VECTOR EMBEDDING TECHNIQUES
FOR PLAYER BEHAVIOUR IN DOTA 2

by

Jinay Shah

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of

MASTER OF INFORMATION TECHNOLOGY: DIGITAL MEDIA WITH
SPECIALIZATION IN DATA SCIENCE

School of Information Technology

at

CARLETON UNIVERSITY

Ottawa, Ontario

August, 2021

© Copyright by Jinay Shah, 2021
1.3 Behaviour Modeling ............................................. 18
  1.3.1 Conventional Methods ...................................... 18
  1.3.2 Usage of Deep Learning .................................... 18
1.4 Reasons To Use a Game Dataset For Behaviour Modeling .... 19
  1.4.1 Ease of Data Access ........................................ 20
  1.4.2 Volume of Data ............................................. 20
  1.4.3 More Confined Environment ................................. 21
  1.4.4 Better Reproducibility ...................................... 21
1.5 Summary ......................................................... 21

Chapter 2 Problem Formulation ................................. 22
  2.1 Challenges Involved ........................................... 22
    2.1.1 Dense Vector Representation .............................. 23
    2.1.2 Meaningful Vector Representation ......................... 23
    2.1.3 Efficiency and Speed .................................... 25
    2.1.4 Feature Selection ......................................... 25
    2.1.5 Ease of Use in Other Environments ....................... 27

Chapter 3 Related Work ........................................ 28
  3.1 The Effectiveness of word2vec ............................... 28
  3.2 Representing Game Objects as Vectors using Physics Information ................................. 30
  3.3 Predicting a Player’s Role in DOTA 2 ....................... 32
3.4 Observing Player Style from Gameplay in Just Cause 2 . . . . . . . . 33
3.5 Representing Emotions using Vectors . . . . . . . . . . . . . . . . . . 35
3.6 Improving Learning Disability Detection Using Vectors . . . . . . . . 36
3.7 Predicting Viewer Engagement Using Game Features . . . . . . . . . 37

Chapter 4 Proposed Approach

4.1 High-Level Overview . . . . . . . . . . . . . . . . . . . . . . . . . . 39
  4.1.1 The Method for Getting Vectors . . . . . . . . . . . . . . . . . 40
  4.1.2 How is the graph made? . . . . . . . . . . . . . . . . . . . . . . 40
  4.1.3 What are we using to embed the graph nodes as vectors? . . . 41
4.2 The OpenDOTA Dataset . . . . . . . . . . . . . . . . . . . . . . . . . 42
  4.2.1 Reasons For Choosing The OpenDOTA API . . . . . . . . . . . 42
  4.2.2 Available Attributes . . . . . . . . . . . . . . . . . . . . . . . . 44
  4.2.3 Selected Attributes . . . . . . . . . . . . . . . . . . . . . . . . . 47
  4.2.4 Selection of Games . . . . . . . . . . . . . . . . . . . . . . . . . 49
  4.2.5 Other Options . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
4.3 Building the Game Graph . . . . . . . . . . . . . . . . . . . . . . . . 50
  4.3.1 Modeling Interactions as Text: Why it Fails . . . . . . . . . . . 50
  4.3.2 Modeling Interactions as a Graph . . . . . . . . . . . . . . . . . 53
4.4 Using Deep Learning to Generate Vectors for Graph Nodes . . . . . 56
  4.4.1 node2vec . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56
4.4.2 LINE ................................................................. 58
4.4.3 Temporal Graph Networks ........................................ 60
4.5 Modeling Interactions After Extracting Vectors ................. 62
4.6 Summary ............................................................... 63

Chapter 5 Evaluation ...................................................... 64
5.1 Testing Framework ................................................... 64
5.1.1 The Big Picture .................................................. 66
5.1.2 Partitioning Games ............................................... 67
5.1.3 Building Vectors ................................................ 68
5.1.4 Building Vector Timelines ....................................... 68
5.1.5 Sampling The Vector Timeline .................................. 70
5.1.6 Evaluation Metrics ............................................... 73
5.2 Experiments ............................................................ 74
5.2.1 Number of Dimensions in the Vector ......................... 75
5.2.2 Ways to Build the Game Graph ................................ 76
5.2.3 Step Size for Linking Two Vectors ............................ 78
5.2.4 Number of Games Used to Make Vectors .................... 79
5.3 Summary ............................................................... 81

Chapter 6 Discussion ...................................................... 84
6.1 Revisiting The Research Problem .................................. 84
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1.1</td>
<td>Dense Vector Representation</td>
<td>84</td>
</tr>
<tr>
<td>6.1.2</td>
<td>Efficiency and Speed</td>
<td>84</td>
</tr>
<tr>
<td>6.1.3</td>
<td>Feature Selection</td>
<td>87</td>
</tr>
<tr>
<td>6.1.4</td>
<td>Ease of Use in Other Environments</td>
<td>87</td>
</tr>
<tr>
<td>6.2</td>
<td>Reflections on the Proposed Approach</td>
<td>90</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Advantages</td>
<td>90</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Shortcomings</td>
<td>92</td>
</tr>
<tr>
<td>6.3</td>
<td>Challenges Faced During the Research</td>
<td>93</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Access to Computing Resources</td>
<td>93</td>
</tr>
<tr>
<td>6.3.2</td>
<td>Access to Usable Source Code</td>
<td>94</td>
</tr>
<tr>
<td>6.4</td>
<td>Future Work</td>
<td>94</td>
</tr>
<tr>
<td>6.4.1</td>
<td>Explore More Graph Embedding Techniques</td>
<td>94</td>
</tr>
<tr>
<td>6.4.2</td>
<td>Test More Hyperparameter Variations</td>
<td>95</td>
</tr>
<tr>
<td>6.4.3</td>
<td>Test More Games and Environments</td>
<td>95</td>
</tr>
</tbody>
</table>

**Chapter 7**  Conclusion  

**Bibliography**
List of Tables

<table>
<thead>
<tr>
<th>Table 5.1</th>
<th>Varying vector dimensions in node2vec</th>
<th>75</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 5.2</td>
<td>Varying vector dimensions in LINE</td>
<td>75</td>
</tr>
<tr>
<td>Table 5.3</td>
<td>Varying vector dimensions in TGN</td>
<td>76</td>
</tr>
<tr>
<td>Table 5.4</td>
<td>Excluding each category type in node2vec</td>
<td>77</td>
</tr>
<tr>
<td>Table 5.5</td>
<td>Excluding each category type in LINE</td>
<td>77</td>
</tr>
<tr>
<td>Table 5.6</td>
<td>Excluding each category type in TGN</td>
<td>77</td>
</tr>
<tr>
<td>Table 5.7</td>
<td>Varying step size in node2vec</td>
<td>79</td>
</tr>
<tr>
<td>Table 5.8</td>
<td>Varying step size in LINE</td>
<td>79</td>
</tr>
<tr>
<td>Table 5.9</td>
<td>Varying step size in TGN</td>
<td>79</td>
</tr>
<tr>
<td>Table 5.10</td>
<td>Varying number of games in node2vec</td>
<td>80</td>
</tr>
<tr>
<td>Table 5.11</td>
<td>Varying number of games in LINE</td>
<td>80</td>
</tr>
<tr>
<td>Table 5.12</td>
<td>Varying number of games in TGN</td>
<td>80</td>
</tr>
<tr>
<td>Table 5.13</td>
<td>Performance summary: Predicting all interaction types for different number of games</td>
<td>81</td>
</tr>
<tr>
<td>Table 5.14</td>
<td>Performance summary: Predicting all interaction types for different step sizes</td>
<td>81</td>
</tr>
<tr>
<td>Table 5.15</td>
<td>Performance summary: Predicting all interaction types for different dimensions</td>
<td>81</td>
</tr>
</tbody>
</table>
Table 5.16 Performance summary: Predicting all interaction types when excluding different interaction types.
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1.1</td>
<td>Image represented as a matrix</td>
<td>6</td>
</tr>
<tr>
<td>Figure 1.2</td>
<td>CNN operating on an image</td>
<td>7</td>
</tr>
<tr>
<td>Figure 1.3</td>
<td>Video represented as a 4D matrix</td>
<td>8</td>
</tr>
<tr>
<td>Figure 1.4</td>
<td>Audio Signal</td>
<td>9</td>
</tr>
<tr>
<td>Figure 1.5</td>
<td>Spectrogram of an audio signal</td>
<td>9</td>
</tr>
<tr>
<td>Figure 1.6</td>
<td>Example of an adversarial attack using noise signals</td>
<td>13</td>
</tr>
<tr>
<td>Figure 1.7</td>
<td>Example of 1-gram embedding</td>
<td>15</td>
</tr>
<tr>
<td>Figure 1.8</td>
<td>Example of 2-gram embedding</td>
<td>15</td>
</tr>
<tr>
<td>Figure 2.1</td>
<td>The Testing Method</td>
<td>24</td>
</tr>
<tr>
<td>Figure 2.2</td>
<td>The most basic form of vectorization</td>
<td>26</td>
</tr>
<tr>
<td>Figure 3.1</td>
<td>Visual representation of word vectors</td>
<td>29</td>
</tr>
<tr>
<td>Figure 3.2</td>
<td>Neural network architecture of an autoencoder</td>
<td>31</td>
</tr>
<tr>
<td>Figure 4.1</td>
<td>Architecture of the proposed approach</td>
<td>39</td>
</tr>
<tr>
<td>Figure 4.2</td>
<td>Common features found in the DOTA 2 game JSON file</td>
<td>44</td>
</tr>
<tr>
<td>Figure 4.3</td>
<td>Teamfights containing start and end timestamps for forming timeslots</td>
<td>45</td>
</tr>
<tr>
<td>Figure 4.4</td>
<td>Available player data in a team fight</td>
<td>46</td>
</tr>
</tbody>
</table>
Figure 6.1 The system adapting to two different systems with a similar data format 89

Figure 6.2 The interaction format needed by my approach 91
Abstract

Recommendation and personalization applications need prior information about a user, and popular techniques that use neural networks need data in a numerical format. Personalization in video games can help adapt the game environment to suit the player’s behaviour and thus increase engagement. This work explores three node embedding techniques to represent player behaviour in a compact vector form. I used data from the game DOTA 2 to produce vectors using each of the three methods, developed a testing framework, and conducted 270 experiments to explore the effect of different parameters on vector quality. The graph embedding techniques I chose for my approach are node2vec, LINE, and TGN.

I explored different values for parameters, including the number of vector dimensions, the number of games, the types of interactions used in training, and step size for updating vectors over time. The results show that using node2vec and a vector dimension of 512 outperforms the other methods in 13 of the 15 parameter variations that I tested. node2vec also computed the vectors 60 and 10 times faster than LINE and TGN, respectively, on the same amount of data.
Acknowledgements

I want to thank my supervisor, Dr. David Thue, for taking a chance on me two years ago, supporting me throughout this journey, being patient, always taking the time and attention to provide feedback. Thank you, Sara Kim, for being such a great friend and making this journey fun. A big thank you to my parents for always believing in me.
Chapter 1: Introduction

Machine Learning (ML) models need their data to be in numerical form to process it. So it is essential to convert real-world data like images, videos, text and graphs into a numerical format (i.e. a vector) to feed them into an ML model. It is also crucial to minimize the size of the vector to ensure that the ML model is smaller in size and thus reduce the possibility of overfitting. We should reduce the vector size while also ensuring that the vector truly represents the input.

Embeddings encode objects such as words or images into dense vector representations in a low-dimensional space. They also try to encode the semantic meaning of those objects, so it’s easier to compare the objects using ML algorithms. For example, word2vec embeds words into a low-dimensional space using a neural network [4]. The addition and subtraction of these vectors result in new representations in the vector space. For example, adding the vectors for a car and the colour red can construct a vector representation for a ”red car.” The idea improved Natural Language Processing applications like Sentiment Analysis, and Language Translation [1] [27].

