* are not readily analyzed outside of Argil since the type is custom-built for Argil. However, to assist with analyzing *Simulation* results, Argil supports a variety of ways to visualize and inspect the data collected from a *Simulation*. Currently, there are three *Producers* that operate on the *Records* - *D3Producer*, *MatplotlibProducer*, and *PandasProducer*. Both the *D3Producer* and *MatplotlibProducer* were designed to operate in the context of a Jupyter Notebook[27]. Jupyter Notebooks are a way of writing and executing code (most often Python code) within a browser and displaying the output of the code directly after each block of code. Figure 7.3 is a screenshot showing an Argil simula-

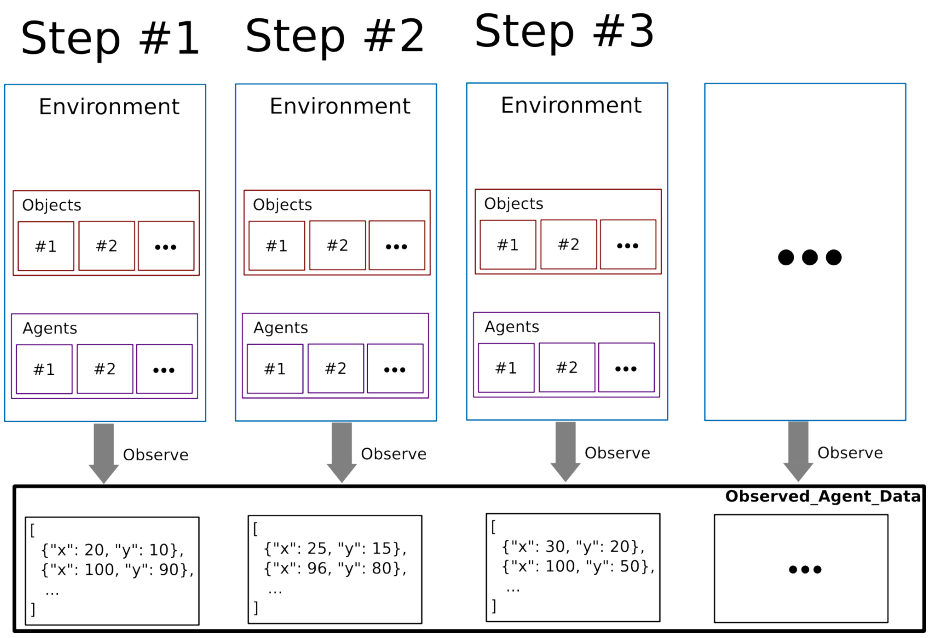


Figure 7.2: Observations while running Simulation

tion embedded in a Jupyter Notebook. Each *Producer* relies on certain fields existing in the records (except for the *PandasProducer* which is agnostic to the field names). Generally, the visualization *Producers* require that the x and y positions of the agents be specified at each time step and that the size and shape of each entity is also defined.

The *D3Producer* uses the Javascript D3 visualization library[7]. D3 is a powerful framework for binding data to elements in the DOM (Document Object Model). DOM elements can be any HTML tag in a webpage. For the *D3Producer*, the simulation results are visualized inside a single SVG (Scalable Vector Graphic) embedded in the webpage. During the loading of the page, the inanimate objects are rendered in the SVG. Then the animation capabilities of D3 are used to dynamically update the agent elements in the SVG at a user-specified rate. The animation loops back to beginning after completing the entirety of the sequence. The animations are produced by serializing the data contained in a *Record* (that was retrieved from *Simulation*) into JSON and injecting the JSON into an HTML/Javascript template. The resulting HTML/Javascript contains all of the markup for describing the visualization as well

