Tile Embeddings: A General Representation for Procedural Level Generation via Machine Learning

by

Mrunal Sunil Jadhav

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

Abstract

Procedural Level Generation via Machine Learning (PLGML) refers to the application of machine learning techniques to the automated generation of game levels. PLGML researchers have investigated different level generation techniques to generate new game levels matching the style of a training corpus. While the PLGML community has made notable progress in designing impressive level generators, we are still far from achieving the holy grail of generalizability. Generalizability refers to a level generators' ability to generate a previously unseen, new game level based on the training data used to build the model. A primary reason for this is the limited availability of PLGML datasets and inconsistent level representation practices across different games. Traditionally employed PLGML datasets are hand-annotated by domain experts and fan communities. The process of curating clean datasets is time-consuming. Hence, even though many video games exist, select few have received a disproportionate amount of research attention.

Towards this goal of generalizability, we propose a representation learning approach for game level design. We introduce *tile embeddings*, a continuous, unified affordance-rich representation of 2D games. This thesis covers an initial implementation of tile embeddings and their further modification to handle the particular case of skewed tile distribution observed in games like Super Mario Bros.. We then introduce a novel, two-step level generation process that can leverage the flexibility of a discrete representation with the expressivity of continuous tile embeddings. We evaluate our tile embedding representation on its ability to predict affordances for unannotated tiles and to serve as a PLGML representation for annotated games. We perform an ablation study for level generation of Super Mario Bros., and further show the ability to apply our approach to level generation for unannotated games. Our outputs cover generative spaces matching the distribution of the original training data, thus demonstrating the potential of tile embeddings for PLGML applications for any tile-based 2D games. The presented thesis attempts to address the core challenges of PLGML around representation and dataset availability. We believe with more work in this direction, our approach has the power to open new horizons for PLGML research.

Preface

The research presented in this thesis is my original work, done under the supervision of Dr. Matthew Guzdial at the University of Alberta. Dr. Guzdial played an instrumental role in shaping the writing of the presented thesis. Parts of this work have been published with him as the co-author. They are as follows:

- Chapter 4: Jadhav, M., Guzdial, M. (2021, October). Tile embedding: a general representation for level generation. In Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (Vol. 17, No. 1, pp. 34-41).
- Chapter 5: Jadhav, M., Guzdial, M. (2022). Clustering-based Tile Embedding (CTE): A General Representation for Level Design with Skewed Tile Distributions. arXiv preprint arXiv:2210.12789 (Presented at Experimental AI in Games Workshop held at the Eighteenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE-22)).

To my brother, my inspiration and my true north.

In god we trust; all others must bring data.

– W. Edwards, Deming.

Acknowledgements

I would like to express my sincere gratitude to my supervisor Dr. Matthew Guzdial for his constant guidance throughout my degree. He is kind, patient and wears an invisible superhero cape every day. He taught me everything I know about research, from approaching a problem to formulating hypotheses, designing experiments and documenting results. Without his insightful discussions and expertise, this thesis would not have been possible. He inspired me to think outside the box and has always supported me in pursuing diverse projects. His tireless efforts in providing valuable feedback helped me develop my academic writing skills. This was my first research experience, and he has been a great mentor, always motivating me to be a better student and researcher. I am forever grateful to him. I would also like to thank all the members of the GRAIL lab. Our weekly lab meetings provided insightful discussions about current academia, industry, time and workload organization.

I would like to extend my sincere thanks to my committee members Prof. Denilson Barbosa and Prof. Levi Lelis, for taking the time to read my thesis and for generously providing their knowledge and expertise through the final phase of my master's.

I would like to acknowledge CIFAR, Alberta Machine Intelligence Institute (Amii), and the University of Alberta for their generous financial support throughout my graduate studies.

I am deeply indebted to my mom and dad for teaching me the right things and for all their care, sacrifices, and concern. This thesis stands as a testament to their unfailing love. No amount of words is enough to thank my grandparents, who always encouraged my education and raised me with love. An enormous thanks to my brother, Pritish, who holds a big place in my heart.

I wouldn't be the person I am today without the drive, inspiration and support he has given me. A very special thanks to my sister-in-law (read as a sister), Anushree, for her heart-warming kindness. Thank you for always caring about me and my work and being genuinely wonderful. I would also like to thank my niece, Ira. Her innocence makes me feel like a little kid again. Thank you for being my stress-buster, my happiness and my all-time cuddle. I am genuinely grateful to my best friend, Soham, who has been my source of sunny optimism through this journey. Thank you for all the love and strength.

I wanted to recognize my extended family, the Suryawanshi's: Anjali, Arvind, and Aradhya and the Patankar's: Mukta, Sunil, and PG. Their unwavering faith in me has been a true asset. I am also grateful to my amazing family in Canada: Vihaan, Anvay, Ankush, Apurva, Swapnali and Manoj for their love and care.

I am thankful to have friends who have been there for me through all the ups and downs, always cheering me on: Aditya, Ketan, Nidhi, Pratik, and Rohan. Thanks for all of the laughs, advice, and memories throughout time.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Procedural Content Generation via Machine Learning (PCGML) involves training machine learning models on existing game data to generate new content such as levels, characters, stories, and music [59]. A significant amount of PCGML research has been devoted to generating game levels with limited human interaction. This discipline of generating levels by employing Machine Learning algorithms is referred to as Procedural Level Generation via Machine Learning (PLGML). A level is a space the player travels through, interacting with objects like enemies and collectibles. For instance, Figure 1.1 shows a level of Super Mario Bros., a platformer game developed by the Nintendo Entertainment System (NES). During gameplay, the player i.e., Mario, races through the level, collects coins, and defeats enemies to reach the end of level. Unlike image generation using machine learning, PLGML models cannot train only on the pixel representations of game levels. Levels obey structural and functional constraints to ensure playability. As seen in the Figure 1.1, a connected series of solid platforms should be present for Mario to have a path to run on till the end, and collectibles and enemies are placed on the player's potential paths to add challenge to the game. A secondary representation is therefore needed to capture the behaviour of game objects in addition to their pixel representation.

A valuable contribution to the PLGML community is the Video Game Level Corpus (VGLC) [61] which provides annotated training corpora for level generation research. A rich amount of PLGML literature has leveraged this

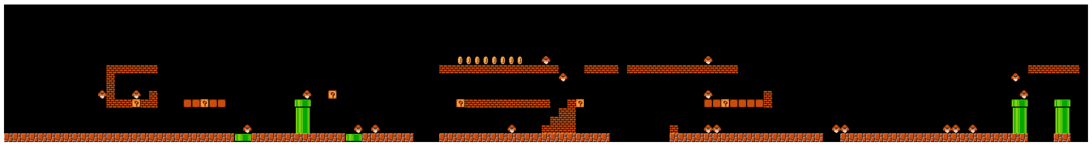


Figure 1.1: Example of a level from the game: Super Mario Bros

representation to generate levels using various machine learning algorithms such as autoencoders [43], [44], GANs [14], [36], and LSTMs [54], [60]. The VGLC maps the pixel representation of game objects to a set of characters called tiles. Each tile is associated with a set of in-game affordances. Affordances convey the conceptual idea of the object and capture the possible interactions of the player with the object [4]. For instance, the VGLC representation associates a Goomba in Super Mario Bros(SMB) with the affordances *Enemy, Damaging, Hazard, Moving* [61].

These representations are game-specific and a substantial amount of manual effort goes into curating them. Consider the problem of training a PLGML model for generating levels for the game Bubble Bobble. Since no annotated representation of its levels exists, we would have to parse the levels ourselves. This typically involves a series of tasks including processing images with OpenCV, human editing, extracting a reduced set of representative tiles, and tagging them with appropriate affordances based on their behaviour [61]. This represents a significant amount of work.

While each tile character is mapped to a set of affordances, the affordances are not directly included in the representation. For instance, the Goomba is represented with the character *'E'* in the VGLC representation, and the affordance-mapping information is present in a separate JSON file. Hence, at their core, the level generation tasks that leverage these representations address problems as a character generation process. Appropriate visual reconstruction also impacts the choice of tiles to include. This enforces the requirement of position-specific tags in the affordance set of the tile/character. For instance, in SMB there are repeated pipe objects of different heights. They are often represented with four different tiles, '[', ']', '<', '>' representing the bottom-left, bottom-right, top-left and top-right of a pipe respectively. In other instances, PLGML practitioners must author secondary processes to visualize levels, such as mapping different characters/tiles to different images depending on their y-position [61]. The current, traditionally drawn VGLC level representation has a number of drawbacks, requiring substantial human effort when collecting data, game-specific representations, and extra processing to visualize generated levels.

This thesis addresses the core challenge of data representation in PLGML. We draw inspiration from word embeddings [34] and introduce *tile embeddings*, which integrate visual and semantic information of tiles. Tile embeddings are a domain-independent, affordance-rich representation of game levels, reducing the reliance on manual translations and domain expertise. Studying the application of the tile embedding representation to level generation demonstrated that they struggled to generate levels for games with imbalanced tile distributions. For example, as seen in Figure 1.1 of Super Mario Bros. (SMB), a majority of the tiles in the level represent the background tiles. Training a level generator on the tile embedding representation of SMB levels resulted in empty levels as illustrated in Figure 1.2. This is a common problem in PLGML when the process of sampling new levels is greedy and biased towards the tile with the highest probability (in the case of SMB: empty sky tiles) [52].

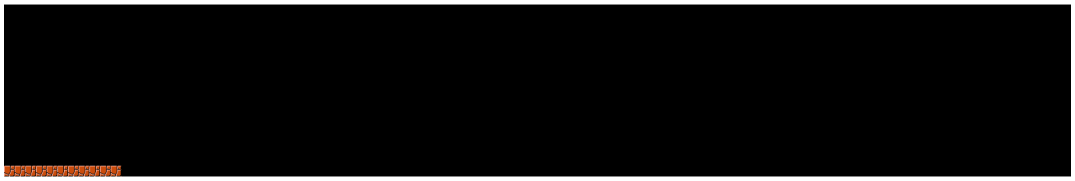


Figure 1.2: Example of generated SMB level with level generator trained on tile embeddings

Traditional PLGML approaches have taken advantage of the discrete nature of the VGLC representation to alleviate the issue of skewed tile distributions. For instance, a level generator can be trained on the VGLC or any discrete representation such that given a sequence of previous tiles in a level, it predicts a distribution over the likelihood of possible next tiles. When generating a new level, tiles at each position can be sampled from this proba-

bility distribution [60]. This sampling process solves the problem of producing empty levels encountered with a greedy tile selection strategy. In order to enable sampling in our level generator, a discrete representation is learned by clustering learned tile embeddings. Thus the presented work leverages the benefits of learning simultaneous discrete and continuous representations to improve level generation for games with skewed tile distributions. This allows us to approximate the benefits of a discrete representation like the VGLC without the cost of hand-processing training data.
Our presented work seeks to answer following research questions:

- Is it possible to automate the extraction of level design data and build large corpora for PLGML research?
- How should a game level be represented in an interaction-aware and domain-independent way?
- Given a domain-independent representation of a game level, would it be possible to train one level generator on level representations of different games?

In relation to these questions, our main contributions through this thesis are as follows:

- Introducing tile embeddings as a general representation for Procedural Level Generation via Machine Learning (PLGML).
- Applying tile embeddings to approximate the affordances of tiles of unannotated games.
- Introducing a novel two-step level generation pipeline based on discrete and continuous tile embedding representation.
- Employing the presented level representation and generation approach to generate levels of annotated games and studying the quality of its outputs in comparison to the outputs of an LSTM trained on the VGLC representation.

- Demonstrating the ability of our approach to generate levels for games with only visual information available.

Chapter 2

Background Material

This chapter introduces readers to the concepts necessary to understand the presented thesis. The rest of the chapter is organized as follows: We lay the groundwork by discussing the basics of Artificial Neural Networks in Section 2.1, followed by an introduction to autoencoders in Section 2.2. An autoencoder is a feedforward unsupervised algorithm which we employ for learning representations of game level design. Section 2.3 describes the Recurrent Neural Network (RNN), which is an autoregressive neural network commonly used for generative language modelling. We then discuss the Long Short Term Memory (LSTM) RNN, a special type of RNN that forms the basis for our level generation model.

In Section 2.4, we cover different clustering techniques and how they can be employed to detect groupings in data. We particularly discuss two clustering algorithms: Gaussian Mixture Models (GMM) and Density-Based Spatial Clustering of Applications with Noise (DBSCAN) in detail along with their advantages. These techniques are relevant to our work as we use them in designing a novel loss function for training our autoencoder and to discretize the learned continuous representation of a game level. In Section 2.5, we define common terminologies related to video games and then briefly introduce a well-known representation in PCGML: the Video Game Level Corpus (VGLC).

2.1 Artificial Neural Networks

Artificial Neural Networks, more commonly referred to as Neural Networks, are a fundamental building block of many deep learning algorithms. They are a simplified approximation of the human brain and its ability to learn from an experience by modifying itself. A neural network consists of connections of nodes organized into multiple layers. Figure 2.1, visualizes a simple neural network. We now walk through the presented neural network from left to right, explaining three categories of neural network layers:

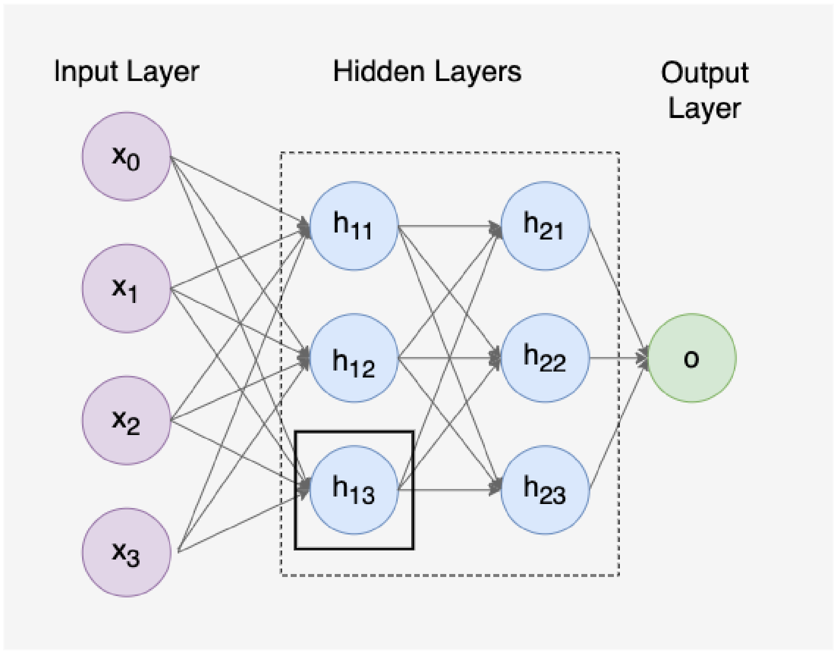


Figure 2.1: Organization of layers in a simple Neural Network.

- The *Input Layer* at the beginning of the workflow accepts all the data into the network for subsequent processing. This is followed by one or more hidden layers.
- *Hidden layers* are responsible for applying mathematical functions and data transformations on the input data. A hidden layer takes the input from the previous layer, performs computations and feeds the output to the next layer. With the number of hidden layers, the complexity of the neural networks increases. As we progress into the network, the

hidden layer tends to detect more abstract features by combining the features of the previous layers. For example, if we consider the task of face recognition, the initial layers might detect edges or parts of the face like eyes, nose, and ears. In contrast, the later layers might detect the overall complexion.