What if we could embed human behaviour as a vector? Research by Mordatch showed that it is possible to embed simple concepts as vectors successfully [23]. Du and Mordatch extended the same idea to represent images as vectors and provided an open-source code implementation [7]. They also showed how to compose two image
embeddings and produce new images from the resultant embedding.

The goal of my thesis is to compare different potential methods for getting vectors of human behaviours. The capacity to get vectors for a behaviour means we can get vectors for any two behaviours, map them to the vector space and compare the distance between them. The distance comparison allows the implementation of other personalizing-based applications. Such applications could, in theory, analyze someone’s behaviour in real-time as some interaction happens, and in turn, customize the interaction in the future.

The benefit of having a vector for human behaviour is that we could pre-train models to embed certain generic behaviours. Downstream applications such as recommender systems, advertising engines, interactive games could then fine-tune the model to embed a more specific behaviour through transfer learning. Also, if we know a vector represents a particular behaviour, it simplifies recognizing such behaviour. Knowing what type of user interacts with the system and fitting into one specific class can help customize the user-system interaction. Big advertising-focused corporations like Google, Facebook, Netflix, and YouTube have their products centred around recognizing user behaviours and tailoring the whole interaction to that user. It shows how there is a big market for research similar to this.

Computer games can significantly benefit from this research since personalization significantly impacts player engagement [18]. Having vectors of player behaviour would make it easy to compare how similar two behaviours are and allow creators to know a player’s style quickly and adjust a game’s elements with that knowledge.
Computer games are also the perfect candidates for testing this idea in isolation. They provide a controlled and reproducible environment, which helps streamline research. This idea can benefit the industry, but there is a lack of research in this specific area, as explained in the related works section.

In this research, I have developed an architecture to understand how three ML techniques can represent user behaviours as vectors, and I evaluated their performance using data from a commercial game. The following points highlight specific details about my research contributions:

1. I used three graph embedding techniques to derive vectors representing a player’s behaviours in the video game DOTA 2 (Defense of the Ancients).

2. I developed a framework to test if vector embeddings accurately represent in-game entities qualitatively.

3. I conducted 270 experiments to explore the effect of parameters on vector quality across all three vector embedding techniques.

I have organized the remainder of this dissertation as follows:

1. Introduction - Background about Deep Learning (DL) methods, past work in this research area, and arguments favouring using interaction data from the OpenDOTA API.

2. Problem Formulation - Laying out the motivation and describing success criteria for vector embedding methods for player behaviour.
3. Related Works - Critiques of 7 papers closely linked to this research and reasons showing its uniqueness.

4. Proposed Approach - A walkthrough of the vector production framework and a summary of the three tested techniques.

5. Evaluation - Explanation of the testing methodology and experiment results.

6. Discussion - Critique of the proposed approach given the requirements laid out in Chapter 2. Reflections on the performance of each graph embedding technique. Weighing the pros and cons of this research, along with future work ideas.

1.0.1 The Purpose Of Having Vectors

Behaviours are subjective; humans judgement to compare with each other, and doing so is even more difficult for computers. I aim to quantify behaviours by embedding them in a real-valued vector space. Having them in numerical form means they can be compared during an interaction and inform the system’s choices in future interactions with a particular user. There has been minimal research investigating this area, but I have designed this approach using Machine Learning (ML). The following section highlights why I have used ML to approach this problem.
1.1 Background

Deep learning is a methodology currently receiving much attention from the research community. It describes a family of learning algorithms rather than a single method, which can learn complex prediction models, e.g., multi-layer neural networks with many hidden units. Furthermore, we can design deep learning models in many ways to suit specific applications [20]. That makes them able to support a large variety of inputs and applications.

1.1.1 Applications

Deep Learning has been the driving force behind several cutting-edge advances in multiple real-world applications [20]. He et al. showcased how convolutional neural networks can classify images in the ImageNet dataset better than humans [13]. GPT-3 is a text generation/completion network based on transformers released in 2020, which outperformed most state-of-the-art NLP benchmarks [2]. AlphaGo was able to defeat the world champion in the game of Go [26]. Neural networks can complete any task given enough data and compute since they have been proven to be universal approximators [3]. This huge potential makes deep learning the perfect vehicle to encapsulate human behaviour and represent it as a vector.

1.1.2 The Types of Inputs Supported

The only type of input neural networks support is an array of floating-point numbers, but we can adapt other input types to represent them as floating-point numbers.
There are many ways to encode text as a vector, like n-grams, word2vec and BERT. N-grams store information about which words or word combinations are present in a sentence as zeros and ones to generate a vector. word2vec uses surrounding words to create a contextual understanding of a word [4]. BERT uses a weighted combination of each word in the sentence to generate a word vector [6]. It uses an attention mechanism to generate those weights.

Images

Images can be seen as a 3D matrix of RGB values, as shown in figure 1.1. There are deep learning networks such as Convolutional Neural Networks that are well suited to images. Figure 1.2 shows a CNN operating on an image.

Figure 1.1: Image represented as a matrix
Videos

We can view videos as a collection of image frames. Each image frame is an RGB pixel matrix. Therefore, videos can be seen as a 4D matrix, as shown in figure 1.3. According to recent research, CNNs perform best on tasks like video recognition [29].
Audio

We can represent audio as a time-series signal of sound amplitudes. Therefore, it’s possible to divide the data into parts using sliding windows to make it a fixed length.
Figure 1.4 illustrates that idea. We can then provide this input to Recurrent Neural Networks for training.

A spectrogram presents a second approach to deal with audio data. For any period in an audio file, we can plot the distribution of the frequencies. The histogram would change as time passes, and we can line up many of those to form an image, as shown in figure 1.5.
Converting audio data into image form means leveraging powerful well-established image processing techniques to train a neural network. This image can now be fed to a neural network using a Convolutional Neural Network (CNN) described previously.

Other Data Types

Other data types, such as graphs, point clouds, or time series, need fine-tuned custom neural networks to leverage these data types’ inherent structure. For example, graph networks need message passing mechanisms, graph convolutions and matrix factorization methods to utilize them for deep learning methods [31]. However, I haven’t discussed those in the interest of brevity.

1.1.3 Benefits of Deep Learning

No Feature Selection

Deep Learning partially eliminates the need for feature selection, assuming that we have sufficient training data available.

Selecting features from the inputs can be presented as a “function” that the deep learning model needs to learn, which is why a DL model can learn to choose the right features given enough data. We assume the model is complex enough to fit the data.
Fits Complex Data

Deep neural networks are universal approximators to any function, assuming sufficient training data and an adequate number of parameters in the model. That is a significant advantage since we can use them for more complex applications than simple classifications. Examples include playing games or summarizing documents [9] [26].

Support Many Types of Inputs

Neural networks need all their inputs to be floating-point numbers, but we can adapt many input types to be represented as floating-point numbers, as explained earlier. That expands the kind of inputs supported to text, images, audio, or video.

1.1.4 Limitations

While being very popular and performant in execution, neural networks have several downsides, which are currently active research areas in the community.

Lack of Transparency

Deep learning models are black-box models, and trying to design transparent tunable models is an essential recent research question [12].

The Need for Data

A neural network is highly inefficient in terms of the number of samples required to train for a task. The number of samples needed to fit a simple image classification
neural network ranges from several thousand to hundreds of thousands. For example, for researchers working on the ImageNet classification, it’s normal to augment the one million image dataset by 2 or 3 times to achieve a state-of-the-art performance [17]. While transfer learning and data augmentation mitigate this issue partially, they are not particularly useful when dealing with unseen data.

**Involves Some Feature Selection**

Real-world data is often incomplete and hard to find in abundance for various reasons. The more complex the data, the larger a model is needed, and likewise, the number of data points or the model will underfit. If the number of data points is not enough, the complexity of the data has to go down, or the model has to be smaller—usually, it’s a combination of both. Making a smaller model is easy enough; the problem is with making the data less complex. It involves a human analyzing the available features and selecting the more valuable ones. The features are not entirely hand-picked, but they aren’t wholly automated either. There is a compromise due to a lack of data.

There’s also a problem of training a large enough model. Usually, the performance of the hardware used to prepare the models is a significant bottleneck when training deep learning models.

**Lack of Global Generalization**

Deep learning models tend to overfit the data they’re trained on and not generalize beyond that [19]. That means that the models perform well as long the distribution
of the testing data is close to the training data distribution. Once the input data deviate far, the model stops functioning correctly.

Introducing some noise in an image can trick a deep learning classifier into misclassifying that image [10]. For example, figure 1.6 shows how a picture of a dog is combined with noise to trick the model into classifying it as a car with high confidence. Even after introducing the noise, the image doesn’t seem to change much, but it’s enough to fool the model. These attacks of intentionally making a model misclassify are known as an adversarial attacks.

![Figure 1.6: Example of an adversarial attack using noise signals](image-url)
This problem extends beyond image classification and applies to many other deep learning models. That means the model has only learned to classify images superficially and not how humans classify them.

Deep learning models are fragile in the sense that we’re not able to assess how they make their decisions, and they often fail to perform in the most obvious use cases. A model can articulate text robustly and exhibits an excellent general understanding of the world while also failing at the most basic and straightforward math tasks that should be easy given its competence at writing logical articles [9]. It’s an indication of a fragile system, which doesn’t truly learn/generalize but manages to function within the data distribution it has seen in training.

1.2 Representing Data as Vectors

In this section, I discuss the example of N-grams and how word2vec solved their shortcomings using word embeddings. I discuss the benefits of word embeddings and how behaviour modelling can get the same benefits using my approach. The example of N-grams and word2vec will help clarify the utility and significance of lower-dimensional vector representations.

1.2.1 N-grams

We see widespread usage of Machine Learning models as classifiers in natural language processing (NLP). For using ML models, the input for those models has to be a set of floating-point numbers (vectors). It’s a challenge to represent words as numbers
correctly. The solutions have evolved, but using N-grams is the most basic way to express sentences as vectors.

N-grams construct vectors using information about the presence of individual words or their combinations in sentences. Figures 1.7 and 1.8 show an example of 1-gram and 2-gram embeddings, respectively. We can apply the same logic to form 3-gram or 4-gram embeddings.

<table>
<thead>
<tr>
<th>Sentence</th>
<th>I play</th>
<th>play</th>
<th>guitar</th>
<th>piano</th>
<th>daily</th>
</tr>
</thead>
<tbody>
<tr>
<td>I play guitar</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>I play piano</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>I play daily</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 1.7: Example of 1-gram embedding

<table>
<thead>
<tr>
<th>Sentence</th>
<th>I play</th>
<th>I guitar</th>
<th>I daily</th>
<th>play guitar</th>
<th>play piano</th>
<th>play daily</th>
<th>piano guitar</th>
<th>piano daily</th>
<th>guitar daily</th>
</tr>
</thead>
<tbody>
<tr>
<td>I play guitar</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>I play piano</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>I play daily</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 1.8: Example of 2-gram embedding

Real-world datasets typically have words in the order of thousands. So if a sentence composed of a few words is represented as a vector this way, only a tiny fraction of 1s end up in the vector, the rest of it being 0. That leads to highly sparse and memory inefficient vectors for each training sample. The more the size of the training data
vectors, the more layers and parameters are needed for fitted that data using a deep learning model. That also means that more data is required to train the model.

The only information an ML model receives is what words are present and along with which other words. Interpreting the deeper meaning of a word or a sentence is still left for the machine learning model to do implicitly. Challenges are abundant with this method’s usage, and that’s why modern NLP systems rarely use this.

