Predictive Dead Reckoning for Online Peer-to-Peer Games

Tristan Walker, Barry Gilhuly, Armin Sadeghi, Matt Delbosc, and Stephen L. Smith

***Abstract*—In online peer-to-peer games, players send periodic updates to each other and each player must locally reconstruct the position of their opponents in between these updates. In scenarios where players are driving cars, high speeds produce more pronounced errors in local replication of online opponents. In this work, we propose a new method of replicating opponents with less data sent and up to 45% less error compared to the state-of-the-art. We use a neural network based approach to predict an opponent's position, combined with a path tracking controller from the field of mobile robotics, to produce smooth, believable trajectories for opponents' vehicles. We also propose a neural network based approach to predict a replicated opponent's trajectory following a collision with a static obstacle.**

keywords: Online multi-player game, Dead Reckoning, peer-to-peer,

I. Introduction

Multiplayer online games are extremely popular, with titles such as *Grand Theft Auto Online* (GTAV), as shown in Figure **??**, having over 100,000 concurrent players [2]. For players to have a smooth playing experience, they must have a reliable internet connection both in terms of bandwidth and latency. For example, GTAV recommends players to have a consistent upload speed of 1 Mbps [3]. However, reliable internet connections are still not available in many parts of the world [4]. In this work propose an approach for reducing bandwidth requirements in Peer-to-Peer (P2P) online games while providing an immersive and competitive experience.

A P2P networked game does not require expensive dedicated servers; rather, P2P games like GTAV use a peer-hosted server, where one player operates a server instance to which as many as 32 other players connect. Another common P2P model uses direct connections: each player sends state information to all other players in the session, and is responsible for maintaining their own true state of the game. The online mode of *Watch Dogs 2* connects up to 8 players per session using this method.

Since P2P games are player hosted, they are impacted by asymmetrical Internet connections where the upload bandwidth is typically a small fraction of that available for download. As a direct consequence, P2P games can be subject to latency and packet loss, which in turn results in a poor online gaming experience [5].

In a car racing game, for example, a player wants the immersive experience of seeing the constantly changing position of an opponent's vehicle with respect to their own. Since sharing updates at game rates would consume the available bandwidth, P2P games are designed to extrapolate the current game state from periodic updates. These extrapolations can result in visual errors, which become more pronounced when players are moving at high speeds.

Fig. 1. Screenshot of driving a vehicle in Grand Theft Auto Online.

Imagine a scenario where a player is racing against an opponent in an online game and the opponent sends their position and velocity once every second (in practice, this happens much more frequently, typically with a period under 100ms [6]). We want the game to render the opponent locally in a way that is both believable and reasonably accurate. Simply rendering the opponent's most recent position at one second intervals would be neither accurate nor believable, resulting in noticeable pauses while the game waits for the next update. Instead, we extrapolate the opponent's changing position from the most recently received message: this is known as Dead Reckoning. In our example, we can 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 2. The opponent starts with a known position and velocity at time t_0, with the true path curving to the right, then at t_1 the curve switches to the left. Since the game engine has no other information, extrapolation using the current state places the car in the incorrect prediction at t_1'. When new information is received, the car snaps to the true position at t_1 resulting in a discontinuous path. This process is then repeated moving from t_1 to t_2. The resulting discontinuities can be jarring to the player. We thus need a method to both predict an opponent's future positions, and 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 post-collision trajectories. This is a difficult problem,

This research is partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and by Ubisoft Toronto.

This work is based on the Master's thesis of Tristan Walker [1], written under the supervision of Prof. S. L. Smith.

T. Walker, B. Gilhuly, A. Sadeghi and S. L. Smith are with the Department of Electrical and Computer Engineering, University of Waterloo, Waterloo ON, N2L 3G1 Canada (tlwalker@uwaterloo.ca; barry.gilhuly@uwaterloo.ca; a6sadegh@uwaterloo.ca; stephen.smith@uwaterloo.ca)

M. Delbosc is with Ubisoft La Forge, Canada (matt.delbosc@ubisoft.com)

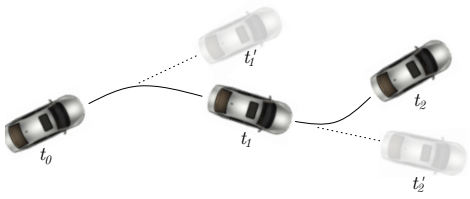


Fig. 2. Extrapolating an opponent's position from periodic updates.

and existing techniques largely aim to hide the effects of this discrepancy without attempting to reduce the error between the trajectories.

In this paper, we show a method of applying neural networks to extrapolate opponent's position in a online game, improving on traditional Dead Reckoning techniques. Combined with a novel application of a path-tracking algorithm, our method results in a more believable, immersive gaming experience for P2P players.

A. Related Work

In online games, sending data over a network introduces significant obstacles to creating an immersive and enjoyable experience. Noticeable latency in the network can impact enjoyment, with one study reporting that players found latencies over 150ms to be unacceptable [5]. Lee et al. propose Outatime [7], a method using Markov predictions and Kalman filtering, to predict future game states and reduce input lag in cloud gaming applications. Savery et al. [8] examine the effect of various lag compensation techniques on player enjoyment, noting that techniques resulting in the best player performance are not always the most enjoyable.

One approach to improving latency is to reduce the amount of data sent. Babei et al. [9] use eye tracking data to determine where players are likely to look so that they can stream higher quality video to only those areas. Bandwidth consumption can also be reduced by improving the prediction of a remote player's position and thereby allowing the interval between information updates to be increased. Harvey et al. [10] shows using Dead Reckoning can reduce the number of information packets sent and conserve energy when gaming on a phone. However, many authors [11]–[13] have shown this to be insufficient for immersive online gaming. A major obstacle, note Pantel and Wolf [13], is that latency between players can be upwards of 100ms. Aggarwal et al. [14] propose a method of using timestamped messages to reduce the impact of latency. Kharatinov [15] proposes an adaptive Dead Reckoning scheme that sends more data during more complex motions, and Almeida and Felinto [12] use the popular Unity [16] game engine to verify that this algorithm holds up under realistic network conditions. For car simulations specifically, Chen and Liu [11] propose an extension to Dead Reckoning that includes environmental cues such as roads, aiding predictions.

Kalman filtering is widely used to perform state estimations in the presence of error. Many variations have been developed, including applying the Kalman filter in the case of intermittent observations [17]. Belhajem et al. [18] use neural networks to approximate a Kalman filter in the event of GPS outages. However, the Kalman filter and its variations require a model of the dynamics of the system. In our scenario such a model is difficult to obtain, as it must capture the physics and dynamics of the full game engine. Additionally, the Kalman filter is intended to provide optimal state estimation in the presence of Gaussian noise. In our scenario, however, there is no error in the measurements or state updates; instead there is uncertainty arising from player inputs changing between updates.

Neural networks have a wide variety of applications in the development of simulated environments. They are used to solve the complex calculations for real-time approximations [19], cloth and fluid simulations [20], [21], turbulence [22], fire behaviour [23], and in Dead Reckoning for ships [24].

