Dead Reckoning for Distributed Network Online Games

by

Tristan Walker

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2021

© Tristan Walker 2021

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Online networked games are becoming increasingly popular. One type of network architecture used in these games is a distributed network architecture, where players send periodic updates to each other and each player must locally reconstruct the position of their opponents in between these updates. In this work, we assume a car model for the players, as errors in this type of network are most pronounced when players have high speeds. We are interested in decreasing this update frequency in order to conserve bandwidth. We are also interested in investigating issues that arise when these locally replicated opponents need to interact and collide with objects in the environment.

In this thesis we decompose the replication problem into two components: first, we must predict the position of our opponents by extrapolating from the received updates, then we must create a smooth trajectory from these predicted positions that appears believable to the player. We introduce a neural network based approach to solving the prediction portion that outperforms the current state of the art. We then propose a neural network based approach and an approach based on a path tracking controller for mobile robots to generate smooth trajectories. We present results to compare these approaches and show that the path tracking approach performs better than both the neural network approach and the established state of the art approaches.

We also investigate collisions between replicated opponents and the environment. This is a complex problem, so for simplicity we are only examining collisions with static obstacles. Collisions can vary dramatically based on small changes in impact point and angle, and so we want to be able to predict collisions based on the predicted position of the opponent because that is theoretically our best estimate of the true position of our opponent. We propose a neural network based approach to this problem, which is able to predict the collision response of a vehicle colliding with a static obstacle. We present results that show this method has potential to outperform the current best practice, but we also discuss several implementation issues that must be addressed.

Acknowledgements

I would like to thank my supervisor, Stephen Smith, for his guidance throughout my Master's degree. I would also like to thank Armin Sadeghi for his assistance during the research and writing of this thesis.

Dedication

This is dedicated to my parents for being such an inspiration and for always believing in me.

Table of Contents

List of Tables ix

List of Figures x

1 Introduction 1

1.1 Contributions . . . 3

1.2 Organization . . . 3

2 Literature Review 5

2.1 Networked Gaming . . . 5

2.2 Dead Reckoning . . . 6

2.3 Machine Learning in Video Games . . . 7

3 Background 8

3.1 Distributed Network Architecture . . . 8

3.2 Terminology . . . 9

3.2.1 Player Model . . . 9

3.2.2 Network Model . . . 10

3.2.3 Replication . . . 11

3.3 Dead Reckoning . . . 13

3.4 Linear Blending . . . 13

3.5 Projective Velocity Blending . . . 14

3.6 Supervised Learning . . . 15

4 Trajectory Prediction **19**
4.1 Trajectory Prediction Problem 19
4.2 Neural Network Approach 21
4.3 Simulation Setup 22
4.4 Training 22
4.5 Results 24

5 Path Blending **27**
5.1 Problem Setup 27
5.2 Neural Network 28
5.2.1 Training 28
5.2.2 Implementation 30
5.3 Path Tracking 31
5.3.1 Formulation 32
5.3.2 Implementation 33
5.4 Results 33
5.4.1 Neural Network Results 33
5.4.2 Path Tracking Results 34
5.4.3 Parameter Tuning 36

6 Collisions **39**
6.1 Problem Setup 40
6.2 Solution Approach 40
6.2.1 Neural Network 41
6.2.2 Training 42
6.2.3 Implementation 43
6.3 Results 44

7 Conclusions **49**

7.1 Future Work 50

References **51**

List of Tables

5.1 Average errors for linear blending and path tracking blending. 36

List of Figures

1.1 Screenshot of driving a vehicle in Grand Theft Auto Online 1
1.2 Example of extrapolating an opponent's position from periodic updates . . 2

3.1 Labeled representation of the player model 10
3.2 Visual representation of the network model 12
3.3 Block diagram of the two player scenario 12
3.4 Projective velocity blending applied to blend $\bar{\mathbf{r}}_0$ to predicted position $\mathbf{r}'_t$ . . 15

4.1 Overview of the prediction network 21
4.2 Screenshot from the Unity simulation environment 23
4.3 Prediction error of dead reckoning and prediction network over different message intervals 24
4.4 Predictions made using dead reckoning and the prediction network compared to the master path 25

5.1 Overview of the blending network 28
5.2 Comparison of optimal path from the SOCP to the path generated by the trained network 30
5.3 Implementation of the blending network when not updating the velocity with new messages 34
5.4 Results of path tracking blending with untuned parameters and a message interval of $300ms$ 35
5.5 Results of path tracking blending with tuned parameters and a message interval of $200ms$ 37

5.6 Results of path tracking blending with tuned parameters and a message interval of 300ms 38

5.7 Results of path tracking blending with tuned parameters and a message interval of 300ms and a drop rate of 50% 38

6.1 Overview of the collision network 41

6.2 Screenshot of a collision with a cylindrical obstacle in Unity 42

6.3 Flow diagram of replication process at timestep $k + i$ with collisions included 44

6.4 Position and Rotation error with a message interval of 100ms 45

6.5 Position and Rotation error with a message interval of 250ms 45

6.6 Position and Rotation error with a message interval of 500ms 46

6.7 Paths of Master compared to Linear and Our replication schemes with a message interval of 500ms 47

Chapter 1

Introduction

Online games are becoming increasingly popular, with games such as Grand Theft Auto Online, shown in Figure 1.1, having over 100,000 concurrent players at a time [26]. Developers strive to provide competitive, yet immersive experiences to players. In online games in particular, one of the key challenges in delivering both of these is the limitations in communication between players. In this work, we investigate games with a distributed network architecture, which is an alternative to the more common host-client architecture that does not require the developer to pay to maintain dedicated servers. In a distributed network, players send information between each other and each player recreates their own "best guess" as to what the true state of the game is. All information that is sent between players may be subject to network latency and packet loss. Additionally, it is prohibitively expensive to send information at the same rate that the game updates, and so players must extrapolate information from periodic updates. The errors caused by these extrapolations

Figure 1.1: Screenshot of driving a vehicle in Grand Theft Auto Online

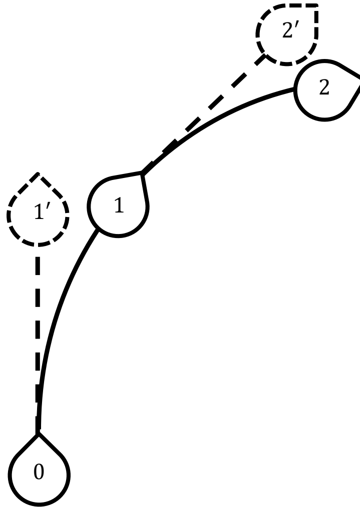


Figure 1.2: Example of extrapolating an opponent's position from periodic updates

are more pronounced when players are moving at high speeds, such as when operating vehicles. The most common vehicles players operate in video games are cars, and thus our investigation focuses on the case where players are driving cars.

Suppose we are playing an online game against another player and our opponent sends their position and velocity once every second. In practice, this happens much more frequently, typically with a period under $100ms$ [3]. In order to maintain an immersive and competitive experience, we want to be able render our opponent locally in a way that is both believable and reasonably accurate so that interactions with the opponent feel fair and natural. If we receive updates from our opponent every second, simply rendering the opponent at the most recent position received from them would not be accurate or believable. Thus we need some way of extrapolating our opponent's position from the information we receive. This is known as dead reckoning. With the information given in our example, the most reasonable extrapolation we can make is to project the position forwards using the received velocity. That is, if we received a position, $\mathbf{r}$, and velocity, $\mathbf{v}$, at n seconds, we can estimate their position at $n + t$ seconds as $\mathbf{v}(n + t) = \mathbf{r}(n) + \mathbf{v}(n)t$. We can see a visualization of this example in Figure 1.2. We start with our opponent's position and velocity at time 0, and we see their true path curves to the right. We can see our extrapolations at 1 and 2 seconds as $1'$ and $2'$ respectively. From the figure we can see that simply rendering the opponent's position using this extrapolation is also not sufficient, as the resulting path will be discontinuous at the points where we receive new information from the opponent, which can be extremely jarring to the player. Thus we

need a method of not only predicting an opponent's position, but also a strategy to blend these predicted positions into a smooth, believable path.

This problem is further complicated when the replicated opponent has to interact and collide with the environment or other opponents. Small discrepancies between a player's estimate of an opponent's position and the opponent's true position at the time of a collision can result in significantly different trajectories after the collision. This is a difficult problem, and existing techniques are largely aim to hide the effects of this discrepancy without making any attempts to reduce the error between the trajectories.

1.1 Contributions

The following are the key contributions in this thesis:

- In Chapter 4, we present a new application of neural networks to extrapolate other players' positions in an online game. We provide simulation results that show our approach has greater robustness to poor network conditions, with at least 25% less error under poor conditions compared to the state of the art.
- Chapter 5 proposes a novel application of a path tracking algorithm to create a smooth trajectory for believably replicating an opponent. We show through simulation results that our approach produces smoother, more believable trajectories with approximately 45%less error than the established state of the art.
- Chapter 6 applies neural networks to predict the response of a vehicle to a collision with a static object. We show with simulation results that our approach can believably predict collisions under much poorer network conditions than the current state-of-the-art.

During my graduate studies I also contributed simulation results and authorship of a journal paper that has been submitted for publication [42].

1.2 Organization

In Chapter 2 we review related literature regarding networked games, dead reckoning, path tracking, and machine learning. In Chapter 3 we introduce our model for a networked game,

and present established algorithms for dead reckoning and path blending. We present our approach for predicting the position of an opponent in Chapter 4 along with results demonstrating the effectiveness of our approach. Chapter 5 proposes our method for using path tracking algorithms to complete the replication problem with results comparing to the established methods described in Chapter 3. Chapter 6 describes our approach to predict the response of a replicated vehicle with a static obstacle. Chapter 7 concludes this work and outlines possible future investigations.

Chapter 2

Literature Review

This section provides a brief overview of relevant literature. We first examine literature related to networked gaming, and how network conditions can affect the player experience. We then examine literature related to predicting an opponent's position, as well as how these predicted positions may be used to form a smooth trajectory. Lastly, we look at other applications of neural networks in video games.

2.1 Networked Gaming

Online games are becoming increasingly popular. Trying to send data over a network introduces significant obstacles to creating an immersive and enjoyable experience. Noticeable latency in the network can impact users' enjoyment, with one study reporting that users found latencies over 150 ms to be unacceptable [11]. Lee et al. propose Outatime [24], a method to use Markov predictions and Kalman Filtering to predict future game states and reduce input lag in cloud gaming applications. Savery et al. [34] examine the effect of various lag compensation techniques on player enjoyment, and note that techniques that result in the best performance of the player are not always the most enjoyable for the player.

On the server side, researchers are looking into ways of reducing the amount of data sent. Babei et al. [2] address this issue in cloud gaming by using eye tracking data to determine where players are likely to look so that they can stream higher quality video to those areas. Harvey et al. [15] use dead reckoning prediction to reduce the amount of information packets sent to save on power when gaming on a phone.

2.2 Dead Reckoning

Dead reckoning is the practice of extrapolating an agent's position given their state from some previous point in time. Dead reckoning has broad applications in many fields related to vehicle positioning, as well as networked games. The simplest method of dead reckoning is to extrapolate using only the estimated velocity. Many authors [5, 8, 31] have shown this to be insufficient for immersive online gaming. Pantel and Wolf [31] note that a major obstacle is that latency between players can be upwards of 100ms, and find some success in mitigating this by incorporating acceleration and user input into their prediction schemes. Aggarwal et al. [1] proposed a method of using timestamped messages to help reduce the impact of latency between players. Kharatinov [19] proposed an adaptive dead reckoning scheme that sends more data during more complex motions, and Almeida and Felinto [8] used the popular game engine Unity to verify that this algorithm held up under realistic network conditions, and showed that its performance was superior to traditional dead reckoning techniques. For car simulations specifically, Chen and Liu [5] proposed an algorithm that uses environmental cues such as roads to aid its predictions, and shows that their algorithm has better performance than traditional techniques in terms of accuracy, computational cost, and bandwidth consumption. In the field of marine navigation, Skulstad et al. [37] used a Recurrent Neural Network to perform dead reckoning for ships, with performance comparable to or surpassing the traditional Kalman Filter approach. Belhajem et al. [4] use machine learning in a similar setting, by combining an Extended Kalman Filter with neural networks to improve accuracy. Shi et al. [36] proposed an application specific dead reckoning approach that uses data collected from human play sessions to parameterize a dead reckoning model specific to one game. Authors in [12] propose a two part machine learning approach for video games to compensate for the inability of traditional dead reckoning approaches to account for changes in player input. Their approach uses a classifier to determine if a player will change direction based on their surroundings, and uses a second network in place of traditional dead reckoning if the classifier determines the player will change direction.

As shown by the example in Figure 1.2, extrapolation only forms part of our problem. We must also be able to form these extrapolations into a smooth trajectory. Murphy [27] describes an alternative dead reckoning technique used in the video game industry called projective velocity blending, which results in smooth predicted trajectories. Schuwerk and Steinback [35] note that while projective velocity blending produces visually smooth paths, users did not enjoy the high accelerations it generated when used to produce haptic feedback. Generating smooth trajectories is also very important in the field of autonomous vehicles. Samson and Abderrahim [33] proposed a linearized feedback control method for a

wheeled robot to stably follow a commanded path. Ostafew et al. [30] successfully applied this controller to a physical robot to enable it to traverse uneven outdoor terrain. Another popular path tracking controller is the controller used by the Stanley robot for the DARPA Grand Challenge [39], which uses wheel angle to control lateral and heading error.

2.3 Machine Learning in Video Games

Machine Learning has been rising in popularity and has many applications, especially in the realm of video games. Besides using machine learning techniques for prediction and dead reckoning as mentioned previously, another application is the use of deep learning to play video games [18]. Machine learning is also used to approximate complex physics calculations much faster than traditional numerical-based approaches. Holden et al. [16] and Luo et al. [25] use neural networks to simulate cloth physics in real time. Tompson et al. [40] uses a convolutional network to simulate incompressible Eulerian fluid physics. Tracey et al. [41] uses supervised learning to model turbulence, and Lattimer et al. [23] uses machine learning to predict behaviour of fires. Summerville et al. [38] explores the use of machine learning to generate new content for video game such as new, unique levels. Geisler [14] uses neural networks to model human behaviour and applies this to make AI opponents that behave more realistically.

Chapter 3

Background

In this chapter, we will present our assumed model of the network and players. We also introduce existing state-of-the-art techniques for replicating an opponent's trajectory from periodic information, including techniques for simply predicting position as well as techniques for generating a smooth, continuous trajectory. We also present a brief background of the machine learning techniques that we will be using in our solution approaches.