1.2.2 The Need For a Dense Vector Representation

2-gram, 3-gram, and 4-gram usually have a vector size of tens of thousands of dimensions. That is a highly inefficient way to vectorize words. Higher dimensions of input samples mean exponentially more parameters are needed in machine learning to accommodate that.

Ideally, the vectorization method should convert each sentence in the text into a dense smaller-sized vector that encapsulates the meaning of the word, the position, and how it relates to other words in the same sentence. The size should be small enough not to need an astronomical number of parameters in the deep learning model.

1.2.3 Using word2vec to Generate Dense Vectors

word2vec presents a solution to achieve that objective by delivering an intuitive way to construct word vectors. Given the word, the word2vec neural network predicts the N-words that surround it (the contextual words). A deep understanding of the word’s meaning is essential to succeed at this task, making word2vec effective.
1.2.4 How Vectors Facilitate Transfer Learning

After training word2vec on a vast dataset, vectors are extracted from the model and are ready to use anywhere else. I have provided a more detailed explanation in the related works section.

If we use a large dataset, the model receives a broader exposure to words and can, in turn, produce better vectors. But after extracting the vectors, they can be used by anyone in other applications and texts. For example, large word2vec models are often available for research purposes like the one provided by Google. ¹ Researchers train these models on millions of documents and publish them online. All the user has to do is download the vectors and use them. That frees the user from training the model, which thus enables the easy development of applications.

Since the vectors are trained on large datasets, they encode a vast amount of prior information about the words. That means that the end-user of these vectors doesn’t need to gather much data. Thus, word2vec enables training application on vastly smaller datasets using transfer learning than training a brand new model. These vectors already contain much prior information, and it saves time for other engineers.

Deep learning models need a massive amount of data to train for a specific task. The models use up most of the data to create a meaningful context for the deeper layers and only a fraction of the particular task. Word2vec takes away the need for creating context since the vectors provide a context equivalent to training ultra-large datasets. That is why word2vec enables transfer learning. It allows using less

¹https://code.google.com/archive/p/word2vec/
data and still having superior performance. That is common in image classifying applications [16].

Word2vec does it for words, and my goal is to solve the same problem for human behaviours. Once it’s possible to train a vector to encode a specific behaviour, it enables many other secondary applications that can leverage those vectors. We can do the training on vast general datasets of behaviours, and the secondary tasks can fine-tune the resultant vectors depending on the application.

1.3 Behaviour Modeling

1.3.1 Conventional Methods

As shown by Eggert and Pirker, conventional methods rely on handcrafted features to represent the behaviour of a game player [8] [24]. Traditional methods may be more relevant, accurate and effective in the short term, but it’s essential to acknowledge the potential deep learning has for this task and develop an effective solution. The most significant upside of deep learning is drawing upon large amounts of data and not selecting features by hand. Traditional methods are inadequate for the exact reasons why vectors are a good idea.

1.3.2 Usage of Deep Learning

Khameneh et al. use an autoencoder neural network to represent game objects as vectors [14]. It successfully uses physics-based features such as velocity, position and size
to represent objects, but that is not my primary research objective because physics-related information does not significantly overlap with an object’s behaviour. Eggert et al. try to use supervised machine learning techniques to predict which roles players are likely to take on in the gameplay based on their gameplay statistics [8]. The work focuses on the prediction of player activity which is a secondary goal of my research. The primary goal remains to have a dense latent representation of behaviour that any other application such as activity prediction can leverage. Pirker et al. observe a set of statistics for each game player as they progress through the game [24]. While the research doesn’t explicitly work on deriving vector representations, it’s essential to study their approach and choice of features to describe a player. Makantasis et al. use audio and visual information from the game to represent the expected reaction of a player as a vector [21]. That vector is a generalized representation of what response a system can expect from a viewer/player. But I’m more focused on extracting a generalized representation of the player’s gameplay behaviour. Klingler et al. use information about students in an online learning system to generate a general vector representation using variational autoencoders [15]. While this method is effective in a small online student learning environment, its performance in other larger datasets is still unknown.

1.4 Reasons To Use a Game Dataset For Behaviour Modeling

I decided to test all of my experiments using the OpenDOTA dataset, which provides move-by-move gameplay logs of the online multiplayer game DOTA 2. There are
several reasons why that dataset was best suited for my thesis. The following reasons also apply to using game datasets for running interactive modelling experiments in general.

1.4.1 Ease of Data Access

There’s an abundance of game traces available online. Getting data is as easy as making an API call, and we receive a complete action-by-action breakdown of an entire game. Collecting data ourselves would be a challenging, time-consuming and complex endeavour. So instead, I chose to focus on taking the openly available data online and focus on the main idea of behaviours as vectors.

1.4.2 Volume of Data

Collecting interaction data in person entails providing incentives to people, usually needing 10 to 20 dollars for their time. That is a severe obstacle for the number of samples we would obtain using the method. Moreover, even if we managed to collect many samples with painstaking effort and after investing much time, we might have to let go of some samples due to errors.

Millions of players play online games every day, and many games choose to store a play-by-play data log for each game. If we can tap into that data, it becomes easier to work at a scale of thousands of samples instead of working with dozens of samples from a real-life data collection with human participants.
1.4.3 More Confined Environment

Using games allows me to narrow down the number of variables compared to in-person experiments, which helps make the experiments easier to control and reproduce. Reducing the number of variables also means that the complexity of deep learning models needed to fit that data is less than more complex applications like image/video classification or language translation models.

1.4.4 Better Reproducibility

This topic builds upon the previous advantage. Since there are a limited number of variables, it’s easier to reproduce a particular situation in a game compared to other in-person interactions. In addition, using games means there is generally a way to obtain game traces or logs reliably and consistently in a defined data structure. That reduces experiment logistics and reduces the time taken to gather data.

1.5 Summary

I presented deep learning, including its capabilities, limitations, and how it adapts to different data types. Using an example of word2vec, I explained the utility and value of having vectors in the context of NLP and behaviour modelling. Lastly, I presented arguments favouring using a game dataset to obtain the behaviour data for this research.
Chapter 2: Problem Formulation

The overarching goal of my research is to gain a new understanding of how different vector representation methods perform on player behaviour data. In my research, I have unified three vector representation methods into a common experimental framework and tested them on interaction data from the video game DOTA 2.

2.1 Challenges Involved

Any vector generation framework for behaviours needs to satisfy the following constraints to be usable in real-life scenarios:

1. Storing and using a complete behaviour log is resource-heavy. Therefore vectors that replace the behaviours data need to be small and compact in comparison to enable training models with fewer parameters and less training data.

2. They also need to represent the player behaviour in a meaningful way.

3. The methods need to be loosely coupled with the input data format so that it’s easy to apply them to other environments.

4. The production or the usage of the vectors needs to consume as few compute resources as possible.

I explain each of these points in more detail in the following sections.
2.1.1 Dense Vector Representation

The goal of having behaviour vectors is to use them for downstream applications in machine learning models. The higher the vector size, the more training samples these ML models need for learning the pattern and converging to optimum weights. Therefore, the vector production method for representing behaviours should produce vectors that are smaller in size. That makes it easier for secondary tasks to utilize the vectors effectively.

2.1.2 Meaningful Vector Representation

Let’s assume that we have access to the interaction data of many entities within an environment and a system capable of producing N-dimensional vectors for each entity using the interaction data. We expect the vectors to serve as a replacement for all the interaction data. If we have a vector for a person interacting with a system, that vector should represent all the interactions a person has done or could do with the environment.

The quality of data plays a huge role in the performance of neural networks [30]. So it’s possible to train a neural network to do something that requires these vectors and use the network’s performance to gauge the quality of the input data. To judge if the vectors are a good representation of each entity’s behaviour, we will test their ability to inform a linear classifier that tries to predict behaviour.

The vectors serve as a compact replacement of all behaviour data for a player. In our case, we will use a linear classifier to test if the vectors have sufficient data in
them to predict an interaction in the future. Figure 2.1 illustrates this.

![Diagram of the Testing Method](image)

Figure 2.1: The Testing Method

It is possible to use other, deeper neural networks to make this prediction, but using a linear classifier makes it a more strict test for the vector generation method. Seeing higher prediction accuracy from this test would indicate a higher semantic quality of the vectors.

One could use F1 score as a metric, since it is robust in face of imbalanced data. I designed a balanced dataset of two classes, which means there is a very clear baseline score of 50 percent accuracy, or near 50 percent in the case of small imbalances. That helps contextualize the model’s performance on different experiments. That is the reason I decided to use accuracy as the metric.
2.1.3 Efficiency and Speed

Real-time applications would need the method to process new data and function in real-time. A game player can make dozens of interactions in a minute, and the vectors should be modified to account for those new interactions. This update process should be as fast as possible to keep up with the speed of a player’s actions. Not only should this vector update happen with low latency, but it should also require minimal resources from the host system.

Efficiency is critical because we need vectors not only for one entity’s behaviour; we need vectors for each entity in the interaction to be represented. There might be several players and game entities involved in the interaction, and we also need to update their vectors in minimal time.

2.1.4 Feature Selection

We have to select what data goes into making a vector, and usually, there is a human involved in this decision. We can easily show someone’s behaviour as a vector by writing all their data in one dictionary and converting that into binary. That would remove the need for human intervention, but that binary string would produce huge vectors even for a small amount of input data. For example, each character takes one byte of space, so using the complete data for a person would involve using megabytes worth of storage. A vector that is just a list of 500 numbers would take only two kilobytes of space.
It is possible to handpick features and arrange a vector representation by hand. That would work in theory, but it does have a significant downside. A human needs to understand all the variables for each new environment, make a subjective judgment, and pick the most meaningful ones to make the resulting vector smaller in size. Since this path involves the help of a human, it means the introduction of subjective judgement into the process and a higher probability of errors. That also means that for every new environment, a person or an entire team would have to dedicate effort to choose variables for that new environment.

For constructing a vector, we can decide what data makes up the vector or influences the making of the vector. The method should have an easy way of selecting...
the relevant data without any help from humans. Feature selection is a problem deep learning is supposed to solve. In theory, the algorithm should take all data, decide what’s important, and assign more weight to it.

It would be good to know what data is making up the vector, but that’s not essential. Instead, the most critical requirement is to require minimal human effort to select what makes up a vector.

2.1.5 Ease of Use in Other Environments

Close coupling between vector generation systems and environments would work well for the coupled environment but would be harder to port to other environments. Doing that would need human effort. The design should be able to move to new environments with minimal effort. This requirement and the previous one about feature selection both focus on minimizing human involvement.

The system should rely on the types of data that are commonly available in multiple types of environments. For example, its performance or working shouldn’t be closely attached to a particular variable in a system since that will hinder the portability of the entire system. We’re looking for a system that is not too dependant on the way one particular environment functions. The design should be at such a level of abstraction that shifting environments wouldn’t handicap its performance or functionality in any way.
Chapter 3: Related Work

I have summarized and analyzed seven works that are related to and have inspired my research direction.

3.1 The Effectiveness of word2vec

In the introduction chapter, I explained how using deep learning methods can be good candidates for pursuing my research. They cannot take words as input for their tasks, so it is essential to encode and represent them in a numerical format. word2vec makes it possible [4]. It presents an approach to derive meaningful vector representations of a word, which we can use to feed a deep learning model without issues. We can find several other more performant ways, but word2vec has a simplicity to its approach and was the pioneer of making vectors for words. My research builds on the conceptual foundation laid by word2vec, so I consider it essential to look at the functioning of word2vec.