In gaming, neural networks produce infinite worlds through procedural generation [25], [26], predict the future location of objects [27], and model network latency [28]. A wider review of the recent role of machine learning in video games can be found in [29].

Developers also use neural networks in video games to predict player behaviour. Duarte [30] uses machine learning to learn trends in player behaviour to better predict future states. Shi et al. [31] propose using data collected from human play sessions to parameterize a Dead Reckoning model specific to one game. Geisler [32] also uses neural networks to model human behaviour and applies this to make AI opponents that behave more realistically.

In this paper, we use machine learning to predict the motion of player operated vehicles. A fully connected model approximates the motion of distant opponents at a low cost, while a deep recurrent neural network takes into account recent player actions to more accurately predict the state of nearby opponents.

As shown in Figure 2, extrapolation solves only part of our problem; we must also render a smooth trajectory. Murphy [33] describes an alternative Dead Reckoning technique called Projective Velocity Blending (PVB), which results in smooth predicted trajectories. Schuwerk and Steinback [34] note that while PVB produces visually smooth paths, users did not enjoy the high accelerations it generated when used to produce haptic feedback. In the field of robotics many approaches use splines to guarantee smooth trajectories; authors in [35] use cubic splines to guarantee continuous acceleration, while authors in [36] use trigonometric splines to guarantee continuous jerk. Robots may also use online path tracking controllers to smoothly follow a commanded path. Samson and Abderrahim [37] propose a linearized feedback control method for a wheeled robot, while Ostafew et al. [38] successfully apply this controller to a physical robot, enabling it to traverse uneven outdoor terrain. Another popular path tracking controller, the Stanley controller, used in the DARPA Grand Challenge [39], uses wheel angle to control lateral and heading error. In this article, we propose using a similar tracking controller to blend the predicted piecewise linear path into a believable smooth curve.

B. Contributions

The key contributions of this work are three-fold. First, we present a new application of neural networks to extrapolate opponents' positions in an online game. Our simulation results show that our approach improves prediction performance under extended message conditions compared to the state-of-the-art. Second, we propose 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 state-of-the-art. Third, we apply neural networks to predict the response of an opponent to a collision with a static object. We show with simulation results that our approach can believably predict collisions better than the current state-of-the-art.

Organization: In Section II we introduce our player and network models, and define the replication problem. In Section III we present our approaches to the three parts of the problem: predicting the position of an opponent; creating a smooth believable opponent trajectory using path tracking algorithms; and predicting the response of a replicated vehicle to a collision with a static obstacle. In Section IV we present experimental results for all parts of our solution approach. Section V concludes this work and outlines possible future investigations.

II. Problem Formulation

In this section we introduce our models for the player and network, as well as formally present the problem.

A. 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 $\boldsymbol{\omega} \in \mathbb{R}^3$.

The player interacts with the game world through the engine in discrete timesteps corresponding to each frame that is rendered by the game. We 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 assume that a single frame is rendered at each timestep of the game, and will use the two interchangeably. We view the game engine as a black box system that uses the game physics to determine how the player interacts with the world and other players.

We 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 a discrete deceleration, coasting, or acceleration control action respectively. We could consider continuous control inputs without adding much complexity, but for simplicity we 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 consider the game engine as a function $\mathcal{F}$ that takes world state at timestep k as $\mathcal{W}[k]$ and the player's state and actions, we can write the evolution of the player's state as $\mathcal{X}[k+1] = \mathcal{F}(\mathcal{X}[k], \mathcal{W}[k], \mathbf{A}[k])$.

In order to focus on the replication problem, this work considers only a single remote player interacting with static elements in the environment. We assume there are no dynamic interactions such as player to player collisions. This simplification allows us to assume the world state is static, and can be encapsulated in the function $\mathcal{F}$. Thus, we think of the evolution of the player's state as simply $\mathcal{X}[k+1] = \mathcal{F}(\mathcal{X}[k], \mathbf{A}[k])$. While this work focuses on the recreation of a single player's trajectory, it is readily extended to multiple remote players by repeating the techniques described for each remote player, ignoring interaction between players. Player to player collisions and other dynamic interactions, such as gravity in a space simulation game, are the subject of future research.

B. Network Model

This work is written from the perspective of the *player* receiving information from another player, whom we refer to as the *opponent*. They are not necessarily competing with each other in the game and simply represent two human agents playing on a network. Each player periodically broadcasts their state and control actions over the network to the other player. This broadcast of information is referred to as a *message*, defined as $s[k] = (\mathcal{X}[k], \mathbf{A}[k], k)$ where k is the timestep the message is sent. To help address potential delays due to latency and packet loss, each message is time stamped and 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 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 fail to reach the other player.

Figure 3 shows latency and message interval when a player is receiving messages from an opponent over a network. In this figure, the number of timesteps or frames between messages is $i = \lfloor \frac{T}{\delta t} \rfloor$. 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 [6]; we are interested in developing a replication scheme that is robust to message intervals of up to 300ms or greater. Being able to predict the state of players with infrequent updates reduces the required network bandwidth and gives the game the opportunity to render more players in the same environment with a richer experience. Our algorithm should also be robust to moderate latency up to 100ms and packet loss of up to 10%, but we leave further study of these conditions for future work. The model of the game engine for the two player scenario is shown in Figure 4.

C. Replication

When playing a networked game, we do not know the ground truth of our opponent's states at all times. We only

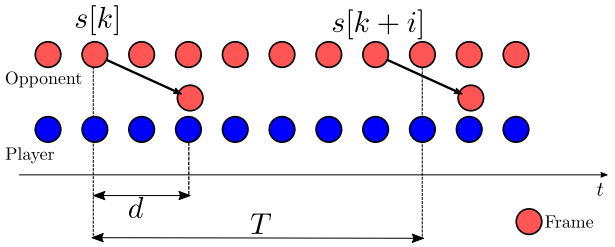


Fig. 3. Every i frames, representing T seconds, the opponent transmits their state information to the player, arriving after a delay of d seconds.

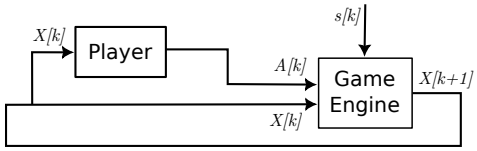


Fig. 4. Block diagram of the two player scenario.

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. Formally, we can define the problem as follows:

Problem II.1 (Replication Problem)**.** Given an opponent that sends a message with a period of T, subject to latency d and packet loss p, generate a series of states $\hat{\mathcal{X}}[0], \hat{\mathcal{X}}[1], ...$ to accurately and believably replicate the motion of the opponent.

In order to generate a smooth and believable trajectory for the replica, we break the problem into three parts. The first component is the prediction problem, where the goal is to predict the future state of the opponent given their most recently received message. The second 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. Believability is difficult to quantify, and is a combination of factors. A believable trajectory should have no discontinuities and appear to obey the dynamics of the vehicle. 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. The third component of the problem is handling collisions that may occur on replicated opponents. 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.