3.1 Distributed Network Architecture

In online video games there are two main network architectures: peer-to-peer or host-client [21]. In a host-client network, one computer is designated as the *server* and is responsible for managing the game state and communicating back and forth with all the players that are connected to the server. The clients can then take the information received from the server and display it to the player without much additional processing. Often this server is an independent computer that does not participate in the game and is called a *dedicated server*, as is the case in many large online games such as Counter Strike: Global Offensive and League of Legends. These are costly to run and maintain, however, and a commonly used alternative is called a *listen server*. In a listen server, one of the players is also designated as the server, which was done in earlier versions of Call of Duty. This can put additional computational load on the designated player's machine as it must run the server as well as run the game for the player. The player designated as the server also has an in-game advantage over the other players because they do not have any latency to the server.

The alternative is a peer-to-peer network, which we refer to as a *distributed* network. In such a model, there is no single governing server; all players, or *peers*, periodically broadcast information to their peers and must take the information received from their peers and resolve it into their own consistent game state to render to the player. A distributed network has the advantage of having less overhead for the game developer by eliminating the need to maintain dedicated servers [28], and as such is sometimes chosen for smaller online games such as the online game mode of Watch Dogs 2. However, a distributed network introduces a number of new challenges. With no single host dictating the game state, each player can have their own slightly different interpretation of what the game looks like at a given time. When a player receives information from other players, they must do their own processing to determine what it thinks the new state of the game looks like, and small discrepancies can escalate over time to have dramatic changes to the game state. For example, if one player's guess of another player's position is off by a small margin, they may think that the other player will collide with a tree when they did not. Additionally, since the players must send information to all other players, bandwidth consumption can be a concern. We will be investigating how to reduce bandwidth consumption while maintaining or improving accuracy in the prediction of other players.

3.2 Terminology

3.2.1 Player Model

In a 3-dimensional video game, a player's state can be described by the tuple $\mathcal{X} = (\mathbf{r}, \mathbf{v}, \mathbf{q}, \mathbf{a}, \boldsymbol{\omega})$ containing the car's position vector $\mathbf{r} \in \mathbb{R}^3$, velocity vector $\mathbf{v} \in \mathbb{R}^3$, orientation quaternion $\mathbf{q}$, acceleration vector $\mathbf{a} \in \mathbb{R}^3$, and angular velocity vector $\boldsymbol{\omega} \in \mathbb{R}^3$. Note the use of a quaternion to represent the player's orientation as is common in video game engines [43]. Figure 3.1 shows a labeled model with these properties.

The player interacts with the game world through the engine in discrete timesteps corresponding to each frame that is rendered by the game. We will define the time between frames as δt. In practice, the timesteps that the game engine uses to update the game state do not have to correspond to frames that are rendered to the player. However, in this work we will assume that a single frame is rendered at each timestep of the game, and will use the two interchangeably. We can view the game engine as a black box system that uses the physics of the game to determine how the player interacts with the world and other players. We will define the player's control actions at a timestep k to be $\mathbf{A}[k] = (h, c)$ with $h, c \in [-1, 0, 1]$ representing a discrete left, straight, or right control action and

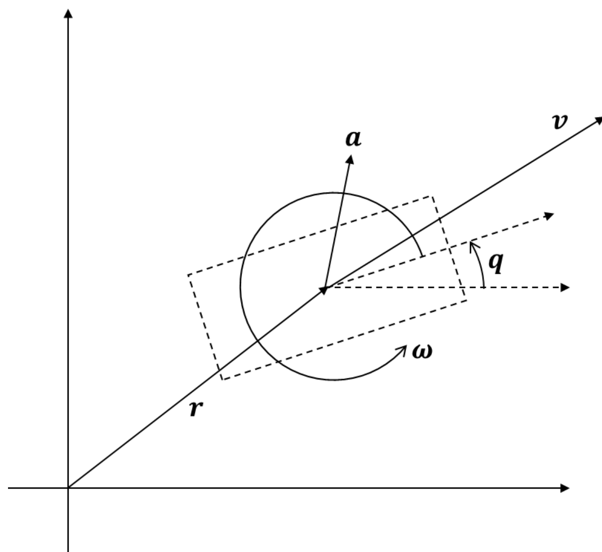


Figure 3.1: Labeled representation of the player model

a discrete deceleration, coasting, or acceleration control action respectively. We could consider continuous control inputs without adding much complexity, but for simplicity we will assume players use a keyboard for control, resulting in discrete inputs. At each timestep, the engine takes the player's state, the player's actions, and the world state, and produces the player's state at the next timestep. If we represent the world state at timestep k as $\mathcal{W}[k]$, and the game engine as a function $\mathcal{B}$ that takes this as well as the player's state and actions, we can write the evolution of the player's state as $\mathcal{X}[k+1] = \mathcal{B}(\mathcal{X}[k], \mathcal{W}[k], \mathbf{A}[k])$. In this work, for simplicity, we can assume that the world state is static, and can be encapsulated in the function $\mathcal{B}$. Thus, we can think of the evolution of the player's state as simply $\mathcal{X}[k+1] = \mathcal{B}(\mathcal{X}[k], \mathbf{A}[k])$.

3.2.2 Network Model

For simplicity in this work, we will only be considering one other player at a time. To extend this work to multiple other players, the techniques described for handling one other player can simply be repeated for each other player, assuming that there is no interaction between players. If the game models some interation between players, such as gravity in a space simulation game, additional work would need to be done to approximate the interaction between two networked opponents. For clarity, this work is written from the

perspective of the *player* receiving information from another player, which we call the *opponent*. This is simply for clarity; they are not necessarily competing with each other in the game and simply represent two human agents playing on a network. In a two-player game on a distributed network, each player will periodically broadcast their state and control actions to the other player. The players receive this information over a network, which means that it may be subject to latency and packet loss. To help deal with potential delays and timing issues, players will also broadcast the time that the information was recorded when broadcasting their state. We will call this set of information that the players broadcast a *message*, defined as $s[k] = (\mathcal{X}[k], \mathbf{A}[k], k)$ where k is the timestep the message is sent. We assume that timesteps are globally synchronized across all players. This is difficult in practice, and messages would usually include a time t instead of a timestep k. However, we will use k to avoid extra conversions from continuous time to discrete timesteps. Additionally, we define the following network parameters: message interval T, as the number of milliseconds the player waits before sending another message, latency d, as the number of milliseconds it takes for the message to reach the other player, and packet loss p as the percentage of messages that will not be received by other players. Figure 3.2 shows latency and message interval when a player is receiving messages from an opponent over a network. In this figure, $i = \lfloor \frac{T}{\delta t} \rfloor$, representing the number of timesteps between messages. Recall that δt is the time between timesteps. In our investigation, we are primarily concerned with making our solution robust to larger values of T. Current games typically use message intervals around $100ms$ or less [3]; we are interested in developing a replication scheme that is robust to message intervals of up to $300ms$ or greater. Our algorithm should also be robust to moderate latency up to $100ms$ and packet loss of up to 10%, but these are not the primary variables that we will examine. The model of the game engine for the two player scenario is shown in Figure 3.3.

3.2.3 Replication

When playing a networked game, we do not know the ground truth of our opponent's states at all times. We only receive periodic messages of our opponent's state, which is subject to latency and packet loss, and leaves many frames in between for which we have no new information. Thus we wish to create a believable estimate of our opponent from these periodic messages. We will call our local estimate of our opponent a *replica*. This replica must appear to be a real player, meaning that it must appear to obey the established physics of the game, and its position should be reasonably close to the true position of the opponent to ensure that we have an accurate image of the state of the game.

In order to generate a smooth and believable trajectory for the replica, we break the

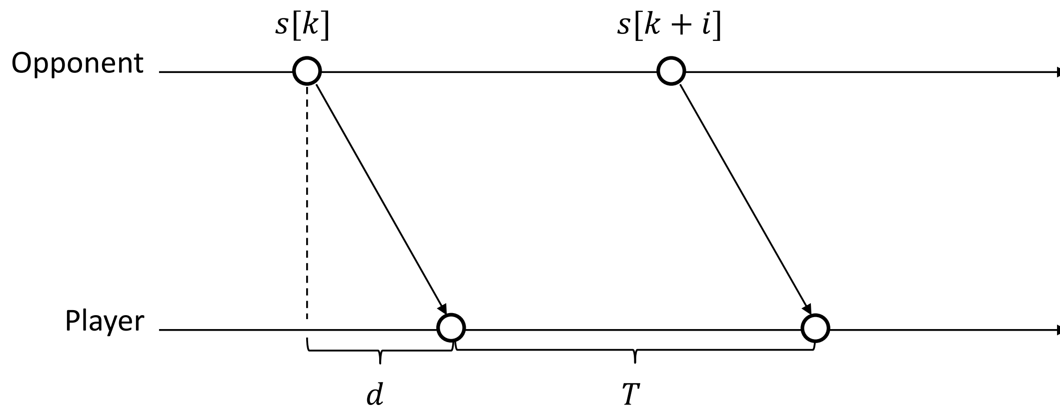


Figure 3.2: Visual representation of the network model

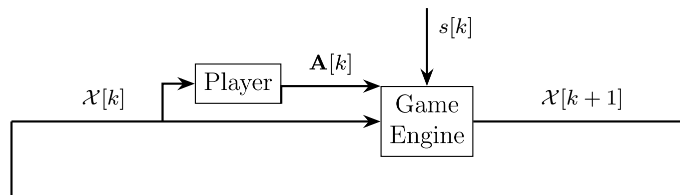


Figure 3.3: Block diagram of the two player scenario

problem into two parts for our approach. The first component is the prediction problem, where the goal is to predict the state of the opponent at some time in the future given the most recently received message from them. The other component we refer to as blending, where we must move our replica of the opponent towards this predicted position in a smooth manner that appears believable to the player. The predicted states do not need to form a smooth trajectory, but the trajectory formed by the blended states should appear realistic to the player. These problems will be defined in more detail later, but we will introduce the convention here of using $\hat{\mathcal{X}}$ to denote predictions and $\bar{\mathcal{X}}$ to denote the state of the blended replica.

3.3 Dead Reckoning

The process of extrapolating an estimated state from some earlier state in time is called dead reckoning. The simplest form of dead reckoning is to project using linear physics; given a position, $\mathbf{r}[k]$ and a velocity $\mathbf{v}[k]$ at timestep k, one can estimate the position at timestep $k+i$ as $\hat{\mathbf{r}}[k+i] = \mathbf{r}[k] + t\mathbf{v}[k], t = i\delta t$, recalling that δt is the time between timesteps, and thus t is the time between timesteps k and $k+i$. If additional information is available, it can also be incorporated. For example, to include acceleration, $a[k]$, one can use basic kinematics to give

$$\hat{\mathbf{r}}[k+i] = \mathbf{r}[k] + t\mathbf{v}[k] + 1/2\mathbf{a}[k]t^2.$$

The state of the art algorithm that we will compare to is a version of dead reckoning that uses velocity and angular velocity. Given position $\mathbf{r}[k]$, velocity $\mathbf{v}[k]$ and angular velocity $\boldsymbol{\omega}[k]$ at timestep k, we predict the position at the next timestep by extrapolating using the given velocity, rotated by the given angular velocity. We will define a rotation operator rot such that $\mathrm{rot}_\theta(\mathbf{v})$ rotates $\mathbf{v}$ by the Euler angles given by $\theta \in \mathbb{R}^3$. Then we have $\hat{\mathbf{r}}[k+1] = \mathbf{r}[k] + \mathrm{rot}_{\boldsymbol{\omega}[k]\delta t}(\mathbf{v}[k]\delta t)$, where δt is the time between frames. To predict further into the future, this process is repeated for each frame, ie. $\hat{\mathbf{r}}[k+2] = \hat{\mathbf{r}}[k+1] + \mathrm{rot}_{\boldsymbol{\omega}[k]2\delta t}(\mathbf{v}[k]\delta t)$. This means that the predicted trajectory from one known state will be a piece-wise linear arc.

3.4 Linear Blending

One of the simplest blending strategies is to replicate the opponent at a set delay behind the current prediction. This has the effect of smoothing the predicted path, at the cost of

introducing error from the delay. Suppose we have a replica of our opponent at timestep $k-1$ with a position $\bar{\mathbf{r}}[k-1]$, and we wish to move it smoothly towards our predicted position $\hat{\mathbf{r}}[k]$. Mathematically, we can find the current position of our replica as $\bar{\mathbf{r}}[k] = \bar{\mathbf{r}}[k-1] + (\hat{\mathbf{r}}[k] - \bar{\mathbf{r}}[k-1])\delta t/\lambda$, where λ is the time behind the prediction that the replica will be.

To implement this algorithm, a new predicted state should be generated for each timestep and the replica should be blended according to this algorithm at each timestep. This strategy can be tuned by varying λ; a smaller λ results in a trajectory that aggressively follows the predictions but can make sharp, unrealistic turns, while a larger λ yields a smoother trajectory that lags further behind the prediction. This algorithm does not guarantee that the blended path will be smooth, and at low values of λ the resulting trajectory may have visibly sharp corners.

3.5 Projective Velocity Blending

The current state of the art blending algorithm is known as projective velocity blending [27]. While relatively simple in theory, it yields smooth paths with quite consistent and robust results. This algorithm performs both the prediction and blending steps; given the current state of the replica $\bar{\mathcal{X}}$ and a message with the last known state $\mathcal{X}$, the algorithm performs a dead reckoned projection of both the current state and the last known state, and uses a linear interpolation between the two to yield the blended position. In practice, this works best when the projection of the current state uses a linear interpolation of the current velocity and the last known velocity. To predict i steps ahead with $t = i\delta t$, recalling that T is the message interval, i.e, the maximum time we are extrapolating to, this looks like:

$$\begin{aligned}
t_{\text{blend}} &= t/T \\
\mathbf{v}_b[k] &= \bar{\mathbf{v}}[k] + (\mathbf{v}[k] - \bar{\mathbf{v}}[k])t_{\text{blend}} \\
\hat{\mathbf{r}}[k+i] &= \bar{\mathbf{r}}[k] + \mathbf{v}_b t_{\text{blend}} + \frac{1}{2}\mathbf{a}[k]t^2 \\
\mathbf{r}'[k+i] &= \mathbf{r}[k] + \mathbf{v}_b t_{\text{blend}} + \frac{1}{2}\mathbf{a}[k]t^2 \\
\bar{\mathbf{r}}[k+i] &= \hat{\mathbf{r}}[k+i] + (\mathbf{r}'[k+i] - \hat{\mathbf{r}}[k+i])t_{\text{blend}}
\end{aligned}$$

A visual representation of projective velocity blending is shown in Figure 3.4.