- The *Output Layer* is the last layer of the network that produces the prediction for the intended task. The number of nodes in the output layer depends on the problem we are trying to solve. For instance, in a classification problem, the number of nodes in the output layer will equal the number of classes present in the dataset.

Each layer is comprised of neurons which are the basic computational units of a neural network. The inputs are connected to each neuron by weights. Each neuron calculates the weighted sum of input features and passes it through an activation function as shown in Figure 2.2. The activation function of a neuron can be mathematically expressed as:

$$y = f(\sum_{i=1}^{n} x_i * w_i + b) \tag{2.1}$$

where,

- x_i $i \epsilon 1, 2, ..., n$ corresponds to the input feature vector.
- w_i is the *weight* associated with the input feature x_i. Weights control the impact an input value will have on the output.
- b is the *bias.* Bias is used to offset the result of the activation function.
- f is the *activation function.* The output of the activation function is the signal a neuron will pass on to later layers of the neural network, in response to a particular input. Activation functions are typically non-linear. Prominent examples of activation functions include *Sigmoid, Tanh, ReLU, and Leaky ReLU.*

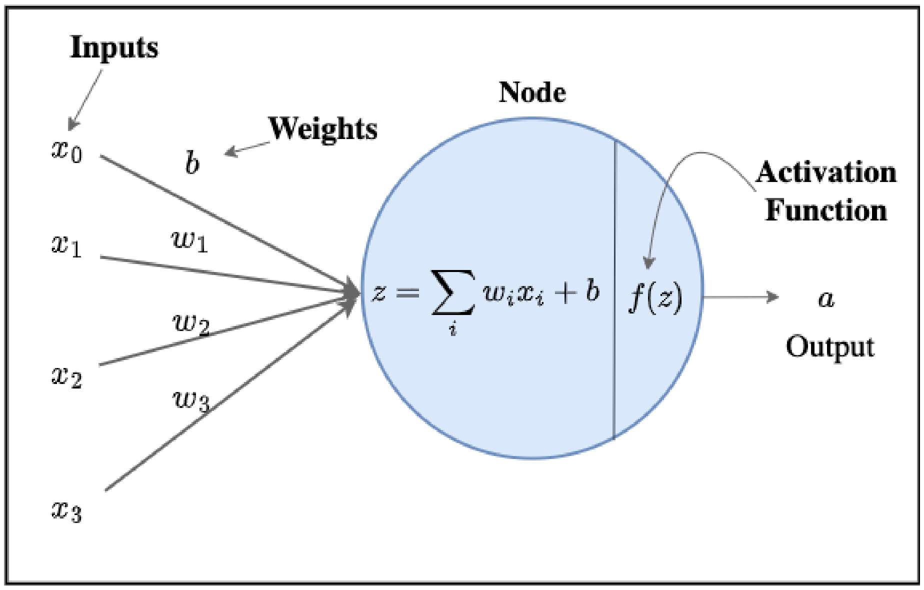


Figure 2.2: Peeking into node h_{11} of the neural network shown in Figure 2.1. The neuron computes preactivation z as the summation of inputs multiplied by the weights. The bias b is also added to the preactivation and then the activation function f is applied to compute the final activation a which is passed on to the neurons in the next layer.

Processing information from the input layer through the hidden layers to calculate the output in the final layer is referred to as *Forward Propagation* in a neural network. As evident from the equation above, weights and biases play a crucial role in calculating the output. For the model performance to improve, the predicted output should grow closer to the optimal solution. A *Loss Function* is a correctional function for the neural network. It is designed to calculate how far the predictions are from the optimal solution, i.e., the error. Thus, learning in neural networks is defined as an optimization problem of minimizing the error by updating the values of weights and biases.

Backpropagation in a neural network is responsible for distributing the total error across the network by propagating it from the output layer through the hidden layers till it reaches the input layer. It calculates a partial derivative of the loss function with respect to network parameters i.e., the weights and biases. This derivative is called a gradient. Each weight and bias of the network is updated based on their computed gradient. Commonly employed backpropagation algorithms are Stochastic Gradient Descent (SGD), Adam, and Adagrad.

Based on the architecture and connections between layers, two types of neural networks are relevant to our thesis:

1. *Feed Forward Neural Network:* Feed-forward neural networks allow the information to flow in a single direction. The output of the previous layer forms the input to the next layer. They are mostly used for pattern recognition, classification and regression problems.

2. *Recurrent Neural Network:* These types of neural networks are dynamic and are widely applied for time-series tasks that require handling temporal data such as stock prediction [46], language translation [62] and image captioning [66]. They distinguish themselves from feed-forward neural networks by their concept of 'memory'. The decisions in recurrent neural networks are based not only on the current input but also on the previous output.

Neural networks form the backbone of our representation learning and level generation network architectures. We elaborate on the particular neural networks our work relies on in the next following sections.

2.2 Autoencoders

An Autoencoder is a feed-forward neural network trained to reconstruct its original input in order to learn a useful abstract representation. It consists of an encoder network that learns to map an n-dimensional input x to an abstract p-dimensional latent space. An encoder A can be mathematically represented with the function: $A : \mathbb{R}^n \to \mathbb{R}^p$. The latent space is also referred to as an embedding space and the learned representation is called an embedding vector. The embedding vector is a real-valued continuous vector containing all the important information needed to represent the original data. The encoder network is followed by a decoder network that learns to take the embedding vector code and reconstruct the original input. It can be represented with function B: $B : \mathbb{R}^p \to \mathbb{R}^n$ To summarize, an autoencoder can be represented as:

$$argmin_{A,B}\ \mathbb{E}[\delta(x, B(A(x))] \tag{2.2}$$

where $\mathbb{E}$ is the expectation over the loss δ.

The ability to learn general representations of data with little or no supervision, which can then be effectively used to develop machine learning applications, makes autoencoders a suitable tool for representation learning. To achieve a valuable representation, it is important to prevent an autoencoder from learning an identity function that simply copies the input to the output. To this effect, an autoencoder can be optimized using additional constraints. These constraints can be imposed on the architecture (Undercomplete Autoencoders [15]), the input (Denoising Autoencoders [65]) or by using a regularized loss function (Sparse Autoencoders [38]). In this thesis, we train an undercomplete autoencoder with a tailored loss function to learn our embedding representation.

2.2.1 Undercomplete Autoencoders

An undercomplete autoencoder compresses the input in a hidden vector representation to be smaller than dimension of its input. By forcing the input through a bottleneck as illustrated in Figure 2.3, an undercomplete autoencoder can capture the most significant features of the training data.

An autoencoder is primarily trained to minimize the reconstruction loss which measures the distance between the original input and decoder output. The reconstruction loss depends on the representation of input-output pairs. For instance, when working with continuous data like images, the reconstruction loss commonly employed is Mean Squared Error [13]. For discrete data, categorical losses such as Binary Cross Entropy [7] or Categorical Cross Entropy [1] can be used. While training an autoencoder with reconstruction loss is common for many applications, their loss function can also be customized to a desired learning goal such as to learn a valuable representation for clustering applications .

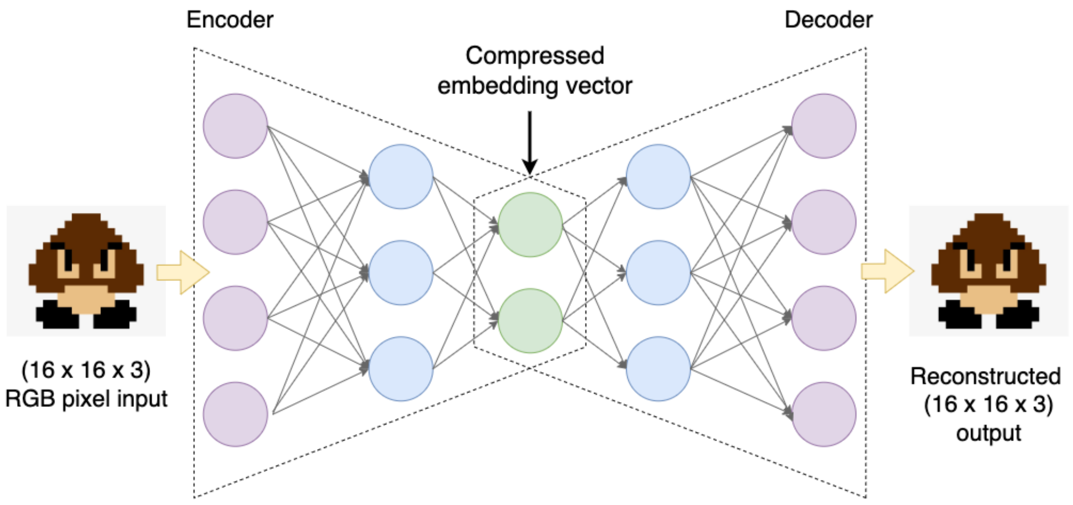

Encoder
Decoder
Compressed embedding vector
(16 x 16 x 3) RGB pixel input
Reconstructed (16 x 16 x 3) output

Figure 2.3: Architecture of an undercomplete autoencoder. An encoder takes in a visual representation of a Goomba and maps it to a compressed embedding vector. The decoder then reconstructs the original image of the Goomba from the compressed representation.

2.3 Recurrent Neural Network

Feed-forward neural networks assume each input to be independent of others and that the decisions are based only on the current input. Due to this assumption, standard feed-forward neural networks show limited ability in modelling sequential data where the data points exhibit dependency between the observations. Examples of such data include text streams, audio and video clips, and time-series data like stock prices and weather. An RNN is a type of neural network specially designed to handle sequential information. It approximates the concept of memory by sharing the weights of hidden layers and allowing them to refer back to earlier input.

Figure 2.4 shows the architecture of an RNN where,

- W_{xh} are the weights for the connection from the input layer to the hidden layer.
- W_{hy} are the weights for the connection from the hidden layer to the output layer.
- W are the weights for the connection from the hidden layer to the hidden layer (memory).

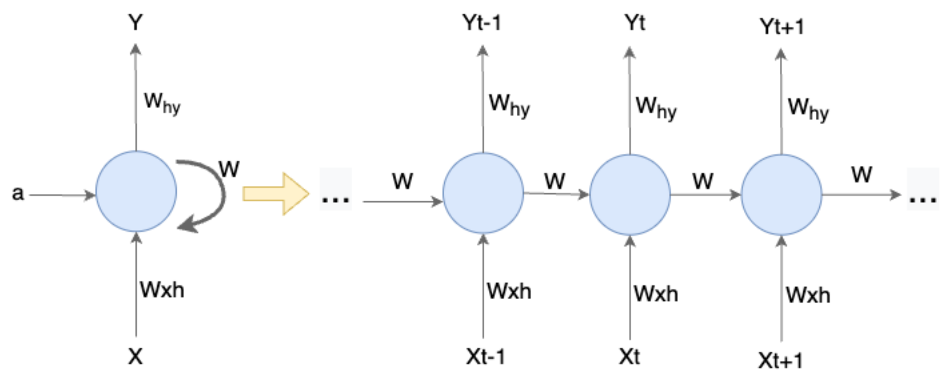


Figure 2.4: An example of a folded Recurrent Neural Network on the left and its unfolded version on the right. This figure visualises how parameters of an RNN network are shared across time.

- a is the activation of the layer.

To train an RNN, we update its network parameters by computing their gradient. The gradient flows backwards across the timesteps. This algorithm is called *Backpropagation Through Time (BPTT)* which computes the gradient at any given time by summing the gradient errors over subsequent timesteps. The number of timesteps in an RNN increases with an increase in sequence length. Therefore for long sequences, the multiplicative term of the gradient dominates the backpropagation. Naturally, the gradient either explodes or vanishes which makes it difficult to train an RNN on long sequences.

The problem of exploding gradients in an RNN can be addressed by employing gradient clipping or by using different weight initializations. Similarly, two RNN variants have been designed to deal with the issue of vanishing gradients 1) Long Short Term Memory RNN [22] and 2) Gated Recurrent Unit [8].

2.3.1 Long Short Term Memory Network

LSTMs are a type of RNN designed to address the problem of vanishing gradients by extending memory. They are well-suited for handling long-term dependencies in sequential data. Similar to a vanilla RNN, an LSTM RNN is also composed of recurrent units. However, there is a difference in the operations performed inside the units and in the connections between them. In addition to the existing hidden state as seen in the RNN, an LSTM maintains a cell state. The cell state acts as long-term memory. An LSTM unit computes the hidden state and the cell state using a gated mechanism. It is comprised of three gates: a forget gate, an input gate, and an output gate. By using these gates, a LSTM unit can decide which information to remove, add, and store in the network. The architecture of an LSTM unit is shown in Figure 2.5.

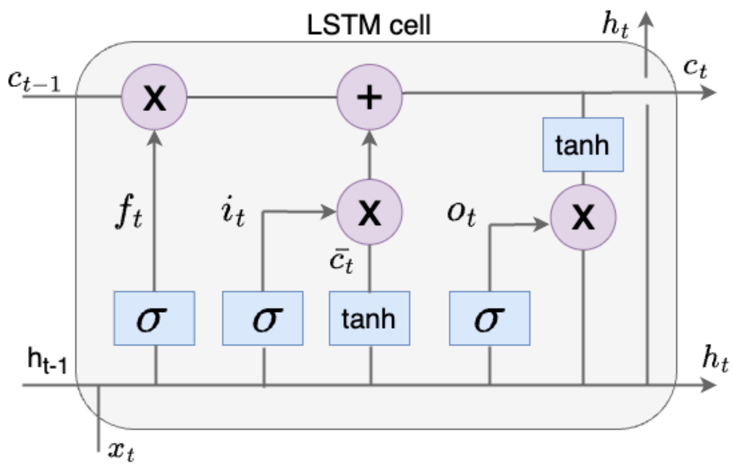


Figure 2.5: A detailed architecture of a Long Short Term Memory cell illustrating how an input gate i_t, a forget gate f_t and an output gate o_t are computed based on the information in the current input x_t, the previous cell state c_{t-1} and hidden state h_{t-1}.

1. A *Forget Gate* f_t indicates which information from the previous cell state is relevant at the current time stamp. It is comprised of a *Sigmoid* function σ based on the previous hidden state information h_{t-1} and the current input x_t. A *Sigmoid* function outputs values between 0 and 1. Values closer to 1 indicate 'to remember' and values closer to 0 indicate 'to forget'. Each value in the previous cell state (c_{t-1}) is then multiplied by the forget gate to decide which values of the cell state are relevant.

2. An *Input Gate* i_t updates the cell state. It accepts the previous hidden state information and the current input, followed by simultaneously applying a *Sigmoid* and *Tanh* activation function to them. The *Sigmoid* output has values between 0 and 1 indicating the importance of each

value. Values closer to 1 indicate more importance. On the other hand, the *Tanh* function scales the values between -1 to 1 to help regulate the network. The *Sigmoid* and *Tanh* activations are multiplied to produce the new cell state c_t.

3. The *Output Gate* o_t uses the *Tanh* function to regulate the output of the newly modified cell state and a *Sigmoid* function to decide which information of the new cell state to carry forward as the next hidden state h_t.