III. Solution Approach

In this section we provide our solutions to the three parts of the replication problem, i.e., 1) prediction of future states, 2) prediction of future collisions, and finally 3) smoothing the

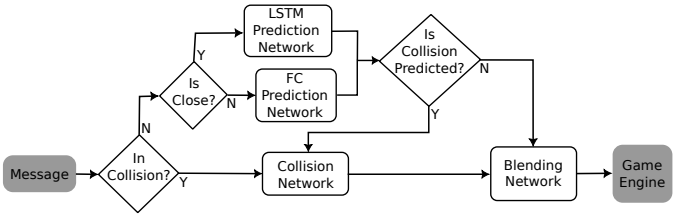


Fig. 5. Proposed framework for motion prediction. Given a message, first we check if we are currently in a collision state and if so use the collision network. Otherwise, based on the proximity to the player we use either the FC or LSTM networks. If a collision is predicted, the collision network is used instead. The motion prediction then passes through the blending network before returning to the game engine.

trajectory. Figure 5 shows the proposed framework. Solutions must be computationally efficient as they need to be calculated separately for each opponent at every timestep. Our solution should approximate the dynamics of the vehicle as well as account for potential changes in opponents' inputs between messages. The dynamics may change for different vehicles, and our solution should not require manual tuning for each vehicle.

A. Predictions

Neural networks are well suited for prediction because they sufficiently address the challenges stated above. They are able to model unknown functions given enough training examples, which is the scenario presented by this problem: we are trying to approximate the game engine, and we can easily record the opponent states within the engine to obtain ample amounts of training data. Moreover, a neural network should be able to learn some patterns in player behaviour. Training of the neural network can be easily automated since we have full access to both the player actions and resulting future states to produce labelled data. With new data and game parameters, the neural network can be easily reconfigured to accommodate other vehicles. Additionally, making predictions with a neural network is a very inexpensive operation.

We take two approaches to making predictions. Fast predictions can be made in a Markov manner, based on the last known state and the number of timesteps since it was received. This network is constructed as a simple deep neural network with three fully connected layers and Rectified Linear Unit (ReLU) activations as shown in Figure 6. This is our Fully Connected (FC) network; it is useful for an opponent located far from the player, where accuracy is less critical. For more accurate predictions of nearby opponents, we replace the fully connected layers of the FC network with a recurrent deep neural network consisting of two layers of Long Short-Term Memory (LSTM) cells, followed by a single fully connected layer that serves as the output. We refer to the recurrent version as the LSTM network.

The input to the FC network is a reduced form of the opponent state. We translate all state information from world reference into the coordinate frame of the opponent's last received message. This reduces the dimensionality of the problem, requiring the network to predict only a simple displacement from the opponent's last position. It is then a

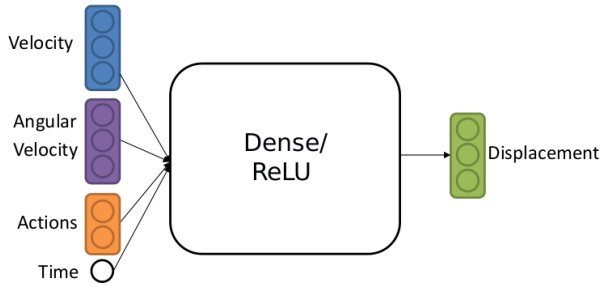


Fig. 6. Overview of the FC prediction network. For the LSTM network, the fully connected network is replaced with two layers of LSTM cells.

straightforward task to translate the displacement back to a world reference for calculation and display purposes. Note that since cars slip, velocity is not necessarily straight ahead in the car's coordinate frame, and must be represented as a vector. As illustrated in Figure 6, the inputs to the FC network are the velocity and angular velocity vectors in $\mathbb{R}^3$, the actions (steering and brake), and the time ahead to predict. The LSTM network accepts the last three reduced states in temporal order, all translated into the most recent opponent frame of reference. The output from both networks is the predicted displacement in $\mathbb{R}^3$. Since the displacement calculation is relative to the last known position, the opponent's current position is not required. Although cars primarily operate in a two dimensional plane, all three dimensions are included to allow this network to be applied to a wider variety of scenarios. Training is accomplished using mean squared error and the Adam optimizer.

B. Blending

Blending adjusts the path output from the prediction network towards the most recent update in a smooth and dynamically feasible manner. We leverage the fact that the motion we are trying to replicate is based on a car's dynamics and borrow the concept of a path tracking controller from mobile robotics. Path tracking is widely used to allow wheeled robots to follow a specified path in the presence of error from sensors, actuators, or the environment. Applying a path tracking controller ensures that the resulting path is smooth and obeys the dynamics of a wheeled vehicle. However, blending also introduces a small delay as it is a form of linear interpolation: the error between the opponent's currently displayed position and the most recent update is corrected over time. The amount of delay introduced affects how much smoothing is applied and is a configurable parameter. The path tracking controller also has the advantage of using feedback control to ensure the system is stable.

Formulation: We leverage the controller proposed in [38] and give a brief derivation of the controller below. We model the opponent as a unicycle robot whose velocity and heading we freely control. 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, $\hat{\mathcal{X}}[k]$. 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:

$$\omega[k] = \frac{-\gamma_1\epsilon_L[k] - \gamma_2 v[k]\sin\epsilon_H[k]}{v[k]\cos\epsilon_H[k]}. \tag{1}$$

This formulation does not control velocity, so we must introduce another equation to govern our opponent's velocity. We introduce a third parameter, γ_3, to govern the velocity. We define $\epsilon_{total} = \epsilon_X[k] + \epsilon_L[k]$, which lets us write

$$\|\mathbf{v}[k]\| = \|\epsilon_{total}\|/\gamma_3. \tag{2}$$

This effectively results in the replica of the opponent running γ_3 seconds behind the predicted position, introducing damping to reduce oscillations.

Implementation: Suppose the opponent is at timestep k with state $\bar{\mathcal{X}}[k]$ and we receive a new message $s[k]$. There are i timesteps until we expect to receive the next message, i.e, $i = \frac{T}{\delta t}$. We implement our blending algorithm as shown below in Algorithm 1. To use the heading and velocity equations described 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 $\|\bar{\mathbf{v}}[k+j]\| = \|\epsilon_{total}\|/\gamma_3$.

Two edge cases need to be accounted for during implementation of this algorithm. First, 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 unaccounted for, the opponent will accelerate away from the predicted position indefinitely. In our implementation we handle this case by simply halving the speed of the opponent, allowing time for the predictions to catch up. A second edge case occurs if the heading error is outside of $\pm 90°$, causing the opponent to track the path in the opposite direction, converging to $180°$ of heading error. To remedy this, we clamp the heading error to $\pm 80°$, which also helps avoid numerical errors from dividing by 0 or close to 0 in the heading equation.