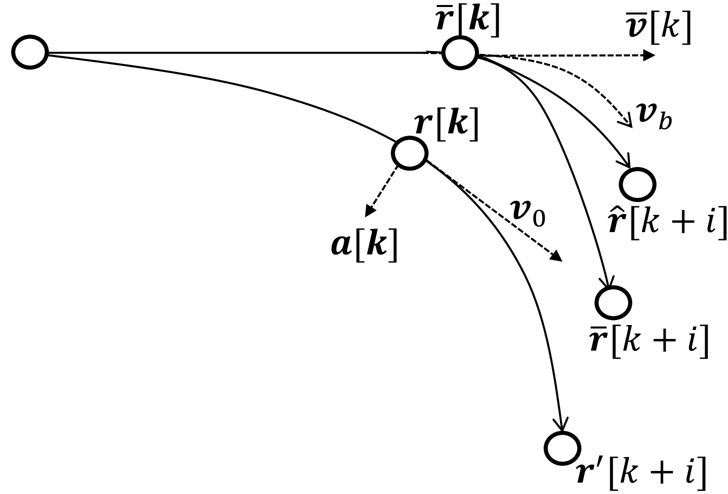


Figure 3.4: Projective velocity blending applied to blend $\bar{\mathbf{r}}_0$ to predicted position $\mathbf{r}'_t$

3.6 Supervised Learning

Supervised Learning is a form of machine learning in which the goal is to learn an unknown function that maps an input to an output, given a training set of sample input-output pairs. Machine learning itself is an umbrella term that encompasses a number of different techniques. These techniques fall into two main categories: classification and regression.

In classification, the goal is to assign a discrete label to a given input. For classification tasks in the realm of Supervised Learning, the classifier can only assign labels for which training data exists. One of the most common classifiers is the Support Vector Machine (SVM) [6]. In its original form it separates classes via linear hyperplanes, however the authors in [6] propose a way to use SVMs for nonlinear classification by applying the kernel trick, which allows the algorithm to implicitly operate in a higher dimensionality. Another simple form of classification is known as k-Nearest Neighbors (k-NN) [13]. With k-NN classification, an input is assigned the label that is most common for the k most similar members of the training set. Classification algorithms are typically evaluated based on their accuracy, or the percentage of inputs that they can classify correctly.

Regression aims to generate a continuous output from a given input. One of the simplest forms is linear regression, which approximates the relationship between input and output with a linear factor. Algorithms like k-NN can be adapted to become regression algorithms by taking an average of the output values of the k most similar members of the training

set. Regression models are evaluated based on their error compared to the true output values, typically measured with root mean squared error (RMSE).

One of the increasingly popular techniques in machine learning is Artifical Neural Networks. This in itself is a very broad concept that can be applied to both supervised and unsupervised learning, and can perform classification as well as regression. Neural Networks are inspired by the structure of neurons in a biological brain; they are composed of a network of nodes or "neurons" that transmit signals to each other through a network of connections, mirroring the transferal of information between neurons through synapses in a real brain. These connections each have a weight that is learned through training the network, which will be discussed later. Each node takes the weighted sum of its inputs as well as its own bias term, which generates the output of the node. The nodes are typically arranged in layers; there are many different ways to connect these layers, each with their own benefits. Feedforward networks are networks in which nodes are only connected to nodes in the next layer. In a fully connected network, each node is connected to every node in the next layer. Networks that allow nodes to connect to the same or previous layers are known as recurrent networks, and can be effective in situations where there may be some temporal correlation in the input data. Networks with one or two layers can be sufficient for many tasks, but additional layers may help to extract higher level features from the input data. Networks with more layers are said to be *deeper*.

Each node outputs some function of the sum of its inputs. This function that generates the output of a node is called the activation function. A simple choice of activation function would be a linear function, in which the sum of the inputs is multiplied by some constant factor. However, this choice has a significant shortcoming in that if the entire network is linear, it is essentially performing linear regression. Therefore, most modern Neural Networks use nonlinear activation functions. It has been proven that with nonlinear activation functions, a neural network with just one hidden layer of sufficient size is capable of approximating any function [7]. Common choices include the sigmoid or hyperbolic tangent (tanh) functions. These are very similar functions, but since the tanh function is centered around the origin it may perform better in applications where the input may be positive and negative. These functions can be computationally expensive especially when there may be hundreds or thousands of them in a network. Additionally, they suffer from the vanishing gradient problem, where the slope of the output becomes very small for very small or large input values, which may cause training to slow significantly or even stop. Recently, a common choice of activation function is the rectifier, or Rectified Linear Unit (ReLU). This is a simple function, defined as the positive part of its argument i.e., $f(x) = max(0, x)$. The simplicity of this function makes it cheap to compute, and its linear nature ensures that it does not suffer from the vanishing gradient. Moreover, the

nonlinearity at the origin ensures that ReLU is capable of approximating any function.

The key algorithms in training Neural Networks are gradient descent and backpropogation. Details regarding training are following the conventions in [29]. At a high level, training a network is an iterative process where the connection weights are varied slightly to try and improve the accuracy of the network at each iteration. The function that captures the accuracy of the network is called the *cost function* (or sometimes *loss function* or *objective function*). For regression tasks, this is typically the mean squared error i.e., $C(w,b) \equiv \frac{1}{2n}\sum_x ||y(x) - a||^2$, where w and b are the weights and biases of the network respectively, n is the number of training inputs, a is the output of the network for a given x, and $y(x)$ is the target we wish to achieve. Training a network aims to find the set of weights and biases that minimizes this cost function. This is typically done through gradient descent, the details of which will not be discussed here. Gradient descent finds the gradient of the cost function and takes small steps in the direction of steepest decreasing gradient. The technique for finding this gradient is called *backpropogation*, which is derived such that it takes advantage of the layered structure of the network to avoid unnecessarily duplicating calculations. There are many extensions to these techniques that aim to speed up training such as variable learning rates that control large the steps taken in gradient descent are, and stochastic gradient descent that uses stochastic sampling and shuffling of the dataset to train on small subsets of the training data. In training Neural Networks, typically a portion of the training dataset is reserved as test data and not used to train the network and is instead used to evaluate the performance of the network once it has converged.

A key advantage of Neural Networks is that they can often be implemented like black boxes, where understanding of the internal mathematics behind their functioning is not required for them to be successful. Many libraries such as *PyTorch* [32] are available that make implementation of such networks straightforward. The important choices that remain for the user have to do with the properties of the network: the number of nodes, the choice of activation function, and how the nodes are structured are some of the most defining features of a network. Some architectures are known to perform better for some applications, for example the field of image recognition is dominated by Convolutional Neural Networks [22], meant to efficiently extract higher level features from an image. Selecting the number of nodes in a network offers a tradeoff; more nodes offers the ability to model more complex functions at the cost of increased computational resources and longer training times. Additionally, having too many parameters in the network relative to the complexity of the function being modeled increases the risk of *overfitting*, where the model performs very well on the training data but can no longer generalize to fit unseen data. Besides choosing an appropriate number of nodes, other techniques exist to

reduce overfitting including cross-validation and regularization. Cross-validation involves performing multiple iterations of training, where a different segment of the training data is used as test data in each iteration. Regularization adds an additional penalty term during training to encourage lower weights. Preprocessing of data, such as normalizing inputs, can also improve the performance of a network.

Chapter 4

Trajectory Prediction

Consider a player in a multiplayer game with one opponent over a distributed network. The player and opponent send each other a message at regular intervals containing information describing their state. As the player, we wish to locally replicate our opponent accurately and believably. Believability is difficult to quantify, and is a combination of different factors. A believable trajectory should be smooth and appear to obey the dynamics of the vehicle, and should not show consistent hiccups when it receives new messages. Our replication scheme is composed of two separate steps; first we predict the position of our opponent after the received message as accurately as possible, then we use a blending algorithm to create a smooth and believable trajectory for our replica. This chapter focuses on the prediction portion of the replication process. Note that in this breakdown of the problem, the prediction encompasses only the position; the blending portion is responsible for determining the orientation. This decomposition places a higher emphasis on believability to the player by giving the blending algorithm full control to select orientations that make the most sense given the blended replica's trajectory.

In this chapter we define our problem, and present our neural network based approach. We also provide simulation results where our approach demonstrates reliable performance over a wider range of message intervals compared to the traditional dead recknoning approach described in Section 3.3.

4.1 Trajectory Prediction Problem

For the prediction portion of the replication process, our goal is, given the most recent message we have received from an opponent, to predict their position up to the time we

receive another message. Recalling that δt is the time between frames, we can formally define the problem as follows.

Problem 1 (Trajectory Prediction) *Given a message $s[k]$ and a time after the message to predict Δt, find a state $\hat{\mathcal{X}}[k+i]$ where i is the number of timesteps to predict, i.e., $i = \lfloor \frac{\Delta t}{\delta t} \rfloor$.*

Note that given the decomposition into a prediction and blending problem, there are two possible approaches to making predictions. Our chosen approach is to make a single prediction for the time in the future that we expect to receive another message, using the blending algorithm at every timestep to create a smooth trajectory towards this predicted point. The alternative is to make a prediction for the next timestep and blend towards this next point on a frame by frame basis. This approach places more emphasis on positional accuracy at every frame, as it encourages the blended state to stay close to the predicted. We have chosen to once again favour smoothness and believability of replication by blending to the single predicted point, which allows for more positional error during the intermittent timesteps while allowing the blending algorithm more freedom to control smoothness.

We are most interested in situations where the opponent's position varies greatly in between messages, where there is higher potential error between the predicted position and the true position. This corresponds to situations where the opponent has a high speed, such as when they are operating a vehicle. For this investigation, we are looking specifically at predicting the position of an opponent when they are driving a car.

Design Considerations: The prediction problem poses many challenges. Recall that we can model the evolution of the player's state as $\mathcal{X}[k+1] = \mathcal{B}(\mathcal{X}[k], \mathbf{A}[k])$, where the function $\mathcal{B}$ represents the game engine. For predicting one time step in the future, the problem is essentially to approximate the unknown function of the game engine. This function itself is difficult to approximate, as it is dictated by complex physics. The function may also vary based on the properties of the player's vehicle, such as its weight and power. Additionally, when predicting multiple time steps in the future, we do not know what actions the player will make in the upcoming time steps, for example if the player chooses to suddenly brake or turn sharply. Solutions to the prediction problem should be computationally efficient as it needs to be computed at every timestep, and in a real implementation would need to be calculated for every opponent. A good solution should also be independent of the properties of the opponent's vehicle; it should be able to be easily applied to different vehicles without needing to manually tune parameters.

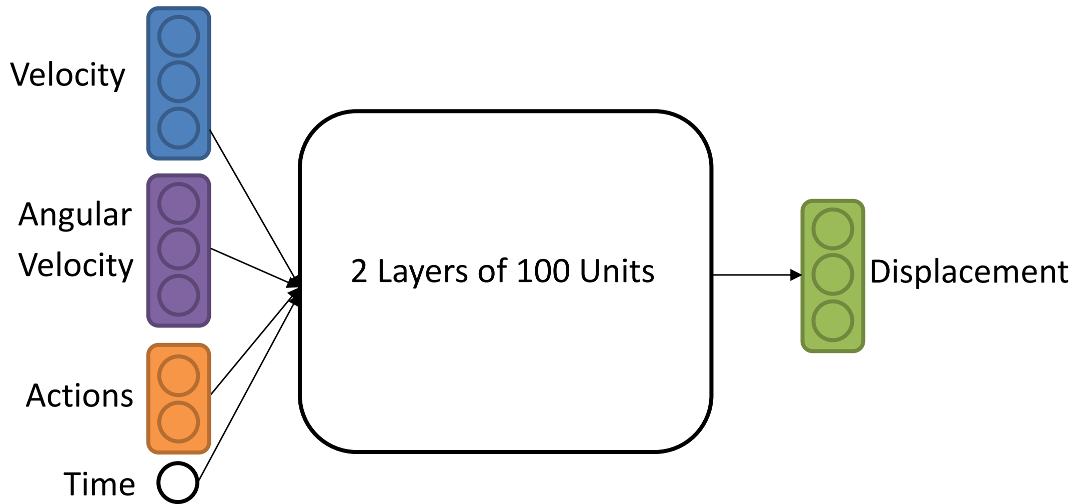


Figure 4.1: Overview of the prediction network

4.2 Neural Network Approach

Neural Networks are well suited for this problem because they sufficiently address the challenges stated above. The core idea of neural networks is that they are able to model unknown functions given enough training examples. This is the scenario presented by this problem: we are trying to approximate the game engine, and we can easily record the position of cars within the engine to obtain ample amounts of training data. Training of Neural Networks is also automated, and so with a new set of training data one can easily train a model for a different vehicle without any training data. Additionally, making predictions with a Neural Network is a very inexpensive operation.

The proposed solution is a network with 13 inputs and 3 outputs, shown in Figure 4.1. The size and number of hidden layers was chosen to be two layers of 100 units, as this was the smallest the network could be while achieving good accuracy. The dimensionality of the input is reduced by converting the velocity and angular velocity to the coordinate frame of the opponent. This means that the output from the network is also in the local coordinate frame of the opponent, and must be applied accordingly. By doing this we do not need orientation as an input. Since the cars can drift, the velocity is not only straight ahead in the car's coordinate frame, and so velocity cannot be reduced to a scalar speed. The velocity and angular velocity vectors are both input as 3 components, the steering and acceleration inputs comprise the next 2 inputs, and the last input is the time ahead

to predict. The output is a 3 component vector representing the displacement from the initial position. Note that the initial position does not need to be known or input into the network to find the displacement. Although cars primarily operate in a 2 dimensional plane, all 3 dimensions are included to allow this network to be applied to 3 dimensional scenarios.

4.3 Simulation Setup

Simulations to validate the approach are conducted in the Unity engine [17]. In Unity, we have a car that we control which we call the *master* car, which is the red car in Figure 4.2. This car represents the opponent's true state. We fully record the state of the car at each timestep, which allows us to collect training data for our network. Additionally, we simulate receiving messages from this master car by reading one of these recorded states every T seconds (recall that T is the specified message interval). We simulate the additional network parameters of latency and packet loss by adding a delay of d milliseconds before we can use a message, and adding a p percent chance that we cannot use a given message, where d and p are the latency and packet loss respectively. From this information, we apply our prediction and blending scheme to create a replica car, shown in yellow in Figure 4.2, which as a player would be the version of our opponent that we would see. These cars have no physical interaction with each other, and are only overlaid to be able to easily make comparison and quickly judge the error of the proposed algorithms. Replication schemes are independent of each other and so any number of different replicas can be run at once, each with different replication schemes in order to easily compare different algorithms.

These cars are simulated with full three dimensional physics including wheel slip. As seen in Figure 4.2, the master car is drifting slightly sideways through its turn, adding additional nuance to the prediction problem. Full physics also means that driving on uneven terrain and collisions are modeled, which will be investigated in later sections. The simulation runs at 50 frames per second, meaning our time between timesteps is $\delta t = 0.02s$.