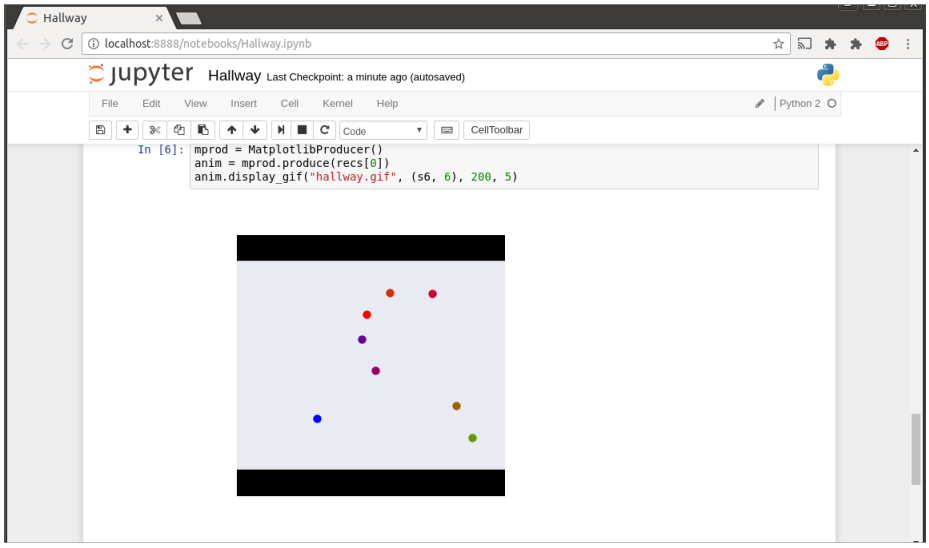


Figure 7.3: Argil Visualization in Jupyter Notebook

as the logic for animating it. Since Jupyter Notebooks are browser-based, the web animations produced by the *D3Producer* can be added into the Notebook.

The *MatplotlibProducer* uses the popular Matplotlib visualization library [24] to construct animations, videos, and images of the simulation. Single-step graphics can be created by rendering the state of all agents and objects in a Matplotlib figure. Animated GIFs and HTML5 videos can be produced by updating the components of a figure using Matplotlib's animation utilities. It is possible to save the graphics, GIFs, and HTML5 videos as files, but they can also be visualized directly in a Jupyter Notebook. The *MatplotlibProducer* is the preferred visualization solution since it is easier to customize, and there are many options for outputting the results. One downside of the *MatplotlibProducer* is that generating videos or animated GIFs can take more several minutes for simulations that are more than 500 steps.

Finally, the *PandasProducer* converts the *Records* into Pandas dataframes. Pandas is a Python library for storing and manipulating data in a structured way [35]. Pandas is a common tool used in Data Science, and it offers the flexibility of storing

heterogeneous data in a single dataframe (strings, integers, floats, booleans, categorical values, etc. are all valid types in a dataframe). Additionally, Pandas provides methods for saving dataframes as CSVs (Comma Separated Values), Excel, SQL, and JSON among several others.

7.2 Comparison

Argil is a novel combination of unstructured Agent-Based-Modeling and crowd simulation. Table 7.1 shows a non-exhaustive list of some of the more popular crowd simulation and Agent-Based-Modeling frameworks. Generally, the crowd simulation solutions define simulations using a markup language like XML, and there is generally limited support for ad-hoc agent behavior. However, many of the crowd simulation packages are written in C++ and offer spectacular performance even when simulations incorporate thousands of agents. Several crowd simulation libraries support multiple types of agents (cars, bikes, carts, etc.) and can use features from real-world maps. The Agent-Based-Modeling frameworks are generally less performant than the crowd simulators, but models are usually written in a programming language and there is tremendous flexibility to customize the behavior of individual agents.

Due to its use of Python, Argil cannot match the speed of dedicated crowd simulators. However, it offers users the unique ability to construct models in a scripting language (Python) and exercise complete control over the parameters and behavior of each agent. Of special importance to many academic projects, it is trivial to construct multiple types of agents (robots, pedestrians, animals, etc.) in Argil while this is an arduous process in crowd simulators. None of the other Agent-Based-Modeling solutions have built-in support for crowd simulation. Several Agent-Based-Modeling libraries use custom scripting solutions that are not as well known as the Python language. Most Agent-Based-Modeling libraries offer custom-built visualizations, while

Argil leverages open-source tools for building visualizations that are easily customized by users who are familiar with Matpoltlib or D3.

Mesa (an ABM library written in Python) is the closest existing solution to Argil and served as a major source of inspiration. Much of the naming conventions and overall software structure are attributable to Mesa (although some of the naming was chosen to correspond to the conventions of reinforcement learning and OpenAI Gym [2]). However, Argil differs from Mesa in several key areas. Argil focuses on crowd simulation, while Mesa is designed for general models (especially ones with discrete positions). In Mesa, collecting data is a separate process and API from visualization, whereas Argil has a unified API for all types of data collection and visualization. Mesa produces visualizations using a mostly custom Javascript system with a Python server, while Argil uses the popular libraries Matplotlib and D3, making Argil quicker and easier to deploy.

7.3 Performance

Since Argil is written in pure Python, it will never achieve the speed that is possible with frameworks like Menge, which are written in C++. However, Argil is currently fast enough to quickly simulate small scenarios (less than 50 agents), and there are several ways to accelerate it. Generally, ABM simulations are run several times in order to generate sufficient data for analysis. Oftentimes, researchers will change several parameters and rerun simulations to assess the effect of each parameter setting. Currently, Argil supports running simulations in parallel using multi-processing in Python. Simultaneous execution of simulations can be useful for many configurations of a small scenarios, but it will not accelerate the simulation of large-scale scenarios where within-simulation parallelism is desirable.