Algorithm 1: BLENDING ALGORITHM

while *replicating* **do**
  $s[k] =$ LATEST MESSAGE
  $i = \lfloor T/\delta t \rfloor$
  $\hat{\mathbf{r}}[k+i], \hat{\mathbf{q}}[k+i] =$ PREDICT$(s[k], i)$
  $j = 1$
  while *no new message* **do**
    $\bar{\mathbf{v}}[k+j] =$ BLEND$(\bar{\mathbf{v}}[k], \hat{\mathbf{r}}[k+i] - \bar{\mathbf{r}}[k], \hat{\mathbf{v}}[k+i] - \bar{\mathbf{v}}[k], j \cdot \delta t)$
    $\bar{\mathbf{r}}[k+j] = \bar{\mathbf{r}}[k+j-1] + \bar{\mathbf{v}}[k+j] \cdot \delta t$
    $j{+}{+}$
    if $j > i$ **then**
      $i = i + \lfloor T/\delta t \rfloor$
      $\hat{\mathbf{r}}[k+i], \hat{\mathbf{q}}[k+i] =$ PREDICT$(s[k], i)$

C. Collision Response Prediction

The problem of predicting an opponent's path may be complicated by interactions with static objects. The current approach is to allow the local physics engine to handle position updates; however, this can lead to synchronization problems as the local engine doesn't support prediction. Instead, we implement a prediction network to smoothly blend an opponent's position towards the predicted collision response rather than let the collision play out. If our replica is offset from where it should be, the collision may not even occur locally, leading to a less desirable result compared to predicting the collision and blending as new updates arrive. In this situation, the model is still $\mathcal{X}[k+1] = \mathcal{F}(\mathcal{X}[k], \mathbf{A}[k])$, except the function $\mathcal{F}$ also encompasses static collisions. Player actions are not considered in the collision model as they little have impact on the collision response for many timesteps after the collision.

Neural Network: We use a fully connected neural network similar in structure to the FC prediction network shown in Figure 6 with the following changes. Since player actions are ignored during a collision event, they are no longer included as an input. In the Unity engine, collisions are calculated based on a collision point, a collision normal and the collision velocity These parameters, all in $\mathbb{R}^3$, are inputs to the collision network. 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. We must also include a time from the collision that we wish to predict. The orientation of the car is important in creating a believable trajectory and cannot be easily estimated from a given path. Therefore, we output displacement and orientation, in $\mathbb{R}^3$ and $\mathbb{R}^4$ respectively.

For a collision at timestep k, letting the collision point be $\mathbf{c}_k$ and the collision normal be $\mathbf{n}_k$, we may format our inputs as $(\mathbf{v}[k], \mathbf{c}_k, \mathbf{n}_k, \Delta t)$. Our outputs are $(\Delta\bar{\mathbf{r}}[k], \bar{\mathbf{q}}[k+i])$, where the displacement $\Delta\bar{\mathbf{r}}[k] = \bar{\mathbf{r}}[k+i] - \bar{\mathbf{r}}[k]$, orientation is $\bar{\mathbf{q}}[k+i]$, and $i = \frac{\Delta t}{\delta t}$.

Training: To train and test collisions, a cylindrical obstacle was placed in the Unity game engine environment. To automatically generate collision data, we ran a script that repeatedly launches a car at the obstacle at random angles, directions, and velocity, 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. We use a circular obstacle to capture all collision angles. Figure 7 shows the car colliding with the cylinder, with the path of the car shown in yellow. The 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. During this process we are able to record the state of the 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 T_{max} must be specified, presenting the same tradeoff between generality and accuracy.

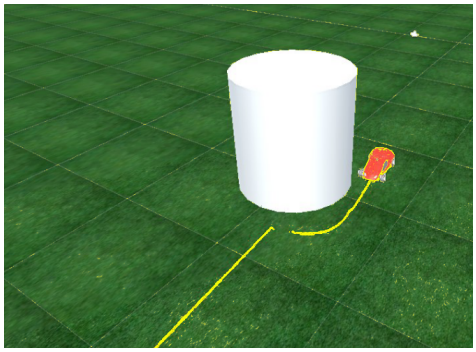


Fig. 7. Screenshot of a collision with a cylindrical obstacle in Unity.

Immediately, we observe two potential challenges. 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, until eventually the wheels will grip and it rolls slowly backward. Second, Figure 7 reveals that the car impacts the the cylinder twice before settling into rolling backwards. This is problematic for collision prediction, as our model is trained only for single impacts and therefore may make erroneous predictions when a second occurs during the same event. Thus, in our implementation we must account for these additional impacts. As before, the network was trained using mean squared error and the Adam optimizer.

D. Framework

A block diagram of the combined framework is illustrated in Figure 5. The trajectory of the opponent is replicated using one of the prediction networks, either FC or LSTM, depending on the opponent's proximity to the player. If, on any timestep, the planner detects a collision between the opponent and another object, the planner switches to a collision prediction state, and remains in that state for an interval of time. The blending network is used to construct a smooth path from any prediction state.

After entering the collision state, the planner remains in that state for two seconds, approximately the time required to establish grip again and settle into smooth, controlled motion. During this interval, the opponent's inputs can be safely ig-

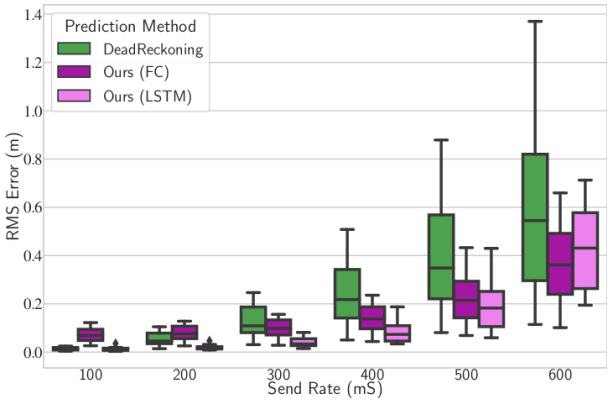


Fig. 8. RMS Prediction error for different message send rates. For longer intervals the neural network methods have significantly less error.

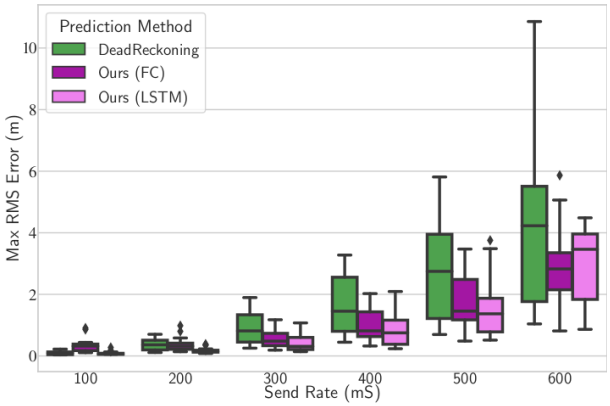


Fig. 9. Maximum RMS Prediction errors. Both of the neural networks have a much smaller deviation from the desired path with the LSTM network performing slightly better.

nored as they have no effect on the trajectory. Once the interval completes, control returns to the predictive network. In this simple model, false detection of collisions can occur resulting in some artifacts while the planner attempts to reconcile the predicted collision path with the opponent's updated position. Adding error handling to detect both false positive and false negative collision predictions as well as the completion of the collision state would improve the overall performance, but is not addressed here.