4.4 Training

To collect training data for the prediction network, we drive the master car and record a message $s[k]$ at each timestep consisting of the state, control actions, and time, i.e., $s[k] = (\mathcal{X}[k], \mathbf{A}[k], t)$. To collect additional data, an automated driving script was run to imitate human control that inputs a random steering and acceleration input, randomly

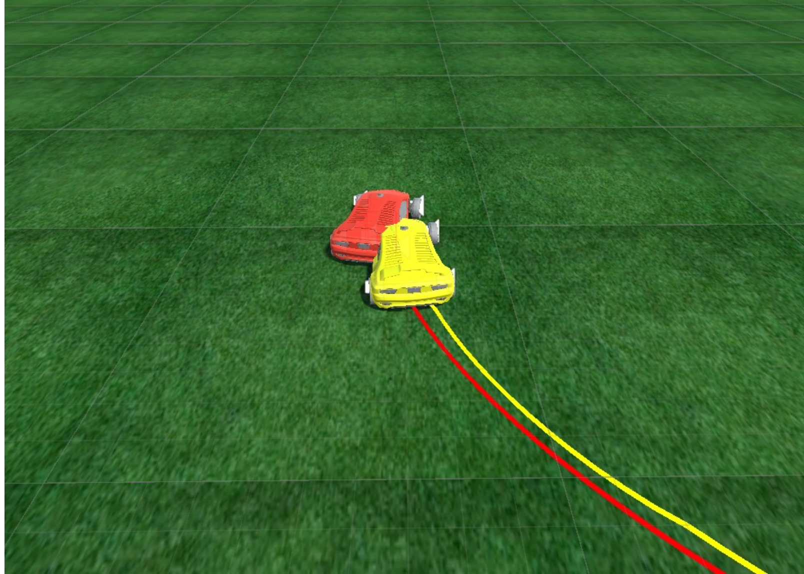


Figure 4.2: Screenshot from the Unity simulation environment

assigning a new one of each at intervals between 0 and 10 seconds. For each recorded message, $s[k]$, an input-output pair can be formed by using a vector of the velocity $\mathbf{v}[k]$, angular velocity $\boldsymbol{\omega}[k]$, and a time to predict Δt as the input. The output vector is obtained by taking the difference in position between the original position $\mathbf{r}[k]$ and the position at the frame Δt after, $\mathbf{r}[k+i]$ where $i = \frac{\Delta t}{\delta t}$. Thus we have the input-output pair $x = (\mathbf{v}[k], \boldsymbol{\omega}[k], \mathbf{A}[k], t), y = (\mathbf{r}[k+i] - \mathbf{r}[k])$.

To train the network, input-output pairs are created with values of Δt up to a maximum desired prediction time Δt_{max}. Typically, this would be slightly more than the message interval T, as this is the longest time we would expect before receiving new information. If we wish for the network to be more robust to packet loss, we can train it for double our desired message interval. Choosing a value for Δt_{max} presents a tradeoff: higher values of δt yield a more general network that is able to make predictions over a larger time window, but may reduce the accuracy of the network even at low values of Δt since the network has to learn a model to fit a much broader function. Moreover, the input-output pairs may have significantly less correlation at larger values of Δt, making training a model for larger times even more difficult.

The recorded training set consists of 1.6 million recorded messages. The amount of

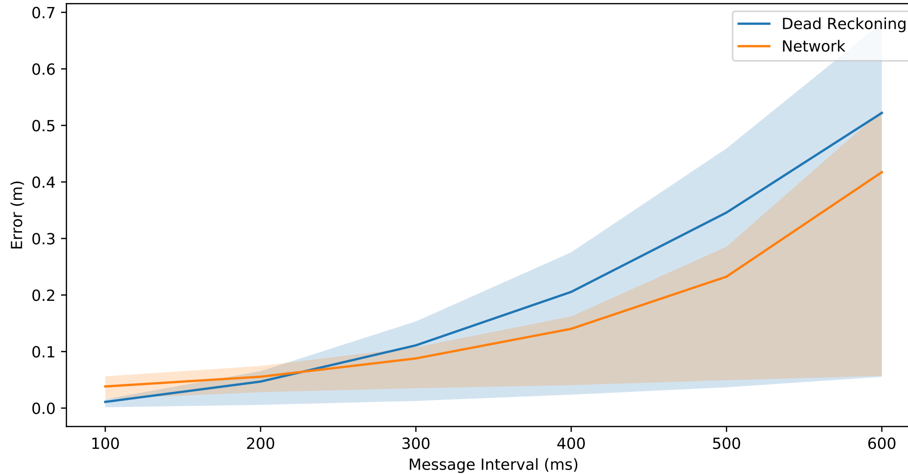
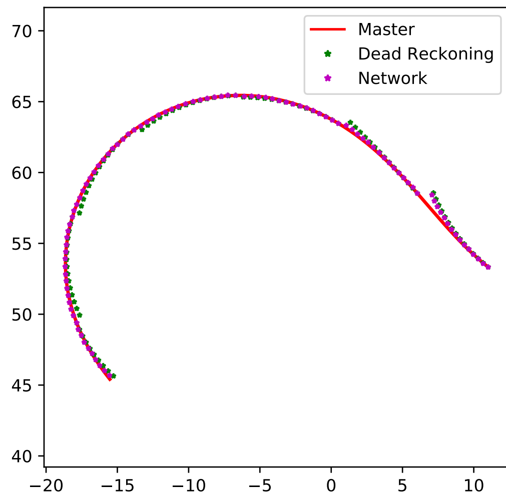


Figure 4.3: Prediction error of dead reckoning and prediction network over different message intervals

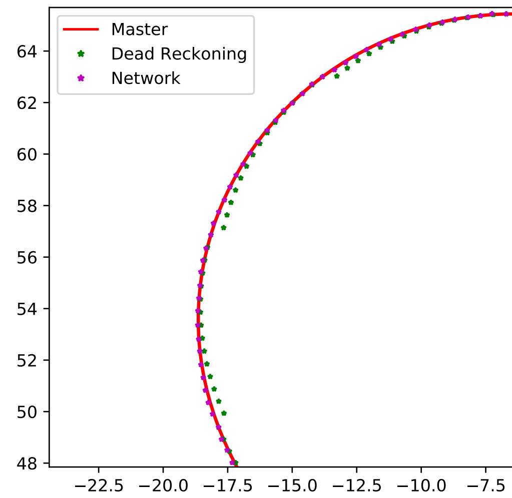
training data available is directly proportional to the maximum number of timesteps we wish to predict for i.e., total training instances is $1600000(\frac{\Delta t_{max}}{\delta t})$. The network was implemented in Python using PyTorch. Mean squared error was used as the cost function, and the Adam optimizer [20] was used for gradient descent, as it is easy to implement without fine tuning and provides good performance in practice.

4.5 Results

The baseline prediction algorithm we compare to is the discrete-time dead reckoning algorithm described in Section 3.3. Our model is trained for $\Delta t_{max} = 400ms$, which is much higher than the typical message interval used in online games. The average prediction error of both these algorithms is shown in Figure 4.3. The mean of the prediction error is shown with the solid line, while the shaded region indicates the $25th$ and $75th$ percentiles, giving a notion of the spread of the data. To give a reference for the scale of errors, the vehicles used in simulation measure roughly $2.4m$ wide and $4.7m$ long. Both algorithms have very low error for message intervals below $200ms$. What we are most interested in is the range between $200ms$ and $400ms$. Our prediction network has considerably lower average error during this portion, as well as lower spread of the error. Note that although our network is



(a) Predicted positions compared to the master path



(b) Zoomed section of master path

Figure 4.4: Predictions made using dead reckoning and the prediction network compared to the master path

only trained for message intervals up to $400ms$, it is able to maintain superior performance to the dead reckoning approach beyond this range, up to message intervals of $600ms$. These results show that our approach offers consistent predictions for higher message intervals, indicating the possibility of saving bandwidth by increasing message intervals. Moreover, our approach maintains lower error for much higher message intervals as well, and although message intervals this high are not realistically be used in practice, they indicate that our approach is more robust to packet loss as it will be able to extrapolate better from the last known message.

Examining the paths predicted by the two algorithms gives a better picture of how the two algorithms compare. A comparison of the predictions of the two algorithms is shown in Figure 4.4. Note the sawtooth pattern created by the prediction algorithms as they extrapolate and diverge further from the master path until they receive a new message. In the close up view, the dead reckoning algorithm clearly creates this sawtooth pattern of repeatedly diverging throughout the curved portion of the path, while the prediction network is able to predict a relatively smooth curve. Also note that when the network's predictions diverge near the start of the path, they do so less severely than the dead reckoning algorithm. This increased consistency in the predictions is highly desirable in creating a believable path for a replica vehicle.

Chapter 5

Path Blending

Suppose a player has a sequence of predicted states for an opponent. Simply rendering these predictions as they come will not result in a believable trajectory. There may be significant discontinuities in the path when new messages arrive, and the overall trajectory will likely not be very plausible in terms of obeying the dynamics of the car. Thus the player will need to apply some form of *blending* to the predicted states in order to believably replicate the opponent's car.

In this section, we first attempt to apply a neural network approach similar to the one in Section 4.2 to generate a blended trajectory. This attempt was unsuccessful, as the resulting solution was unpredictable and occasionally unstable. We instead use an approach based on a feedback-linearized controller used in robotics that is able to give better and more consistent results.

5.1 Problem Setup

Given a sequence of predicted states $(\hat{\mathcal{X}}[0], ..., \hat{\mathcal{X}}[k])$, we wish to produce a sequence of *blended* states $(\bar{\mathcal{X}}[0], ..., \bar{\mathcal{X}}[k])$ to replicate an opponent. While reducing $\sum_{j=k}^{k+i} \|\bar{\mathbf{r}}[j] - \hat{\mathbf{r}}[j]\|$ is desired, it cannot be used as the sole metric to evaluate potential solutions. The blended states must be accurate as well as believable. We therefore propose a generic evaluation function g, such that our goal is to produce a sequence of blended states $(\bar{\mathcal{X}}[0], ..., \bar{\mathcal{X}}[k])$ where $g(\bar{\mathcal{X}}[0], ..., \bar{\mathcal{X}}[k])$ is maximized. Note that unlike the prediction problem, the quality of a solution to the blending problem is dependent on the other blended states and thus the problem cannot be approached one timestep at a time.

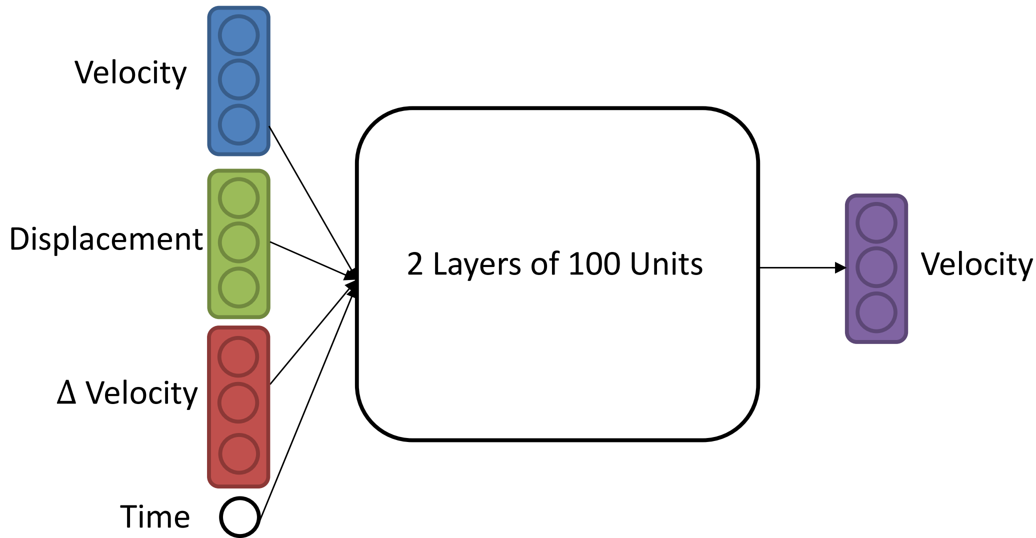


Figure 5.1: Overview of the blending network

5.2 Neural Network

Given the success of the Neural Network approach to the prediction problem, we first tried a machine learning method for the blending problem. We wish to once again leverage a neural network's ability to provide computationally efficient outputs that can be generated in real time. Creating a solution for the blending problem is more difficult because there is no definite quantity that we wish to optimize, which means we have no clear cost function or training targets to use to train our network.

5.2.1 Training

To utilize supervised learning we must first create a method to generate targets to train the network with. Again, the criteria for a good blending algorithm is largely qualitative, and so there is some amount of human discretion in generating these training instances. This method must also be automated in order to produce sufficient data for training. We choose to formulate and solve a Second Order Cone Program (SOCP) to generate training instances. An optimization problem such as this should be able to produce desireable trajectories given proper formulation, but it is too computationally expensive to be able to run at every timestep in real time. Thus, we train a neural network to try and learn the solution to this SOCP.

To formulate this SOCP, we must attempt to quantify the evaluation criteria for the blended path. Recall that we want our blended path to be accurate as well as smooth; we use the positional error as the objective function to minimize, and add constraints to ensure smoothness of the path. Since the acceptable smoothness of the path is up to human discretion, we control smoothness via a soft constraint penalizing acceleration magnitudes greater than a specified $\mathbf{a}_{max}$. For a segment of the master trajectory between timesteps k and $k+i$, we can formulate the SOCP as follows:

$$\begin{aligned}
\min \quad & \sum_{j=k}^{k+i} \|\mathbf{r}_j - \bar{\mathbf{r}}_j\|_2^2 + \beta\theta \\
\text{s.t.} \quad & \mathbf{r}_{j+1} = \mathbf{r}_j + \mathbf{v}_j \delta t, \\
& \theta \geq \|\mathbf{v}_{j+1} - \mathbf{v}_j\|_2^2 - a_{max}, \\
& \theta \geq 0
\end{aligned}$$

This is a convex optimization problem that can be solved using existing solvers. In our case, we used CVXPY [10]. The parameters a_{max} and β control how smooth the resulting path will be: a_{max} governs how quickly the trajectory can change direction without being penalized, and β dictates how much the trajectory is penalized for exceeding a_{max}. Larger values of β mean the trajectory is less likely to exceed a_{max} and will be smoother at the cost of potentially greater positional error. These parameters were manually tuned by solving the blending problem for segments of predicted paths and examining the resulting blended trajectories.

For our network, we first try to implement it without considering orientation to evaluate its potential as a solution. If successful, orientation can be added to the network or calculated using another method afterwards. We are given a message $s[k]$, a predicted state $\hat{\mathcal{X}}[k+i]$, and a time to predict, Δt. An overview of the network is shown in Figure 5.1. For inputs to our blending network we require the current velocity $\mathbf{v}[k]$, the displacement between the current and predicted position $(\hat{\mathbf{r}}[k+i] - \mathbf{r}[k])$, the difference in current and final velocity $(\hat{\mathbf{v}}[k+i] - \mathbf{v}[k])$, and the time to predict Δt. The output of the network is a velocity $y = \bar{\mathbf{v}}[k+j]$ to apply to the current replica, i.e., $\bar{\mathbf{r}}[k+j+1] = \bar{\mathbf{r}}[k+j] + \bar{\mathbf{v}}[k+j] \cdot \delta t$.