Like an RNN, an LSTM network can also be unfolded in time along the input sequence as seen in Figure 2.6.

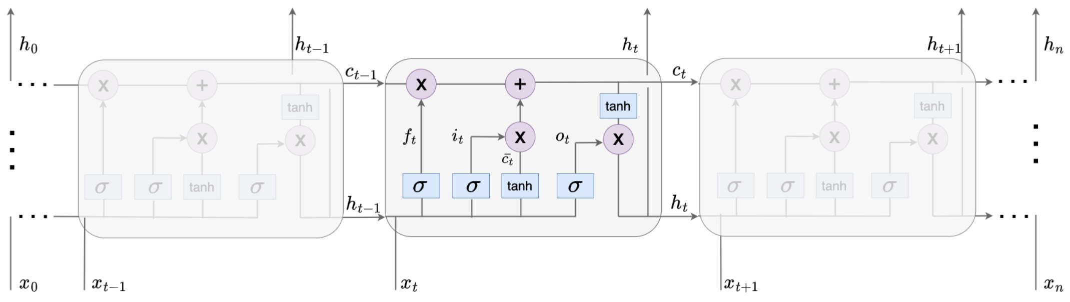


Figure 2.6: Unfolded LSTM Network along the input sequence of length n.

LSTMs are capable of learning structural composition of sequential datum. For textual data, an LSTM can be trained to predict the next word given a previous sequence of words in a sentence. Such a trained predictive LSTM model can also be employed for sampling new sentences. This can be achieved by first feeding a sequence of seed words to the trained LSTM to get a predicted word as output. The predicted word is then appended to the initial seed sequence and fed to the LSTM again to predict the next word. The process of shifting the window of the seed text to include the previous output and again feeding it back to the LSTM to generate next words in sequence is performed iteratively to generate an entire sequence of text. Such generative networks in which the outputs are fed back into the model as inputs are called autoregressive.

In PCGML, 2D video game data can be viewed as sequential data [60]. Therefore, similar to text generation, an autoregressive LSTM network can be used for generating new sequences representing game levels.

2.4 Clustering

Clustering is an unsupervised learning task that identifies groups of similar objects in an unlabelled dataset. It implicitly identifies hidden patterns of features and divides the underlying data into discrete clusters. Based on different criteria on which groupings can be identified, clustering algorithms can be categorized as follows:

1. **Centroid-based**: Centroid-based clustering algorithms partition the data into a specified number of clusters k, based on the proximity of the data points to the cluster centroids.

2. **Distribution-based**: Distribution-based clusters assume different distributions in the underlying data. These approaches assign data points to clusters based on their likelihood of being drawn from the same distribution.

3. **Density-based**: Density-based clustering algorithms define clusters as regions of high density separated by low-density regions.

4. **Connectivity-based**: Connectivity-based clustering methods organize the data into a hierarchical structure based on group similarities.

Within the scope of this thesis, we narrow our focus on two types of clustering algorithms: Distribution-based clustering (Gaussian Mixture Models) and Density-based clustering (DBSCAN).

2.4.1 Gaussian Mixture Models

A Gaussian distribution is a bell-shaped, continuous probability distribution that is symmetrical around the mean. A Gaussian Mixture Model views the underlying dataset as a mixture of multiple Gaussian distributions, each representing a cluster (Figure 2.7). Mathematically the model can be represented as:

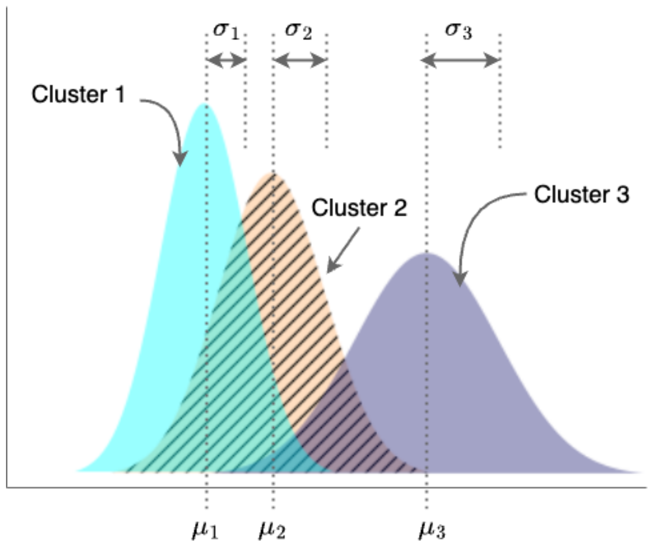


Figure 2.7: The figure shows three clusters ($K = 3$) stemming from a mixture of three Gaussians with different mean and standard deviation values.

$$p(x) \;=\; \sum_{k=1}^{K} \pi_k \, \mathcal{N}(x|\mu_k, \Sigma_k) \tag{2.3}$$

where,

- μ_k defines the mean of the distribution k, $k \,\epsilon\, 1, ...K$
- K is the total number of distributions or clusters.
- Σ is the covariance matrix defining the width of each curve.
- π is the mixing coefficient which assigns weight to each distribution. Mixing coefficients are probabilities and are subject to constraints $\sum_{k=1}^{K} \pi_k \;=\; 1 \; and \; \pi_k \geq 0 \; \forall k$

A GMM uses the Expectation-Maximization algorithm (EM) for estimating the model's parameters. The EM algorithm alternates between two steps until the model converges: (1) The E-step that calculates the posterior probability of each data point being generated by each Gaussian k given the model parameters (2) The M-step updates the mean, covariance matrix and component weights based on the points assigned to each Gaussian k.

A GMM is a powerful tool for clustering that offers flexibility in the shapes and sizes of clusters. It is particularly useful for applications where data can be assumed to be generated as a mixture of different Gaussian distributions. For instance, a mixture of game levels from different genres can be pooled together to form training data. A GMM estimates the likelihood of a data point belonging to a particular cluster. This probabilistic cluster assignment is beneficial in cases where it's hard to assign a single label to each data point. For example, consider the task of classifying levels of different games according to their genre. Puzzle Quest is a turn-based puzzle game that is backed by action. It involves tile-matching puzzles to win gold, spells, and equipment, in order to battle against other players. Since a Puzzle Quest level could be categorized as a puzzle or action game level, it wouldn't make sense to assign it to only one genre. GMM is a soft-clustering algorithm i.e., it predicts the likelihood of a data point belonging to each cluster thus allowing partial cluster assignments. In the example above, employing a GMM would prove beneficial as it can assign a level to more than one genre.

2.4.2 Density-Based Spatial Clustering of Applications with Noise (DBSCAN)

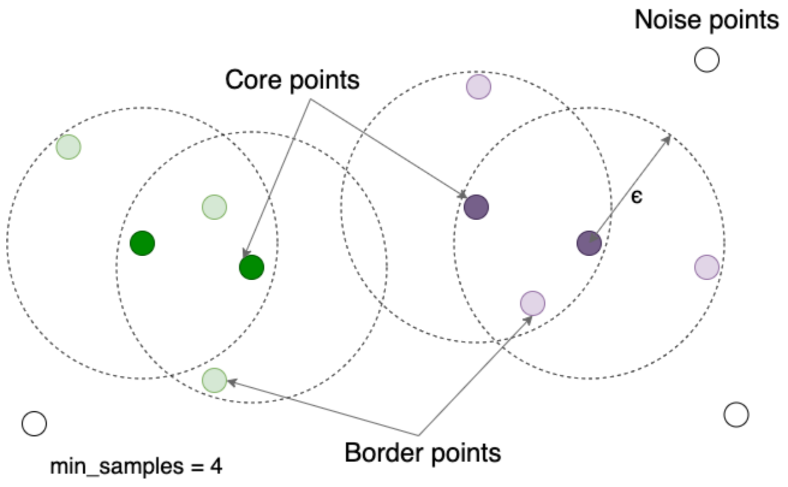


Figure 2.8: A demonstration of DBSCAN Clustering with eps: ϵ and min_samples: 4 to group points in two clusters. The two clusters are indicated by two colours: Green and Purple. A deeper tone of each colour indicates core points and a lighter tone indicates border points.

DBSCAN is a popular density-based clustering algorithm that divides data points based on their spatial distribution. DBSCAN defines a cluster as a continuous space of high-density points separated from other clusters by lower-density regions. To detect clusters, it relies on three parameters:

1. *eps*: The maximum distance at which two points can be considered neighbours.

2. *min_samples*: The minimum number of points required to consider a region dense, i.e., to form a cluster. As a rule of thumb, $min\_samples >= D + 1$ where D is the number of dimensions in a dataset.

3. *metric*: The distance function used to calculate the distance between two points. Commonly used metrics include Euclidean, Cosine, and Manhattan.

Based on these parameters, DBSCAN classifies each point in the dataset as either a core point, border point or noise point as shown in Figure 2.8.

1. A *Core Point* has at least min_samples number of points within its neighbourhood defined by eps.

2. A *Border Point* has fewer points than min_samples within the radius of eps but is in the neighbourhood of one or more core points.

3. A point that can neither be classified as a core point nor a border point is a *Noise Point*.

Compared to other clustering algorithms, DBSCAN offers an important practical advantage of noise detection. Unlike most of the other clustering algorithms, which use distance as a measure to detect clusters, DBSCAN performs clustering based only on the density in a region. Thus, DBSCAN does not require us to specify the number of clusters and allows clusters of arbitrary shapes.

2.5 Common Video Game Terminology

In this chapter's earlier sections, we talked about the Machine Learning techniques employed in the presented thesis. Our work relies on the application of these ML algorithms for learning representation and generation of 2D tile-based platformer game levels. To familiarize readers with our application domain, we pick some essential concepts related to level design and define them in this section.

2.5.1 Platformer Games

Platformer refers to the genre of video games where a player navigates through an environment to reach a goal. The gameplay space is broken up into subspaces called levels that the player must navigate one at a time. During a game, the player may need to battle enemies, avoid obstacles, and gather rewards. We focus on platformer games as they make up a majority of our training data and evaluation domains. Following are the commonly shared components of platformer game level design as identified by [48]:

Platforms:

Platformer game levels include a series of 'platforms' on which the player can stand, run or jump. Platforms can have various characteristics. For example, a sticky platform or a platform made of high-resistance material can slow down a player whereas an ice-based platform or the presence of oil on a platform can make it slippery. A platform can also be destructible or collapse after an event, such as after a specific number of jumps.

Obstacles:

Obstacles provide a challenging component to game levels. They are the game elements capable of damaging the player's movement. Depending on the type of obstacle, a player may choose to fight, kill, or avoid them. Obstacles can be static such as platform gaps and spikes, or they can be dynamic like a moving enemy or firing cannons.

Collectibles:

Collectibles consist of reward items such as gems, emeralds, weapons, cards and coins that a player can gather. Depending on their type, collectibles can have different values. Some collectibles increase a player's score, while some provide power-ups by improving a player's health, granting extra lives, or transforming a character. For instance, a mushroom in Super Mario Bros. turns Mario into his super form.

Movement Aids:

Movement aids include elements such as trampolines and ropes that support the player's navigation and basic gameplay mechanics.

Triggers:

Triggers include objects capable of changing the state of the level. For example, hitting a certain block can open a path for a short period.

2.5.2 Level Design

Designing a game level involves bringing together all the game elements to create an immersive experience for a player. Along with specifying the positions of game objects such as platforms, rewards, and obstacles, a level design also defines their visuals. For efficient memory utilization, many platformer games are designed using tilemaps where different game objects are represented using tiles. Tiles are small regularly-shaped graphical units repeated to form a game level. Each tile also has a set of associated mechanical affordances that define its behaviour. For instance a *Brick* tile can have affordances such as *Solid* and *Breakable* referring to its structural properties. A tileset refers to a complete set of all tiles available for use in the environment and a tile sprite is the pixel art designed for different surfaces or object types. Games relying on tiles are commonly referred to as tile-based games. Examples of tile-based platformer games include, but are not limited to, Super Mario Bros., Megaman, Kid Icarus, Legend of Zelda and Lode Runner. We draw on these games as the training data in this thesis.

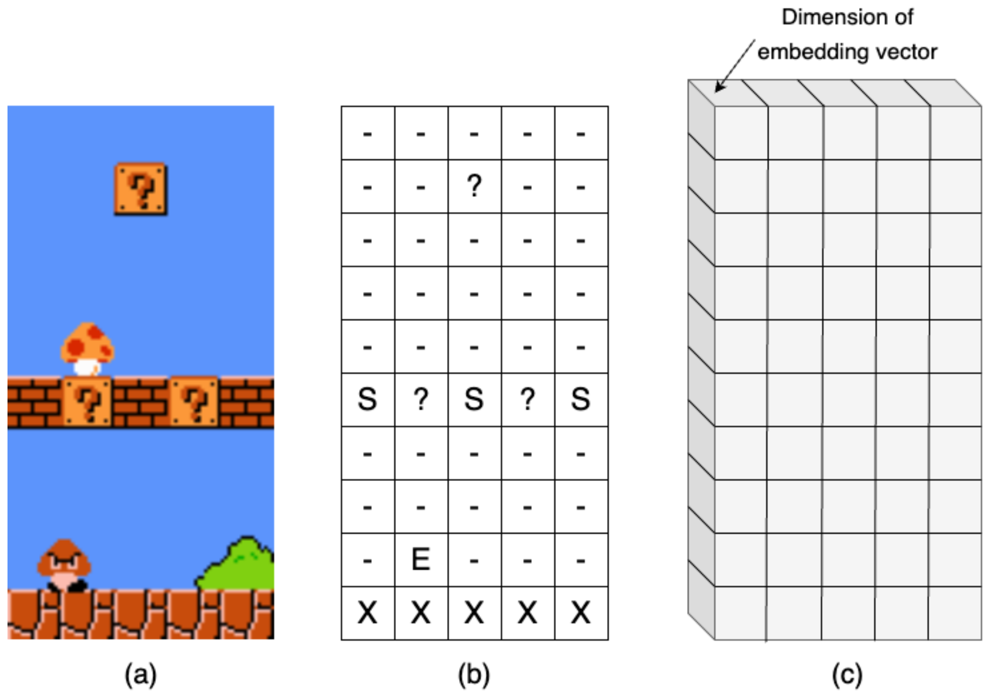


Figure 2.9: (a) SMB level Image (b) Corresponding VGLC representation (c) Proposed continuous tile embedding representation

Procedural Content Generation techniques are leveraged for algorithmic generation of game content such as levels, music, textures and characters. **Procedural Content Generation via Machine Learning** is a branch of PCG that applies machine learning algorithms to generate new content matching the style of the training dataset.

This thesis focuses particularly on representation learning for game levels and studies the importance of representation for the PCGML task of level generation. For successful generative modelling, the underlying representation must embed crucial level design information. We propose an alternative to the commonly used representation in PCGML: the Video Game Level Corpus (VGLC) [61]. The VGLC representation includes a set of 2D tile-based video game levels represented as a sequence of repeating discrete symbols, where each symbol represents a tile. Our goal is to learn a continuous representation that embeds the behavioural and visual information of game objects Figure 2.9. We can then use this representation to learn a discrete representation.

The usage of the terms ‘discrete representation’ and ‘continuous representation’ is in reference to the discrete and continuous nature of the level design representations. A categorical or discrete variable takes a countably finite number of values whereas a continuous variable can take on an infinite set of values.