IV. Results

In the following section, we present our simulation results[1] based on data collected from the Unity game engine [16].

A. Predictions

The baseline prediction algorithm we compare to is a discrete-time Dead Reckoning algorithm that uses velocity and angular velocity, a method widely used in industry [40]. Given position $\mathbf{r}[k]$, velocity $\mathbf{v}[k]$ and angular velocity $\boldsymbol{\omega}[k]$ at timestep k, the position at the next timestep can be predicted by extrapolating from the current position using the current velocity, rotated by the given angular velocity. We can define a rotation operator $\mathbf{q}$ such that $\mathbf{q}_\theta(\mathbf{v})$ rotates $\mathbf{v}$ by the Euler angles given by $\theta \in \mathbb{R}^3$. Then we have $\hat{\mathbf{r}}[k+1] = \mathbf{r}[k] + \mathbf{q}_{\boldsymbol{\omega}[k]\delta t}(\mathbf{v}[k]\delta t)$, where δt is the time between frames. To predict further into the future, this process is repeated for each frame, i.e. $\hat{\mathbf{r}}[k+2] = \hat{\mathbf{r}}[k+1] + \mathbf{q}_{\boldsymbol{\omega}[k]2\delta t}(\mathbf{v}[k]\delta t)$. This means that the predicted trajectory from one known state will be a piece-wise linear arc.

The FC network was constructed with three fully connected layers of 100 perceptron cells. The LSTM network used two layers of 128 LSTM cells. The relatively small layer sizes were found empirically to produce accurate results while being as small as possible to minimize computation costs.

B. Training and Testing

Our prediction data sets were generated in an empty Unity environment, using feasible randomly generated paths. No context queues are taken from the environment in the prediction process. Using available map information would likely contribute to the predictions; however, this is not considered here. While incorporating the environment states can be helpful in racing games where the environment greatly dictates player actions [11], in open world games the actions of the players and the roads/rules of the environment are only loosely correlated.

The training data for trajectory prediction consists of samples drawn from the trajectory data combined with the control inputs that generated them. First the recorded trajectories are separated into training and testing data. The process starts by sampling an initial position and player control from a trajectory. Then, stepping forward along the trajectory in 50ms steps, a new training/test entry is created by combining the position, control, time since position occurred, and the current true position at the current step. For example, to create training data for a 400ms interval test, an initial state and control is selected from the data at time t. Then, at times $t + 50ms$, $t+100ms$, ... the car's position is read from the trajectory and a new training entry is created. This creates eight samples with inputs of starting state, a control input, step time, and a ground truth position. For the LSTM network, we simply extend the number of position and control samples required, separated by the desired training interval. For the model described in this paper, we require three samples of input data drawn at 400ms intervals prior to the target state data that follows the last sample. Using this method, samples were generated with message intervals from $T = 100$ms to $T = 400$ms, much higher than the typical message interval used in online games. The process can be entirely automated, making the support of self-supervised learning straightforward.

Note that the model is trained on data that incorporates the dynamics of the player vehicle and the environment. Complex behaviour, such as a changing model, requires a larger prediction network and training for each vehicle type, or additional parameters to capture vehicle characteristics.

In the training stage, the data set is randomly sampled, drawing an input state, controls, time since the last update, and the paired ground truth. Each state, input, and prediction

[1]Training models and code are available at https://github.com/Ubisoft-LaForge/ubisoft-laforge-PredictiveDeadReckoning

time interval is provided to the network and the resulting output compared to the ground truth. The error is then back-propagated through the network to correct any error. For testing, the samples are passed through the network and the result compared with the ground truth using L2 distance.

One of the challenges with predicting the opponent's position is the subjective nature. Every player has a different tolerance for visual artifacts; thus, we've found the best approach is focus on minimizing the visual error as an analog for belief. To this end, we evaluate the quality of the reconstructed trajectory by comparing the RMS displacement error from the ground truth trajectory. The opponent's orientation is assumed to be preserved. The distribution of mean RMS displacement errors over trajectories for all three methods is shown in Figure 8. The coloured boxes indicate the range of values that capture from 25% to 75% of the trajectories, with the overall mean being the solid line in the middle. The whiskers show the full distribution, giving a notion of the spread of the data. We also show the distribution of maximum error values observed for each trajectory in Figure 9.

While the smaller errors at lower intervals ($< 200ms$) may be barely noticeable, above 300ms, the errors start to become significant. At 400ms, the maximum error for Dead Reckoning can result in a jump of the opponent's position by almost 2m on average, roughly half the body length of a 4.7m car. The LSTM network is the best performing at our target interval of $300 - 400$ms. All of the algorithms have very low error for message intervals at or below 200ms, though the FC network does not perform quite as well. This may be improved with a training focus on shorter intervals; however, we didn't investigate further as intervals of 100ms and below are well served by existing methods.

More importantly, in the range between 200ms and 400ms, we see significant improvement from the network implementations. The LSTM prediction network has considerably lower average error during this portion, as well as smaller spread. The difference in spread is perhaps more apparent in Figure 9 where the mean worst case error shows almost a meter decrease. Note that although our networks are only trained for message intervals up to 400ms, they maintain superior performance 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.

C. Execution Time

In terms of execution time, the Dead Reckoning and FC networks are very similar, both taking 6 seconds to complete 100,000 tests. The LSTM network is approximately seven times slower, requiring 43 seconds for the same task when run on nVidia 3090 hardware, or about 0.5ms per iteration.

D. Path Tracking Results

The state-of-the-art we compare to initially is a discrete linear blending approach that replicates 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 toward 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 represents the replica's time delay from the prediction. In our results we will compare to this linear blending approach using $\lambda = 0.4$ which was found to be the best-performing value.

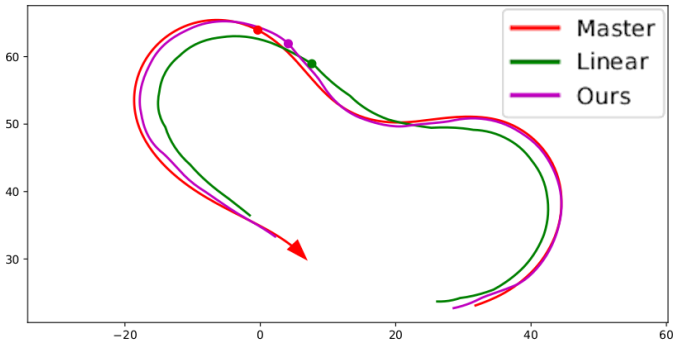


Fig. 10. Results of path tracking blending with untuned parameters and a message interval of 300 ms.