For training this network, an upper limit on the time to blend Δt_{max} must be specified, presenting similar tradeoffs to the selection of the maximum time to predict for the prediction network. To use the SOCP, we sample a random state of the master car and predict a trajectory that is as long as Δt i.e., $(\mathcal{X}[k], ..., \mathcal{X}[k+i]), i = \frac{\Delta t}{\delta t}$. We then generate states near the start of the path by applying small disturbances to the starting position,

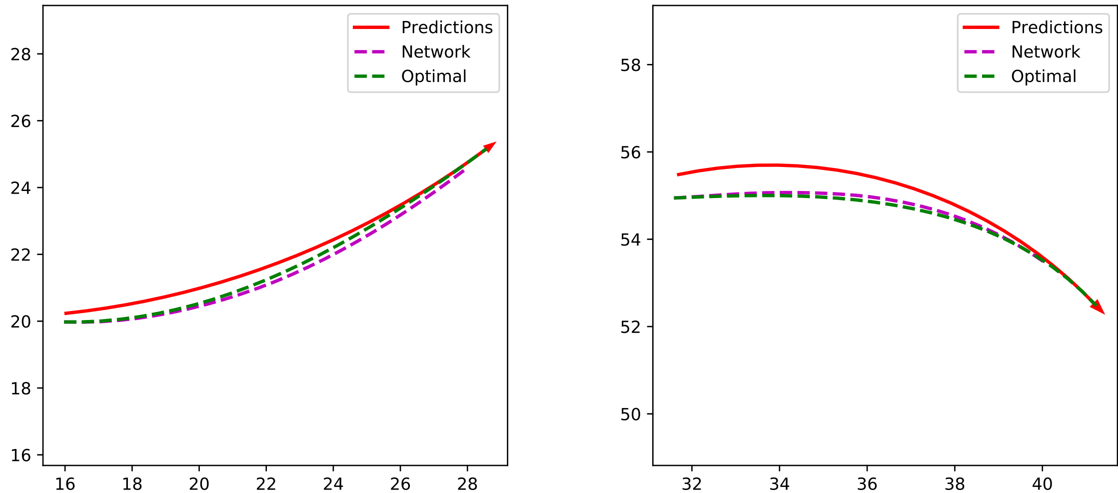


Figure 5.2: Comparison of optimal path from the SOCP to the path generated by the trained network

velocity, and orientation to mimic the offset between the replica and the predicted path when a new message is received. We can then use the solver to generate a sequence of *optimally* blended points to use as training targets. Note that while these solutions are optimal with respect to our SOCP formulation, there is no truly optimal solution to the blending problem as the quality of a solution is ultimately up to human judgement. The SOCP solver was used to generate 2.1 million training instances. Similar to the prediction network, the blending network was trained using the Adam optimizer with Mean Squared Error as the cost function. A comparison of the paths generated by the network and the optimal paths from the SOCP is shown in Figure 5.2. Note that while the network does a good job of replicating the path, there is some error in the final position of the blended path, while the SOCP solutions have virtually no error in the final position.

5.2.2 Implementation

Suppose we are at timestep k with a replica car with state is $\bar{\mathcal{X}}[k]$ and we receive a new message $s[k]$. There are i timesteps until we expect to receive the next message, i.e, $i = \frac{T}{\delta t}$. To implement the blending network, we use the algorithm described in Algorithm 1.

Algorithm 1: BLENDINGALGORITHM

```
1  while replicating do
2      s[k] = LATEST MESSAGE
3      i = T/δt
4      r̂[k+i], v̂[k+i] = PREDICT(s[k], i)
5      j = 1
6      while no new message do
7          v̄[k+j] = BLEND(v̄[k], r̂[k+i] − r̄[k], v̂[k+i] − v̄[k], j · δt)
8          r̄[k+j] = r̄[k+j−1] + v̄[k+j] · δt
9          j++
10         if j > i then
11             i = i + T/δt
12             r̂[k+i], v̂[k+i] = PREDICT(s[k], i)
```

In simpler terms, when a message is received we predict the position and velocity of the opponent at the time we expect to receive a new message. We then apply the blending algorithm at each frame. Line 10 checks if we have passed the time we predicted up to without receiving a new message, which can happen in the event of packet loss. Notice that our prediction approach described in Section 4.2 does not explicitly predict a velocity. In our implementation, we approximate the velocity using $\hat{\mathbf{v}}[k+i] = (\hat{\mathbf{r}}[k+i] - \hat{\mathbf{r}}[k+i-1])/\delta t$.

5.3 Path Tracking

As an alternate blending algorithm, we leverage the fact that the motion we are trying to replicate is based on a car's dynamics and borrow concepts from mobile robotics. Path tracking is widely used in mobile robotics, particularly for wheeled robots, to allow robots to follow a specified path in the presence of error from sensors, actuators, or the environment. In our scenario, error in our position comes from the deviation between the predicted position and the true position. Applying a path tracking controller ensures that the resulting path is smooth and obeys the dynamics of a wheeled vehicle. The path tracking controller also has the advantage of using feedback control to ensure the system is stable.

5.3.1 Formulation

For our path tracking controller, we use the controller proposed in [30]. For completeness, we give a brief derivation of the controller below. We model the opponent as a unicycle robot where we can freely control the velocity and heading of the robot. The longitudinal, lateral, and heading errors at time k are given by $\epsilon[k] = [\epsilon_X[k], \epsilon_L[k], \epsilon_H[k]]^T$ respectively, and are calculated using the Euclidean distance to the predicted state at time k, $\hat{x}_k$. With the linear and angular velocities at time k as $v[k]$ and $\omega[k]$ respectively, and the time between frames as δt, the resulting error dynamics are:

$$\begin{bmatrix} \epsilon_L[k+1] \\ \epsilon_H[k+1] \end{bmatrix} = \begin{bmatrix} \epsilon_L[k] \\ \epsilon_H[k] \end{bmatrix} + \delta t \begin{bmatrix} v[k] \sin \epsilon_H[k] \\ \omega[k] \end{bmatrix}$$

where $v[k]$ and $\omega[k]$ are inputs and $\omega[k]$ is given by the controller. If we assume the velocity $v[k]$ is constant, and let $z_1[k+1] := \epsilon_L[k]$, $z_2[k] := v[k] \sin \epsilon_H[k]$, and $\eta[k] := v[k] \cos \epsilon_H[k] \omega[k]$, the system becomes:

$$\begin{aligned} \begin{bmatrix} z_1[k+1] \\ z_2[k+1] \end{bmatrix} &= \begin{bmatrix} 1 & \delta t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} z_1[k] \\ z_2[k] \end{bmatrix} + \delta t \begin{bmatrix} 0 \\ v[k] \cos \epsilon_H[k] \omega[k] \end{bmatrix} \\ &= \begin{bmatrix} 1 & \delta t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} z_1[k] \\ z_2[k] \end{bmatrix} + \delta t \begin{bmatrix} 0 \\ \eta[k] \end{bmatrix} \end{aligned}$$

We choose a proportional controller with the form $\eta[k] = -\gamma_1 z_1[k] - \gamma_2 z_2[k]$ with $\gamma_1, \gamma_2 > 0$, which yields the following stable, closed-loop system:

$$\begin{bmatrix} z_1[k+1] \\ z_2[k+1] \end{bmatrix} = \begin{bmatrix} 1 & \delta t \\ -\delta t \gamma_1 & 1 - \delta t \gamma_2 \end{bmatrix} \begin{bmatrix} z_1[k] \\ z_2[k] \end{bmatrix}$$

Solving for $\omega[k]$ gives the following relation to govern the heading:

$$\omega[k] = \frac{-\gamma_1 \epsilon_L[k] - \gamma_2 v[k] \sin \epsilon_H[k]}{v[k] \cos \epsilon_H[k]}$$

This formulation does not control velocity, so we must introduce another equation to govern our replica's velocity. Borrowing from the linear blending described in Section 3.4, we will use a similar equation to govern the velocity for this approach. We define $\epsilon_{total} = \epsilon_X[k] + \epsilon_L[k]$, which lets us write $\|\mathbf{v}[k]\| = \|\epsilon_{total}\| / \gamma_3$. This gives us three parameters to tune, γ_1, γ_2, and γ_3.

5.3.2 Implementation

The integration of the path tracking controller uses the same overall algorithm as the blending network, shown in Algorithm 1. To use the heading and velocity equations described above to generate a single blended velocity $\bar{\mathbf{v}}[k+j]$, we rotate the previous blended velocity $\bar{\mathbf{v}}[k+j-1]$ by $\omega[k]\delta t$ and set its magnitude as $\|\bar{\mathbf{v}}[k+j]\| = \|\epsilon_{total}\|/\gamma_3$.

Some edge cases need to be accounted for during implementation of this algorithm. Firstly, with our choice of velocity control, we need to consider the case where our blended position is ahead of the predicted position, which can happen in cases of extreme deceleration. If this case is unhandled, the replica will accelerate away from the predicted position indefinitely. In our implementation we handle this case by simply halving the speed of the replica, allowing time for the predictions to catch up to the replica. Another edge case is that if the heading error is outside of $+/-90°$, the replica will end up tracking the path in the opposite direction, converging to $180°$ of heading error. To remedy this, we clamp the heading error to $+/-80°$, which also helps avoid numerical errors from dividing by 0 or close to 0 in the heading equation.

5.4 Results

In this section we present results comparing our proposed approaches to the established state of the art methods including the linear blending approach described in Section 3.4 and the projective velocity blending technique described in Section 3.5.

5.4.1 Neural Network Results

Preliminary results for the blending network were obtained by training the network for a Δt_{max} of $500ms$ and simulating using a message interval of $T = 500ms$ with no latency or packet loss. These simulations were run with predictions made using the existing state-of-the-art dead reckoning algorithm described in Section 3.3 to reduce potential variables. Our neural network approach is compared to the linear blending approach described in Section 3.4 with $\lambda = 0.4$. Initial results when implementing the network are shown in Figure 5.3 and show good performance along smooth curves, but show noticeable error at high speed turns or other areas where the predictions would have greater error. More importantly however, the blended path from the network eventually diverges completely from the master path. While the errors in the final positions of the blended path appeared

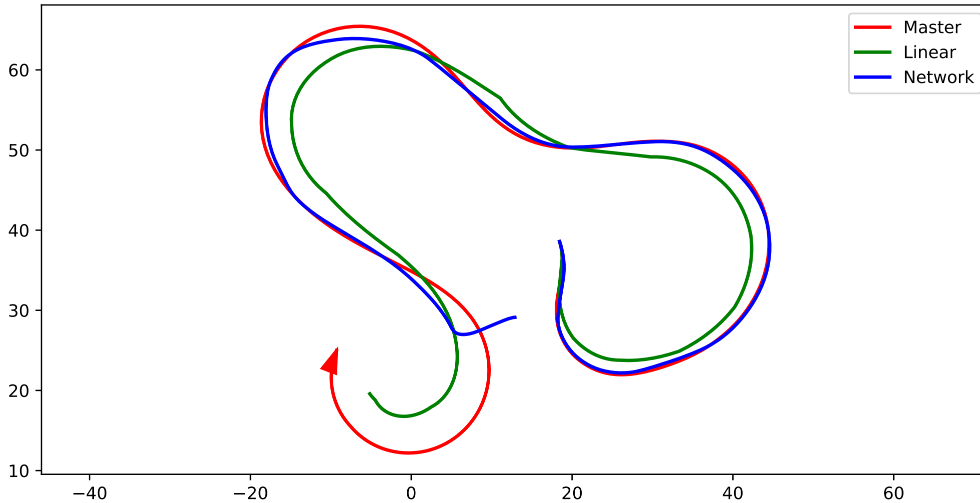


Figure 5.3: Implementation of the blending network when not updating the velocity with new messages

insignificant in Figure 5.2, they result in complete divergence when they are allowed to accumulate over multiple predicted segments. With no explicit feedback control mechanisms to mitigate error, there are no guarantees that the blended path will be able to consistently track the predictions. The black box nature of neural networks also makes the behaviour of this approach difficult to predict. Ultimately, due to these shortcomings along with the manual tuning required in the training process, this approach was deemed unsatisfactory and was not investigated further.

5.4.2 Path Tracking Results

The state-of-the-art we compare to initially is the discrete linear blending approach described in Section 3.4 with $\lambda = 0.4$. With initial parameters of $\gamma_1 = 80, \gamma_2 = 20, \gamma_3 = 0.2$ obtained through trial and error, initial results are promising, showing much smoother paths than the existing state-of-the-art with considerably less error. A γ_3 of 0.2 corresponds to roughly half the delay of the the state-of-the-art algorithm. These results are shown in Figure 5.4. The direction of the master path is indicated with an arrow, and a point is marked on each path at a timestep where a message was sent from the master. This

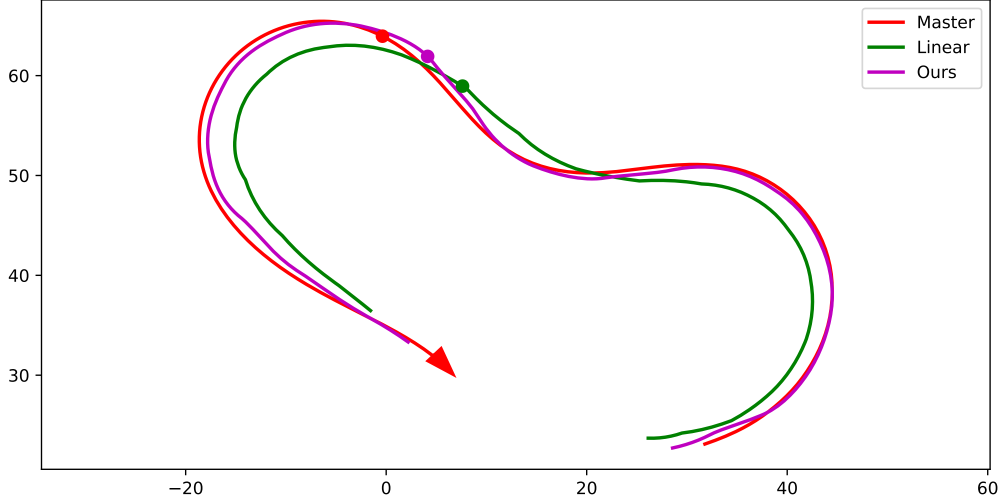


Figure 5.4: Results of path tracking blending with untuned parameters and a message interval of $300ms$

marked point shows that using $\gamma_3 = 0.2$ results in roughly half the delay compared to the linear blending approach. Our approach also replicates the path of the master considerably better than the linear blending technique.

Although it is not the only metric to evaluate blending algorithms, examining the positional error between the algorithms summarizes the performance nicely. We will examine both the absolute error, as well as what we call the *adjusted error*, which is calculated by comparing the error between blended position $\bar{\mathbf{r}}[k]$ and the actual time shifted position $\mathbf{r}[k - \lambda/\delta t]$, or in the case of path tracking blending, $\mathbf{r}[k - \gamma_3/\delta t]$. This is meant to somewhat eliminate the impact of the different delay values used and compare how closely the algorithms follow the true path, independent of delay, by comparing the blended points with the points on the master path that corresponds to that amount of delay. The averages of these two errors are shown in Table 5.1. While the path tracking blending is expected to have lower absolute error due to running with a smaller delay, it also has significantly lower adjusted error, suggesting that it also does a better job of following the master car's original path. While this metric is not precise by any means, its results combined with the smoothness of the path upon visual inspection are encouraging for the potential of path tracking blending.