It takes a corpus of text to train itself and creates an N-dimensional vector for each word in the corpus. The vector representation embeds the semantic meaning of the words. The larger the text corpus, the better the vectors are. The Euclidean distance between two vectors represents the semantic similarity of any two words. Figure 3.1 illustrates that idea more clearly.
Since the vectors represent the semantic meaning of their words, other researchers can effortlessly use vectors produced by someone else. The word2vec vectors trained by Google on the Google News dataset is an excellent example of this benefit. The dataset has over 100B words, and Google provided the vectors free of charge\(^1\). That makes it easy for other researchers to use Google’s vectors in their experiments and harness the performance of vectors trained on such a large dataset.

\[\text{Figure 3.1: Visual representation of word vectors}\]

**Gap Analysis**

It is possible to use word2vec directly in the construction of behaviour vectors. I explain and critique the approach in chapter 4.

\(^1\)https://code.google.com/archive/p/word2vec/
3.2 Representing Game Objects as Vectors using Physics Information

Khameneh et al. present an approach to derive entity embeddings of game objects using their visual and mechanical data [14]. An autoencoder deep learning network takes that data and embeds it in a lower dimension in the vector space. The overall goal of this paper was to have a generalized representation of game objects. Procedural Content Generators can use these representations in the future.

They use mechanical information about objects like size, velocity, and location as attributes to describe the object. After handpicking data and organizing it in a vector form, each game object has a vector of size 1600. The authors selected a few attributes, but their vector representation ends up making the vector size 1600. A variational autoencoder network takes the data as input and works in two steps. At first, it tries to compress the data into 25 dimensions. After that, it tries to recreate the original vector from the compressed one. The network compares the squared sum of the input vector and the recreated version to receive feedback on its performance. This calculation informs the network about how similar the resulting version is to the original. When the network converges and it’s able to generate the original version from the 25-D vector, it indicates that the compressed vector is a good representation of the full-sized vector. Eventually, the network can take attributes of an unseen game object and compress it to a lower dimension. Figure 3.2 showcases the neural network architecture for this study.
Gap Analysis

This paper aligns strongly with the direction of my research since it explores a way to represent game objects in a vector space. However, the information used to produce these embeddings is all physics-based information like the position or location of the object. My research focuses on representing behaviours as a vector, not just game
objects. The method can take a large vector of an object with all the games obtained from the game and compress it down to 25 dimensions, but that’s the extent of its capabilities. It doesn’t try to or have the capacity to take the behaviour of an object into account. The attributes used to generate the embeddings are also too specific and tightly coupled to the game. Applying the same method to another game would need a person to reselect the essential features of an object from scratch.

3.3 Predicting a Player’s Role in DOTA 2

DOTA 2 is an online multiplayer game that also provides move-by-move interaction logs for each player. Players often converge to a specific playing style or a role in the game to tackle the game challenges successfully. These roles are not formally defined, but players implicitly take them. Eggert et al. have identified nine such specific roles in the game into which they are trying to classify the players [8]. They did not intend for the roles to indicate how good a player is; they focused more on the style of playing. They handpicked 18 attributes from the available ones and used them for the classification task. They successfully classified the players into nine roles with 75 percent accuracy using logistic regression. If the number of classes is reduced and narrowed down, the accuracy increases to 96 percent. They concluded that logistic regression is the best algorithm for this task.
Gap Analysis

The authors derived more than half of the 18 attributes from manually labelling the data after watching the entire game. Since the authors could not do the labelling themselves, they crowdsourced that task. While the technique may successfully predict player roles in the current game, it would take a tremendous amount of effort to recreate this in another game. Because doing that means starting from a blank slate again and designing handcrafted features. It might also involve manual annotations by humans, as done in this research. The features and the game have very tight coupling, which means it’s challenging to achieve the same result in a different environment. The focus was to classify players into nine categories. The authors fine-tuned the attributes to optimize for that specific task, which means these attributes might not have the information necessary to perform well on other tasks, like representing behaviours. The central focus of this paper is more on classifying players, whereas my focus is more on representing the players’ behaviours in a vector space. My focus is to make these vectors more general than the ones in this research.

3.4 Observing Player Style from Gameplay in Just Cause 2

Pirker et al. tracked some critical metrics of Just Cause 2 gameplayers and analyzed how they changed as the players progressed through the game’s main storyline [24]. They observed which attributes varied and which ones remained constant as the players progressed. For each level in the game, players were grouped in clusters using
Archetypal Analysis. The authors tracked how players moved between clusters as the

game progressed.

Just Cause 2 is an open-world game, and the authors used data for 5000 players
from 2010. The game’s premise is that the player has to cause chaos in an open-
world environment to progress through the levels. The dataset contains data about
the player’s coordinates, chaos, kills, vehicles driven, distance driven, and timestamps
for all those interactions. The authors visualized the flow of players across clusters
as the levels progress. The authors used Sankey flow diagrams data to accomplish
that since the diagrams allow a visual inspection of the player’s movement between
clusters. Out of the many available features, the authors have handpicked only a
subset of features for characterizing players’ behaviours. According to the author’s
observations, some player attributes remain constant as the game progresses, while
some shift with the player’s style.

Gap Analysis

The features chosen by the authors are particular to the core mechanics of the game.
They included aggregate level features like kills per minute and kills to deaths ratio.
The data is on a higher level of abstraction than what we would need to predict a
specific future interaction for the player. The features may uniquely classify play-
ners into roles and track their progress through the game but may not hold enough
information for larger, more general tasks like representing behaviours.
3.5 Representing Emotions using Vectors

Makantasis et al. aimed to capture a general-purpose representation of player emotion using deep learning techniques [21]. They propose that audio and visual data from a game is enough to predict what emotion it might evoke in a player. It’s possible to predict that emotion even before presenting the scene to a player. The research also identified specific regions on the screen that lead to a particular feeling.

The authors built two deep learning networks, one taking video and the other taking audio as input. These networks process their respective inputs, and another network combines their output to form a vector. The vector is supposed to embed the expected emotion the scene will evoke in the player. A third network uses that vector to predict the player’s sentiment for that given ”scene vector.” The two networks are penalized based on how well the third network can predict emotions. The authors can predict the expected sentiment of the player with high accuracy using this framework. When the network takes a scene as input and predicts the emotion, the authors can build a heatmap on the parts of the image that contributed to the network’s decision. That means they can identify what emotion a scene might evoke in a player and the part of the scene responsible for that. They also tested this approach on various genres of games to prove that it’s genuinely generalized.
Gap Analysis

This paper tries to construct vectors of something other than behaviour, but it’s valuable to observe the methodology used to do that. The authors focused on the player playing and predicting how they would feel, but my focus is on how the player is playing and capturing their behaviour in a vector. Both goals focus on the player but in different directions.

3.6 Improving Learning Disability Detection Using Vectors

Klingler et al. tried to improve the accuracy of detecting developmental dyscalculia in students [15]. While their focus was not on having generalized vector representations, they produce and use them to reach their goal. They used a variational autoencoder to obtain vector embeddings for each student’s data, which helped improve the detection of the learning disability. They used two types of networks for the encoder and the decoder; a fully connected network and a convolutional neural network.

They handpicked essential features for the students and tracked them across time. Since these features are constantly changing, they took snapshots of their values and compiled them in a matrix. Conventional methods choose to feed that matrix directly into a neural network that predicts learning disability. But they had access to many unlabeled data that they could leverage to improve the classification task. They trained an autoencoder to compress the matrix into a smaller vector and recreate it into a matrix form. In the process, it learned to retain the essential information and
reject the rest. The encoder part of the network can then take new data and compress it to a lower dimension, which they use for the classification task instead of the larger matrix. This process is known as semi-supervised learning, where unlabeled data can augment the classification task using an autoencoder.

**Gap Analysis**

While the encoder’s output is equivalent to the vector representation I’m pursuing, it is essential to note that the encoder needs fixed-length data from the entire interaction to generate the vector. There is no straightforward way to reproduce the vector to account for a new set of interactions. The vector is limited to representing a student, not the entities within the learning environment. This model would also fail when the number of interactions increases or decreases since it takes a fixed-length input. The only way to account for a different type of input is to train a new model. The input also used handpicked features, which makes it hard to replicate easily in a new environment.

**3.7 Predicting Viewer Engagement Using Game Features**

Melhart et al. focus on predicting the engagement of live stream watchers based on the gameplay metrics in the game PUBG [22]. They were able to achieve the goal with 83 percent accuracy. They measured engagement as the number of chat messages in the live stream. This research provides insight into how feature selection happens in a game.
PUBG provides an API with comprehensive data about all the games on the platform. The authors selected five streamers with enough data to suit a machine learning model and narrowed the API’s data down to 40 features. A deep learning model uses these 40 features to predict a score between 0 and 1. They compare that score with the inverse of the number of chat messages and penalize the model based on the difference. This process happens in each frame. Even though the authors have access to data for all players, they only give the model for the specific player and compare the score with messages for that particular player.

**Gap Analysis**

The PUBG API provides comprehensive data about each player, so the authors only work with each player’s data. But they note that they didn’t make any feature selection in that data beyond the 40 functional features, making the approach more generalizable and applicable to other games or environments. The critical gap here is that the authors took the feature for one timestamp and used the model to predict engagement. They ignore the past feature values and trends, which could potentially provide meaningful insight. They also assume that more chat messages equate to more engagement. While the proposed approach does work for predicting the number of messages, the authors do not have access to the ground truth, the absolute engagement level of the viewers. The amount of chat messages is just a proxy for that metric.
Chapter 4: Proposed Approach

4.1 High-Level Overview

![Diagram of proposed approach](image)

Figure 4.1: Architecture of the proposed approach

My goal is to test the performance of different graph embedding methods on behaviour data from the DOTA 2 game. Figure 4.1 shows a visual flow of my approach along
with step numbers. After getting log data from OpenDOTA API (step 1), I clean the
data and select the interactions necessary for the other steps (step 2). After that, I
partition the data by reserving ten percent of it for testing and use the remaining data
to generate a graph (step 3). Step 4 includes using the graph to create vectors. In
step 5, I use the fraction reserved for testing and the vectors developed in the previous
step to generate testing samples. I finally use those testing samples to benchmark
the quality of the vectors in step 5. I will describe each step in more detail later in
this chapter.

4.1.1 The Method for Getting Vectors

Several graph embedding techniques exist in the research community, such as node2vec,
TGN and LINE [11] [25] [28]. These methods have strong performance on standard
graph embedding benchmarks indicating that they are robust. It is possible to trans-
late game data to a graph and use these graph embedding methods to produce vectors
for the graph. Representing game data as a graph is a challenge, and I have explored
one way to do that. I have described the system in more detail in the following
sections.

4.1.2 How is the graph made?

I made game entities into graph nodes, and an interaction between two entities is
considered an edge. Since there is action by action data available in the OpenDOTA
API response, this allows me to make a detailed, connected graph of the entities
involved. I describe the process in more detail in the following sections.

4.1.3 What are we using to embed the graph nodes as vectors?

The graphs constructed in the previous step can work as the input for graph embedding techniques. Out of the several varieties in the research community, I have chosen node2vec, LINE, and TGN. Having easy access to the open-source code was my primary reason, but I have laid out more specific reasons for selecting them in the next paragraph.

I chose node2vec because it used word2vec internally, which performs well on numerous datasets [11]. I chose LINE for the simplicity and effectiveness of its approach since the authors avoid complicated mathematics and achieve performance comparable to other popular node embedding benchmarks [28]. Finally, I have selected TGN since it outperformed almost all the techniques in the standard node embedding benchmarks [25].