[2] https://gym.openai.com/

Table 7.1: Comparison of Simulators

Software	Language	Type	Applications	Visualization
Menge	C++	Crowds	academic	Custom
Massis	Java	Crowds	academic	Custom
SUMO	XML, Python	Crowds	traffic	Custom
PedSim	C/C++	Crowds	academic	Custom
Massive	N/A	Crowds	films/games	3dsMax, Maya
Miramy	Maya	Crowds	films/games	Maya
Golaem	Maya	Crowds	films/games	Maya
Agentbase	CoffeeScript	ABM	academic	Custom
Repast	Java, ReLogo	ABM	general	Custom
NetLogo	NetLogo	ABM	general	Custom
Swarm	Obj-C, Java	ABM	general	Custom
Mesa	Python	ABM	general	Custom
Argil	Python	Both	academic/general	Matplotlib, D3

Two experiments were conducted on Argil to measure its performance. A laptop with a 3.0Ghz i7 processor and 8GB of RAM was used. Figure 7.4 shows the time required for Argil to execute 20 runs of a simple pedestrian scenario with varying numbers of agents. Figure 7.5 shows the time required to run 100 simulations of a 10 pedestrian scenario using varying numbers of processes. For the simulations produced in this thesis, Argil was able to generate sufficient data for training in less than 5 minutes.