Table 5.1: Average errors for linear blending and path tracking blending.

Message Interval	100ms		300ms		500ms	
	mean	std dev	mean	std dev	mean	std dev
Linear	5.25	3.02	6.19	2.27	5.54	3.17
Path Tracking	2.71	1.53	3.27	1.16	2.97	1.68
Linear Adjusted	1.17	0.99	1.75	1.12	1.49	1.21
Path Tracking Adjusted	0.33	0.20	0.57	0.34	0.72	0.64

5.4.3 Parameter Tuning

While the initial results with randomly selected parameters are promising, this approach has further potential available by tuning its 3 parameters. Through manual experimentation, it was found that lower values of γ_1 and γ_2 were desirable as they produced smoother paths, but setting these too low resulted in greater error. A lower value of γ_3 is desired as it directly correlates to less delay in the blending algorithm. However, it was found that for higher message intervals there was a firm lower limit for how low γ_3 could be while maintaining stability. Thus we can select γ_3 based on our desired message interval as the only parameter we tune by hand. In our case, for a desired message interval of $250-300ms$, we selected $\gamma_3 = 0.1$. We can then use a simple grid search to find our tuned values of γ_1 and γ_2 by simulating our path tracking blending algorithm with different values of γ_1 and γ_2 and calculating the average errors of each combination. We want to select values that result in low error, but we also wish to choose low values of γ_1 and γ_2. Therefore, we selected the pair of γ_1 and γ_2 with the lowest sum whose average error was within 20% of the minimum average error. This gave us our tuned values of $\gamma_1 = 130, \gamma_2 = 10, \gamma_3 = 0.1$.

The tuned algorithm is then compared to the dead reckoning and linear blending approach described in Sections 3.3 and 3.4, and the projective velocity blending technique described in Section 3.5. Figure 5.5 shows the algorithms performing with a message interval of 200ms and no packet loss or latency. The master vehicle is traveling from right to left, and a point has been marked on the path of each replica at the same timestep where a message was received from the master. This point shows the differences in how delayed each replica is: the linear blending approach causes noticeable delay, while the PVB replica runs next to the master, with our approach running slightly behind. These results illustrate the tradeoff between delay of the replica and smoothness of the trajectory. The linear blending approach runs significantly behind, but maintains a very smooth trajectory; its trajectory is even too smooth, filtering out some of the detail of the master

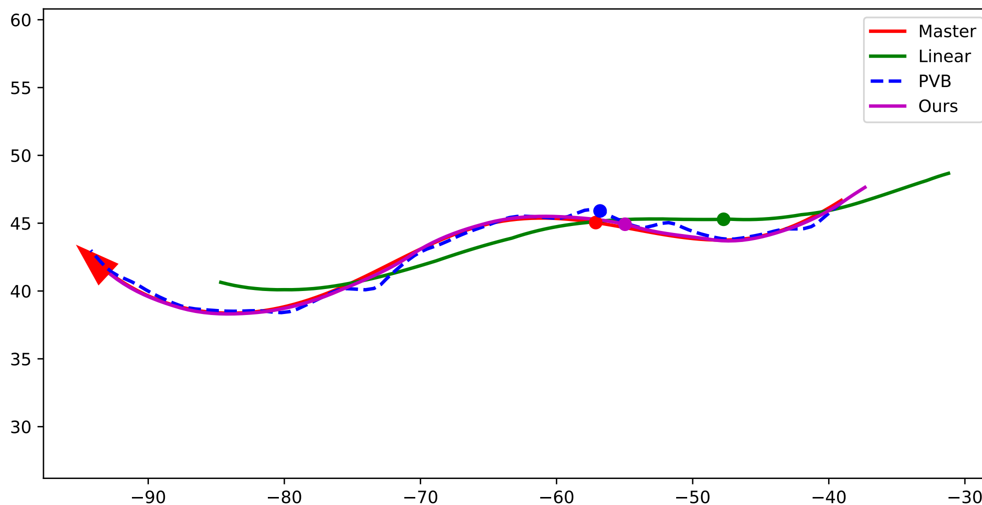


Figure 5.5: Results of path tracking blending with tuned parameters and a message interval of 200ms

path. The PVB approach on the other hand, has no delay behind the master car at the cost of some irregularities in the path when it must double back when its extrapolation is incorrect. Our approach sacrifices a small amount delay for considerably better replication of the master path. Our approach is also robust to worse network conditions, as shown in Figure 5.6. At these network conditions, the flaws of all the replication algorithms are magnified. At this message interval, even the conservative linear blending approach shows jarring changes in direction. Our blending approach shows some divergence from the master path, but it is masked to a much better degree than the PVB approach. Figure 5.7 shows even worse network properties, with a message interval of 300ms and 50% packet loss. Our algorithm still performs adequately at these worst-case network properties, while the other two algorithms show significant issues. The effect of latency is not compared here as it affects all algorithms in the same way by making them perform as if the message interval has the latency added to it, and including it as a variable would only obfuscate the results.

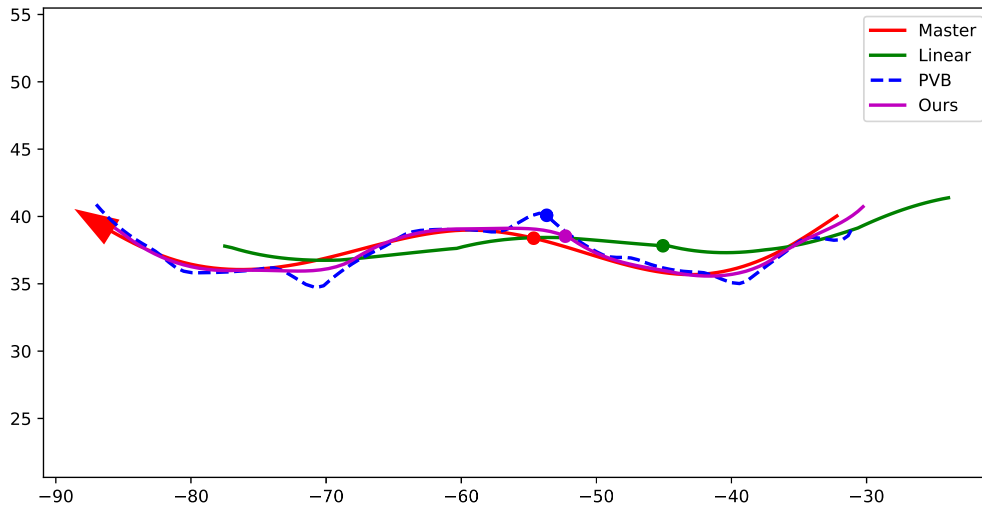


Figure 5.6: Results of path tracking blending with tuned parameters and a message interval of 300ms

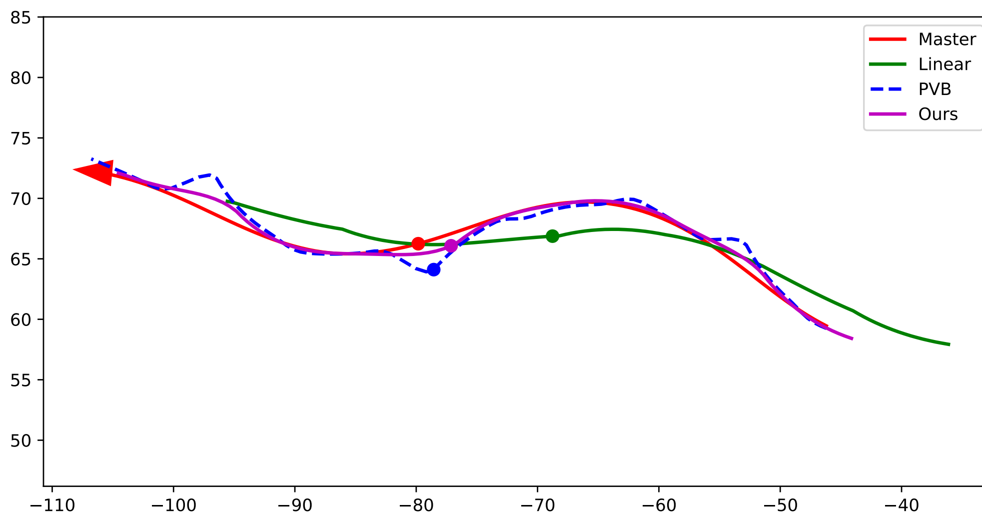


Figure 5.7: Results of path tracking blending with tuned parameters and a message interval of 300ms and a drop rate of 50%

Chapter 6

Collisions

A large challenge, particularly in networked games, is how to deal with collisions. In networked games, we do not have a perfect model of our opponents, but as shown in previous sections, with a good prediction and blending scheme we can create a decent approximation of our opponents. Maintaining an accurate and believable model of our opponents is made more difficult in the presence of collisions, however, since small differences in the impact point can have a dramatic effect on the outcome of the collision. In some cases, small differences in position between the replica and the true position can make the difference between one colliding with an obstacle and the other not colliding. This problem is further complicated by considering collisions between multiple other players, where all agents involved in the collision are replicas whose exact position is not known.

We are restricting our investigation to only look at collisions between a replicated opponent and a static obstacle. Even in this case, there are significant challenges presented. If the replica does not have physics enabled and collisions are not otherwise accounted for, the replica will pass through obstacles without colliding until a new message is received from the master car after a collision has occurred. This can be especially jarring for the player, as this blatantly violates the physics of the game engine. Alternatively, if the replica does have physics enabled and we attempt to implement our blending algorithm in the presence of obstacles, we may still be attempting to blend the replica to a predicted point that is inside an obstacle. This disagreement between the blending algorithm and the physics engine can result in extreme cases where the replica is ejected from the collision with a very unrealistic angle and velocity. This is equally jarring for the player and is also undesirable.

One approach that is used in practice is to disable predictions when a collision is

detected by the replica, and allow local physics to control the replica for some tuned period of time before merging back into the prediction scheme [9]. One shortcoming of this is that it depends on the position of the replica, which may be inaccurate. As discussed above, small differences in position between the replica and the master can have a significant impact on the collision response. We would like to explore the possibility of predicting a collision and determining a response based on the predicted positions of the replica, since these should be much closer to the true position of the car we are attempting to replicate. This is a very nuanced problem with many implementation pitfalls, and so our goal will be mainly to establish a proof of concept that predicting collisions based on predicted positions of the replica is reliable and reasonably accurate.

6.1 Problem Setup

Given a message from an opponent $s[k]$ and an obstacle P, determine if the opponent will collide with the obstacle. If not, perform prediction and blending as described in Chapters 4 and 5. If the opponent will collide, generate a series of believable states $\bar{\mathcal{X}}[k], ..., \bar{\mathcal{X}}[k+i]$ to approximate the true states, $\mathcal{X}[k], ..., \mathcal{X}[k+i]$, after collision. Success of an approach is evaluated similarly to the blending results, with some emphasis placed on Euclidean error while also considering the believability of the replication. Assume we can override the implemented prediction and blending mechanisms when a collision is detected, thus there is no requirement for the solution here to utilize the previous algorithms. The solution for this problem should, however, be able to smoothly return to the regular prediction and blending after the car is no longer affected by the collision dynamics.

6.2 Solution Approach

The problem presented in the previous section is to approximate the complex physics of the game engine, similar to the prediction problem, but this time including interactions with static obstacles. With our model of the game engine, this still looks like $\mathcal{X}[k+1] = \mathcal{B}(\mathcal{X}[k], \mathbf{A}[k])$, only this time the function $\mathcal{B}$ also encompasses static collisions. Unlike the trajectory prediction problem, player actions do not play a large role in this case, as the control inputs of the car have little impact on its collision response for many timesteps after the collision.

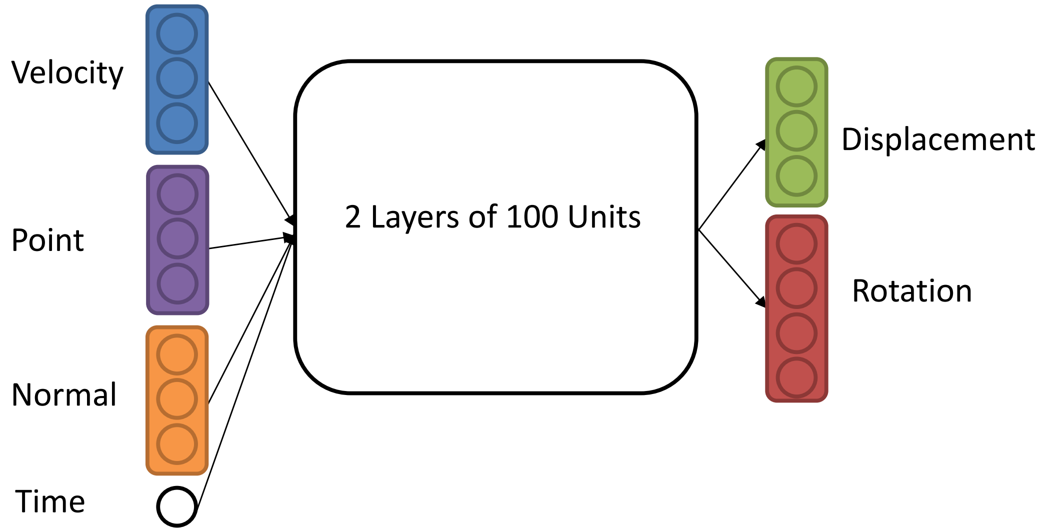


Figure 6.1: Overview of the collision network

6.2.1 Neural Network

We applied a neural network to this problem as well. In the Unity engine, collisions are calculated based on a collision point and a collision normal. We also need to input the velocity of the vehicle at the point of collision in order to calculate the collision response. Vehicle parameters such as mass and friction coefficients are also necessary for calculating a collision response, but in our case these will be captured implicitly within the network. Our goal for this section is to establish a proof of concept for being able to predict collision response using a neural network, and thus creating an easily generalizable model by including such vehicle parameters as inputs is not a priority. The orientation of the car is important in creating a believable trajectory, and cannot be easily estimated from a given path and so we must include orientation as an output along with displacement. Similar to the previous networks, we must also include a time from the collision that we wish to predict. The dimensionality of the inputs and outputs is similar in magnitude to the prediction problem described in Section 4.2 and so we will employ a similar network with 2 hidden layers of 100 units. An overview of the network is shown in Figure 6.1. We can represent a collision at timestep k and letting by the point of the collision $\mathbf{c}[k]$ and the normal to the surface at the collision $\mathbf{n}[k]$. Thus we may format our inputs and outputs as $x = (\mathbf{v}[k], \mathbf{c}[k], \mathbf{n}[k], \Delta t), y = (\Delta\bar{\mathbf{r}}[k], \bar{\mathbf{q}}[k+i])$, where $\Delta\bar{\mathbf{r}}[k] = \bar{\mathbf{r}}[k+i] - \bar{\mathbf{r}}[k]$ and $i = \frac{\Delta t}{\delta t}$.