The standard benchmarks involve datasets of citations and social media where they use resultant vectors to predict a link between two papers or users of social media long with classifying a vector [5]. My approach involves using interaction data from games that are not in a graph format. So, I adapted that data into a graph. I have targeted my approach towards real-time usage in a game, so I adjusted the benchmark to suit the application.

Storing detailed behaviour data takes several megabytes of data compared to
keeping a single vector of 2 kilobytes to replace that. Having low-dimensional representations of entities in interaction is a novel application, and there is still a lack of research in this niche. Chapter 5 will discuss how the proposed approach stacks against the criteria set in the problem formulation chapter.

4.2 The OpenDOTA Dataset

Investigating behaviours and quantifying them using Machine Learning models means that it’s essential to have data for training those models. We need access to an extensive reservoir of user interaction data on which the models can rely. This data also needs to be detailed and plentiful for the model to extract useful information. I expect the final application of these kinds of models to be in games and other personalization-based applications. So it makes sense to work with a game dataset for the pilot testing. I have highlighted other arguments for using a game dataset in chapter 1.

4.2.1 Reasons For Choosing The OpenDOTA API

DOTA 2 is a popular massively multiplayer online role-playing game with millions of active daily users. During gameplay, players interact with many objects and other players. Therefore, we can expect the interaction log to be quite detailed. I have chosen to use player data for various reasons, but the primary ones are the massive volume of available data and the ease of getting access.

DOTA has an easy-to-access API, which has a cost of one dollar per 10,000 calls.
I accessed the logs for 700k games during my research, rejected most due to lack of complete data, and kept 3.3k games for my experiments. The entire set of API calls cost less than 30 CAD. This economic access was essential to making the whole study happen since having a large volume of data for training the machine learning models is vital.

The data provided by the API is in JSON format. There are many ways to access the information from the API, like getting games played by a specific player or a team. In this case, I decided to access information about individual games. The API provides a detailed log of interactions during the gameplay and the timestamps for those interactions. In the next section, I explain how this data structure helped me construct its graph representation.

The number of games available from the API is in the order of millions, which is perfect since deep learning models often need vast amounts of data to converge.

The data represents real people playing games but can be accessed quickly and cheaply. The same data would cost much more time to obtain if taken manually in an in-person setting. The scope of applications for my approach may be broader than just computer games, but I have chosen to use this API for my experiments since it is cheap, plentiful, and generated by real people. An important note is that applications like these that require working with and analyzing player data are the primary targets for my approach, but using a game dataset doesn’t limit the study in any way.
4.2.2 Available Attributes

When a user queries the API for a game with a specific ID, the server responds with data for the entire game. Some of this data is common to all players, like match ID, duration, and replay URL. Some data is specific to the player, like a log of purchases and kills.

Figure 4.2 lists all the common attributes obtained from the API response for a single game. These features are not valuable for my approach since I aim to leverage player-specific features and construct a vector for describing the player’s behaviour.

<table>
<thead>
<tr>
<th>Match ID</th>
<th>Game Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barracks Status Dire</td>
<td>Human Players</td>
</tr>
<tr>
<td>Barracks Status Radiant</td>
<td>League ID</td>
</tr>
<tr>
<td>Chat</td>
<td>Lobby Type</td>
</tr>
<tr>
<td>Cluster</td>
<td>Match Sequence Number</td>
</tr>
<tr>
<td>Radiant Score</td>
<td>Picks Bans</td>
</tr>
<tr>
<td>Dire Score</td>
<td>Negative Votes</td>
</tr>
<tr>
<td>Dire Team ID</td>
<td>Positive Votes</td>
</tr>
<tr>
<td>Duration</td>
<td>Objectives</td>
</tr>
<tr>
<td>First Blood Time</td>
<td>Start Time</td>
</tr>
<tr>
<td>Lobby Type</td>
<td>Tower Status Dire</td>
</tr>
<tr>
<td>League ID</td>
<td>Tower Status Radiant</td>
</tr>
<tr>
<td>Cosmetics</td>
<td>Series ID</td>
</tr>
<tr>
<td>Series Type</td>
<td>Replay Salt</td>
</tr>
<tr>
<td>Region</td>
<td>Replay URL</td>
</tr>
<tr>
<td>Dire Team</td>
<td>Radiant Team</td>
</tr>
</tbody>
</table>

Figure 4.2: Common features found in the DOTA 2 game JSON file

We can find the player-specific features in two places in the API. We can leverage these for the graph and vector construction process. During the entire game, the two teams of players have several fights and the JSON file stores each interaction of the fights. The team fight data have detailed data for each interaction of players in each
team fight, including when the fights start and end. Figure 4.3 showcases the start and end times in team fights, and figure 4.4 shows the data for a single player in a team fight. The complete list of attributes is too large to be included here.

Apart from team fight-related data, the API response has a section where the data is separated player-wise. This player-specific data is a collection of all the data stored during the game for each player. I show them in figure 4.5.
Figure 4.4: Available player data in a team fight
4.2.3 Selected Attributes

I have chosen to use only a subset of available features due to the API’s format of providing data. I have done so partly because some data was not suitable for my approach and partly because of other reasons explained in this section. I have left out the data common to the players since there is no personal information in that. Player-specific data is present inside two areas. One is in the section which contains all the team fights, and the other is in the player summary section.

Out of all the attributes in the team fights, ability uses and item uses are in a
format suitable for making graphs. Figure 4.6 illustrates that. For each ability use, we can make an edge from the player node to the ability node. We know the timestamps for each fight, which helps us figure out the timestamp for the ability use. The same applies to item uses.

Figure 4.6: Ability and Item use data of a player in a teamfight

The same logic applies to data from the player summary. Purchases, kills, and rune uses are in a perfectly suitable format for making a graph. Figure 4.7 shows the structure for each purchase log in the data. Kill logs and rune logs have the same structure. It contains the name of the purchase and the timestamp when it happened. This temporal information in both team fights and player summary helps build a vector timeline of interactions that I will explain in-depth in the next chapter.
4.2.4 Selection of Games

For implementing my approach, it’s essential to have access to timestamped interaction data. After looking at a significant number of games from the OpenDOTA API, I discovered that many of the casual games do not have the relevant data I need. That is why I have chosen to work with only popular professional matches that are complete and usable for my pipeline. Out of the 700k games I downloaded, only 3.3k had complete data for building a graph.
4.2.5 Other Options

There were other datasets or APIs that I looked into, but I couldn’t find the same level of detail and timestamped information that OpenDOTA provides. Other games like League of Legends and StarCraft have APIs but do not provide the data I need to implement this approach. PUBG is a strong competitor for the dataset I’m using, but I chose OpenDOTA simply because it is more popular than other games.

4.3 Building the Game Graph

Once we have access to the interaction data, we need to shape and mould the data into a form that the neural network can use to make vectors. This data shaping should be general enough to apply to a wide array of environments. First, I explain how we might naively do this, and then I present the approach that I chose to follow.

4.3.1 Modeling Interactions as Text: Why it Fails

word2vec popularized the idea of getting general representations in a vector form. word2vec is popular and effective, so it might seem logical to use it to get behaviour vectors. word2vec can not directly accomplish the task; it only takes sentence training input. It’s a matter of adapting the task to something that suits word2vec.

word2vec takes a bulk of text material, performing its process and yielding meaningful vectors for each word in that text material. When a new piece of text comes in, the vectors for each word are combined to form a final vector representing the entire document. Figure 4.8 illustrates this process.
The way word2vec can be used to make player vectors is that the training data from step one can be all the details about players written in an organized manner. word2vec would learn to associate each word with the word that comes after that and hopefully make vectors that hold that information. For example, if word2vec sees
[player-one] and [item-seven] words appear together, it might learn to associate both those words together and bring them closer in the vector space.

When new data from a game comes in, we need to get the vectors for each word and combine them to get a vector for each player. If even more interaction data comes in, we have to connect the vectors for those words and get the resulting vector.

The challenge is to model the player information in a suitable way for word2vec to perform this association task well enough. The step for vector combining vectors can use addition, multiplication, or any other operation to combine them. Trial and error are required to find the perfect method of combination. It follows that we could lose much of the original information after combining several vectors for a new player in a game. The presented technique is straightforward but has a major conceptual flaw that I will highlight in the next paragraph.

**Flaws in This Approach**

The words we give as input serve as information about the player's interactions. But there’s no way to explicitly tell the model that we care about these interactions and that they should affect the vector in a meaningful way. So the model is left to figure out the connection between players and interactions implicitly. The methodology might be simple, but we’re implicitly asking the ML model to make relevant connections about the interactions. That is a huge conceptual flaw in this method, so I have not used it for my approach. My approach addresses this concern by modelling interactions as a graph, where interactions can be presented explicitly.
4.3.2 Modeling Interactions as a Graph

Instead of relying on the model to look at words and make implicit connections, I chose to make a graph out of the interactions. Doing so provides two key advantages. One of them is that the relationships between two entities are explicit. Another advantage is the availability of methods to embed graph nodes as vectors.

The interactions would look like what figure 4.9 shows. Players and the items they interact with are nodes in a graph, and any interaction between them counts as an edge. I highlighted Entity A to show the correspondence between interactions and edges.

The items all have uniquely identifiable names which do not vary across games. This invariance is essential because it ensures that the names will be the same in the training and testing phases. The same is not true for players since the players we see in training may not be the ones we evaluate in a live environment. For that reason, I used each player’s hero ID as their identifier instead of their account number. The player can choose to play the game using a particular character, each with different abilities. The hero ID refers to the ID of those characters, which are limited in quantity and are consistent across games. When the system receives data from a new unseen account ID, the system can rely on the known hero ID to work. Also, too many unique account IDs would pollute the graph and sabotage the vector-building process. That is why limiting our scope to just hero IDs would help the system become more abstract and perform better.
Figure 4.9: Making a graph from interaction data
Figure 4.10: Pipeline for my approach, which resembles the one for word2vec
In the first step, we take all training data, construct a large combined graph, and let the node embedding methods create vectors for each node. If two entities interact multiple times over all games, I count all of those interactions as a single edge between them.

4.4 Using Deep Learning to Generate Vectors for Graph Nodes

After building the graph, we can get vectors for each node using existing deep learning graph embedding techniques. I briefly describe the working of the three graph embedding techniques that I used for my experiments.

4.4.1 node2vec

In the previous section, I have described how word2vec enables us to generate word vectors once we supply it with text to perform its training. Text is fundamentally a collection of sentences. The words can be of any language and may also use unique identifiers since the only information relevant for word2vec is which words appear together in a sentence. node2vec leverages this abstract requirement of word2vec and generates sentences out of a graph using random walks on the graph. That means nodes that are connected appear adjacent to one another in the random walk sentences. word2vec can use the ”node” sentences and provide meaningful vector representations for them. Figure 4.11 shows the entire process.
The authors have open-sourced the code base, the method fares well across graph embedding benchmarks, and the approach is intuitive. These are the reasons why I have used this method in my experiments.
Figure 4.12: Three graph nodes mapped to a vector space

1. Closer since they are connected in the graph
2. Farther since they are not connected in the graph
The premise of this method is that two nodes connected in the graph should be closer in the vector space and vice versa. The model optimizes for that assumption using deep learning. The reasons for including this method in my experiments are the same as node2vec.

Figure 4.12 presents an example graph and its vector space representation. Nodes 8 and 14 have an edge in the graph, so they are closer to each other in the vector space. The opposite is true for nodes 8 and 18. They are far away from each other in the graph, and the vector space diagram reflects that.