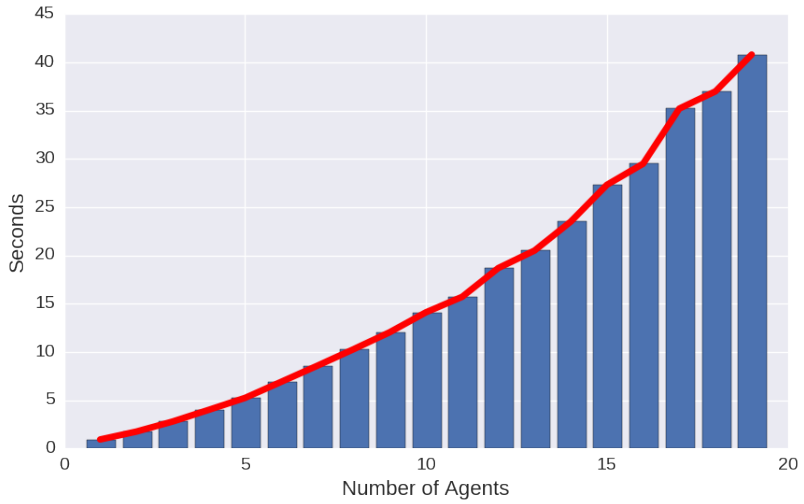


Figure 7.4: Effect of Number of Agents on Performance

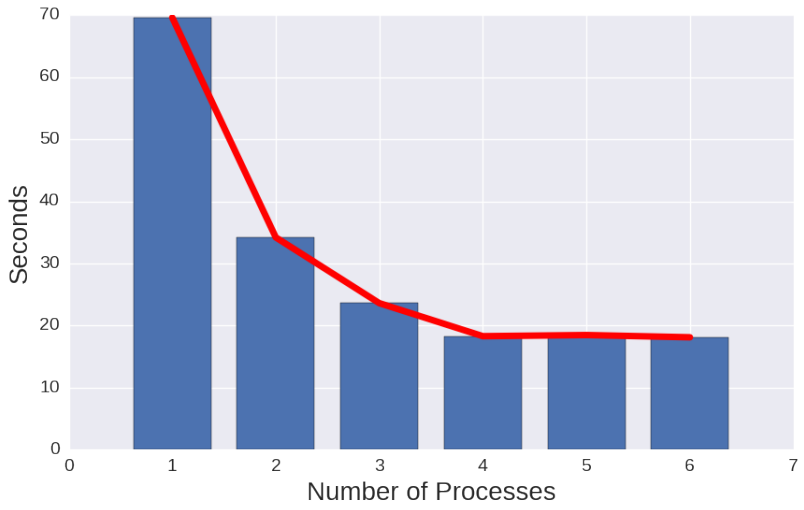


Figure 7.5: Effect of Multi-Processing on Performance

Chapter 8

FUTURE WORK

8.1 Trajectory Prediction

In this thesis, like other works on trajectory prediction, it is assumed that the exact coordinates of all agents are none with certainty at each timestep. However, this assumption is almost never satisfied in real-world situations. Sensors are not perfect, so robots typically use probabilistic representations of other objects or agents. Robots can generally be less certain about the position and velocity of objects that are far away since the sensor readings loose accuracy with increasing distance. Kalman Filters and Particle Filters are commonly used in autonomous robot systems since they can increase prediction certainty after repeated readings from sensors and a model of the motion for the entity that is tracked. Besides uncertainty about the positions of other agents, robotic systems may intermittently fail to detect a pedestrian when they are occluded by objects or other pedestrians. A useful extension to this work would be to consider real-world sensor readings as inputs to a trajectory prediction system. The Occupancy Grid architecture comes close to handling uncertain measurements by allowing coordinates to be specified as a probability distribution, but the Occupancy Grid has the severe limitation that it has no way of representing the velocities and previous positions of neighbors. A better solution is needed so that uncertain measurements can be used in a way that still includes information about the trajectory of the agent. One possibility for this is to include velocity measurements along with coordinates in any of the architectures. These velocities could also be represented by a probability distribution. There is considerable prior art on neural networks outputting uncertainty in their predictions, but including uncertain inputs

into a neural network is still an active area of research.

As mentioned previously, an improved trajectory prediction network would include information about the environment. Obviously, pedestrians typically prefer to walk on sidewalks instead of across bushes. One way to include this information would be create a dense semantic map and provide it as an input to the network at each timestep. The semantic map could be a n x m x k grid where the width of the environment is n, the height is m, and there are k different types of terrain. Each cell in the three dimensional map would be 1 if that location belonged to that certain type of terrain. This map would closely resemble an image with many channels (where typical images have three channels - red, green, and blue). There are many techniques for neural networks to learn from images, so these methods (especially convolutional layers) could be used to incorporate the relevant components of the map into the prediction of the next location of an agent.

In 2014, Goodefllow et al. introduced Generative Adversarial neural networks (GANs) [16]. GANs are actually two neural networks - one network (generator) takes random noise as input and outputs a sample that is intended to be similar to an observation from some dataset. The second network (discriminator) tries to predict whether a given sample was produced by the generator or came from the real dataset. A GAN could be constructed by training a generator to produce plausible trajectories while the discriminator predicts whether the trajectory was produced by real-humans from a dataset of trajectories or was made by the generator. In the basic form, the generators in a GAN are provided a noise vector, but there has been some work on conditioning generators to produce outputs that conform to specific criteria. For navigation, the generator could be conditioned with the destination of the robot. The networks are trained together. The loss for the generator is the ability of the generator to distinguish its outputs from the real trajectories. The loss for the discriminator is its misclassification of trajectories. At convergence, the discriminator

should output an approximately 50% chance that each input is from the real dataset or was generated. The resulting trajectories produced by the generator would have the desirable property that they cannot be distinguished from human trajectories. GANs have shown promising results in a variety of areas, but they remain challenging to implement and train.

8.2 Planning in Dynamic Environments

An end-to-end neural network for planning paths in dynamic environments would be a major achievement. Such a neural network could first be trained to predict the trajectories of pedestrians (as was done for Social-LSTM and the models presented in this thesis). However, the next step could be training through reinforcement learning to output a path that optimized certain criteria (shortest route to the goal, collision avoidance, etc.). A solution similar to this proposal was used in AlphaGo - the first computer program to beat grandmasters at the strategic game of Go [46]. The researchers first trained a neural network to predict the moves of grandmasters, then they fine-tuned the network through repeated games of self-play. Another potential method to achieve the same result would be to build a realistic simulator of pedestrian movements that can create new scenarios and update the positions of the agents appropriately through time. Such a simulator could be created using a network architecture similar to those presented in this thesis but transformed into a Variational Autoencoder. By sampling from the distribution of the autoencoder, new scenarios could be constructed. Then another neural network could be trained to output the benefits of state, action pairs where the state is the sequence of previous positions of the robot and the action is the movement of the robot. This is called Q-learning, and recently neural networks were shown to be highly adept at learning these functions that map states and actions to their value (a notable example is DQN, which learned

to play Atari games[38]). The network would be trained against the simulated scenes to produce high values for state, action pairs that maximize a reward. This reward could be inversely proportional to the length of the route or a penalty for collisions or other criteria that define a good path.

One major roadblock to the implementation of these planning algorithms for real-world robots is the uncertainty in how humans may react to robots. Much of this work assumed that people will treat robots much the same way that they treat humans, but this is unlikely to be an accurate assumption. Obviously, experiments with a physical robot in uncontrolled pedestrian environments would yield data about how humans tend to react. The most significant experiments in this area were performed by Trauntman et al.[53]; however, there is much more work to be done to validate these planning approaches in real situations. One especially important problem is how a robot should react to an adversarial agent. Humans may be interested in testing the robot by blocking its path or intentionally hitting it or colliding with it. Reacting in a controlled and predictable way in such situations is critical for humans to develop trust in these autonomous mobile robots. Future research should consider how robots can respond to unusual behavior. Real-world experiments are fraught with ethical and logistical challenges, but these experiments are essential to widespread deployment of robots that can interact with humans in pedestrian environments.