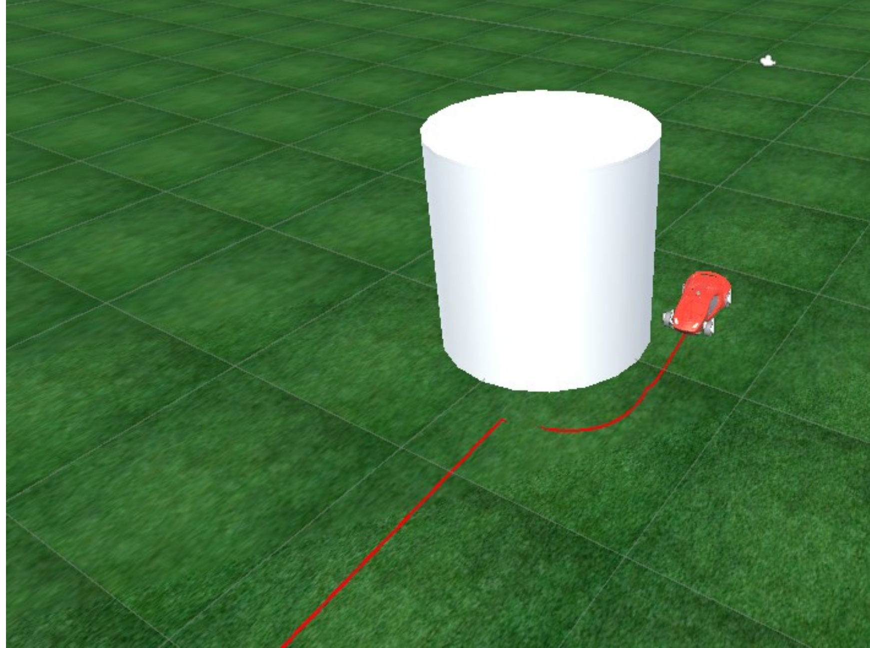


Figure 6.2: Screenshot of a collision with a cylindrical obstacle in Unity

6.2.2 Training

To train and test collisions, a static cylindrical obstacle was placed in the Unity environment. To automatically generate collision data, a script was run that repeatedly launches the master car at the obstacle at random angles and directions with a random speed within the typical range of the car. Since collisions are resolved using a point and a normal, the overall shape of the obstacle is not important and by using a circular obstacle, we are able to easily capture all collision angles. A collision with this obstacle is shown in Figure 6.2. The path of the master car throughout the collision is shown in red, and examining it reveals a few potential challenges. During this process we are able to record the state of the master car at every timestep, as well as whether or not a collision is occurring in a given frame, and the point and normal of any detected collisions. As with the previous networks, a maximum desired time to predict Δt_{max} must be specified, presenting the same tradeoff between generality and accuracy.

First note that after the collision, the car is facing backwards. Given the parameters of the vehicle and the collision surface, this is how most collisions look; the car will rotate

away from the collision surface and its momentum will carry it through as it slides and rotates around, where eventually the wheels will grip and it will roll slowly backwards. The slight disconnect in the path near the impact point is due to the car rotating slightly in relation to the ground plane, causing the trail left by the car to clip into the ground slightly. Also notice in Figure 6.2 that after the initial collision the car impacts the obstacle again slightly before settling into rolling backwards. This is problematic for collision prediction, as our model is only trained for impacts while driving, and attempting to use our network to predict a response from this secondary collision may result in error due to predicting outside of what the network is trained for. Thus, in our implementation we must account for these additional impacts. The overall collision response takes up to 2 seconds to establish grip again and settle into smooth, controlled motion. Similar to before, the network was trained using mean squared error and the Adam optimizer.

6.2.3 Implementation

When training the network, it was found that the collision network could be trained for up to $500ms$ while maintaining reasonable accuracy. Additionally, the blending scheme is specific to controlled driving motion of the vehicle and thus is not suitable for use in a collision response. However, during the initial collision response, the path of the master may not be smooth and so smoothness of the replica is not as important, thus we can simply use the predictions from the collision network as the position of the replica, with no blending applied.

With the collision network valid for up to $500ms$, we still have up to $1.5s$ of collision response remaining to predict. Simply applying the prediction scheme from before at this point yielded unsatisfactory results, as the car is still sliding uncontrollably at this point in the collision response in a way the prediction network was not trained for. A natural progression would be to try and train the prediction network to handle a wider range of motions, including this uncontrolled sliding. Doing so yielded better post-collision predictions, but resulted in worse predictions in cases with no collisions where the predictions would slide with no throttle applied. Thus another network, dubbed the *sliding network* was trained with identical structure to the prediction network, using data from this post-collision sliding period up to $2000ms$ after the collision. After this sliding period, we can return to the normal prediction and blending scheme.

Our overall replication scheme with collisions included is shown in Figure 6.3 for determining the blended position of the vehicle at time $k + i$, where $s[k]$ is the most recent message received from the master. Upon receiving a message, we first check if we have

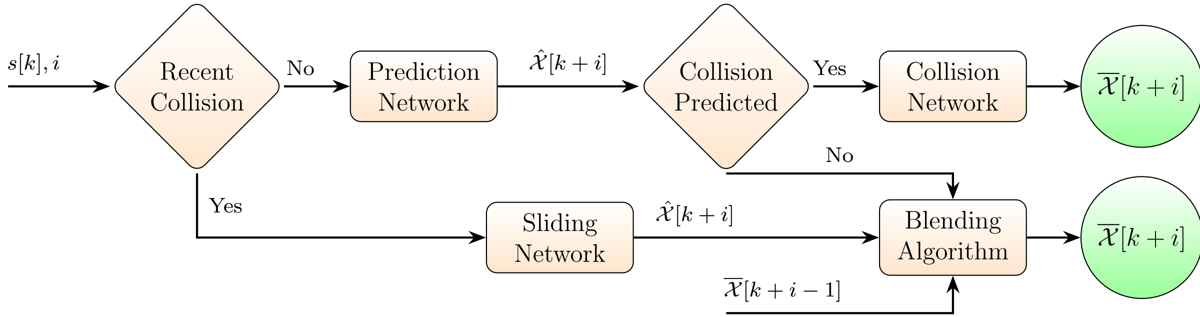


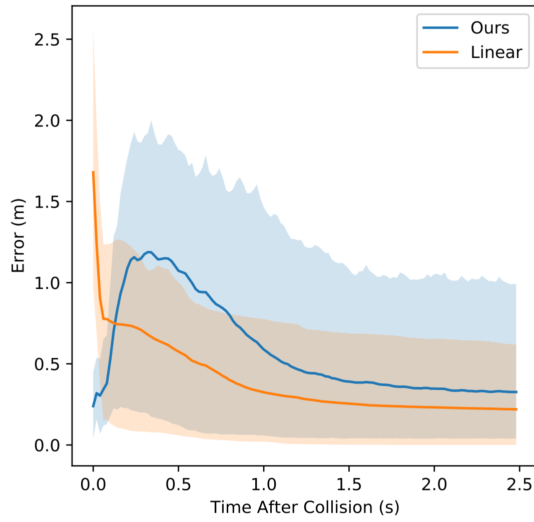
Figure 6.3: Flow diagram of replication process at timestep $k+i$ with collisions included

recently collided with an obstacle (within $2000ms$ of a collision in our implementation). If so, we use the sliding network to obtain a predicted position of the vehicle at time $k+i$, then apply our blending algorithm to find the blended position $\overline{\mathcal{X}}[k+i]$. If not, we use the prediction network, then cast a box between predicted positions $\hat{\mathcal{X}}[k+i-1]$ and $\hat{\mathcal{X}}[k+i]$ and determine if that collides with an obstacle. If a collision is detected, we obtain our blended position $\overline{\mathcal{X}}[k+i]$ directly from the collision network. If no collision is detected, we apply our blending algorithm to the predicted state obtained from the prediction network to obtain $\overline{\mathcal{X}}[k+i]$. Note that the blending algorithm also takes as input $\overline{\mathcal{X}}[k+i-1]$.

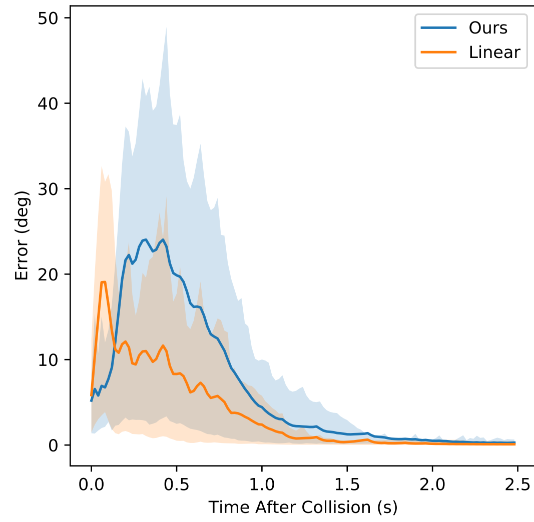
6.3 Results

In this section, our algorithm is compared to the state of the art dead reckoning and linear blending described in Sections 3.3 and 3.4 with local physics enabled on the replica car. with no considerations made for collisions, this scheme results in the replica overshooting around the obstacle or jittering around the point of collision as the game engine attempts to reconcile the replica colliding with an obstacle and the blending algorithm trying to force the replica to a prediction that is inside the obstacle.

A summary of our algorithm's performance compared to the linear blending approach for message intervals of $100ms$, $250ms$, and $500ms$ is shown in Figures 6.4, 6.5, and 6.6 respectively. To characterize the average performance, around 100 random collisions were simulated using the same script that was used to generate training data. For consistency in these trials, the first message after the collision was forced to be sent T milliseconds after the collision for each message interval T. This way, these results can be seen to loosely represent the worst-case performance for a given message interval. During simulation, we experienced some implementation issues with our approach that were unrelated to our

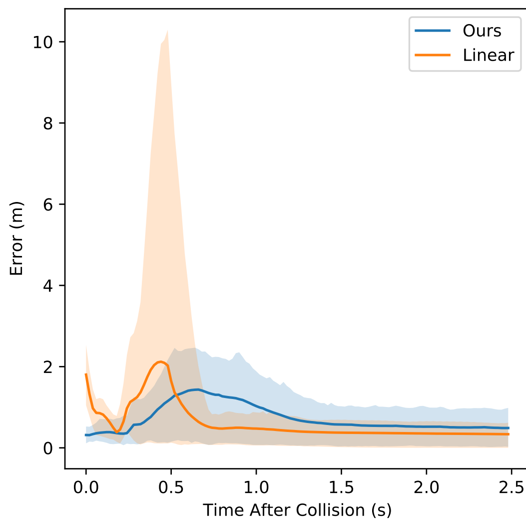


(a)

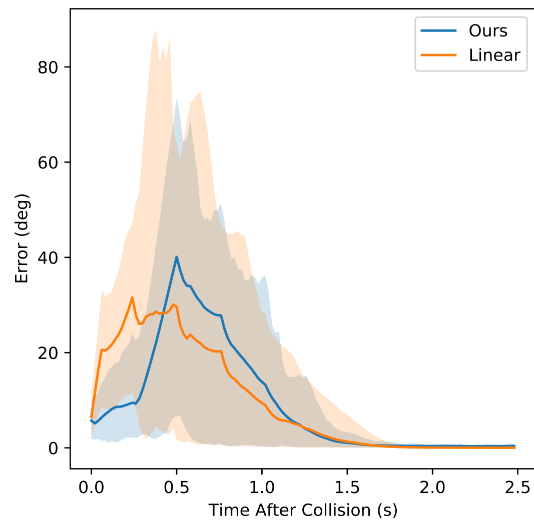


(b)

Figure 6.4: Position and Rotation error with a message interval of $100ms$



(a)



(b)

Figure 6.5: Position and Rotation error with a message interval of $250ms$

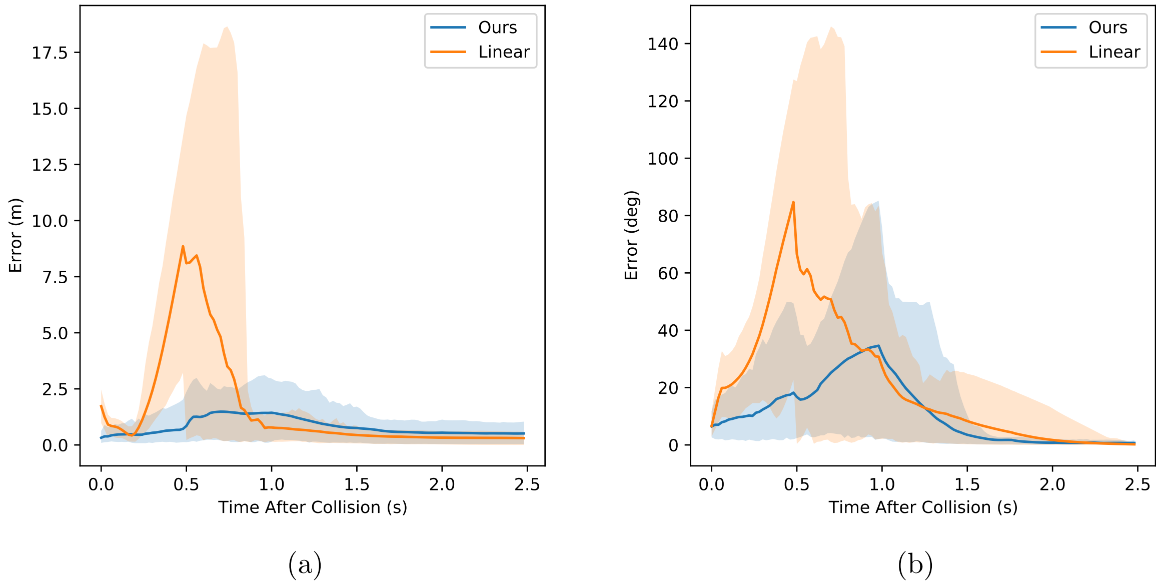
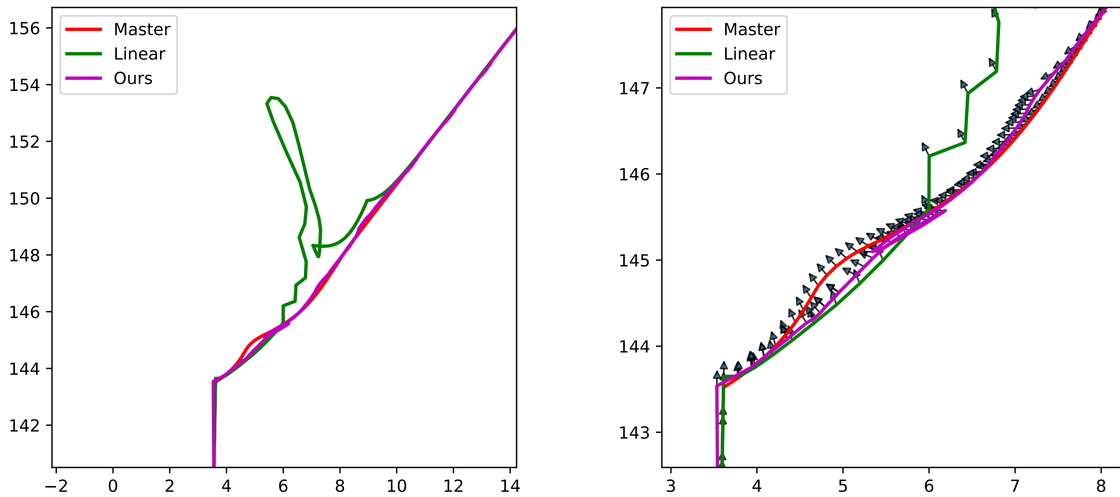


Figure 6.6: Position and Rotation error with a message interval of 500ms

solution. Occasionally, the game engine's collision detection system would fail to report a collision. The other source of issues was our use of a local Python server to run the neural network. Sometimes the communication between the Python server and Unity was delayed by a frame or two, which resulted in the simulation using the predicted state of the previous collision. Roughly 25% of trials experienced these issues, and were not included in these results.