In this chapter we reviewed the required background for our work. In the next chapter, we discuss the prior related work.

Chapter 3

Literature Review

In Section 3.1 of this chapter, we review prior studies that employ an autoencoder network for PLGML tasks such as level generation and blending. We then discuss existing game level representation practices in PCGML in Section 3.2. Both these sections particularly highlight the approaches that employ clustering, as our work relies on it to discretize tile embeddings. In Section 3.3, we review literature on word embeddings, and also cover some prior work on game embeddings.

3.1 Autoencoders

Prior research has successfully employed autoencoders [21] and Variational Autoencoders (VAEs) [30] for PCGML application such as level generation and blending [44], [63]. Jain et al. [25] were the first to demonstrate that autoencoders could learn representations useful in downstream PCGML tasks. Guzdial et al. [16] presented an explainable co-creative tool by training an autoencoder on existing level structures and associated design pattern labels. Yang et al. [68] employed a Variational Autoencoder with a Gaussian mixture as a prior distribution (GMVAE) for level generation. Their work essentially relies on clustering to identify similar (16×16) chunks from levels of multiple games. The learned components of the Gaussian Mixture Model are then used to generate new chunks of the same style. Karth et al. [26] proposed neurosymbolic map generation using a VQ-VAE and Wave Function Collapse (WFC). A VQ-VAE quantizes patches of level images to a finite tileset on which WFC is applied to generate levels. While these works do not directly focus on embeddings, the essence of our approach is in learning and optimizing the latent tile embedding representation.

Alvernaz and Togelius [2] trained an autoencoder to generate a lower dimensional representation of a videogame environment which was then used in a reinforcement learning framework. This is similar to our approach as we learn a level representation using level structure and affordances. However, while their work focused on automating gameplay, we focus on automating design. Additionally, a majority of these previous approaches are based on representations of either chunks of levels or entire levels. In our presented work, we instead focus on learning the representation of the level's basic building blocks, tiles.

3.2 Level Representation

Most PCGML approaches addressing level design tasks rely on datasets of annotated images [3], [44], [53], [60], or gameplay videos [17]. A notable contribution to the current Game AI research community is the Video Game Level Corpus (VGLC) [61]. It presented a training corpus for 12 games consisting of level images and parseable text files in three different formats: tiles, graphs, and vectors. This work has gained popularity with 106 citations at the time of this writing.

How one determines a set of affordances for tilesets is an open area of research. While most PCGML approaches rely on the hand-authored set from the VGLC or similar representations, there has been some effort to derive these in a more grounded way. Summerville et al. [57] attempted to learn the semantic properties of tiles from gameplay. Snodgrass [51] clustered potential tiles into groups and estimating their quality based on levels generated using these potential tiles. The tiles surrounding the candidate tile played an important role in clustering. Similar to this work, we use clustering to learn discrete representation by grouping continuous embedding representation. However, in our presented work instead of using neighbouring tiles for clustering decisions, we base them on the RGB pixel representation of candidate tile, its behavioural and edge information. For level blending tasks such as Sarkar et al. [43], which combines different game representations, there's a need to come up with a joint set of affordances across games. However, this is typically done by hand. Bentley and Osborn [4] presented an annotation tool and a common set of nine affordances. We leverage the affordances from this tool.

3.3 Embedding Vectors

Word embeddings [34] are extensively used in modern NLP tasks. Each word is represented as a continuous d-dimensional vector denoted by $w^i \in \; R^d$. The low-dimensional representation captures the word's meaning (semantics) from streams of text. Words related to each other are placed closer in the vector space, and relationships between words are encoded as the differences between these points. A popular word analogy that can be demonstrated by this vector space is $\vec{king} - \vec{man} + \vec{woman} \approx \vec{queen}$, which demonstrates an understanding of concepts and context by the model. While the potential of word embeddings to hold these analogies is questionable, embedding vectors have proven to be efficient and fruitful in representing words for several NLP applications [9], [12], [35], [67].

World models represent a novel approach to learning to represent an entire game or similar virtual environment as a neural network [18], [29]. Related to this, Yousefzadeh Khameneh and Guzdial [28] used a VAE to extract embeddings of the entities in a game, which they call *entity embeddings*, which encoded information of gameplay elements. We instead focus on capturing the level structure in our representation and define *tile embeddings* as a d-dimensional vector in an embedding space encoding the semantic information of a tile.

To the best of our knowledge, we are the first to tie clustering and embeddings together for representation learning in PCGML. However, this approach has been explored in other fields like reinforcement learning for games. [32] introduced the shrinkage effect in training an encoder for extracting representations of players in professional ice hockey. It allows the model to transfer information between the observations of different players such that statistically similar players lead to similar representations under similar game contexts. We draw a parallel to this work and implement clustering loss to enforce intrinsic clustering and assign similar representations to tiles with similar RGB pixel representation, affordances and edges.

Chapter 4

Tile Embedding: Initial Approach

The goal of our work is to learn an affordance-rich embedding of a tile as a PLGML representation. In this chapter, we describe our initial attempt at learning tile embeddings by training an undercomplete autoencoder on visual, contextual and behavioural information of tiles. We discuss the specifications of our training data and its preprocessing in Section 4.1, followed by our model architecture and training in Section 4.2. Our trained autoencoder is employed to extract a 256-dimensional embedding vector as our tile representation, which we refer to as a *Tile Embedding.* In Section 4.3, we evaluate the obtained tile embeddings on their ability to approximate the affordances of unannotated tiles. We then apply tile embeddings to level generation for annotated and unannotated games. In Section 4.4, we walk through the results of above evaluations and demonstrate the utility of tile embeddings to serve as a PLGML representation for level generation.

4.1 Data Preparation

Our training data consists of five classic Nintendo Entertainment System (NES) games: Super Mario Bros, Kid Icarus, Legend of Zelda, Lode Runner and Megaman, which are all 2D, tile-based games. Figure 4.2 illustrates our architecture, described in detail below.

We draw on local pixel context and affordances associated with the tiles

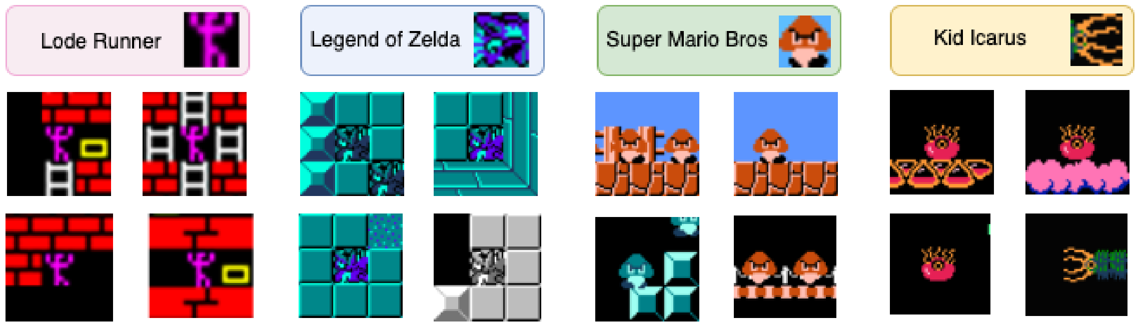


Figure 4.1: Neighbourhood Context for Tiles.

from the VGLC [61]. We incorporate affordances as an input since the visual similarity between tiles can be deceptive. Tiles that differ in pixel appearance may have the same behaviour, such as the recoloured tiles in Figure 4.1. Further, when affordances are not known, the neighbourhood context could be crucial for the embedding vector. For instance, a brick may depict a [‘solid’, ‘breakable’] object in one game, but a background pattern in another game with the affordances [‘empty’, ‘passable’]. However, in this case, the bricks in the latter case would repeat in a way similar to solid coloured sky tiles in other games. Thus the placement of a tile’s embedding value in the latent space is influenced not only by the visuals of the tile but also by its behaviour and relationship with the neighbouring tiles.

4.1.1 Local Pixel Context

One part of the input to our autoencoder is the pixel representation of a tile and its neighbouring tiles as demonstrated in Figure 4.1. For this, we use the level images from the VGLC Corpus. To capture local context, we slide a 48*48 pixel window, as we use a 16*16 tile representation, over the images to extract all unique contexts. By unique we indicate all possible combinations of VGLC tile types in the neighbourhood of the candidate tile, it does not matter if the tiles differ in terms of their pixel appearance. We made this choice to reduce class imbalance in tile types, as “empty” background tiles occur much more frequently than all others.

4.1.2 Semantic Context and Unified Affordances

The other input to our model is the affordances of the candidate tile. The annotations for each of the tiles are obtained from the JSON files stored in the VGLC Corpus [61]. However, these are all game-specific, thus it is necessary to map the different game affordances to a single, unified set. Based on prior work [4], [43], we employ the following 13 common tags: *Block, Breakable, Climbable, Collectable, Element, Empty, Hazard, Moving, Openable, Passable, Pipe, Solid, Wall.* For example, *Climbable, Passable* refers to tiles such as stairs, ropes, and ladders. The player can use these tiles to move in the vertical direction or can choose to pass the tile and continue on their original path. *Hazard* covers all harmful obstacles to the player such as spikes, cannons, and enemies. The affordances for each tile are then expressed as a multi-hot vector, with 1 at the index of features that are present for this tile, and 0 otherwise.

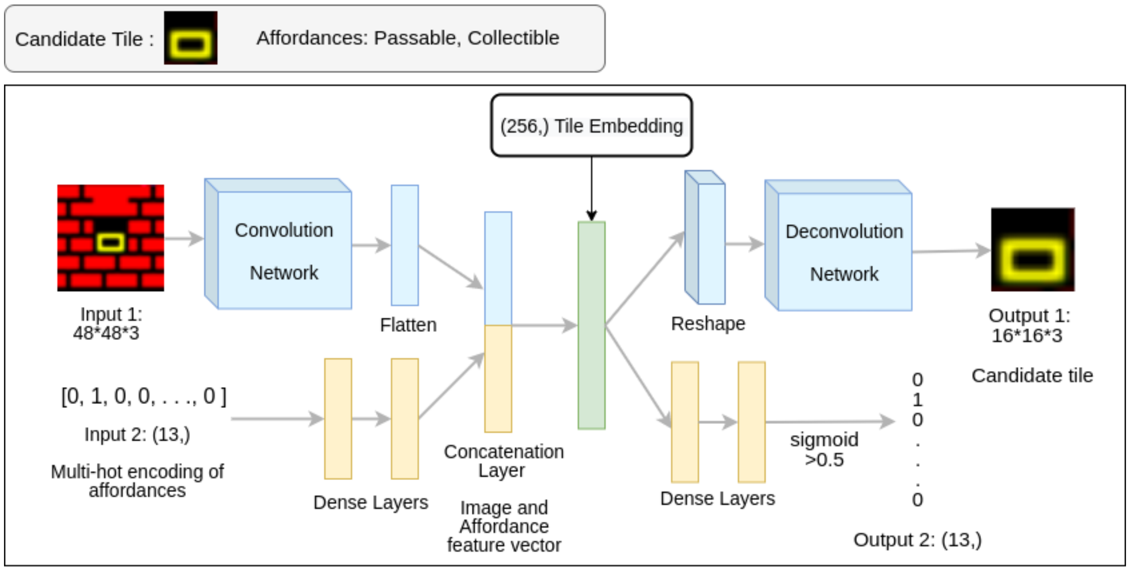


Figure 4.2: Network Architecture.

4.2 Model Architecture

An autoencoder is a feedforward multilayer neural network architecture that learns a compressed representation of the input to capture key structures. In our work, we adapt the X-Shaped VAE architecture proposed by Simidjievski et al. [47]. The encoder consists of two branches that process the individual inputs. The outputs of the two branches are merged and compressed into a single embedding vector which we employ as our tile embedding. The decoder network again splits into two branches to reconstruct the desired outputs.

The 48*48 pixel input is fed to a three-layer encoder convolutional network - the first with 32 (3*3), then 32 (3*3) and finally 16 (3*3) filters. Each layer is followed by Batch Normalization and then *Tanh* activation. Batch normalization applied before a non-linear activation function stabilizes the distribution of the input and reduces the divergence risk [23]. This output is flattened to form a one-dimensional image feature vector. In parallel, the multi-hot feature vector of affordances is passed through two fully connected layers of sizes 32 and 16 with *Tanh* activation for a feature vector encoding of the affordances.

We concatenate the output of both branches and pass it through a fully connected layer to get a (256,) dimensional tile embedding. This captures the relationships between branches in a common latent representation. This merging of information is crucial in cases where the affordance information is unknown, such as when we wish to derive tile embeddings for a new game. We hypothesize in these cases that we can approximate reasonable affordances based on pixel data alone. The decoder is close to an inverse of the encoder. A three-layer deconvolutional network upsamples the embedding vector to reconstruct the pixel portion of our output. Given that we want an embedding for individual tiles, we reconstruct just the 16*16 centre tile. In parallel, in order to reconstruct the affordances, we include two fully connected layers of sizes 16, and 32. The output of these layers is finally connected to a dense layer with *Sigmoid* activation representing the affordances of the centre tile.

We trained this model with the adam optimizer and two-loss functions. For the image output, we use mean square loss. The multi-label prediction task for our N affordances can be formulated as N independent binary classification problems and so we use binary cross-entropy loss as our second loss function. However, our training dataset does not have equal instances of each label. To counter this problem of class imbalance, we derived a TF-IDF vectorizer to compute an importance score for each label based on its frequency. We use this as the weight for each label and define our binary cross-entropy as,

$$Weighted\ BCE = -\sum_{i=1}^{N} y_i log(P(y_i)) * w_i \qquad (4.1)$$

where, y_i is the ground truth, $P(y_i)$ is the predicted probability for label i in N, and w_i is the TF-IDF weight for label i. The objective function combines the two above loss functions with a weighted linear combination. We use the weight 0.8 for the image loss and 0.2 for the affordance loss, which we derived empirically. During training, we employ 20% of our training data as a validation set and apply early stopping to avoid overfitting [19], [37].

We include a t-SNE [64] visualization of our learned latent space (Figure 4.3). It shows a good mix of our tile embeddings across different games. Lode Runner is over-represented as it has the most samples of any game. However, even games like Legend of Zelda, which are very different from the other games, are fairly evenly distributed across this latent space, indicating it has been able to generalize across the different games.

4.3 Evaluation

In this section, we discuss the three evaluations of our system. First, we approximate affordances for tiles of unseen games. Second, we compare tile embeddings and the VGLC tile representation on a level generation task. Finally, we demonstrate the application of tile embeddings for generating levels of a game with no annotated data.