Figure 4.13 shows how LINE generates its vectors. The encoder is a fully connected neural network. Node 1 and Node 2 are their respective slices from the adjacency matrix of the graph. The encoder takes them as input and generates the vector representation for both. It then measures the Euclidean/Cosine distance between Vector 1 and Vector 2 and penalizes the encoder depending on these factors. If Node 1 and 2 have a mutual edge in the graph, LINE optimizes the encoder to minimize the distance between Vector 1 and 2. The opposite is true when Node 1 and 2 do not have an edge between them.
4.4.3 Temporal Graph Networks

TGN works by building node vectors one interaction at a time [25]. It supports building vectors dynamically as the interactions happen. The process happens in four stages, as shown in figure 4.14. When a new interaction comes in, it’s converted into a message vector by the message module. The memory module takes the message vector and updates the memory vector for the two entities involved in the interaction. Whenever a node vector is required, the embedding module computes and provides it. It takes the memory vector for the node needed and from the nodes that have
interacted with that one recently and combines them to form a final embedding vector.

Figure 4.14: Neural network architecture for TGN

TGN gets its gradients from trying to predict if two vectors interact in the future. A suitable vector means a simple classifier should have an easier time predicting and achieving a higher score. This description is only a brief overview of how TGN works; I have intentionally omitted minute details in the interest of brevity.
Imagine a scenario where we have vectors for entities A, B, C, D after training on previous interactions. Assume that there is a new interaction between A and B. This interaction means the vectors for A and B need to be updated to account for that. We can have a function that can update the vectors slightly for both. In my case, I decided to add a fraction of the B vector to A. All my experiments follow the same way of linking vectors after two entities interact. The process takes minimal resources from a system and is scalable for a large number of entities. Figure 4.15 shows how I have designed the vector update process.
4.6 Summary

This chapter discussed the available data from the Open DOTA API and the reasons for choosing some attributes over others. It outlines and describes how the data flows through each step in the proposed approach, from the raw API response to a complete vector form. The chapter also covered a brief overview of each graph embedding method used in the proposed approach and an example of modelling vectors in a live interaction.
Chapter 5: Evaluation

Having a system to test the quality of vectors is essential to the applications of the vectors. However, quality is a subjective criterion by definition and hard to define for a vector, which is only a series of numbers. I have used a proxy to determine the quality of the vector since there is no direct way of accomplishing that. I have used the vectors as input to a classifier and used their performance as a proxy to measuring their quality.

5.1 Testing Framework

I have outlined how I obtained vectors in the previous chapters. Testing the quality of those vectors, however, requires additional work. The output of the approach is one vector for each entity in the game. Each vector is just an array of numbers. They are supposed to represent the behaviour of the respective entity, but it’s challenging to figure out how well they do so. It is helpful to predict the player’s likelihood of interacting with an element in the game based on their playing style. That information allows the personalization of the game and enhancing the player’s experience. Therefore, I have used the following approach to gauge vector quality.

Imagine we pick two entities in the game and try to predict if they will interact before the game ends. A machine learning model should be able to make a reasonable
prediction if they have access to the behaviour data for the two entities. The primary purpose of the vectors is to encapsulate and represent an entity’s past behaviours. It’s supposed to be a replacement for all of the behaviour data. So the conclusion follows that an ML model should be able to use only the vectors for the two entities and make the prediction we decided on. Therefore, I will use the performance of such an ML model to indicate the quality of the vectors obtained from my approach. More specifically, for each configuration of my approach, I trained an ML model to predict if two entities interact in the future, given their present vector representations. Higher prediction accuracies indicate better vector quality.
5.1.1 The Big Picture

Figure 5.1: Test Pipeline

Figure 5.1 shows the high-level architecture of my testing framework, which I explain in more detail in the following sections. Starting with all the games, I use 90 percent of them to generate vectors using my approach. I use a combination of the remaining games and the vectors to create vector timelines and samples from them. The binary
classifier uses these samples with ten-fold cross-validation to score the quality of the vectors.

We need enough data for testing to ensure that there’s minimal variance in the testing results. Along with that, it’s important to reserve more data for training to optimize model performance. There are dataset splitting options like 70:30, 80:20 or 90:10. 10 percent of my training data amounts to 300 games. I gathered 100 testing samples from each game, which meant I had 30k testing samples for each experiment. Having 30k samples decreases the likelihood of under-fitting. That is the reason I went for a 90:10 split. It maximizes training data along with reserving a sufficient data for testing.

5.1.2 Partitioning Games

I downloaded 3300 games for performing the experiments. These games are from top players in DOTA 2 and their data was complete and had the parts I needed to utilize my approach. I decided to use a fraction of the total amount of games to construct vectors for game entities and the remaining fraction to test the vectors’ quality. The game data came in the form of a JSON file from the OpenDOTA API.

Games Used For Vector Generation

I conducted experiments with a range of parameters. For each experiment, I kept aside 90 percent of the games for this task, randomly chosen by the code. That helps avoid lucky coincidences and overfitting to a specific set of games.
Games Used For Classifier Tests

I made the vectors using one set of interactions, and it is essential to test them on a different set of unknown interactions to get an unbiased result. I have used this partition for that purpose. The entities are the same ones that appeared in training, but the set of interactions is new.

5.1.3 Building Vectors

This step operates on the data kept aside for vector generation in the previous step. I converted each game into a graph, keeping the entities as nodes and edges between them denoting an interaction. I collected all nodes and edges for the games into one big graph. I used either one of node2vec, LINE, TGN to take the large graph and generate vectors for each node in the graph. Chapter 4 describes this process.

5.1.4 Building Vector Timelines

Before explaining vector timelines, it’s necessary to introduce the concept of timeslots. Instead of using the exact timestamp, I have used buckets of timeslots based on when the team fights happen. That limits the number of possible values the timeslot variable can have. I show an example below in figure 5.2.
The vector timeline is a snapshot of all the vectors for each timeslot, as shown in figure 5.3. This data structure is essential to the sampling process that follows. As shown in figure 5.1, this step needs the following data to operate: The data kept aside for the classifier, and the vectors generated by either node2vec, LINE or TGN.

As entities interact, it’s essential to change the vectors to reflect those interactions. Chapter 4 shows the vector update process in more detail. The vectors have different values in each timeslot due to this process.
5.1.5 Sampling The Vector Timeline

This process aims to gather samples for training and test the binary classifier in the following procedure. I describe the sample collection process for each sample using the example Figure 5.4. The first step is to pick two random entities from the game and find the last timeslot in which they interact; T2 in our case. Once we know this, we have two ways to sample data from the game. If we choose the vectors for these two entities at a timeslot earlier than T2 (e.g. T1), we know they’l interact in the
future at T2. That means we can save their vectors at timeslot T1 and answer 1 as a classification training sample. Answer 1 means the two entities interact in the future.

Figure 5.4: Picking a sample where the two entities react in the future timeslots

The other solution is to sample the vectors for these two entities at a timeslot after T2 (e.g. T4). That means they don’t interact in any later timeslots. We can save their vectors at timeslot T4 and set the answer as 0 since they don’t interact after T4. Figure 5.5 illustrates the idea.
My sample generator gathers a fixed number of samples (100) from each game like this. I have kept a 50-50 ratio of samples with 0 and 1 as their answers, respectively. That is because using accuracy as a metric requires the classes in a dataset to be balanced.
5.1.6 Evaluation Metrics

Figure 5.6 shows how I have trained the classifier. Given two vectors for some entities, all it has to do is predict if they will interact in any future timeslot. I used part of the samples to train the classifier and another part for testing.

The classifier I used is Logistic Regression, with 10-fold cross-validation on the given samples. A better quality vector can better inform the classifier and thus achieve better accuracy. That is why I have chosen the classifier’s accuracy on this task as the metric to judge the vectors’ quality.

My goal for these experiments was to make the classifier a good indicator for the vectors’ quality instead of optimizing the classifier’s accuracy. That is why I have used such a simple classifier for the tests and not complex multilayer neural networks.
5.2 Experiments

My aim for these experiments is to explore the effect of different parameters on vector quality. I developed a framework to perform these tests efficiently and in an automated fashion. The code is available on GitHub with an MIT license. \(^1\) The four parameters that I studied were step size, dimensions, number of games and the types of included interactions. For each parameter, I selected a range of values to test. For each parameter value, I held the others constant (see values below), built a graph, and produced vectors using each embedding technique. I then tested each set of vectors in six experiments: five that tried to predict only a single type of interaction (purchases, kills, rune uses, ability uses, item uses) and one that predicts all five types. In the context of this work, an experiment is the process of training and testing the classifier using vectors produced by a particular technique using a particular set of parameters.

When I held parameter values constant, I used the following values:

1. Step size - 0.05
2. Dimensions - 512
3. Number of games - 3300
4. Categories of interactions included - All

I used 15 ways of varying these four parameters, as explained in the following sections.

\(^1\)https://github.com/jinayshah7/behaviours-as-vectors-thesis-code
5.2.1 Number of Dimensions in the Vector

Higher dimensions of vectors can embed more information. Too many dimensions can risk overfitting, so I decided to try four different dimensions for vector generation and see which dimension provides the best performance.

I tried testing each type of interaction separately along with doing them together. The first five rows of the following tables indicate results for every kind of interaction. The last row shows the results with all interaction types combined. Each result is the prediction accuracy of the classifier that I trained.

<table>
<thead>
<tr>
<th>node2vec</th>
<th>16</th>
<th>128</th>
<th>256</th>
<th>512</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purchase</td>
<td>0.5239</td>
<td>0.6514</td>
<td>0.6513</td>
<td>0.6568</td>
</tr>
<tr>
<td>Kills</td>
<td>0.5261</td>
<td>0.6000</td>
<td>0.6288</td>
<td>0.6342</td>
</tr>
<tr>
<td>Runes</td>
<td>0.5915</td>
<td>0.6086</td>
<td>0.6201</td>
<td>0.6295</td>
</tr>
<tr>
<td>Ability Uses</td>
<td>0.5308</td>
<td>0.6180</td>
<td>0.6340</td>
<td>0.6380</td>
</tr>
<tr>
<td>Item Uses</td>
<td>0.5641</td>
<td>0.5828</td>
<td>0.6622</td>
<td>0.6866</td>
</tr>
<tr>
<td>All</td>
<td>0.5195</td>
<td>0.5822</td>
<td>0.6034</td>
<td>0.6170</td>
</tr>
</tbody>
</table>

Table 5.1: Varying vector dimensions in node2vec

<table>
<thead>
<tr>
<th>LINE</th>
<th>16</th>
<th>128</th>
<th>256</th>
<th>512</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purchase</td>
<td>0.5250</td>
<td>0.6540</td>
<td>0.6520</td>
<td>0.6516</td>
</tr>
<tr>
<td>Kills</td>
<td>0.5276</td>
<td>0.5937</td>
<td>0.6024</td>
<td>0.6105</td>
</tr>
<tr>
<td>Runes</td>
<td>0.5937</td>
<td>0.6097</td>
<td>0.6136</td>
<td>0.6185</td>
</tr>
<tr>
<td>Ability Uses</td>
<td>0.5350</td>
<td>0.5924</td>
<td>0.5965</td>
<td>0.5995</td>
</tr>
<tr>
<td>Item Uses</td>
<td>0.5596</td>
<td>0.5943</td>
<td>0.6167</td>
<td>0.6562</td>
</tr>
<tr>
<td>All</td>
<td>0.5203</td>
<td>0.5798</td>
<td>0.5859</td>
<td>0.5907</td>
</tr>
</tbody>
</table>

Table 5.2: Varying vector dimensions in LINE
The result indicates an upward trend in accuracy scores with an increase in the
dimensions; however, the differences in scores are minor. node2vec performs better
than the other two methods in more than half of the total experiments, followed
closely by LINE. TGN underperformed compared to the other methods in almost
all the experiments. All the methods also perform better on predicting purchase
interactions than the “All” score, which includes all types of interactions.