8.3 Crowd Simulation and Modeling

The major limitation of Argil is its performance compared to other crowd simulators. Native Python code is too slow for large-scale scenarios, but fortunately there are a number of ways to improve performance while still maintaining the convenient Python API. One option is to use Cython[5] for the performance critical code. Cython is a way for Python to call native C code. Rewriting some of the equations that calculate

the Social Forces in C could produce significant speedups. Alternatively, it may be possible to vectorize using Numpy[55], which is a vector/matrix library that uses C and Fortran for mathematical operations. The most promising solution to increase the performance of Argil is to utilize a framework like Theano[51] or Tensorflow[1] for the simulations. Theano is a matrix/vector library like Tensorflow that is used for scientific applications and neural networks. Both Theano and Tensorflow offer transparent use of GPUs and highly optimized vector operations. Argil could support an API for constructing models in idiomatic Python and then convert the Python expressions into equivalent Tensorflow or Theano graphs. These graphs could then be executed in parallel on CPUs or GPUs. All of the optimizations present in Theano or Tensorflow could be leveraged for massive speedups in Argil simulations. Additionally, Argil could stay focused on modeling and visualizations instead of adding the complexity of maintaining native code for accelerating simulations. The use of Tensorflow or Theano would require careful planning and the implementation may be difficult, but the performance benefit would likely be substantial.

Argil would also benefit from the ability to perform live simulations. The original design of Argil was to run a simulation, record the relevant information and then use that information to visualize the simulation. Unfortunately, this original approach requires that the simulation run to completion before it can be analyzed, which is a major impediment to rapid iteration on complex models. By showing each frame of the simulation at each timestep while the next frame is being calculated, researchers can quickly diagnose issues. In addition, less memory is required because the entirety of the simulation's states are not stored in memory since they can be released after the next frame is ready. There is already an alpha feature in Argil that uses pygame[1] to show a simulation while it is being run, but the code is rough and prone to issues. Pygame is a Python library primarily designed for building games. The major ad-

[1] https://www.pygame.org/

vantage of pygame is that it has excellent support for multiple platforms including Windows, Mac, Linux, and even some mobile operating systems. Further work will be needed to make the pygame simulation robust enough for live simulations.

Chapter 9

CONCLUSION

Predicting the future positions of pedestrians in crowded scenes and navigating in these dynamic scenes are difficult problems. Pedestrian movements are stochastic, and even humans sometimes struggle to estimate where other pedestrians are going. Since so many factors can influence the routes that pedestrians take, it is unlikely that a rule-based system could effectively predict trajectories in the myriad possible scenarios that a robot could encounter. Therefore, using machine-learning techniques has the advantage that no rules are necessary; however, these machine-learning methods are generally convoluted and computationally expensive. This project focused on neural network solutions to trajectory prediction because neural networks have achieved exceptional results in a variety of fields, and they can learn complex nonlinear relationships (and trajectories are typically complex and highly nonlinear). The inspiration for this thesis was a current state-of-the-art algorithm Social-LSTM that uses recurrent neural networks to estimate the most likely subsequent positions of pedestrians. This algorithm was highly successful because it learns how the neighboring pedestrians impact the movement of the current pedestrian using a grid that contains latent representations of nearby pedestrians. For this thesis, several extensions and modifications to the Social-LSTM algorithm were designed, implemented, and tested. Two methods for incorporating these trajectory predictions into a full navigation system were explored as well. Additionally, a new Python library for simulating pedestrian interactions was developed.

The first contribution of this research was the use of Mixture Density Networks in pedestrian trajectory prediction. The original Social-LSTM paper predicted the subsequent position of pedestrians using a single Bivariate Gaussian. A Bivariate

Gaussian is unimodal and cannot represent the multiple likely paths that a pedestrian could take. The solution that used the neural networks to output the parameters for a mixture of many Bivariate Gaussians significantly improved the accuracy]in both simulations and the real-world UCY dataset. In addition to improving the ability of the neural network to predict the next position of pedestrians, the Mixture Density Network outputs enable the same neural networks to predict pedestrian locations much farther into the future. The long-range prediction capabilities of the neural networks using mixtures of Gaussians was evaluated on a simulated Intersection scenario. Overall, Mixture Density outputs are a promising area for human trajectory prediction, and future models should incorporate them.