We define the controller parameters of Equations (1) and (2) in Section III.B as $\gamma_1 = 80, \gamma_2 = 20, \gamma_3 = 0.2$ obtained through trial and error. The results are promising, showing much smoother paths than the state-of-the-art with considerably less error. These results are shown in Figure 10. 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. From the marked point we see that a γ_3 of 0.2 corresponds to roughly half the delay of the state-of-the-art algorithm.

We evaluate the blending algorithms using positional error, considering both the absolute error, as well as what we call the *adjusted error*. The adjusted error 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, $\mathbf{r}[k - \gamma_3/\delta t]$. The adjusted error minimizes impact of the different delay values used, allowing the comparison of the algorithms independent of delay. The averages of these two errors are shown in Table I. 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. This suggests 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 visual smoothness of the path are encouraging for the potential of path tracking blending.

Parameter Tuning: While the initial results with manually selected parameters are promising, further improvements are possible by tuning γ_1, γ_2, and γ_3. During the manual experimentation phase, we 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, for higher message intervals there was a firm lower limit for how low γ_3 could be while maintaining stability.

TABLE I
AVERAGE ERRORS FOR DISCRETE LINEAR BLENDING AND PATH TRACKING BLENDING.

	100ms		300ms		500ms	
	mean	std dev	mean	std dev	mean	std dev
Error						
Linear	5.3	3.0	6.2	2.3	5.5	3.2
Path Tracking	2.7	1.5	3.3	1.2	3.0	1.7
Adjusted Error						
Linear	1.2	1.0	1.8	1.1	1.5	1.2
Path Tracking	0.3	0.2	0.6	0.3	0.7	0.6

Thus we select γ_3 based on our desired message interval. In our case, for a desired message interval of $250 - 300$ms, we selected $\gamma_3 = 0.1$. By simulating our path tracking blending algorithm with different values of γ_1 and γ_2 we calculate the average errors of each combination. A simple grid search then gives our tuned values. We want values of γ_1 and γ_2 that both result in a lowest error and are low values. Therefore we selected the pair with the lowest sum whose average error was within 20% of the minimum, yielding tuned values of $\gamma_1 = 130, \gamma_2 = 10$, and $\gamma_3 = 0.1$.

The tuned algorithm is then compared to both the discrete linear blending approach, and PVB [33]. Note that PVB performs both the prediction and blending steps. Given the current estimated state of the replica $\bar{\mathcal{X}}$ and a message with its last known state $\mathcal{X}$, the algorithm uses Dead Reckoning to predict future states for both. A linear interpolation between the two future states yields 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. We are not considering acceleration when making projections.

Figure 11 shows the algorithms performing with a message interval of 300ms 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 discrete 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 discrete 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 path. The PVB approach on the other hand, has no delay behind the master car at the cost of some irregularities in the path where it must double back when its extrapolation is incorrect. Our approach sacrifices a small amount delay for considerably better replication of the master path. Under more difficult network conditions, with higher packet intervals, the flaws of all the replication algorithms are magnified. At this message interval, even the conservative discrete 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. At even higher message intervals and with message loss, our algorithm still performs adequately, while the other two algorithms show significant issues. The effect of latency is not compared here, as it affects all algorithms identically, and including it as a variable would only obfuscate the results.

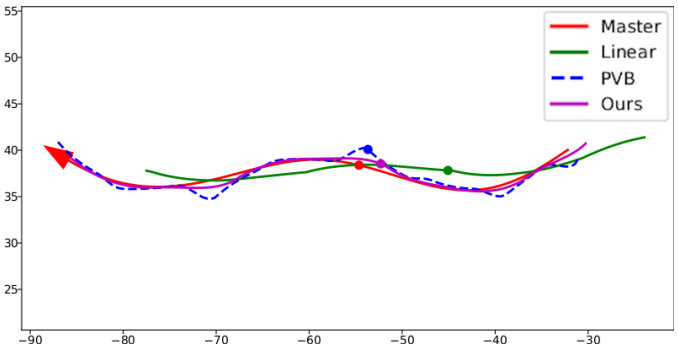


Fig. 11. Results of path tracking blending with tuned parameters and a message interval of 300ms. Notice the bumps that occur in the PVB path, leading to unrealistic motion of the opponent vehicle.

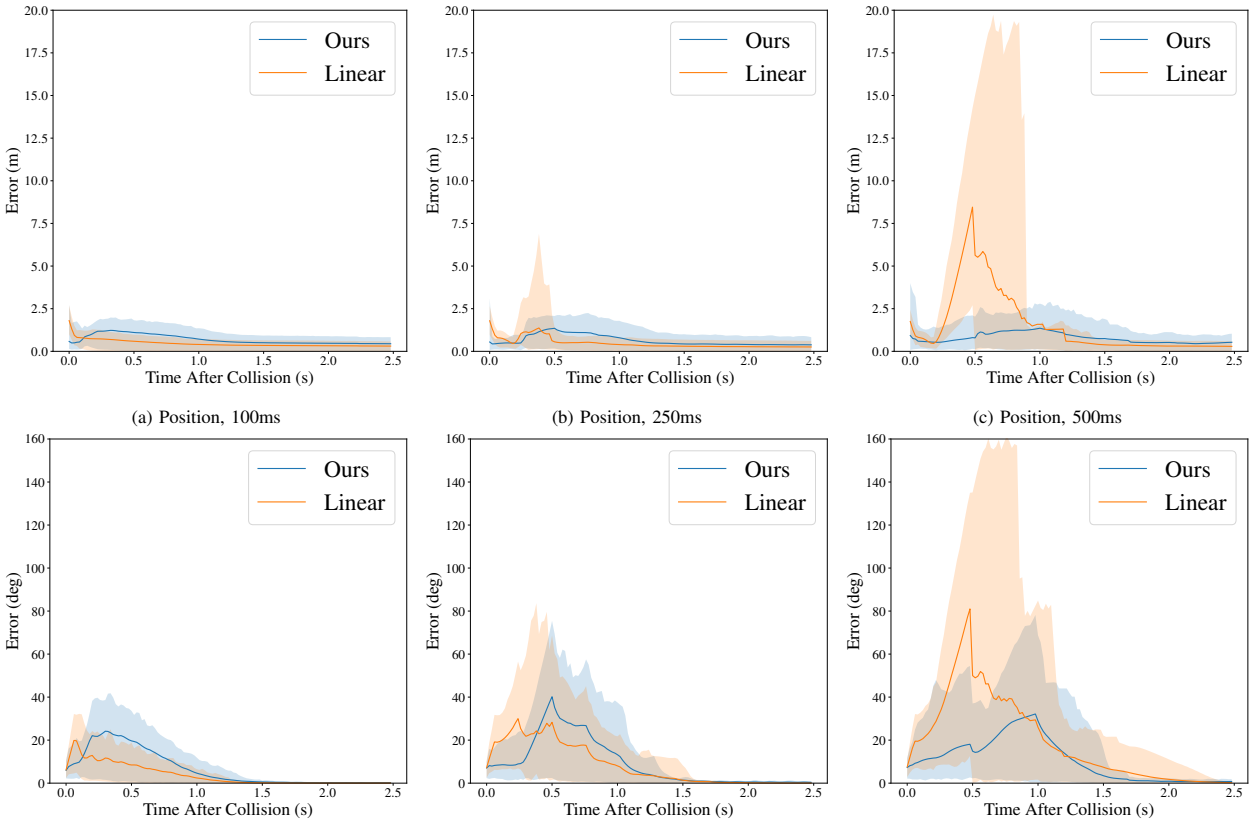
E. Collision Response Prediction

We constructed the collision network with 100 cells per fully connected layer, opting again for the smallest size of network that still produced reasonable results. We compare the output from our collision network with the state-of-the-art discrete linear blending approach described in the previous section.