5.2.2 Ways to Build the Game Graph

I included the following interaction types in the graph building process: Ability uses,
Item uses, Purchase Log, Runes Log and Kills Log.

To see which interaction contributes the most to the vector’s quality, I have tried
removing each interaction type and performing the tests. If the test performance drops
significantly after removing one kind of interaction, that category of interaction is an
essential contributor to the vector’s quality. If the test performance improves after
removing a set of interactions, those interactions were not helpful and polluted the
dataset.

Some cells are empty in these tables. For example, if we haven’t used “purchase”
interactions to construct the graph, we cannot test that category of interactions afterwards. The columns in the table indicate the results when we exclude that particular category of interactions from the whole vector pipeline for and testing vectors.

<table>
<thead>
<tr>
<th>node2vec</th>
<th>Purchases</th>
<th>Kills</th>
<th>Runes</th>
<th>Ability Uses</th>
<th>Item Uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purchase</td>
<td>-</td>
<td>0.6551</td>
<td>0.6646</td>
<td>0.6055</td>
<td>0.6337</td>
</tr>
<tr>
<td>Kills</td>
<td>0.7103</td>
<td>-</td>
<td>0.6906</td>
<td>0.5837</td>
<td>0.6605</td>
</tr>
<tr>
<td>Runes</td>
<td>0.7359</td>
<td>0.6846</td>
<td>-</td>
<td>0.5807</td>
<td>0.6198</td>
</tr>
<tr>
<td>Ability Uses</td>
<td>0.7073</td>
<td>0.6636</td>
<td>0.6614</td>
<td>-</td>
<td>0.6164</td>
</tr>
<tr>
<td>Item Uses</td>
<td>0.7188</td>
<td>0.6768</td>
<td>0.6693</td>
<td>0.6189</td>
<td>-</td>
</tr>
<tr>
<td>All</td>
<td>0.7078</td>
<td>0.6449</td>
<td>0.6448</td>
<td>0.5708</td>
<td>0.5880</td>
</tr>
</tbody>
</table>

Table 5.4: Excluding each category type in node2vec

<table>
<thead>
<tr>
<th>LINE</th>
<th>Purchases</th>
<th>Kills</th>
<th>Runes</th>
<th>Ability Uses</th>
<th>Item Uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purchase</td>
<td>-</td>
<td>0.6132</td>
<td>0.6399</td>
<td>0.5923</td>
<td>0.6299</td>
</tr>
<tr>
<td>Kills</td>
<td>0.7142</td>
<td>-</td>
<td>0.6688</td>
<td>0.5290</td>
<td>0.6803</td>
</tr>
<tr>
<td>Runes</td>
<td>0.7340</td>
<td>0.6531</td>
<td>-</td>
<td>0.5591</td>
<td>0.6199</td>
</tr>
<tr>
<td>Ability Uses</td>
<td>0.7093</td>
<td>0.6423</td>
<td>0.6384</td>
<td>-</td>
<td>0.6144</td>
</tr>
<tr>
<td>Item Uses</td>
<td>0.7144</td>
<td>0.6478</td>
<td>0.6536</td>
<td>0.5461</td>
<td>-</td>
</tr>
<tr>
<td>All</td>
<td>0.7057</td>
<td>0.5962</td>
<td>0.6134</td>
<td>0.5413</td>
<td>0.6044</td>
</tr>
</tbody>
</table>

Table 5.5: Excluding each category type in LINE

<table>
<thead>
<tr>
<th>TGN</th>
<th>Purchases</th>
<th>Kills</th>
<th>Runes</th>
<th>Ability Uses</th>
<th>Item Uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purchase</td>
<td>-</td>
<td>0.5740</td>
<td>0.5551</td>
<td>0.5885</td>
<td>0.5843</td>
</tr>
<tr>
<td>Kills</td>
<td>0.6997</td>
<td>-</td>
<td>0.6547</td>
<td>0.5254</td>
<td>0.5338</td>
</tr>
<tr>
<td>Runes</td>
<td>0.7202</td>
<td>0.5525</td>
<td>-</td>
<td>0.5535</td>
<td>0.5235</td>
</tr>
<tr>
<td>Ability Uses</td>
<td>0.6699</td>
<td>0.5674</td>
<td>0.5899</td>
<td>-</td>
<td>0.5646</td>
</tr>
<tr>
<td>Item Uses</td>
<td>0.7017</td>
<td>0.5911</td>
<td>0.6103</td>
<td>0.4944</td>
<td>-</td>
</tr>
<tr>
<td>All</td>
<td>0.6789</td>
<td>0.5414</td>
<td>0.6036</td>
<td>0.5407</td>
<td>0.5195</td>
</tr>
</tbody>
</table>

Table 5.6: Excluding each category type in TGN

The data shows that removing purchase interactions from the game graph boosts the test scores for all methods. Most purchase interactions happen at the beginning of the game and outnumber the other interactions by a considerable margin. That
means the presence of purchase interactions didn’t provide the model with any helpful information and instead served to pollute the other data. Removing them enables the model to learn the different types of interactions better and hence score higher. The experiments showed how all methods performed better without the purchase interactions. That means purchase interactions caused the model to learn them mostly and suffered when predicting other categories. Removing them resulted in a noticeable performance boost for the other categories.

5.2.3 Step Size for Linking Two Vectors

These experiments are relevant for building a reasonable vector timeline. When two entities interact, the code slightly modifies their vectors to reflect the interaction. I describe this process in figure 5.7. Each column in the table indicates results for different step sizes.

Figure 5.7: Updating two vectors in light of a new interaction
The data indicates there’s no noticeable difference in the performance of the methods with the different step sizes that I tested.

### 5.2.4 Number of Games Used to Make Vectors

The vectors capture behaviour data and represent it for the entity they represent. They infer this data from the game graph. Although I conducted all the previous
experiments using 3300 games, I performed the following experiments to see how few games are enough to make suitable vectors.

For example, when we use 100 games, we use 90 games to construct the vectors, and interactions from the remaining ten games to test the vector quality.

<table>
<thead>
<tr>
<th>node2vec</th>
<th>100</th>
<th>500</th>
<th>3300</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purchase</td>
<td>0.6953</td>
<td>0.6887</td>
<td>0.7101</td>
</tr>
<tr>
<td>Kills</td>
<td>0.6258</td>
<td>0.6686</td>
<td>0.6737</td>
</tr>
<tr>
<td>Runes</td>
<td>0.6060</td>
<td>0.6786</td>
<td>0.6717</td>
</tr>
<tr>
<td>Ability Uses</td>
<td>0.6094</td>
<td>0.6267</td>
<td>0.6446</td>
</tr>
<tr>
<td>Item Uses</td>
<td>0.6060</td>
<td>0.6563</td>
<td>0.6800</td>
</tr>
<tr>
<td>All</td>
<td>0.6208</td>
<td>0.6608</td>
<td>0.6590</td>
</tr>
</tbody>
</table>

Table 5.10: Varying number of games in node2vec

<table>
<thead>
<tr>
<th>LINE</th>
<th>100</th>
<th>500</th>
<th>3300</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purchase</td>
<td>0.7223</td>
<td>0.6797</td>
<td>0.7244</td>
</tr>
<tr>
<td>Kills</td>
<td>0.6247</td>
<td>0.6288</td>
<td>0.6447</td>
</tr>
<tr>
<td>Runes</td>
<td>0.5830</td>
<td>0.6160</td>
<td>0.6355</td>
</tr>
<tr>
<td>Ability Uses</td>
<td>0.5972</td>
<td>0.6267</td>
<td>0.6124</td>
</tr>
<tr>
<td>Item Uses</td>
<td>0.5710</td>
<td>0.6568</td>
<td>0.6378</td>
</tr>
<tr>
<td>All</td>
<td>0.6206</td>
<td>0.6232</td>
<td>0.6443</td>
</tr>
</tbody>
</table>

Table 5.11: Varying number of games in LINE

<table>
<thead>
<tr>
<th>TGN</th>
<th>100</th>
<th>500</th>
<th>3300</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purchase</td>
<td>0.5521</td>
<td>0.5399</td>
<td>0.5574</td>
</tr>
<tr>
<td>Kills</td>
<td>0.5523</td>
<td>0.6232</td>
<td>0.5724</td>
</tr>
<tr>
<td>Runes</td>
<td>0.5336</td>
<td>0.6441</td>
<td>0.6319</td>
</tr>
<tr>
<td>Ability Uses</td>
<td>0.5852</td>
<td>0.5990</td>
<td>0.5327</td>
</tr>
<tr>
<td>Item Uses</td>
<td>0.5320</td>
<td>0.5866</td>
<td>0.5656</td>
</tr>
<tr>
<td>All</td>
<td>0.5535</td>
<td>0.5893</td>
<td>0.5301</td>
</tr>
</tbody>
</table>

Table 5.12: Varying number of games in TGN

The data shows that moving from 100 to 500 games improves the results in most cases, but the scores stagnate when moving to 3300 games. The difference between
500 and 3300 games is not coherent. In Chapter 6, I explain how there is a possibility that the models are currently under-fitted, and scores may get a boost if we were to increase the number of games used exponentially.

5.3 Summary

Considering the accuracy score for all interaction types as a general metric, node2vec outperforms the other methods in 13 of the 15 ways of varying the parameters. Tables 5.13 to 5.16 summarize these results.

<table>
<thead>
<tr>
<th>Number of Games</th>
<th>100</th>
<th>500</th>
<th>3300</th>
</tr>
</thead>
<tbody>
<tr>
<td>node2vec</td>
<td>0.6208</td>
<td>0.6608</td>
<td>0.6590</td>
</tr>
<tr>
<td>LINE</td>
<td>0.6206</td>
<td>0.6232</td>
<td>0.6443</td>
</tr>
<tr>
<td>TGN</td>
<td>0.5535</td>
<td>0.5893</td>
<td>0.5301</td>
</tr>
</tbody>
</table>

Table 5.13: Performance summary: Predicting all interaction types for different number of games.

<table>
<thead>
<tr>
<th>Step Size</th>
<th>0.0025</th>
<th>0.05</th>
<th>0.10</th>
</tr>
</thead>
<tbody>
<tr>
<td>node2vec</td>
<td>0.6497</td>
<td>0.6547</td>
<td>0.6467</td>
</tr>
<tr>
<td>LINE</td>
<td>0.6325</td>
<td>0.6384</td>
<td>0.6419</td>
</tr>
<tr>
<td>TGN</td>
<td>0.5218</td>
<td>0.5208</td>
<td>0.5204</td>
</tr>
</tbody>
</table>

Table 5.14: Performance summary: Predicting all interaction types for different step sizes.

<table>
<thead>
<tr>
<th>Dimensions</th>
<th>16</th>
<th>128</th>
<th>256</th>
<th>512</th>
</tr>
</thead>
<tbody>
<tr>
<td>node2vec</td>
<td>0.5195</td>
<td>0.5822</td>
<td>0.6034</td>
<td>0.6170</td>
</tr>
<tr>
<td>LINE</td>
<td>0.5203</td>
<td>0.5798</td>
<td>0.5859</td>
<td>0.5907</td>
</tr>
<tr>
<td>TGN</td>
<td>0.5157</td>
<td>0.5180</td>
<td>0.5201</td>
<td>0.5223</td>
</tr>
</tbody>
</table>

Table 5.15: Performance summary: Predicting all interaction types for different dimensions.
### Table 5.16: Performance summary: Predicting all interaction types when excluding different interaction types.