4.3.1 Cross-fold Affordances Analysis

We employ a cross-fold analysis over our five games: Super Mario Bros, Kid Icarus, Legend of Zelda, Lode Runner, and Megaman. Our model is trained on four games with the fifth game held out as test data. We extract and pass 48*48 pixel contexts from test levels as the input to our trained model. We act as though their affordances are not known and pass a (13,) array of zeros

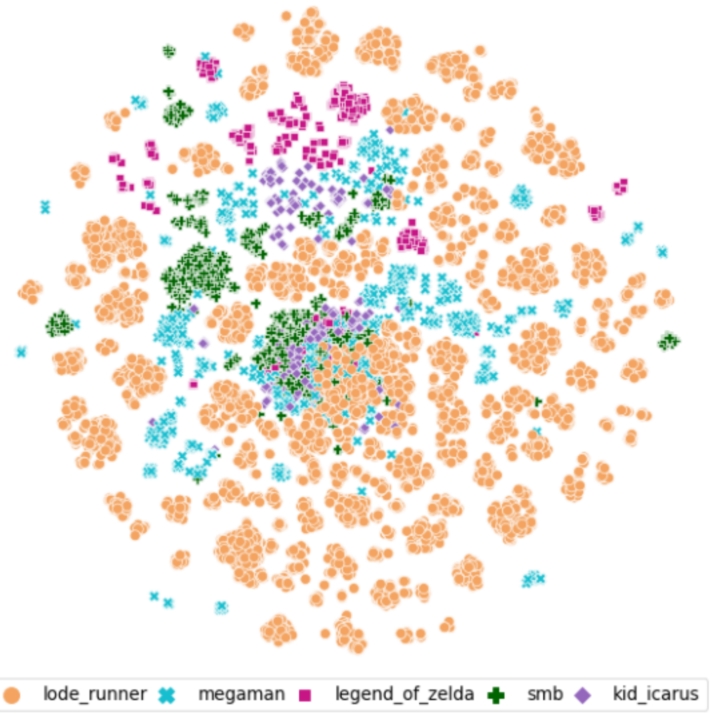


Figure 4.3: t-SNE Visualization of Embedding Space.

as the second input. This allows us to approximate a situation in which we are attempting to predict the affordances for an unseen game.

Evaluating the predicted affordances is a multi-label prediction task where the predicted output may be fully correct, partially correct, or fully incorrect. We therefore employ a number of metrics. *Exact Matching Ratio (EMR)* indicates the percentage of test examples where the predicted labels are exactly correct. EMR can be harsh in a multi-label setting. Hence we adopt example-based and label-based evaluations from [55] with the metrics: *Precision, Recall, Accuracy* to evaluate our model for partial correctness. We include *Example-based* versions of these metrics, which are applied on each instance and averaged over the number of instances in the dataset. For the *Label-based* version of these metrics, we investigate their values for individual labels and compute the average on each label's precision, recall and accuracy independently. The example-based metrics allow us to determine our performances in terms of all the labels (affordances) of each tile, whereas the label-based metrics capture the performance in terms of individual labels (empty, hazard, etc.). Accuracy provides an intuition of the model's correctness in predicting true positives (TP) and true negatives (TN). However, for a sparse prediction vector, accuracy may be misleading. To understand the performance of the model at predicting positives accurately, we employ Precision and Recall. Of all the labels that the model predicted (TP+FP), precision indicates the percentage of labels that were actually true (TP). On the other hand, Recall is the percentage of true labels that the model was able to capture (TP/ (TP+FN)). To further investigate misclassification and missing-label errors, we adopt a more robust metric: *Alpha Evaluation* [5]. Alpha Evaluation weighs missing-label errors (M_x) and misclassification errors (F_x) separately using parameters β (for missing-label) and γ (for misclassification). α controls the forgiveness for errors. Alpha Evaluation is given by the formula,

$$alpha\ score = (1 - \frac{|\beta M_x \ + \ \gamma F_x|}{|Y_x \ \vee \ P_x|})^{\alpha} \tag{4.2}$$

such that $\alpha \geq 0,\ 0 \leq \ \beta, \beta = 1|\gamma = 1$, where Y_x is the ground truth and P_x are the predicted labels.

4.3.2 LSTM for Level Generation-Annotated Game

In this evaluation, we directly compare our tile embedding representation to the state-of-the-art VGLC representation for one game. LSTMs are a special type of RNN with a memory mechanism at the heart of their architecture. LSTMs have been extensively used in PLGML. We adapt the work of Summerville and Mateas [60] and train two similar LSTM networks, one with the VGLC tile representation and the other with our tile embeddings to generate levels for the game Lode Runner. We chose Lode Runner due to the results of the first evaluation. Lode Runner tiles are 8*8 pixels in size. To fit this to our autoencoder architecture, we upscaled the level images using the Python Imaging Library (PIL) such that each tile has a dimension of 16*16 pixels. We trained our model to consider a history of the last 3 rows (approximately 100 tiles) and generate the next 3 rows at a time. Similar to Summerville and Mateas' work, to track the progression of the level, we include column depth as an input to the network. The only differences between the two network implementations are in the input and output layer due to the differences in representation.

Input Layer: Before training an LSTM on tile embeddings, each level is converted to an embedding representation with our trained autoencoder model using context windows and affordances. For instance, a (512 * 352*3) level image of Lode Runner is converted to a (32*22*256) representation. The other LSTM is trained on the (32*22) character representation obtained from the VGLC dataset.

Output Layer: For the LSTM trained on our embedding representation, the output layer predicts the embedding directly. It is modelled as a (256,) Dense layer with *Tanh* activation. Before visualizing a level, we map the predicted embedding to the nearest actual embedding. We use the memory efficient Annoy library[1] to index the embeddings and find the nearest neighbor based on the Manhattan distance. For the VGLC representation, the output of the LSTM is connected to a dense layer with Softmax activation indicating the probability of a tile character. We perform an expressive range analysis of generated levels with the metrics: Linearity and Leniency [33], [50], [56].

- *Linearity* profiles the structure of a level in terms of how well it fits to a line. Linearity is computed by performing linear regression on centre points of all the platforms. We then compute the average distance between each centre point and its projection on the regression line. The score is normalized between [0,1] by dividing by the total number of centre points.

- *Leniency* measures the difficulty of the level. We assign rewards with weight 1 and enemies with weight -1. We then calculate the sum of leniency values and average it with the total number of tiles.

4.3.3 LSTM for Level Generation- Unannotated Game

As our third evaluation, we apply tile embeddings for generating levels for the game Bubble Bobble. We chose this game because no annotated dataset for it

[1]https://github.com/spotify/annoy

currently exists. We download 100 Bubble Bobble level images as our training dataset.[2] We extract tile embeddings by passing the 48*48 pixel context and a (13,) zero vector to an autoencoder trained on all five NES games. We employ the same architecture as we did for Lode Runner to train an LSTM on the embedded level representation. The majority of the Bubble Bobble levels are vertically symmetric, and so we parse the levels column-wise. Our model is trained to generate the right half of the level when the left half is fed as an input. During inference, we mirror the generated right half to produce an entire level.

4.4 Results

4.4.1 Cross-fold Affordances Analysis

Test Data	Example-based				Label-based		
	EMR	Prec	Recall	Acc	Prec	Recall	Acc
SMB	0.17	0.52	0.49	0.39	0.22	0.23	0.11
Kid lcarus	**0.44**	0.63	0.55	0.54	0.27	0.30	**0.14**
Megaman	0.36	0.60	**0.61**	0.53	*0.25*	**0.32**	**0.14**
Lode Runner	*0.11*	*0.44*	*0.27*	*0.27*	0.26	*0.17*	*0.05*
LOZ	0.39	**0.78**	**0.61**	**0.59**	**0.34**	*0.17*	0.10
Mean	0.29	0.59	0.51	0.46	0.27	0.23	0.11
MFL Baseline	0.32	0.46	0.46	0.42	0.46	0.15	0.07

Table 4.1: Results of evaluation metrics for predicting affordances on unseen tiles

Table 4.1 and Table 4.2 present the results of all the evaluation metrics for predicted affordances of unseen game tiles. The most frequent label combination in our dataset is [*'empty','passable'*] accounting for approximately 32% of the dataset. The *Most Frequent Label (MFL) Baseline* indicates the value of our metrics if only the most-frequent label combination is predicted. We include it as a comparison point in the table and in our discussion of the results below. For all the metrics, the closer the value is to 1, the better. **Bold**

[2]https://www.adamdawes.com/retrogaming/bbguide/

Test Data	α-Evaluation with α=1		
	β=0.75, γ=0.25	β=1,γ=1	β=0.25,γ=0.75
SMB	*0.66*	0.29	0.63
Kid lcarus	0.75	**0.45**	0.69
Megaman	0.71	0.40	**0.70**
Lode Runner	0.67	*0.17*	*0.50*
LOZ	**0.78**	0.43	0.65
Mean	0.71	0.35	0.63
MFL Baseline	0.64	0.30	0. 67

Table 4.2: Results of alpha evaluation

indicates the highest value and *italic* indicates the lowest value across the test games for our model.

Exact match ratio (EMR) indicates the percentage of label combinations identified exactly by the model. On average, EMR is 0.29 with a standard deviation of 0.14. The performance is mainly because the metric is aggressive and does not attribute any value to partially correct predictions. The MFL Baseline achieves 0.32 due to the fact that the label makes up 32% of the dataset. However, we still outperform it for three of the five games.

We observe stronger performance on example-based measures that evaluate partial correctness. The average values observed across all example-based evaluations are better in comparison to our MFL baseline. However, if we evaluate individual labels, we find lower values. These lower values on label-based metrics is likely due to the poor performance in predicting rare labels, and due to the over-abundance of the most common labels.

In all five games of our dataset, *Solid, Passable, Empty* tiles occupy a majority of the level as compared to other tiles. Concretely, these labels together account for 70.8% of our training instances. Comparatively higher values on the example-based evaluations than label-based evaluations demonstrate that the model is capable of predicting frequent labels and struggles to predict rare labels such as *Climbable, Collectibles, Element, Block, Wall, Hazard.* For instance, the level design for Legend of Zelda has dungeons composed of *Solid* tiles which our model is good at predicting. Hence metrics for Legend of Zelda have higher values than other games. In comparison, Lode Runner has the lowest values for most of the metrics as its levels have a well-proportioned set of tiles including *Enemy, Collectable, Breakable, Solid, Empty, Passable.* It also had the largest set of overall data, and our model clearly struggled when we withheld these training samples. However, our tile embeddings are able to effectively represent Lode Runner levels when trained on this data, as we demonstrate in the next evaluation.

Table 4.2 highlights the effect of different values of β and γ on the α-evaluation scores. Lowering the weight of misclassification errors (γ) and increasing the weight of missing errors (β), increases the α-evaluation score. This indicates the presence of more misclassification errors and fewer missing labels i.e more False Positives.

Overall, we find these results to be heartening, as our model outperformed our MFL baseline for seven of our ten metrics, and always performed better than it for at least two games. This suggests we can approximate affordances on unseen games. Additionally, for certain use cases like level generation, getting the exact correct affordances is not required as long as the latent space representations of similar entities are close together. This is due to the fact that identifying the entities with similar behaviour will ensure they are appropriately handled in terms of placement during level generation. For instance, as long as enemy tiles are grouped together and separate from solid tiles, a secondary model can be trained to place them in appropriate positions.

4.4.2 LSTM for Level Generation-Annotated Game

Figure 4.4 gives the results for our second evaluation. We generated 150 levels with each LSTM: one trained on tile embeddings and the other trained with the VGLC representation. Figure 4.4 shows the Kernel Density Estimation with Leniency and Linearity. While there is a small section of the plot that the VGLC levels cover that the tile embedding levels do not, overall the levels generated with the tile embeddings representation cover more of the original distribution. In particular, since the VGLC representation did better in terms of linearity, we expect that the VGLC's hand-authored representation was

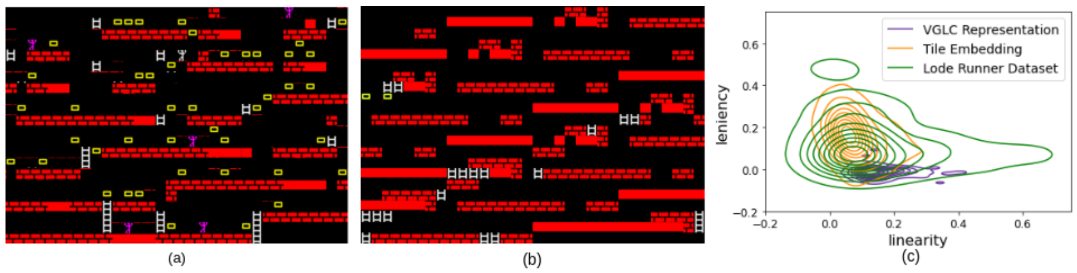


Figure 4.4: (a) Level generated for Lode Runner by training LSTM on tile embeddings. (b) Level generated for Lode Runner by training LSTM on VGLC tile character representation. (c) Kernel Density Estimation with Linearity and Leniency

better able to encode structural knowledge. However, the LSTM struggled to model less common elements with it, including enemies and rewards, which can be seen in the Kernel Density Estimation and example level.

4.4.3 LSTM for Level Generation-Unannotated Game

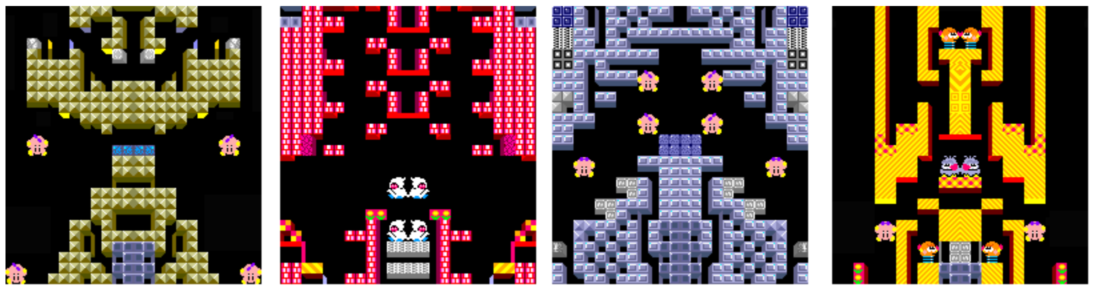


Figure 4.5: Levels Generated for Bubble Bobble

Figure 4.5 shows the Bubble Bobble levels generated by the LSTM trained on tile embeddings. While we note some oddities (floating enemies) the levels overall are of surprisingly high quality, indicating the appropriateness of this approach for generating levels on unseen games. We note that these levels were output as a tile embedding and then visualized with Annoy as described above. One benefit of our approach is that we naturally model tiles with the same affordances (e.g. solid tiles) with all of the visual variety from the original content, leading to the yellow, blue, and pink structures in the output levels. Approaches like the VGLC representation cannot due this, and require a secondary process to map tiles with the same affordances to multiple, distinct output tiles. To play these levels one would need to map them to in-game objects (which is also necessary for the VGLC representation) or employ the embeddings in a playable, deep neural network-based game [29].

4.5 Discussion and Takeaways

In this chapter, we trained an undercomplete autoencoder to take in a tile's mechanical affordances and the local pixel context to learn a 256-dimensional tile embedding representation. The placement of embedding vectors in a latent space is thus influenced by their visual as well as behavioural (affordances) similarity and their relationship with neighbours. We then presented evidence that tile embeddings can reasonably approximate the affordances on unannotated tiles. By generating levels of annotated and unannotated games, we demonstrated that our tile embedding representation could be successfully drawn for the PLGML task of level generation. However, certain shortcomings still need to be addressed. While tile embeddings have shown promising results in generating Lode Runner levels, we observe their limitation in effectively representing and generating levels of Super Mario Bros. (SMB).