With no considerations made for collisions, this scheme results in the replica overshooting the obstacle or jittering around the point of collision as the game engine attempts to reconcile the replica collision and the blending algorithm, trying to force the replica to a prediction inside the obstacle.

A summary of our algorithm's performance for message intervals of 100ms, 250ms, and 500ms is shown in Figure 12. To characterize the average performance, around 100 random collisions were simulated using the same method 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.

In Figure 12, 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 discrete linear blending approach performs very well in both categories. At higher message intervals, in which we are more interested, our approach shows some advantages over the discrete linear blending approach. At message intervals of 250ms and above, the discrete linear blending approach starts to show the overshooting that was mentioned earlier, as evidenced 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



(a) Position, 100ms (b) Position, 250ms (c) Position, 500ms

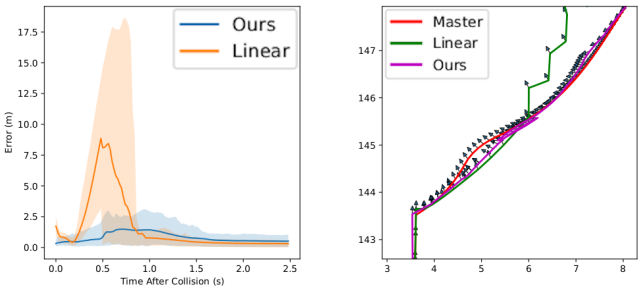
(d) Orientation, 100ms (e) Orientation, 250ms (f) Orientation, 500ms

Fig. 12. Position and Rotation error after a collision for message intervals of 100ms, 250ms and 500ms. Both methods perform well at 100ms; however, at higher message intervals the neural network is much more consistent.

used in practice, but could arise if packet loss occurs during a collision at a lower message interval. Overall, our approach demonstrates much more consistent replication, with fewer immersion-breaking jumps and overshooting.

During simulation, we experienced some implementation issues with our approach that were unrelated to our solution. Occasionally, the game engine's collision detection system would fail to report a collision. Another source of issues was an occasional delay of one or two frames between the Python server running the neural network and Unity engine. This delay 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.

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. Figure 13 shows the paths of the linear replica and our replication scheme as compared to the master car during a collision. The most obvious advantage is that our algorithm does not suffer the same overshooting problem as discrete 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 return transition from the collision network to the prediction network. In the post-collision path of our replica, there is a sharp point where the replica backtracks for a timestep when it switches. In a fast-moving collision, this is nowhere near as noticeable as



(a) Collision Paths (b) Close up view

Fig. 13. Effect of a collision on predictions(500ms interval). a) The full path after collision – the Linear method is unable to reconstruct a reasonable copy of the master path. b) In the close up, Our method can be seen to closely predict the orientation of the Master.

the overshooting of the discrete linear blending, but it is still undesirable. In practice, there are many ways that this could be addressed, and we leave this investigation for future work.

V. Conclusions

Replicating player positions from state information updates is one of the most bandwidth consuming components of operating online P2P games. We have shown how neural networks can be used to improve the prediction of opponent actions, allowing for a lower data transmission rate between

players and possibly allowing more players per session. We applied a path tracking algorithm from robotics to create a smooth, believable reconstruction of a remote player's trajectory. Finally, we demonstrated the use of a small neural net in the prediction of collision responses to render a trajectory that closely reproduces the true trajectory.

Our goal with this work was to establish a proof of concept for predicting future motion and collision responses using a neural network. To do so, we captured some aspects of the problem, notably the physical properties of the objects involved, implicitly in the neural networks. We leave the problem of creating a generalized model for future work. We also note that while our approach maintains lower error for much higher message intervals, further research is required into packet loss and delay.

Our simplified environments did not make use of map information. As noted in [11], road maps can be used as a prior over a player's future trajectory. Adding some notion of the upcoming road information to the model inputs may further improve predictions of future actions.

Finally, our investigation of collisions omitted player-to-player impacts and did not consider methods to detect the point at which the player regains control. Expanding the research to dynamic events and improving the transition from the collision network back to the prediction network where player controls are responsive again, are likely to be additional valuable contributions.

REFERENCES

[1] T. Walker, "Dead reckoning for distributed network online games," Master's thesis, University of Waterloo, May 2021, http://hdl.handle.net/10012/16960.

[2] S. Mirani, "How Many People Play GTA 5? Know More About This Successful Rockstar Release," *Republic TV*, 2020, https://www.republicworld.com/technology-news/gaming/how-many-people-play-gta-5-know-more-about-this-successful-rockstar-release.html.

[3] "Grand Theft Auto Online PC Connection Troubleshooting." [Online]. Available: https://support.rockstargames.com/articles/200525767/Grand-Theft-Auto-Online-PC-Connection-Troubleshooting

[4] "Advancing meaningful connectivity: Towards active and participatory digital societies," Alliance for Affordable Internet (A4AI), Tech. Rep., 2022, https://a4ai.org/wp-content/uploads/2022/02/ExecsummaryENGLISH.pdf.

[5] M. Dick, O. Wellnitz, and L. Wolf, "Analysis of factors affecting players' performance and perception in multiplayer games," in *ACM SIGCOMM Workshop on Network and System Support for Games*, 2005.

[6] C. "Battle(non)sense", "How netcode works, and what makes 'good' netcode," *PC Gamer*, 2017, https://www.pcgamer.com/netcode-explained.

[7] K. Lee, D. Chu, E. Cuervo, J. Kopf, Y. Degtyarev, S. Grizan, A. Wolman, and J. Flinn, "Outatime: Using Speculation to Enable Low-Latency Continuous Interaction for Mobile Cloud Gaming," in *International Conference on Mobile Systems, Applications, and Services*, 2015, pp. 151–165.

[8] C. Savery, N. Graham, C. Gutwin, and M. Brown, "The Effects of Consistency Maintenance Methods on Player Experience and Performance in Networked Games," in *ACM Conference on Computer Supported Cooperative Work & Social Computing*, 2014, pp. 1344–1355.

[9] E. Babaei, M. R. Hashemi, and S. Shirmohammadi, "A state-based game attention model for cloud gaming," in *IEEE Workshop on Network and Systems Support for Games*, 2017.