In the figures, the mean error is plotted as a solid line, and the shaded region represents the 5th and 95th percentiles of error. At low message intervals, i.e., 100ms, both algorithms have low positional and rotational error, but the linear blending approach performs very well in both categories. At higher message intervals, which is where we are more interested, our approach shows some advantages over the linear blending approach. At message intervals of 250ms and above, the linear blending approach starts to show the overshooting that was mentioned earlier, as shown by the high peaks in the 95th percentile of positional error. At message intervals of 250ms and 500ms, our approach has lower peak positional error for both the 95th percentile and the mean. At message intervals of 500ms, our approach also shows lower rotational error. This is an extreme interval that would not be used in practice, but could arise if packet loss occurs during a collision at a lower



(a) Paths of master compared to replicas during collision

(b) Close up view of impact point with orientation shown

Figure 6.7: Paths of Master compared to Linear and Our replication schemes with a message interval of $500ms$

message interval. Overall, our approach demonstrates much more consistent replication, with fewer immersion-breaking jumps and overshooting.

Examining the paths of the replicas during a collision with a send rate of 500ms demonstrates the advantages of our algorithm, while also revealing some shortcomings. The paths of the linear replica and our replication scheme compared to the master car during a collision in Figure 6.7. The most obvious advantage is that our algorithm does not suffer the same overshooting problem as the linear blending. The close up view also shows how our algorithm better matches the orientation of the master throughout the collision response. One issue that was not addressed is the transition between the collision network and the sliding network. In the post-collision path of our replica, there is a sharp point where the replica backtracks for a timestep when it switches between the collision network and the sliding network. In a fast-moving collision this is nowhere near as noticeable as the overshooting of the linear blending, but it is still undesirable. In practice, there are many ways that this could be addressed, but this was deemed outside the scope of this investigation. Another shortcoming of this approach is that the collision network is vehicle-specific,

and is also specific to the obstacle unless additional parameters such as obstacle material, vehicle mass, and vehicle rigidity were also input into the network. This was not done for this investigation, as training such a network would require immense amounts of data for each different possible combination of obstacle and vehicle properties.

Chapter 7

Conclusions

In this work we investigated new approaches to replicating an opponent given periodic messages in an online game over a distributed network, with the goal of increasing the message interval to reduce bandwidth. We decomposed this problem into two parts: predicting the opponent's position from their latest message, and rendering a replica of the opponent to follow these predictions with a smooth and believable trajectory. We used a neural network to perform the predictions and showed results that were comparable to the state of the art for low message intervals, and results that were far better than the state of the art for higher message intervals. This indicates potential for using higher message intervals while maintaining believable replication. We tried to apply a neural network to produce a smooth trajectory for the replica vehicle as well, but the results were inconsistent and unreliable and were deemed inadequate for the problem. We then applied a path tracking algorithm borrowed from the field of mobile robotics to take advantage of the fact that we were replicating wheeled vehicles, and found results that were superior to the state of the art algorithms.

We also investigated the problem of resolving collisions in a distributed network environment. This is a very complex problem, and our investigation focused on collisions between a replica vehicle and a static obstacle. In particular, we wanted to establish a proof of concept that we could predict a collision response based on the predicted positions of the replica as opposed to the current position of the replica, as these should be the closest to the replica's true position. We apply a neural network to predict the collision response and show that it has superior performance to the current state of the art for high message intervals.

7.1 Future Work

One of the next steps for this work is to integrate it into other existing game engines. Our solutions were designed with the intention of being easily implemented with little hand tuning required; integration into other engines would be able to demonstrate this ease of implementation. This may also reveal unforeseen shortcomings of our approaches. It would also be worthwhile to train new networks for different vehicles to verify that our approach can be easily applied without additional steps or tuning for different vehicles.

There are many next steps for the collision prediction work. Firstly, the unresolved implementation details could be explored, such as when to switch between the different networks, and how to transition smoothly between them. Additionally, the sliding network could potentially be merged with the prediction network to create a more comprehensive network that better models a wider range of the vehicle's range of motion. While we attempted this in our investigation and could not do so successfully, we believe that with a deeper network and more training data, a comprehensive network could be constructed. This would streamline implementation, and would potentially improve predictions when the master vehicle loses traction.

We also believe that, despite its poor performance in our testing, the neural network blending approach has potential. Although it was potentially unstable, the trajectories it generated while it was tracking successfully were very close to the true trajectory of the vehicle. The most straightforward way to improve robustness would be to add some recovery method to allow the blended path to reconverge if too much error is detected. Methods to improve the training of the blending network could also be investigated such as different network architectures, or different ways of normalizing and preprocessing the data.

References

[1] Sudhir Aggarwal, Hemant Banavar, Amit Khandelwal, Sarit Mukherjee, and Sampath Rangarajan. Accuracy in dead-reckoning based distributed multi-player games. In *Proceedings of 3rd ACM SIGCOMM workshop on Network and System Support for Games*, pages 161–165, 2004.

[2] Ebrahim Babaei, Mahmoud Reza Hashemi, and Shervin Shirmohammadi. A state-based game attention model for cloud gaming. In *2017 15th Annual Workshop on Network and Systems Support for Games (NetGames)*, pages 1–3. IEEE, 2017.

[3] Chris "Battle(non)sense". How netcode works, and what makes 'good' netcode. *PC Gamer*, 2017.

[4] Ikram Belhajem, Yann Ben Maissa, and Ahmed Tamtaoui. Improving low cost sensor based vehicle positioning with machine learning. *Control Engineering Practice*, 74:168–176, 2018.

[5] Youfu Chen and Elvis S Liu. Comparing dead reckoning algorithms for distributed car simulations. In *Proceedings of the 2018 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, pages 105–111, 2018.

[6] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[7] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

[8] Luis Fernando Kawabata de Almeida and Alan Salvany Felinto. Evaluation of the motion-aware adaptive dead reckoning technique under different network latencies applied in multiplayer games. In *17th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)*, pages 137–13709. IEEE, 2018.

[9] Matt Delbosc. Replicating chaos: Vehicle replication in 'Watch Dogs 2', 2017. Game Developers Conference.

[10] Steven Diamond and Stephen Boyd. CVXPY: A Python-embedded modeling language for convex optimization. *Journal of Machine Learning Research*, 17(83):1–5, 2016.

[11] Matthias Dick, Oliver Wellnitz, and Lars Wolf. Analysis of factors affecting players' performance and perception in multiplayer games. In *Proceedings of 4th ACM SIGCOMM workshop on Network and System Support for Games*, pages 1–7, 2005.

[12] Elias P. Duarte, Aurora T.R. Pozo, and Pamela Beltrani. Smart reckoning: Reducing the traffic of online multiplayer games using machine learning for movement prediction. *Entertainment Computing*, 33:100336, 2020.

[13] Evelyn Fix. *Discriminatory analysis: nonparametric discrimination, consistency properties*, volume 1. USAF school of Aviation Medicine, 1985.

[14] Ben Geisler. Integrated machine learning for behavior modeling in video games. In *Challenges in game artificial intelligence: papers from the 2004 AAAI workshop. AAAI Press, Menlo Park*, pages 54–62, 2004.

[15] R Cameron Harvey, Ahmed Hamza, Cong Ly, and Mohamed Hefeeda. Energy-efficient gaming on mobile devices using dead reckoning-based power management. In *9th Annual Workshop on Network and Systems Support for Games*, pages 1–6. IEEE, 2010.

[16] Daniel Holden, Bang Chi Duong, Sayantan Datta, and Derek Nowrouzezahrai. Subspace neural physics: fast data-driven interactive simulation. In *Proceedings of the 18th annual ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 1–12, 2019.

[17] Arthur Juliani, Vincent-Pierre Berges, Ervin Teng, Andrew Cohen, Jonathan Harper, Chris Elion, Chris Goy, Yuan Gao, Hunter Henry, Marwan Mattar, and Danny Lange. Unity: A general platform for intelligent agents, 2020.

[18] Niels Justesen, Philip Bontrager, Julian Togelius, and Sebastian Risi. Deep learning for video game playing. *IEEE Transactions on Games*, 2019.

[19] Vasily Y. Kharitonov. Motion-aware adaptive dead reckoning algorithm for collaborative virtual environments. In *Proceedings of the 11th ACM SIGGRAPH International Conference on Virtual-Reality Continuum and Its Applications in Industry*, VRCAI '12, page 255–261, New York, NY, USA, 2012. Association for Computing Machinery.

[20] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *ICLR (Poster)*, 2015.

[21] B. Knutsson, Honghui Lu, Wei Xu, and B. Hopkins. Peer-to-peer support for massively multiplayer games. In *IEEE INFOCOM 2004*, volume 1, page 107, 2004.

[22] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.

[23] B.Y. Lattimer, J.L. Hodges, and A.M. Lattimer. Using machine learning in physics-based simulation of fire. *Fire Safety Journal*, 114:102991, 2020.

[24] Kyungmin Lee, David Chu, Eduardo Cuervo, Johannes Kopf, Yury Degtyarev, Sergey Grizan, Alec Wolman, and Jason Flinn. Outatime: Using speculation to enable low-latency continuous interaction for mobile cloud gaming. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, pages 151–165, 2015.

[25] R. Luo, T. Shao, H. Wang, W. Xu, X. Chen, K. Zhou, and Y. Yang. NNWarp: Neural network-based nonlinear deformation. *IEEE Transactions on Visualization and Computer Graphics*, 26(4):1745–1759, 2020.

[26] Sahil Mirani. How many people play gta 5? know more about this successful rockstar release. *Republic TV*, 2020.

[27] Curtiss Murphy and E Lengyel. Believable dead reckoning for networked games. *Game engine gems*, 2:307–328, 2011.

[28] Christoph Neumann, Nicolas Prigent, Matteo Varvello, and Kyoungwon Suh. Challenges in peer-to-peer gaming. *SIGCOMM Comput. Commun. Rev.*, 37(1):79–82, January 2007.

[29] Michael A Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

[30] Chris J Ostafew, Angela P Schoellig, and Timothy D Barfoot. Visual teach and repeat, repeat, repeat: Iterative learning control to improve mobile robot path tracking in challenging outdoor environments. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 176–181. IEEE, 2013.

[31] Lothar Pantel and Lars C Wolf. On the suitability of dead reckoning schemes for games. In *Proceedings of the 1st workshop on Network and System Support for Games*, pages 79–84, 2002.

[32] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019.

[33] Claude Samson and Karim Ait-Abderrahim. Feedback control of a nonholonomic wheeled cart in cartesian space. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1136–1141, 1991.

[34] Cheryl Savery, Nicholas Graham, Carl Gutwin, and Michelle Brown. The effects of consistency maintenance methods on player experience and performance in networked games. In *Proceedings of the 17th ACM Conference on Computer Supported Cooperative Work & Social Computing*, CSCW '14, page 1344–1355, New York, NY, USA, 2014. Association for Computing Machinery.

[35] C. Schuwerk and E. Steinbach. Smooth object state updates in distributed haptic virtual environments. In *2013 IEEE International Symposium on Haptic Audio Visual Environments and Games (HAVE)*, pages 51–56, 2013.

[36] Wei Shi, Jean-Pierre Corriveau, and Jacob Agar. Dead reckoning using play patterns in a simple 2d multiplayer online game. *International Journal of Computer Games Technology*, 2014:138596, May 2014.

[37] Robert Skulstad, Guoyuan Li, Thor I Fossen, Bjornar Vik, and Houxiang Zhang. Dead reckoning of dynamically positioned ships: Using an efficient recurrent neural network. *IEEE Robotics & Automation Magazine*, 26(3):39–51, 2019.

[38] Adam Summerville, Sam Snodgrass, Matthew Guzdial, Christoffer Holmgård, Amy K Hoover, Aaron Isaksen, Andy Nealen, and Julian Togelius. Procedural content generation via machine learning (pcgml). *IEEE Transactions on Games*, 10(3):257–270, 2018.

[39] Sebastian Thrun, Mike Montemerlo, Hendrik Dahlkamp, David Stavens, Andrei Aron, James Diebel, Philip Fong, John Gale, Morgan Halpenny, Gabriel Hoffmann, Kenny Lau, Celia Oakley, Mark Palatucci, Vaughan Pratt, Pascal Stang, Sven Strohband,

Cedric Dupont, Lars-Erik Jendrossek, Christian Koelen, Charles Markey, Carlo Rummel, Joe van Niekerk, Eric Jensen, Philippe Alessandrini, Gary Bradski, Bob Davies, Scott Ettinger, Adrian Kaehler, Ara Nefian, and Pamela Mahoney. Stanley: The robot that won the darpa grand challenge. *Journal of Field Robotics*, 23(9):661–692, 2006.

[40] Jonathan Tompson, Kristofer Schlachter, Pablo Sprechmann, and Ken Perlin. Accelerating eulerian fluid simulation with convolutional networks. In *International Conference on Machine Learning*, pages 3424–3433. PMLR, 2017.

[41] Brendan D Tracey, Karthikeyan Duraisamy, and Juan J Alonso. A machine learning strategy to assist turbulence model development. In *53rd AIAA aerospace sciences meeting*, page 1287, 2015.

[42] Florence Tsang, Tristan Walker, Ryan A. MacDonald, Armin Sadeghi, and Stephen L. Smith. Lamp: Learning a motion policy to repeatedly navigate in an uncertain environment, 2020.

[43] Jim Van Verth. Math for game programmers: Understanding quaternions, 2013. Game Developers Conference.