The output layer of the tile embedding-based LSTM level generator discussed above has a dense layer at its output. Given a previous sequence of tile embeddings, it outputs a 256-dimensional embedding vector of the next tile in succession. Naturally, at every step, it generates the embedding vector corresponding to the tile having the highest probability of occurrence. This behaviour of a tile embedding-based level generator making locally optimal decisions is in resonance with the mechanism of greedy sampling.

The drawback of greedy sampling in a level generator becomes evident especially when dealing with skewed tile distributions. As shown in Figure 4.6, Lode Runner and SMB levels exhibit contrast in their tile distribution. In comparison to SMB, Lode Runner levels has a balanced distribution of tiles. A typical SMB level has 90% of *Empty* background tiles, making it the most probable tile at any given position. Therefore a tile embedding-based level generator that performs greedy sampling generates empty levels when trained on SMB.

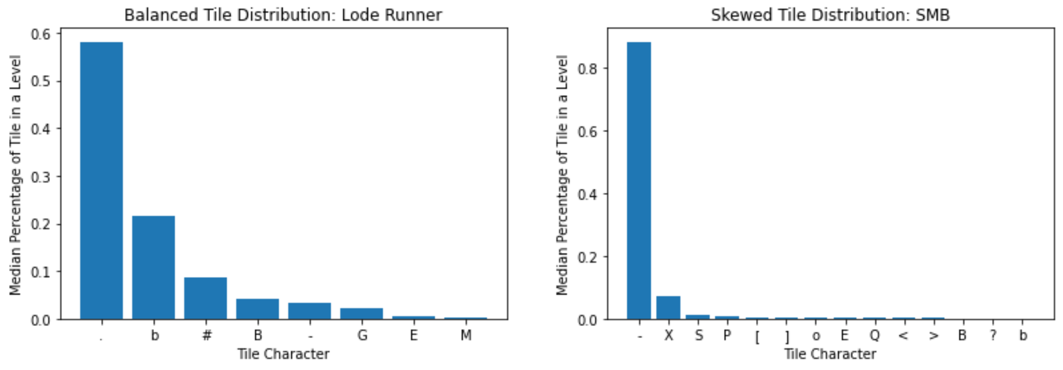


Figure 4.6: Median Frequency of a tile in Lode Runner vs SMB levels.

In the next chapter, we take a step toward alleviating the issue of greedy sampling in a tile embedding-based level generation by discretizing the representation. We take inspiration from prior PLGML level generators that have used probabilistic sampling, made possible due to the discrete nature of VGLC representation. While VGLC is a natural fit for many PLGML applications, it has limited expressivity and benefits from hand-authoring. We believe, the PLGML community would hugely benefit if we could learn the same quality of discrete representation without human transcription as well as preserve its continuous counterpart.

Chapter 5

Clustering-based Tile Embedding: Dealing with games having skewed tile distribution

The goal of this chapter is to learn an improved tile embedding for games with skewed tile distributions for level generation. Towards this objective, we begin this chapter by discussing our modifications to the original tile embedding autoencoder to learn our new Cluster-based Tile Embeddings (CTE). Next, we explain the limitations of an LSTM level generator trained on the original tile embedding representation for games with skewed tile distributions. We then present our novel two-step level generation pipeline that learns a discretization of our CTE through clustering and leverages both representations for level generation. We employ our two-step level generation pipeline for level generation of Super Mario Bros. (SMB). We compare the results of our approach on SMB level generation against the results of LSTMs trained on CTE, the original tile embeddings and the VGLC representation of SMB levels. We then demonstrate our approach's ability to generate levels for two games that no prior PLGML approach has attempted: Bugs Bunny Crazy Castle and Genghis Khan, based solely on images of their levels.

5.1 CTE: Cluster-based Tile Embeddings

The VGLC tile-based representation of a level L is an $h \times w$ dimensional array. Here h and w are the height and width of the level, respectively. Each character

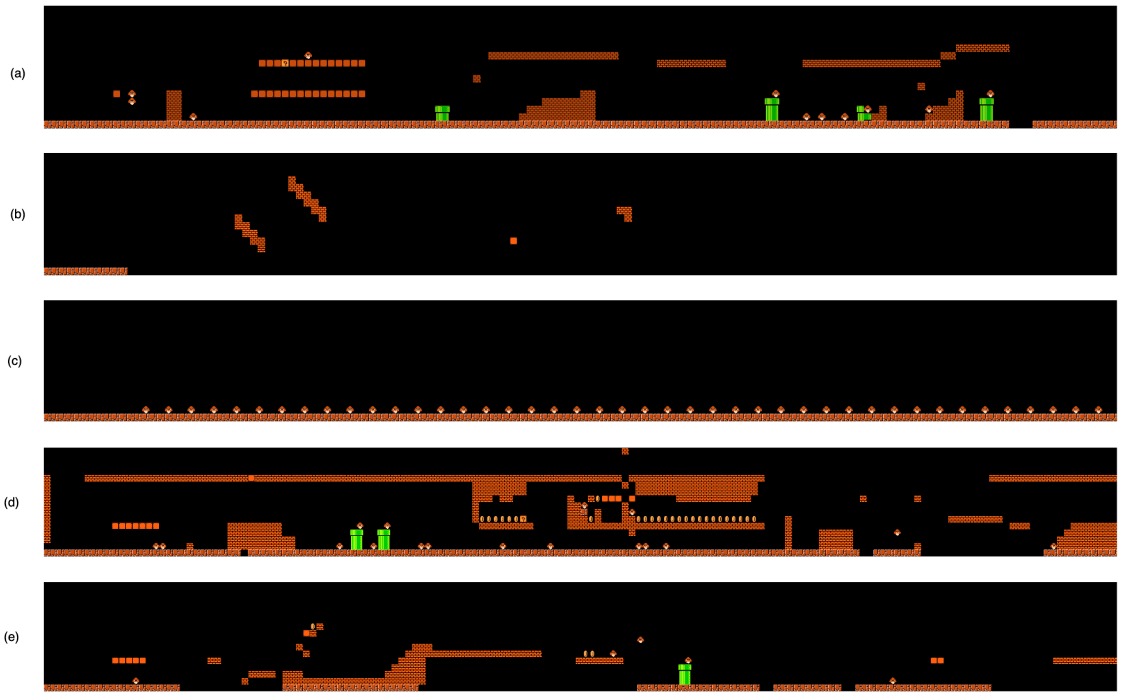
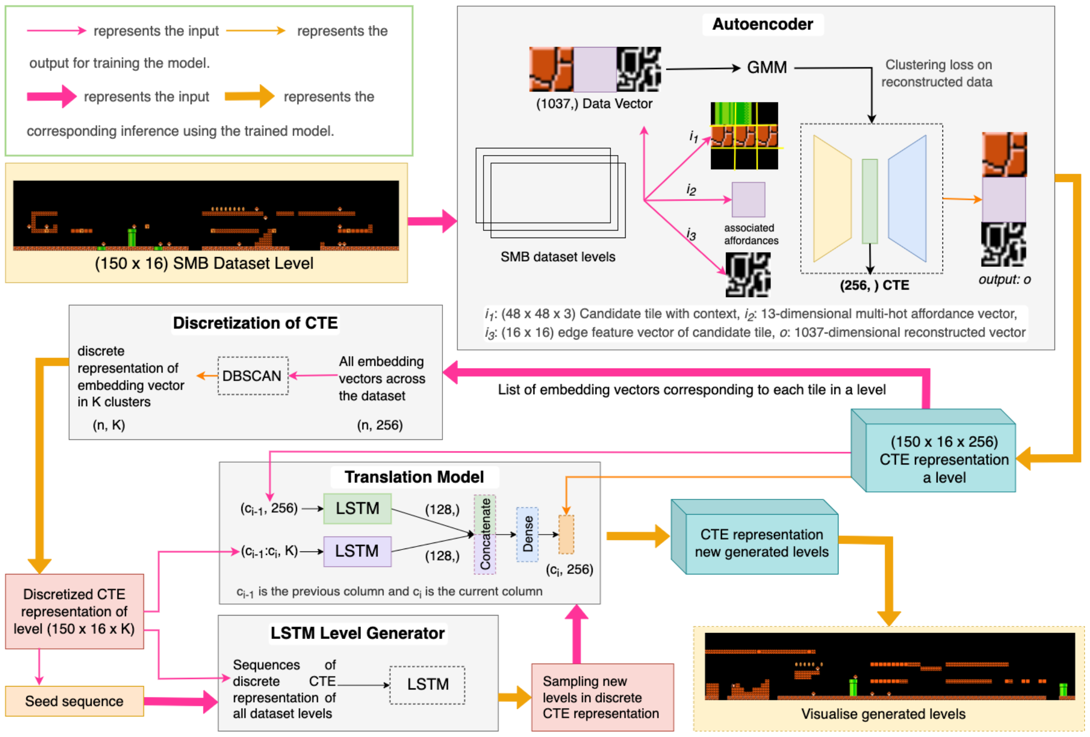


Figure 5.1: SMB LSTM level generator outputs with: (a) VGLC representation (b) original tile embedding (c) CTE. We also include good (d) and bad (e) examples for our two-step CTE level generation process.

of L is called a tile which is associated with a 16×16 pixel representation in a level image and a corresponding set of affordances. Affordances convey a tile's mechanical behaviour.

Our original tile embedding work employed a dual branched autoencoder to learn a 256-dimensional embedding vector representation of a tile [24]. The network accepted two inputs: 1) a 3*3 grid of the candidate tile at the centre with its neighbours surrounding it in the 16*16*3 RGB pixel representation ($48 \times 48 \times 3$), 2) the candidate tile's 13-dimensional one-hot affordance vector. To compare more easily to the original tile embedding work, we utilise the same set of games (Super Mario Bros., Kid Icarus, Megaman, Lode Runner and Legend of Zelda) as our training corpus and maintain the same tile-affordance mapping. The tile-based level data is taken from the VGLC corpus[1] and the JSON files for tile-affordance mapping are from the original tile embedding

[1]`https://github.com/TheVGLC/TheVGLC`



represents the input
represents the output for training the model.
represents the input
represents the corresponding inference using the trained model.
(150 x 16) SMB Dataset Level
Autoencoder
(1037,) Data Vector
GMM
Clustering loss on reconstructed data
SMB dataset levels
associated affordances
(256,) CTE
output: o
i_1: (48 x 48 x 3) Candidate tile with context, i_2: 13-dimensional multi-hot affordance vector, i_3: (16 x 16) edge feature vector of candidate tile, o: 1037-dimensional reconstructed vector
Discretization of CTE
discrete representation of embedding vector in K clusters
DBSCAN
All embedding vectors across the dataset
(n, K)
(n, 256)
List of embedding vectors corresponding to each tile in a level
(150 x 16 x 256) CTE representation a level
Translation Model
(c_{i-1}, 256)
LSTM
(128,)
(c_{i-1}:c_i, K)
LSTM
(128,)
Concatenate
Dense
(c_i, 256)
c_{i-1} is the previous column and c_i is the current column
CTE representation new generated levels
Discretized CTE representation of level (150 x 16 x K)
Seed sequence
LSTM Level Generator
Sequences of discrete CTE representation of all dataset levels
LSTM
Sampling new levels in discrete CTE representation
Visualise generated levels

Figure 5.2: A complete system diagram. We train an autoencoder on the RGB, affordance, and edge information using a cluster-based loss to learn our Cluster-based Tile Embedding (CTE). We then discretize this representation via DBSCAN, and train an LSTM level generator on this discretized CTE. We train a translation model (also an LSTM) to convert back to CTE from the discrete representation output by the level generator.

implementation[2]. We make two modifications to the training of the original autoencoder to better handle level design tasks for games with skewed tile distributions and refer to the newly extracted 256-dimensional embedding vector as the **Cluster-based Tile Embedding** (CTE).

5.1.1 Incorporating Edge Information

When applying the original tile embeddings to games where the affordance information was unknown, we found that the latent space representations depended predominantly on coloured pixel information of a tile. For instance, an empty blue sky tile was placed close to a solid blue brick tile. To discourage this, we included edge information into our embedding. Canny edge detection

[2]`https://github.com/js-mrunal/tile_embeddings`

[6] is a common algorithm for identifying edge information. We convert the $16 \times 16 \times 3$ pixel representation of a tile to grayscale and apply the canny edge detection algorithm to obtain a 16×16 edge feature vector. Thus for each candidate tile, we feed three inputs to our autoencoder: the pixel representation of the candidate tile along with its neighbours ($48 \times 48 \times 3$), a 13-dimensional multi-hot affordance vector and (16×16) edge features.

5.1.2 Introducing Clustering Loss

In the original tile embedding work, the learned latent space was fairly continuous, without clear separation between types of tiles. Learning more distinct groups can improve the utility of a final representation [20]. With an aim to push representations of similar elements closer while keeping representations of dissimilar elements apart, we introduce an explicit cluster-based loss L_c in the training process. For this cluster-based loss, we must cluster our data prior to training our autoencoder. The idea is to leverage the clusters as a guide for representation learning. For each candidate tile, its $16 \times 16 \times 3$ RGB pixel representation, 13-dimensional multi-hot affordance vector, and 16×16 edge vector are fed to a Gaussian Mixture Model (GMM) [39].

A tile can belong to multiple clusters. For instance, it is appropriate to assign a Cannon in MegaMan to a cluster of *Hazards* as well as to a cluster of *Solids*. We rely on a GMM in order to account for such potential overlap in tile groups. We pick an elbow point based on the Silhouette score and Bayesian Information Criterion (BIC) to determine the optimal number of clusters [40], [45]. For the given VGLC dataset, we observe an elbow point at 10 clusters.

We compute our clustering loss (L_c) as the categorical cross entropy error between the GMM cluster assignment of a given tile and its corresponding embedding during training. Along with L_c, our loss function includes the mean squared error on the reconstructed edge feature vector (L_e), the mean squared error over the reconstructed image data (L_i) and the binary cross entropy loss on the reconstructed affordances (L_a). In totality, the loss function can be mathematically represented as:

$$Total\ loss\ =\ (0.5\ L_i) + (1.5\ L_a) + (0.5\ L_e) + (0.5\ L_c) \qquad (5.1)$$

To accurately embed affordance information, we increase the relative weight of its reconstruction.

5.2 Level Generation for Super Mario Bros.

In this section, we describe the difficulty in generating SMB levels using an LSTM trained on the original and CTE tile embeddings, which motivated our novel two-step level generation process described below.

5.2.1 The Curious Case of SMB Level Generation:

We train two LSTM models, one on the original tile embeddings and the other on our CTE representation, for SMB. We follow the training process from [24]. Sampling from an LSTM trained on a continuous representation is deterministic and hence for a given seed input, these models generate only one output as shown in Figure 5.1(b) and (c) respectively. In both cases we feed in the same 200 tiles of flat ground as input. While the CTE representation helps the LSTM learn to generate more reasonable output than the original tile embedding, the output is repetitive and does not reflect Mario-like structure. These outputs show clear limitations of an embedding-based generator in modelling levels with skewed tile distributions, given that the outputs for games with balanced tile distribution appear much more like the original levels [24]. In Table A.2 of the Appendix, we outline the difference in tile distributions for a skewed and a comparatively balanced tile-based game.