<table>
<thead>
<tr>
<th>Dimensions</th>
<th>Purchases</th>
<th>Kills</th>
<th>Runs</th>
<th>Ability Uses</th>
<th>Item Uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>node2vec</td>
<td>0.7078</td>
<td>0.6449</td>
<td>0.6448</td>
<td>0.5708</td>
<td>0.5880</td>
</tr>
<tr>
<td>LINE</td>
<td>0.7057</td>
<td>0.5962</td>
<td>0.6134</td>
<td>0.5413</td>
<td>0.6044</td>
</tr>
<tr>
<td>TGN</td>
<td>0.6789</td>
<td>0.5414</td>
<td>0.6036</td>
<td>0.5407</td>
<td>0.5195</td>
</tr>
</tbody>
</table>

I used different random seeds for each experiment category like changing dimensions, number of games, etc. This is introduced some variation in the test results. For example, node2vec with 0.05 step size in table 5.14 uses the same parameters as node2vec with 512 dimensions in table 5.15, but they still have a different accuracy score. To test the extent of this source of variance, I performed a 10-fold cross validation across the 3300 games I had. That meant the script used 10 different parts of the dataset to test the performance of all methods. Here is the mean of accuracy for all three methods:

- TGN - 0.5259
- LINE - 0.5869
- node2vec - 0.6218

I used the following parameters for the 10-fold testing:

- Step size - 0.05
- Dimensions - 512
- Number of games - 3300
• Categories of interactions included - All

The variance for all these results is less than 0.0001 in all cases. Running a baseline dummy classifier which only predicts the most frequent class produced an average result of 0.53. That means that TGN has a comparable performance to the baseline classifier, while the other two methods showcased superior performance.
Chapter 6: Discussion

6.1 Revisiting The Research Problem

This section details how the current proposed approach fulfills all the requirements for the research problem.

6.1.1 Dense Vector Representation

As showcased in the earlier sections, each player has many unique attributes that change every moment in the game. If we track all such changes across the game, the total number of features might end up being 5000+ for a single player. Building a large vector to hold this information can be cumbersome and inefficient. My approach allows compressing that information down to 128 or 512 dimensions depending on the level of performance needed. Using the proposed method, it’s easy to go from a set of interactions to having vector representations of each entity in that interaction. The process also benefits from directly comparing vectors since they’re just a list of numbers and embeds all past behaviour. The same sort of comparison is tough with a complete set of behaviour data logs.

6.1.2 Efficiency and Speed

The method works in two steps as follows:
Computing The Vectors

This component needs access to a significant sample size of interactions from the past. We should know about all entities in advance and have training data for them. Machine Learning models are delicate and require significant training data to account for all the possible scenarios in the tests. The system should, in turn, have sufficient parameters to account for the amount and complexity of the data. I used 3300 games for my experiments, making a graph with 1000 nodes and 120,000 edges on average.

Once that data is available, we can train the graph embedding network to extract vectors out of the data. There is no need to account for time, amount of resources required or performance in this step since it happens outside of the vector usage. In my case, it took node2vec 1 minute, TGN 10 minutes and LINE 60 minutes to compute vectors for each experiment. It limited my ability to conduct more experiments but in an industrial application, the time taken and resources used in this step are less critical.

That also allows well-funded companies like Google or OpenAI to train a model with enormous data and prepare vectors, instead of individual researchers. Google openly provided Word2vec vectors for the Google News dataset containing 100B words. Researchers could then directly use those vectors and fine-tune them for specific applications with minimal compute power since the vectors are already a great point to start from. This separation of the heavy lifting step from the actual usage is a massive advantage as it allows individual researchers like me to work and
explore applications more accessibly—my approach has a similar benefit.

Because of the training data size, it might take significant time for the vector embedding methods to take the interaction logs, convert them into a graph, and extract vectors out of the graph. But the following step requires minimal computing power. Extracting vectors is a one-time process.

Using the Vectors Live in an Environment

Assuming we have vectors for each entity in the environment, it’s easy to get an updated representation of the vector after seeing a new interaction, as shown in section 4.5.

The vector incrementally changes as new interactions happen. That is not true for other systems which only have a single vector for an entire set of interactions and have no way of adapting them if new activity happens [15].

If a new player joins the game, we can take their hero ID and work with it. The proposed approach assumes that all the nodes in testing also appear in the training data. That means the system fails when an entirely new type of node appears in testing. For example, if a new hero gets added to DOTA 2, the best we can do is use a zero vector to represent them and let their interactions define their vector as it gets updated. Or we can retrain the graph embedding methods to include the new hero.
6.1.3 Feature Selection

My approach requires some human input to select which features have timestamps and only pick those features, but there’s nothing needed beyond that. In conventional methods of behaviour modelling, this doesn’t hold [8].

Selecting features using my approach is programmable in a script, and it’s easy to perform a brute force search in an automated fashion without the need of a subject matter expert to provide input. That saves time and effort when using the approach with a new environment, which I address in the next point.

6.1.4 Ease of Use in Other Environments

My method’s only assumption about how the environment works is that one can obtain a list of individual interactions between entities. The system then creates a graph out of those interactions for the vector building process. As a result, my method can easily be ported over to other environments without significant changes.

Figure 6.1 shows an example of the system as it would work with data from the OpenDOTA API and a social media application. The system doesn’t need any code refactoring to adapt to the data coming from the social media application. The users on the social media application might have different patterns, which means the system might need different hyperparameters to run optimally. But that is an obvious and unavoidable expectation. The amount of work involved in porting the system to another environment is minimal compared to other traditional approaches [8] [15].

In contrast, conventional systems need human experts to design a completely
new algorithm for selecting and processing the essential features for an entity in a domain. The effort invested in handpicking features in DOTA would be useless when moving to a new system. Understandably, an expert's handpicked features might yield marginally better performance out of a classifier, but the loss of versatility is too significant to ignore.
Figure 6.1: The system adapting to two different systems with a similar data format
6.2 Reflections on the Proposed Approach

6.2.1 Advantages

As I showcased in the related works section, this approach is unique and untapped by other researchers in the community. It presents four primary advantages:

**Simple, intuitive and effective**

The process occurs in two steps: one constructing the vectors and one using them. Producing the vectors can happen on a powerful machine, and users can reuse those pre-computed vectors to analyze a live game. The process is similar to how word2vec works, where it’s easy to find pre-computed vectors trained on large textual datasets.

**End To End Process**

It’s an end-to-end process of taking interactions and directly giving vectors with minimal feature selection. The approach needs a particular kind of data to construct the graph, so it was easy to narrow down the options available in the OpenDOTA JSON response for a game. Out of the remaining options, it was simple to try removing one category of data each time and test its performance. This simplified feature selection process makes the approach easy to use.
Less Coupling With The Game

There’s less coupling with how the game works than a system that uses data that is unique to a game. The only information we need access to is interaction data for players and entities in the format shown by figure 6.2. After we make the graph, constructing the vectors is straightforward and doesn’t change for other environments. Therefore, we can use the approach with any environment as long as the data is in the desired format.

![Figure 6.2: The interaction format needed by my approach](image)

Modular Components

I was able to try my approach on three different graph embedding methods due to a modular architecture. Chapter 4 shows the vector combination method to update vectors as interactions happen, where new techniques can be inserted and tried out. That allows the usage of multiple independent modules to suit different environments.
6.2.2 Shortcomings

The graph embedding methods I employed take a single graph state and compute node vectors from that state. If we add a new interaction to the graph, the methods cannot modify the vectors to account for the latest update. To tackle this, I decided to change the vectors in a certain way to reflect a new interaction, as shown in section 5.2.3. It could be helpful to discover and include a better way to combine the vectors, but doing so was out of scope for my project.

There are, however, dynamic graph embedding methods that inherently support changing the vectors in light of new interactions. That would remove the need to research a custom way to do it. Using those methods hasn’t been tried before but can perform better than the current set of techniques I used. TGN belongs to that family of methods, but I have only used it for static graphs to keep its functioning consistent with node2vec and LINE in these experiments.

There is also the problem of encountering new nodes in the graph while working with a new game. Once the vectors are computed and we find a new unseen node on the graph, it is impossible to get a vector for that node. That means we have to leave out the node from further computations, and we lose potentially valuable data from that node. A potential solution is to initialize that node to zero and update its vector using the vectors from its neighbours.
6.3 Challenges Faced During the Research

6.3.1 Access to Computing Resources

My approach consists of two processes. The first process consists of building a large graph and making vectors from that. To compute vectors for a graph of 120k edges, node2vec took 2 minutes, while LINE and TGN took an average of 10 minutes and 60 minutes, respectively.

The second step is to use the vectors in a game by building a vector timeline, which also consumed significant time in the experiments. The classifier needs samples to train itself, and computing vector timelines is an essential part of that step. We need the classifier for testing the quality of a particular way to generate vectors. Making a vector timeline, sampling data from it and training the classifier took roughly 1 hour for each experiment. That meant I was limited in the number of experiments that I could try. I did, however, try 15 parameter variations for each method. That makes a total of 45 variations as follows:

1. Number of dimensions - 16, 128, 256, 512 [4 variations]

2. Number of games - 100, 500, 3300 [3 variations]

3. Removing a category of interactions - Ability uses, item uses, purchase logs, rune logs, kill logs [5 variations]

4. Step size - 0.0025, 0.05, 0.10 [3 variations]
I implemented multiprocessing in my code and used a 16 core Google Cloud Platform (GCP) instance to bring down the sampling and testing time to 2 minutes. But using this option still costs a considerable amount of money to operate the GCP instances.

6.3.2 Access to Usable Source Code

The biggest reason I chose node2vec, TGN, LINE was because their code was open source. There were several other candidate methods, but their code was inaccessible, or there was no way to execute the provided code. For the same reason, I have kept my code open source to make it accessible to the research community. I have also taken steps like using fixed random seeds in the experiments to make them completely reproducible.

6.4 Future Work

6.4.1 Explore More Graph Embedding Techniques

There are better methods for graph embedding than the ones I used, but I couldn’t do a comprehensive coverage in the interest of time. My primary goal with this research was to test how different graph embedding methods perform on behaviour data. There were ways to tune the current classifier or use a different one to optimize its accuracy, but I chose to use a simple Logistic Regression Classifier to limit the experiment permutations.
6.4.2 Test More Hyperparameter Variations

There are several places where we can fine-tune the hyperparameters to produce better vectors. I only tried limited variations to make sure the research finished in time. Fine-tuning the hyperparameters of node2vec, TGN, LINE can boost the vectors’ quality and choose a good way to combine the vectors after seeing interactions.

6.4.3 Test More Games and Environments

I have tested this approach in the DOTA 2 environment, which turned out to have good performance. Future research can focus on applying the method to similar games or completely different environments to see how practical the system is. The approach functions at a high level of abstraction, making it applicable to a variety of environments. It would be worthwhile to see how it performs there.
Chapter 7: Conclusion

Humans are unique, along with their behaviours. Comparing two behaviours is a subjective task and hard to quantify. In this research, I explored an approach to quantify human behaviour into N-dimensional vectors using deep learning. The vectors encapsulate a person’s past behaviour and thus serve as a compressed replacement for a person’s behaviour data.

My research focussed on evaluating the performance of three graph embedding methods on player behaviours data from 3300 DOTA 2 games. I used node2vec, LINE and TGN to translate game interaction logs to vectors. I designed a test framework to quantify the vectors’ quality and performed 270 experiments to explore the effect of different process parameters, including vector dimensions, number of games using for training, and type of interactions used. The results indicate that node2vec outperformed the other methods on 13 of the 15 parameter variations I tested. It also computed the vectors 60 and 10 times faster than the LINE and TGN, respectively, for the same dataset size.

The research explores a generalized approach to representing behaviours that enables its application across different environments with relative ease. It can improve personalization in interactions using deep learning with the use of vector representations for behaviours.
Bibliography