The next contribution was the development of four novel architectures for representing the influence of neighbors on the trajectory of pedestrians. The first architecture was an Occupancy Grid that is unique among pedestrian prediction models in that it allows for the explicit inclusion of uncertainty about the current location of neighboring pedestrians. The sensors on robots are never perfectly accurate, so the ability to input the confidence in the measurement of the location of an agent will be of practical utility to real-world robots. The second architecture was the Hierarchical LSTM, which mimics the way humans scan a scene and remember the relevant components. The Hierarchical LSTM does not limit the inclusion of neighbors based on a grid or pool the neighbors. This makes the Hierarchical LSTM capable of detecting long-range influences that might not be possible for the original Social-LSTM model. The major issue with the Hierarchical LSTM is that it is difficult to train and when the number of neighboring agents is large, its memory usage spikes and inference can take a long time. These issues may be irrelevant in robotics applications because the number of agents seen by a robot is limited by the capacity of its sensors; however, if the number of agents is still too large, then heuristics could be used to prune agents that are known to be irrelevant and only input potentially relevant neighbors into

the Hierarchical LSTM. The next architecture is the Spatial Attention model that learns to weight the importance of neighbors based on their displacement from the agent whose next position is being predicted. The Neighbor Attention model uses the coordinates of the neighboring pedestrians in relation to the current agent to decide how to weight the importance of each other pedestrian on the trajectory of the current agent. Through evaluation on a simulated dataset and real-world data from the UCY dataset, the Hierarchical LSTM was the most promising architecture. In most tests, the Hierarchical LSTM was comparable with the state-of-the-art Social-LSTM model, but the Hierarchical LSTM does not require careful hyper-parameter choices for the selection of a neighborhood region. The removal of the hyper-parameters could potentially accelerate the rate at which the Hierarchical LSTM model could be deployed on a robot platform since less fine-tuning is required. The Occupancy Grid performed better than the baseline that did not account for neighbors, but it was not as successful as the other models. The Spatial Attention model had disappointing performance (barely outperforming the baseline), but future work may be able to improve it. The Neighbor Attention architecture was nearly as performant as the Hierarchical LSTM and Social-LSTM, but it uses fewer parameters than the other architectures and is a promising alternative.

Dynamic Horizon A* is a modification to the traditional A* algorithm that treats certain locations as invalid when the probability of another pedestrian occupying that location in the next timestep is above a threshold. These locations were only considered impassable if the location was within a set radius from the current location of the robot. The horizon keeps the robot from planning around agents that are far away from the robot because these agents will probably move before the robot approaches their current position. Tree search was also applied to finding routes by searching for actions that get the robot closest to its destination without causing the robot to collide with other pedestrians. Tree search planned several timesteps into the future

and incorporated predictions of the other pedestrians into the future. Dynamic Horizon A* is mostly compatible with other planning systems that rely on A*, and it requires only one inference of the neural network per planning iteration. Tree search requires many inferences on the neural network for each planning iteration, but it offers the flexibility of a customized reward/cost functions and the ability to incorporate long-term movements of other pedestrians in its planning. The planning approaches were tested on the UCY dataset where a pedestrian in the scene was replaced by a simulated robot that then planned a route to the original pedestrian's destination. The tree search most closely mimicked the behavior of the pedestrian and also had the fewest near collisions. The Dynamic Horizon A* method also outperformed the baselines that only included static obstacles or no obstacles at all.

Finally, Argil, an Agent-Based-Modeling (ABM) and crowd simulation, was designed and built to help others run crowd simulations. Argil offers a number of advantages over existing ABM and crowd simulation tools. Argil has built-in support for pedestrian models (which no other general ABM framework currently supports). Argil offers greater flexibility than existing crowd simulation utilities, which typically require models be written in customized markup languages or using proprietary software. There are two visualization options in Argil that together allow for simulations to be shared as GIFs, videos, images, and web animations. The interface and structure of Argil is intentionally kept simple; Argil has a unified, cohesive API for the extraction of data and the visualization of results. The performance of Argil simulations is currently adequate for many research projects with fewer that 50 agents, but larger simulations are time-consuming. The speed of Argil simulations certainly detracts somewhat from Argil's utility to some potential users, but there are still many applications that can use Argil without any noticeable performance issues.

9.1 Recommendations

For trajectory prediction, Mixture Density Networks have rigorously demonstrated their value over a single Bivariate Gaussian. Mixture Density outputs do not increase the complexity of networks significantly, sampling from them is trivial, and they show significant accuracy improvements. They should likely be incorporated into future trajectory prediction systems. The four novel architectures that were proposed had mixed results. A well-tuned Social-LSTM model with sum-pooling in a grid is still likely to offer the best performance and accuracy. Yet, in novel situations where it is not known how far away pedestrians may influence another pedestrian, the Hierarchical LSTM model may be an excellent candidate. The Occupancy Grid architecture would be a reasonable choice for applications where there is significant uncertainty in the actual location of neighboring pedestrians. The Spatial Attention model requires further refinement before using in real-world application, but the Neighbor Attention architecture could be useful when the system is memory-constrained.