Analysis: A possible explanation for these poor results is the lack of a sampling mechanism in the generator, since the CTE output is similar to the most likely Mario level output from a probabilistic generator [52]. In comparison, we observe higher quality results when our LSTM is trained on the VGLC representation as seen in Figure 5.1(a). The only difference between the two models is in the output layer [24]. For the model trained on the

VGLC representation, the output layer is a probability distribution p over possible tile types [60]. The next tile is sampled from p. If we simply pick the most likely next tile, we output levels similar to Figure 5.1(c) even with the VGLC representation. We cannot sample from an LSTM trained on either tile embedding, as the LSTM would output the closest tile embedding, not a probability distribution. To remedy this, we present a two-step level generator which discretizes CTE.

5.2.2 Two Step Level Generation

The two steps of this level generator are to first generate levels in a discrete representation, allowing sampling to occur. Then we have a secondary translation model that converts the levels in this discrete representation back into our CTE representation, so that we can visualize them and extract the predicted affordances. This two-step level generation process naturally requires training two distinct models, one for each step. For both models we use the same LSTM architecture used throughout this paper.

Step I. Training LSTM on Cluster Levels:

To obtain the discrete representation of each level, we leverage the latent structure imposed by the clustering-based loss function. We first begin by converting each level to the CTE representation and then feed all the CTE embedding vectors to the density-based clustering algorithm, DBSCAN [11]. Unlike partitioning-based and distribution-based clustering algorithms, DBSCAN has a number of benefits for clustering in a latent space [27], which makes it highly relevant to this task.

Figure 5.2 shows the overview of our system architecture. If we consider an SMB level of 150×16 VGLC tiles, and replace each tile with its 256-dimensional embedding, we get the $150\times16\times256$ dimensional CTE representation. Further, if each of these embeddings is assigned to a cluster we can simply represent a 256-dimensional embedding by a cluster identifier to get a $150 * 16 * K$ discretized representation, where K is the optimal number of clusters. We refer to this as our cluster representation. For SMB, the optimal number of clusters (K) detected by DBSCAN was 11 with a silhouette score of 0.91. We note that we recalculate these clusters for each new game, unlike the clusters used to inform the CTE cluster loss. We note that we do this to learn a discrete representation as we cannot use the VGLC level representation directly or generating levels for games outside the VGLC corpus would be impossible.

Step II. Translation Model:

The generated output of the previous step is in the cluster representation and cannot be used directly. A cluster may consist of many member tiles, thus a cluster identifier may not be adequate for accurate visual and affordance reconstruction. Therefore, we need a translation mechanism to convert the cluster representation of a level to its associated CTE representation. We train an LSTM network to translate from the cluster representation to CTE. Such a translation mechanism requires the knowledge of context as well as affordances. For instance, to rebuild a solid red brick tile pattern, red bricks cannot be followed by blue bricks even though they may belong to the same cluster. Therefore, as illustrated in Figure 5.2, a CTE representation of column c, depends on: a) the underlying cluster representation of column c and $c-1$, b) the CTE representation of column $c-1$. With this approach, we observed instances where the translation model did not output CTE tiles from the correct clusters. Thus, we replace translated CTE output with its nearest neighbour from the correct cluster. Translated SMB test dataset levels are shown in the Appendix.

5.3 Experiments

We evaluate our two-step level generation pipeline and CTE representation by sampling levels for Super Mario Bros., a game with a skewed tile distribution. We employ commonly-used PCGML metrics to assess the quality of our outputs in comparison to the outputs of an LSTM trained on the VGLC representation, original tile embeddings, and CTE. Additionally, we test our approach's ability to represent and generate levels for two unannotated games:

Bugs Bunny Crazy Castle and Genghis Khan. In this section, we describe our experiments and report our results.

	Dataset	**LSTM on VGLC**	**Two-step level generation**	**LSTM on CTE**	**LSTM on original tile embeddings**
Leniency	-0.0069 ±0.0084	-0.0096 ±0.0077	**-0.0054 ±0.0102**	-0.0021 ±0.0078	*-0.0130 ±0.0155*
Density	0.1315 ±0.0642	0.1669 ±0.0654	**0.1625 ±0.0600**	0.0485 ±0.0310	*0.0721 ±0.0172*
Linearity	0.0515 ±0.0729	0.0362 ±0.0514	**0.0466 ±0.0737**	0.7234 ±0.3540	*0.8208 ±0.3435*
Interestingness	0.0254 ±0.0133	**0.0279 ±0.0114**	0.0227 ±0.0082	0.0002 ± 0.0005	*0.0002 ±0.0003*
Enemy Sparsity	42.0036 ±17.512	**34.6699 ±7.7747**	32.3738 ±10.4389	0.25 ± 0.25	*0.0 ±0.0*
Playability	86.4864	54.0	40.0	**100.0***	0.0*

Table 5.1: Comparative study of SMB generators based on PCGML tile metrics. **Bold** indicates mean values nearest and *Italic* indicates values farthest from the original mean dataset values. Asterisks indicate theoretical values.

5.3.1 Level Generation for SMB

The training corpus for this experiment consists of the 37 levels from Super Mario Bros. and Super Mario Bros. 2 (Japan) from the VGLC Corpus [61]. We analyze the performance of our two-step level generator for SMB level generation and compare it against the results of LSTMs trained directly on the original tile embeddings, CTE, and the VGLC representation [24], [60]. For all the level generation models the history sequence is maintained at 200 tiles and the network consists of three layers each comprised of 512 LSTM cells. We partition the data as 80-10-10% train, test and validation split. LSTMs trained directly on the original tile embeddings and CTE output the continuous embedding vector of the next tile whereas the LSTMs trained on discrete CTE (two-step level generation) and the VGLC output a distribution over tiles with softmax activation at the final layer. This makes sampling possible. The new levels are sampled tile by tile by generating rows left to right then bottom to top.

Ideal output levels would match the style of existing SMB levels. [49] suggested several metrics to assess the style of generated content in comparison to the dataset.

- **Leniency** captures the difficulty of the level. Values closer to one indicate more lenient levels [49]. We compute leniency as,

$$leniency = \frac{2r - (0.5 * g) - e}{T} \tag{5.2}$$

  where r, g, and e represent the counts for rewards, gaps, and enemies respectively, and T is the total number of tiles in a level $(l \times w)$.

- **Linearity** measures how well a level fits to a line. It is calculated as the mean squared error between the centre points of each platform and its projection on the linear regression line [49].

- **Interestingness** is an important metric especially for evaluating generators for skewed tile distributions because the most probabilistic tile is unlikely to be interesting. It measures the fraction of tiles that bring visual variety to the level [58].

- **Density** is the proportion of solid tiles in the level. Density is a relevant here, as we observe that it is possible for SMB generators to produce only empty tiles because of their high probability [58].

- **Enemy Sparsity** measures the horizontal spread of the enemies across the level [58]. Because SMB levels include lines of enemies, it is possible for a generator to get stuck generating a continuous string of enemies. We calculate enemy sparsity as:

$$EnemySparsity = \frac{\sum_{e \in E} |x(e) - \overline{x}|}{|E|} \tag{5.3}$$

  where x is the x-position of an enemy, $\overline{x}$ the average x-position of enemies, and $|E|$ the total number of enemies.

- **Playability** measures the percentage of playable levels generated. We run an A* agent provided in the VGLC to check for the existence of path in a level [61].

As illustrated in Figure 5.1, we observe a notable improvement in the quality of levels generated by our proposed two-step level generator with CTE in comparison to the LSTM on original tile embeddings and LSTM on CTE. Compared to the good examples of level generation, the bad ones are fairly empty and consist of unreachable sections because of large platform gaps or height (Figure 5.1(e)). Meanwhile, the good examples show the presence of more interesting tiles and have a coherent structure better matching the style of the dataset (Figure 5.1(d)). But these are only two examples.

Table 5.1 shows the results of the metrics-based evaluation between 50 output levels generated by our two-step level generator, LSTM on VGLC representation, LSTM on original tile embeddings, LSTM on CTE and the original SMB dataset. Level generation using discrete representations (VGLC and discrete-CTE) consistently outperforms level generation using continuous representations (original tile embedding and CTE) across all tile metrics. The distribution of levels generated by the LSTM trained on the original tile embeddings and the LSTM trained on CTE is nowhere close to the distribution of the original dataset. This is also evidenced in Figure 5.1 (b) and (c). We take this to indicate that the two-step generation process allows CTE to compete with the hand-authored VGLC representation.

These results reinforces the importance of a discrete representation and sampling in level generation for levels with skewed tile distributions. Further, our two-step level generator's levels more closely resemble the training distribution compared to the VGLC generator levels in Leniency, Density and Linearity. In a similar vein, although the VGLC generator outperforms our approach for the other two metrics, we find the performance comparable. The playability results are an oddity, since there are levels the provided A* agent cannot complete in the original dataset but the LSTM on CTE levels (because they only include flat ground) can always be completed. On the other hand, the VGLC and discrete-CTE level playability values are comparatively close. We find these results valuable as unlike discrete-CTE, the VGLC benefits from being human-authored.

Approximating the actual distribution of game levels accurately is difficult given the limited size of the test split. Therefore the Dataset column summarizes metrics across the entire dataset. To provide evidence that the model is not overfit, we report the minimum tile edit distance between cluster representations of generated levels and the dataset levels of games in the Appendix.

5.3.2 Level Generation for Unseen Games

We train our two-step CTE level generator to generate levels for two unseen games: Bugs Bunny Crazy Castle (BBCC) and Genghis Khan. We downloaded 20 levels of BBCC and 41 of Genghis Khan as our training corpus[3]. We chose these particular games because of their contrasting degree of structure variance, with BBCC being comparatively higher. For both games the affordance information is missing. In such cases, the clustering relies on visual and edge data.

BBCC is an action-puzzle Nintendo Entertainment System (NES) game where the player moves Bugs through rooms collecting carrots. It has a set of representative tiles consisting of solid brick patterned background ; boxing gloves , invincibility potions , safes , crates , and ten thousand-pound weights that can be used against the enemies in the game; and solid tiles , , , on which bugs can stand. Genghis Khan is a turn-based strategy game. Its tiles exhibit comparative similarity in structure as well as colour. Thus, generating levels for both games allows us to study the impact of structural variance in our learned representation.

We train a two-step level generator on both games by employing a similar process as for SMB. The only difference is that we pass a zero vector for the affordances when extracting the CTE representation. We found the optimal number of clusters for the two-step generator was 8 and 24 for BBCC

[3] `https://vgmaps.com/`

and Genghis Khan respectively. Examples of output levels are given in the Appendix.

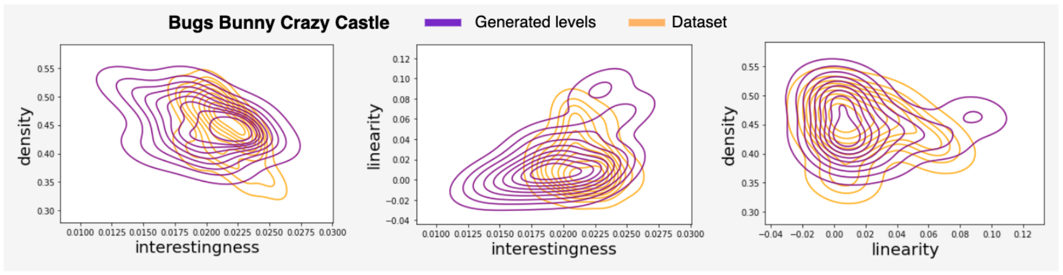


Figure 5.3: Expressive range analysis for the unseen game: Bugs Bunny Crazy Castle.

To evaluate the performance of the generator, we employed expressive range analysis on the generated content in comparison to their respective datasets [49]. Due to the limited size of test split, we use the entire dataset to estimate the true distribution. We perform expressive range analysis on Interestingness, Linearity and Density for BBCC. For Genghis Khan, we approximate only Interestingness and Leniency. We do not calculate Linearity and Density since it is not a platformer game. The BBCC metrics remain unchanged, as the game is sufficiently similar to SMB. Interestingness in BBCC is calculated as the frequency of tiles representing the items. Similarly, interestingness in Genghis Khan is measured as the proportion of tiles that bring visual variety to levels such as mountains, forests, towns, and castles. Further, we use the movement cost associated with each tile to calculate the Leniency. We assign negative costs to tiles that are difficult to move across such as -5 for mountains and castles, -6 for deserts, and -8 for rivers. In comparison, it costs less to move across regular land, forests and towns thus we assign movement costs of 3, 3, and 4 respectively. We calculate Leniency by summing these movement costs normalized by the total number of tiles in a level.

Figure 5.3 and Figure 5.4 shows the expressive range analysis performed on the generated levels of unannotated games in comparison to the entire original datasets. As seen in Figure 5.3, for BBCC, our model covers a considerable amount of the generative space, with a large amount of overlap with the original levels. However, we found lower output interestingness than the true distribution. For BBCC, the density of levels increases as linearity decreases, this is due to the fact that as more platforms are generated vertically, levels become denser due to the presence of platforms and stair tiles.

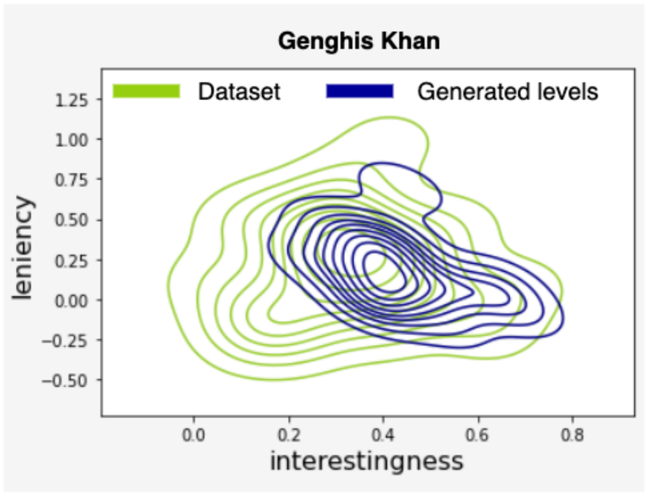


Figure 5.4: Expressive range analysis for the unseen game: Genghis Khan.

In comparison to BBCC, the Interesting-Leniency expressivity analysis for Genghis Khan (Figure 5.4) does not match the training distribution as closely, though there is still significant overlap. We find that the generated outputs have more challenge, more difficult terrain, compared to the training dataset. Although these results can be improved, we find them promising, indicating the capability of the generator to design levels for games based only on image data.

Chapter 6

Conclusion

Procedural Level Generation via Machine Learning (PLGML) involves training machine learning models on existing data to generate novel game levels. While different level generation techniques have been studied in this domain, data scarcity and lack of a consistent data representation across different games remain core challenges in the PLGML community.

This thesis presented an initial approach to learn an initial set of tile embeddings and a more specialized representation, clustering-based tile embeddings (CTE). We initially employed a two-branch autoencoder network that compresses a tile's visual, behavioural and contextual information into a 256-dimensional embedding vector. Our original tile embedding representation demonstrated its ability to reasonably approximate unseen tiles' behaviour and to serve as a representation for the level generation of annotated and unannotated games. However, we found that our initial tile embedding representation performed poorly at generating levels for games with skewed tile distributions, such as Super Mario Bros. (SMB). A majority of a SMB level is occupied by empty tiles. The continuous nature of tile embeddings does not allow a tile embedding-based level generator to approximate a probability distribution over the next possible tiles. Therefore, a tile embedding-based level generator adopts a greedy sampling strategy, generating tiles with the highest probability of occurrence. This led to the generation of empty levels for Super Mario Bros..