[10] R. C. Harvey, A. Hamza, C. Ly, and M. Hefeeda, "Energy-efficient gaming on mobile devices using dead reckoning-based power management," in *IEEE Workshop on Network and Systems Support for Games*, 2010.

[11] Y. Chen and E. S. Liu, "Comparing Dead Reckoning Algorithms for Distributed Car Simulations," in *ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, 2018, pp. 105–111.

[12] L. F. K. de Almeida and A. S. Felinto, "Evaluation of the Motion-Aware Adaptive Dead Reckoning Technique Under Different Network Latencies Applied in Multiplayer Games," in *IEEE Brazilian Symposium on Computer Games and Digital Entertainment*, 2018, pp. 137–146.

[13] L. Pantel and L. C. Wolf, "On the Suitability of Dead Reckoning Schemes for Games," in *Workshop on Network and System Support for Games*, 2002, pp. 79–84.

[14] S. Aggarwal, H. Banavar, A. Khandelwal, S. Mukherjee, and S. Rangarajan, "Accuracy in Dead-Reckoning Based Distributed Multi-Player Games," in *ACM SIGCOMM Workshop on Network and System Support for Games*, 2004, pp. 161–165.

[15] V. Y. Kharitonov, "Motion-Aware Adaptive Dead Reckoning Algorithm for Collaborative Virtual Environments," in *ACM SIGGRAPH International Conference on Virtual-Reality Continuum and Its Applications in Industry*, 2012, pp. 255–261.

[16] A. Juliani, V.-P. Berges, E. Teng, A. Cohen, J. Harper, C. Elion, C. Goy, Y. Gao, H. Henry, M. Mattar, and D. Lange, "Unity: A General Platform for Intelligent Agents," 2020.

[17] B. Sinopoli, L. Schenato, M. Franceschetti, K. Poolla, M. Jordan, and S. Sastry, "Kalman filtering with intermittent observations," *IEEE Transactions on Automatic Control*, vol. 49, no. 9, pp. 1453–1464, 2004.

[18] I. Belhajem, Y. B. Maissa, and A. Tamtaoui, "Improving low cost sensor based vehicle positioning with Machine Learning," *Control Engineering Practice*, vol. 74, pp. 168–176, 2018.

[19] D. Holden, B. C. Duong, S. Datta, and D. Nowrouzezahrai, "Subspace Neural Physics: Fast Data-Driven Interactive Simulation," in *ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, 2019.

[20] 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*, vol. 26, no. 4, pp. 1745–1759, 2020.

[21] J. Tompson, K. Schlachter, P. Sprechmann, and K. Perlin, "Accelerating eulerian fluid simulation with convolutional networks," in *International Conference on Machine Learning*. PMLR, 2017, pp. 3424–3433.

[22] B. D. Tracey, K. Duraisamy, and J. J. Alonso, "A Machine Learning Strategy to Assist Turbulence Model Development," in *AIAA Aerospace Sciences Meeting*, 2015.

[23] B. Lattimer, J. Hodges, and A. Lattimer, "Using machine learning in physics-based simulation of fire," *Fire Safety Journal*, vol. 114, 2020.

[24] R. Skulstad, G. Li, T. I. Fossen, B. Vik, and H. Zhang, "Dead Reckoning of Dynamically Positioned Ships: Using an Efficient Recurrent Neural Network," *IEEE Robotics & Automation Magazine*, vol. 26, no. 3, pp. 39–51, 2019.

[25] A. Summerville, S. Snodgrass, M. Guzdial, C. Holmgård, A. K. Hoover, A. Isaksen, A. Nealen, and J. Togelius, "Procedural content generation via machine learning (pcgml)," *IEEE Transactions on Games*, vol. 10, no. 3, pp. 257–270, 2018.

[26] R. R. Torrado, A. Khalifa, M. C. Green, N. Justesen, S. Risi, and J. Togelius, "Bootstrapping conditional GANs for video game level generation," in *IEEE Conference on Games*, 2020, pp. 41–48.

[27] T. Rath and N. Preethi, "Application of AI in Video Games to Improve Game Building," in *IEEE Conference on Communication Systems and Network Technologies*, 2021, pp. 821–824.

[28] A. Wong, C. Chiu, G. Hains, J. Behnke, Y. Khmelevsky, and C. Mazur, "Modelling Network Latency and Online Video Gamers' Satisfaction with Machine Learning," in *IEEE International Conference on Recent Advances in Systems Science and Engineering*, 2021.

[29] G. Edwards, N. Subianto, D. Englund, J. W. Goh, N. Coughran, Z. Milton, N. Mirnateghi, and S. A. A. Shah, "The Role of Machine Learning in Game Development Domain-A Review of Current Trends and Future Directions," *Digital Image Computing: Techniques and Applications*, 2021.

[30] E. P. Duarte, A. T. Pozo, and P. Beltrani, "Smart Reckoning: Reducing the traffic of online multiplayer games using machine learning for movement prediction," *Entertainment Computing*, vol. 33, 2020.

[31] W. Shi, J.-P. Corriveau, and J. Agar, "Dead Reckoning Using Play Patterns in a Simple 2D Multiplayer Online Game," *International Journal of Computer Games Technology*, May 2014.

[32] B. Geisler, "Integrated machine learning for behavior modeling in video games," in *Challenges in Game Artificial Intelligence: Papers From the 2004 AAAI Workshop*, 2004, pp. 54–62.

[33] C. Murphy and E. Lengyel, "Believable Dead Reckoning for Networked Games," *Game Engine Gems*, vol. 2, pp. 307–328, 2011.

[34] C. Schuwerk and E. Steinbach, "Smooth object state updates in distributed haptic virtual environments," in *IEEE International Symposium on Haptic Audio Visual Environments and Games*, 2013, pp. 51–56.

[35] C. Lin, P. Chang, and J. Luh, "Formulation and optimization of cubic polynomial joint trajectories for industrial robots," *IEEE Transactions on Automatic Control*, vol. 28, no. 12, pp. 1066–1074, 1983.
[36] D. Simon and C. Isik, "A trigonometric trajectory generator for robotic arms," *International Journal of Control*, vol. 57, no. 3, pp. 505–517, 1993.
[37] C. Samson and K. Ait-Abderrahim, "Feedback control of a nonholonomic wheeled cart in cartesian space," in *IEEE International Conference on Robotics and Automation*, 1991, pp. 1136–1141.
[38] C. J. Ostafew, A. P. Schoellig, and T. 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*, 2013, pp. 176–181.
[39] S. Thrun, M. Montemerlo, H. Dahlkamp, D. Stavens, A. Aron, J. Diebel, P. Fong, J. Gale, M. Halpenny, G. Hoffmann *et al.*, "Stanley: The Robot that Won the DARPA Grand Challenge," *Journal of Field Robotics*, vol. 23, no. 9, pp. 661–692, 2006.
[40] S. Liu, X. Xu, and M. Claypool, "A Survey and Taxonomy of Latency Compensation Techniques for Network Computer Games," *ACM Computing Surveys*, 2022.