Tree search is a more promising approach to planning than the Dynamic Horizon A* both because of its ability to find shorter and collision-free paths as well as its flexibility to incorporate a custom cost/reward function. There are a number of ways for the tree search or Dynamic Horizon A* methods to be fine-tuned for a specific scenario including modifying the horizon parameter, the threshold for determining whether a state is valid, the granularity of the grid in A*, and the number of steps executed by the tree search. Future navigation systems for navigation in dynamic environments should investigate the advantages of tree search and the general idea of planning using the estimated positions of agents multiple timesteps into the future.

Argil is an useful tool for small-scale experimentation and for visualizing simulations. For small-scale robotics research projects, Argil will be more than adequate for simulating dynamic environments. Large-scale crowd simulations with hundreds of

agents are best performed with dedicated simulation software like SUMO or Menge. Argil is also a great tool for reinforcement learning and educational projects where students can program (or train) one agent and let the agent interact with other agents.

BIBLIOGRAPHY

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning.

[2] A. Alahi, K. Goel, V. Ramanathan, A. Robicquet, L. Fei-Fei, and S. Savarese. Social lstm: Human trajectory prediction in crowded spaces. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 961–971, 2016.

[3] D. Amodei, R. Anubhai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, J. Chen, M. Chrzanowski, A. Coates, G. Diamos, et al. Deep speech 2: End-to-end speech recognition in english and mandarin. *arXiv preprint arXiv:1512.02595*, 2015.

[4] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[5] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. Seljebotn, and K. Smith. Cython: The best of both worlds. *Computing in Science Engineering*, 13(2):31 –39, 2011.

[6] C. M. Bishop. Mixture density networks. 1994.

[7] M. Bostock, V. Ogievetsky, and J. Heer. D3 data-driven documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2301–2309, Dec. 2011.

[8] T. Carlson and J. del R. Millan. Brain-controlled wheelchairs: A robotic architecture. *IEEE Robotics Automation Magazine*, 20:65–73, 2013.

[9] Y. F. Chen, M. Liu, M. Everett, and J. P. How. Decentralized non-communicating multiagent collision avoidance with deep reinforcement learning. *CoRR*, abs/1609.07845, 2016.

[10] S. Curtis, A. Best, and D. Manocha. Menge: A modular framework for simulating crowd movement. *Collective Dynamics*, 1(0):1–40, 2016.

[11] Y. Du, W. Wang, and L. Wang. Hierarchical recurrent neural network for skeleton based action recognition. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1110–1118, June 2015.

[12] T. Fernando, S. Denman, S. Sridharan, and C. Fookes. Soft + Hardwired Attention: An LSTM Framework for Human Trajectory Prediction and Abnormal Event Detection. *ArXiv e-prints*, Feb. 2017.

[13] T. W. Fong, C. Kunz, L. Hiatt, and M. Bugajska. The human-robot interaction operating system. In *2006 Human-Robot Interaction Conference*. ACM, March 2006.

[14] Y. Gal and Z. Ghahramani. Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning. *ArXiv e-prints*, June 2015.

[15] N. S. Geoffrey Hinton and K. Swersky. Lecture 6a overview of minibatch gradient descent. `http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`. Accessed: 2017-09-19.

[16] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative Adversarial Networks. *ArXiv e-prints*, June 2014.

[17] A. Graves. Generating sequences with recurrent neural networks. *CoRR*, abs/1308.0850, 2013.

[18] A. Graves. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*, 2013.

[19] K. Gregor, I. Danihelka, A. Graves, D. J. Rezende, and D. Wierstra. Draw: A recurrent neural network for image generation. *arXiv preprint arXiv:1502.04623*, 2015.

[20] D. Helbing. Traffic and related self-driven many-particle systems. *Reviews of modern physics*, 73(4):1067, 2001.

[21] D. Helbing, I. Farkas, and T. Vicsek. Simulating dynamical features of escape panic. *Nature*, 407(6803):487–490, 2000.

[22] D. Helbing and P. Molnar. Social force model for pedestrian dynamics. *Physical review E*, 51(5):4282, 1995.

[23] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[24] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

[25] M. S. Ibrahim, S. Muralidharan, Z. Deng, A. Vahdat, and G. Mori. Hierarchical deep temporal models for group activity recognition. *CoRR*, abs/1607.02643, 2016.

[26] T. Kimoto, K. Asakawa, M. Yoda, and M. Takeoka. Stock market prediction system with modular neural networks. In *Neural Networks, 1990., 1990 IJCNN International Joint Conference on*, pages 1–6. IEEE, 1990.

[27] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay, et al. Jupyter notebooks—a publishing format for reproducible computational workflows. In *Positioning*

and Power in Academic Publishing: Players, Agents and Agendas: Proceedings of the 20th International Conference on Electronic Publishing, page 87. IOS Press, 2016.

[28] H. Kretzschmar, M. Spies, C. Sprunk, and W. Burgard. Socially compliant mobile robot navigation via inverse reinforcement learning. *The International Journal of Robotics Research*, 35(11):1289–1307, 2016.

[29] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[30] T. Kruse, A. K. Pandey, R. Alami, and A. Kirsch. Human-aware robot navigation: A survey. *Robotics and Autonomous Systems*, 61(12):1726 – 1743, 2013.

[31] R. J. Kuo, C. Chen, and Y. Hwang. An intelligent stock trading decision support system through integration of genetic algorithm based fuzzy neural network and artificial neural network. *Fuzzy sets and systems*, 118(1):21–45, 2001.

[32] A. Lerner, Y. Chrysanthou, and D. Lischinski. Crowds by example. In *Computer Graphics Forum*, volume 26, pages 655–664. Wiley Online Library, 2007.

[33] J. Li, M. Luong, and D. Jurafsky. A hierarchical neural autoencoder for paragraphs and documents. *CoRR*, abs/1506.01057, 2015.

[34] B. Majecka. Statistical models of pedestrian behaviour in the Forum. Master's thesis, University of Edinburgh, 2009.

[35] W. McKinney. Data structures for statistical computing in python. In S. van der Walt and J. Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 51 – 56, 2010.

[36] R. Mehran, A. Oyama, and M. Shah. Abnormal crowd behavior detection using social force model. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 935–942. IEEE, 2009.

[37] D. Mehta, G. Ferrer, and E. Olson. Autonomous navigation in dynamic social environments using multi-policy decision making. In *Intelligent Robots and Systems (IROS), 2016 IEEE/RSJ International Conference on*, pages 1190–1197. IEEE, 2016.

[38] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[39] B. Okal and K. O. Arras. Learning socially normative robot navigation behaviors with bayesian inverse reinforcement learning. In *Robotics and Automation (ICRA), 2016 IEEE International Conference on*, pages 2889–2895. IEEE, 2016.

[40] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[41] S. Pellegrini, A. Ess, K. Schindler, and L. Van Gool. You'll never walk alone: Modeling social behavior for multi-target tracking. In *Computer Vision, 2009 IEEE 12th International Conference on*, pages 261–268. IEEE, 2009.

[42] Z. Peng, R. Zhang, X. Liang, X. Liu, and L. Lin. Geometric scene parsing with hierarchical LSTM. *CoRR*, abs/1604.01931, 2016.

[43] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

[44] K. Richmond. A trajectory mixture density network for the acoustic-articulatory inversion mapping. In *Interspeech*, 2006.

[45] A. Robicquet, A. Sadeghian, A. Alahi, and S. Savarese. Learning social etiquette: Human trajectory understanding in crowded scenes. In *European Conference on Computer Vision*, pages 549–565. Springer, 2016.

[46] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[47] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[48] R. S. Sutton. *Introduction to reinforcement learning*, volume 135.

[49] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015.

[50] Y. H. Tan and C. S. Chan. phi-lstm: A phrase-based hierarchical LSTM model for image captioning. *CoRR*, abs/1608.05813, 2016.

[51] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016.

[52] P. Trautman and A. Krause. Unfreezing the robot: Navigation in dense, interacting crowds. In *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, pages 797–803. IEEE, 2010.

[53] P. Trautman, J. Ma, R. M. Murray, and A. Krause. Robot navigation in dense human crowds: the case for cooperation. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 2153–2160. IEEE, 2013.

[54] B. Uria, I. Murray, and H. Larochelle. Rnade: The real-valued neural autoregressive density-estimator. In *Advances in Neural Information Processing Systems*, pages 2175–2183, 2013.

[55] S. van der Walt, S. C. Colbert, and G. Varoquaux. The numpy array: A structure for efficient numerical computation. *Computing in Science Engineering*, 13(2):22–30, March 2011.

[56] A. Vemula, K. Muelling, and J. Oh. Modeling cooperative navigation in dense human crowds.

[57] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, L. Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, and J. Dean. Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144, 2016.

[58] S. Yi, H. Li, and X. Wang. Understanding pedestrian behaviors from stationary crowd groups. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3488–3496, 2015.

[59] H. Zen and A. Senior. Deep mixture density networks for acoustic modeling in statistical parametric speech synthesis. In *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*, pages 3844–3848. IEEE, 2014.

[60] B. D. Ziebart, N. Ratliff, G. Gallagher, C. Mertz, K. Peterson, J. A. Bagnell, M. Hebert, A. K. Dey, and S. Srinivasa. Planning-based prediction for pedestrians. In *Intelligent Robots and Systems, 2009. IROS 2009. IEEE/RSJ International Conference on*, pages 3931–3936. IEEE, 2009.