To address this issue, we employed clustering to learn a discrete counterpart of the continuous tile embedding representation. In cases where the affordance information is missing, we observed a lack of clear separation between types of tiles. The clustering decisions were predominantly based on the RGB representation of a tile. Therefore, tiles of similar colour were clustered together. To discourage this, we made two modifications to our original approach. We presented a modified tile embedding representation by incorporating edge feature information and introduced a cluster-based loss to our autoencoder training. We refer to the new 256-dimensional embedding vectors as clustering-based tile embeddings (CTE), and their discrete counterpart as our discrete CTE representation. We then presented a novel two-step level generation pipeline that can leverage the benefits of our discrete representation for sampling new levels and can take advantage of the expressivity of persistent tile embeddings.

6.1 Limitations

The CTE representation and our two step level generation pipeline demonstrated improved performance in generating levels for Super Mario Bros., and the ability to generate levels for unannotated games. Notably, our approach also shows potential in generating levels of non-platformer games such as Genghis Khan, a turn-based strategy game. However, we can still improve our pipeline further, especially for games with structurally similar tiles and missing affordances. We employ Silhouette Score and the Structural Similarity Index to evaluate the performance of the Clustering and Translator modules in the two-step level generation pipeline.

	Clustering		Translation
	Number of Clusters	Silhouette Score	Structural Similarity Index
SMB	11	0.91	0.9976 ± 0.0014
Genghis Khan	8	0.39	0.8689 ± 0.01
BBCC	24	0.53	0.9792 ± 0.0079

Table 6.1: Evaluating clustering and translation modules of our two-step level generation pipeline.

Table 6.1 shows the results of these evaluations. We find that clustering is a crucial component of our level generation pipeline and the performance of the translation model improves with the quality of the clustering. For Genghis Khan, the missing affordances and lack of structural variability between representative tiles yielded a low silhouette score. A low silhouette score is an indication of arbitrary clustering. If the cluster participants have no particular structure, the translation model struggles to map cluster numbers to embeddings and hence does not converge well.

6.2 Future Work

Our tile embedding representation can support level generation tasks for both annotated and unannotated games, greatly expanding the set of possible domains where we can apply PLGML. We propose the following avenues for future research to learn stronger embedding representations: (1) Variational Autoencoders (VAE) are an advancement over regular autoencoders that approximate a distribution for each latent variable and thus can effectively be used as a representation learning model. A recent trend has been towards optimizing the latent space of a VAE for clustering, commonly referred to as Deep Clustering [31]. In a similar vein, a Gaussian Mixture Variational Autoencoder (GMVAE) is a type of VAE imposing a mixture of Gaussians as a prior on the latent space [10]. While in the presented work, we approximate the mixture of Gaussians before training an autoencoder, Deep Clustering or a GMVAE could be used to learn a more robust representation [68]. (2) Discrete representations have benefits over continuous representation for several PCGML tasks. A Vector Quantized VAE is a variant of the VAE that quantizes the latent space to learn a discrete latent representation [41]. Leveraging a VQ-VAE could potentially simplify our current level generation pipeline. Such an implementation opens the possibility to having a common discrete representation across multiple games and thus learning a generalized level generator. Before applying a VQ-VAE to learning tile embeddings, we would need to address a number of caveats. In our current work, we reflect on the idea that continuous and discrete representations are both needed for level generation. While discrete representations are a natural fit for many applications, learning only discrete representations can limit the expressivity of the generator. Further, these representations cannot be applied directly in tasks based on interpolating between points in a learned latent space. For example, in generating novel tiles. This might be relevant in another future application of CTE: the PLGML task of level blending [42]. (3) In our current implementation, we use a weighted binary cross-entropy loss on reconstructed affordances to tackle dataset imbalance. While this provided a significant improvement, we need to generalize better over the distribution of labels with fewer instances. To address this challenge, future research could investigate data augmentation and employ data sampling techniques. We also suggest expanding the affordances as a set of 13 labels is fairly limiting for the model to be able to express any 2D tile-based game level. If we expand the affordances, we can include additional games of different styles and genres, which will enrich the training corpus.

6.3 Takeaways

The PLGML community has made great strides in designing level generation and blending techniques. However, most of the work is limited to academia. A primary reason for this is the limited availability of high-quality representations of game levels. As a result, a significant percentage of PLGML studies draw on a select few games for which clean annotations are available. Through our presented work, we aim to push the boundaries of PLGML research by introducing tile embeddings, a general representation of game levels. Tile embeddings have the potential to represent game levels across various domains without human effort, allowing us to build much more generalisable PLGML models. Therefore, we encourage PLGML researchers to play a part in making them stronger. We hope the PLGML community benefits from our contribution and look forward to seeing their application in future research.

References

[1] G. Alain and Y. Bengio, "What regularized auto-encoders learn from the data-generating distribution," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 3563–3593, 2014.

[2] S. Alvernaz and J. Togelius, "Autoencoder-augmented neuroevolution for visual doom playing," in *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, IEEE, 2017, pp. 1–8.

[3] S. Beaupre, T. Wiles, S. Briggs, and G. Smith, "A design pattern approach for multi-game level generation," in *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 14, 2018.

[4] G. R. Bentley and J. C. Osborn, "The videogame affordances corpus," in *2019 Experimental AI in Games Workshop*, 2019.

[5] M. R. Boutell, J. Luo, X. Shen, and C. M. Brown, "Learning multi-label scene classification," *Pattern recognition*, vol. 37, no. 9, pp. 1757–1771, 2004.

[6] J. Canny, "A computational approach to edge detection," *IEEE Transactions on pattern analysis and machine intelligence*, no. 6, pp. 679–698, 1986.

[7] Y. Chen and M. J. Zaki, "Kate: K-competitive autoencoder for text," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2017, pp. 85–94.

[8] K. Cho, B. Van Merriënboer, D. Bahdanau, and Y. Bengio, "On the properties of neural machine translation: Encoder-decoder approaches," *arXiv preprint arXiv:1409.1259*, 2014.

[9] S. Clinchant and F. Perronnin, "Aggregating continuous word embeddings for information retrieval," in *Proceedings of the workshop on continuous vector space models and their compositionality*, 2013, pp. 100–109.

[10] N. Dilokthanakul, P. A. Mediano, M. Garnelo, *et al.*, "Deep unsupervised clustering with gaussian mixture variational autoencoders," *arXiv preprint arXiv:1611.02648*, 2016.

[11] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, ser. KDD'96, Portland, Oregon: AAAI Press, 1996, pp. 226–231.

[12] G. Finley, S. Farmer, and S. Pakhomov, "What analogies reveal about word vectors and their compositionality," in *Proceedings of the 6th joint conference on lexical and computational semantics (* SEM 2017)*, 2017, pp. 1–11.

[13] K. Ghasedi Dizaji, A. Herandi, C. Deng, W. Cai, and H. Huang, "Deep clustering via joint convolutional autoencoder embedding and relative entropy minimization," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 5736–5745.

[14] E. Giacomello, P. L. Lanzi, and D. Loiacono, "Doom level generation using generative adversarial networks," in *Proceedings of the 2018 IEEE Games, Entertainment, Media Conference (GEM)*, IEEE, 2018, pp. 316–323.

[15] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[16] M. Guzdial, J. Reno, J. Chen, G. Smith, and M. Riedl, "Explainable pcgml via game design patterns," *arXiv preprint arXiv:1809.09419*, 2018.

[17] M. Guzdial and M. Riedl, "Game level generation from gameplay videos," in *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 12, 2016.

[18] D. Ha and J. Schmidhuber, "World models," *arXiv preprint arXiv:1803.10122*, 2018.

[19] D. M. Hawkins, "The problem of overfitting," *Journal of chemical information and computer sciences*, vol. 44, no. 1, pp. 1–12, 2004.

[20] J. R. Hershey, Z. Chen, J. Le Roux, and S. Watanabe, "Deep clustering: Discriminative embeddings for segmentation and separation," in *Proceedings of 2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2016, pp. 31–35. DOI: `10.1109/ICASSP.2016.7471631`.

[21] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *science*, vol. 313, no. 5786, pp. 504–507, 2006.

[22] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[23] S. Ioffe and C. Szegedy, *Batch normalization: Accelerating deep network training by reducing internal covariate shift*, 2015. arXiv: `1502.03167 [cs.LG]`.

[24] M. Jadhav and M. Guzdial, "Tile embedding: A general representation for level generation," in *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 17, 2021, pp. 34–41.

[25] R. Jain, A. Isaksen, C. Holmgård, and J. Togelius, "Autoencoders for level generation, repair, and recognition," in *Proceedings of the ICCC Workshop on Computational Creativity and Games*, 2016, p. 9.

[26] I. Karth, B. Aytemiz, R. Mawhorter, and A. M. Smith, "Neurosymbolic map generation with vq-vae and wfc," in *The 16th International Conference on the Foundations of Digital Games (FDG) 2021*, 2021, pp. 1–6.

[27] H. Keller, H. Möllering, T. Schneider, and H. Yalame, "Balancing quality and efficiency in private clustering with affinity propagation," in *Proceedings of the 18th International Conference on Security and Cryptography, SECRYPT 2021, July 6-8, 2021*, S. D. C. di Vimercati and P. Samarati, Eds., SCITEPRESS, 2021, pp. 173–184. DOI: `10.5220/0010547801730184`. [Online]. Available: `https://doi.org/10.5220/0010547801730184`.

[28] N. Y. Khameneh and M. Guzdial, "Entity embedding as game representation," *arXiv preprint arXiv:2010.01685*, 2020.

[29] S. W. Kim, Y. Zhou, J. Philion, A. Torralba, and S. Fidler, "Learning to simulate dynamic environments with gamegan," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 1231–1240.

[30] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," *arXiv preprint arXiv:1312.6114*, 2013.

[31] K.-L. Lim, X. Jiang, and C. Yi, "Deep clustering with variational autoencoder," *IEEE Signal Processing Letters*, vol. 27, pp. 231–235, 2020.

[32] G. Liu, O. Schulte, P. Poupart, M. Rudd, and M. Javan, "Learning agent representations for ice hockey," in *Proceedings of the Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., vol. 33, Curran Associates, Inc., 2020, pp. 18 704–18 715. [Online]. Available: `https://proceedings.neurips.cc/paper/2020/file/d90e5b6628b4291225cba0bdc643c295-Paper.pdf`.

[33] J. Mariño, W. Reis, and L. Lelis, "An empirical evaluation of evaluation metrics of procedurally generated mario levels," in *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 11, 2015.

[34] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[35] M. Nissim, R. van Noord, and R. van der Goot, "Fair is better than sensational: Man is to doctor as woman is to doctor," *Computational Linguistics*, vol. 46, no. 2, pp. 487–497, 2020.

[36] K. Park, B. W. Mott, W. Min, K. E. Boyer, E. N. Wiebe, and J. C. Lester, "Generating educational game levels with multistep deep convolutional generative adversarial networks," in *2019 IEEE Conference on Games (CoG)*, IEEE, 2019, pp. 1–8.

[37] L. Prechelt, "Early stopping-but when?" In *Neural Networks: Tricks of the trade*, Springer, 1998, pp. 55–69.

[38] M. Ranzato, Y.-L. Boureau, Y. Cun, *et al.*, "Sparse feature learning for deep belief networks," *Advances in neural information processing systems*, vol. 20, 2007.

[39] D. A. Reynolds, "Gaussian mixture models.," *Encyclopedia of biometrics*, vol. 741, no. 659-663, 2009.

[40] P. J. Rousseeuw, "Silhouettes: A graphical aid to the interpretation and validation of cluster analysis," *Journal of computational and applied mathematics*, vol. 20, pp. 53–65, 1987.

[41] A. Saravanan and M. Guzdial, "Pixel vq-vaes for improved pixel art representation," *CoRR*, vol. abs/2203.12130, 2022. DOI: `10.48550/arXiv.2203.12130`. arXiv: `2203.12130`. [Online]. Available: `https://doi.org/10.48550/arXiv.2203.12130`.

[42] A. Sarkar and S. Cooper, "Generating and blending game levels via quality-diversity in the latent space of a variational autoencoder," in *Proceedings of the 16th International Conference on the Foundations of Digital Games (FDG) 2021*, 2021, pp. 1–11.

[43] A. Sarkar, A. Summerville, S. Snodgrass, G. Bentley, and J. Osborn, "Exploring level blending across platformers via paths and affordances," in *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 16, 2020, pp. 280–286.

[44] A. Sarkar, Z. Yang, and S. Cooper, "Controllable level blending between games using variational autoencoders," *arXiv preprint arXiv:2002.11869*, 2020.

[45] G. Schwarz, "Estimating the Dimension of a Model," *The Annals of Statistics*, vol. 6, no. 2, pp. 461–464, 1978. DOI: `10.1214/aos/1176344136`. [Online]. Available: `https://doi.org/10.1214/aos/1176344136`.

[46] S. Selvin, R. Vinayakumar, E. Gopalakrishnan, V. K. Menon, and K. Soman, "Stock price prediction using lstm, rnn and cnn-sliding window model," in *2017 international conference on advances in computing, communications and informatics (icacci)*, IEEE, 2017, pp. 1643–1647.

[47] N. Simidjievski, C. Bodnar, I. Tariq, *et al.*, "Variational autoencoders for cancer data integration: Design principles and computational practice," *Frontiers in genetics*, vol. 10, p. 1205, 2019.

[48] G. Smith, M. Cha, and J. Whitehead, "A framework for analysis of 2d platformer levels," in *Proceedings of the 2008 ACM SIGGRAPH symposium on Video games*, 2008, pp. 75–80.

[49] G. Smith and J. Whitehead, "Analyzing the expressive range of a level generator," in *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, ser. PCGames '10, Monterey, California: Association for Computing Machinery, 2010, ISBN: 9781450300230. DOI: 10.1145/1814256.1814260. [Online]. Available: https://doi.org/10.1145/1814256.1814260.

[50] G. Smith, J. Whitehead, M. Mateas, M. Treanor, J. March, and M. Cha, "Launchpad: A rhythm-based level generator for 2-d platformers," *IEEE Transactions on computational intelligence and AI in games*, vol. 3, no. 1, pp. 1–16, 2010.

[51] S. Snodgrass, "Towards automatic extraction of tile types from level images," in *Joint Proceedings of the AIIDE 2018 Workshops co-located with 14th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE 2018), Edmonton, Canada, November 13-14, 2018*, J. Zhu, Ed., ser. CEUR Workshop Proceedings, vol. 2282, CEUR-WS.org, 2018. [Online]. Available: http://ceur-ws.org/Vol-2282/EXAG%5C_119.pdf.

[52] S. Snodgrass and S. Ontanón, "Generating maps using markov chains," in *Ninth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2013.

[53] ——, "Procedural level generation using multi-layer level representations with mdmcs," in *2017 IEEE conference on computational intelligence and games (CIG)*, IEEE, 2017, pp. 280–287.

[54] K. Sorochan, J. Chen, Y. Yu, and M. Guzdial, "Generating lode runner levels by learning player paths with lstms," in *The 16th International Conference on the Foundations of Digital Games (FDG) 2021*, 2021, pp. 1–7.

[55] M. S. Sorower, "A literature survey on algorithms for multi-label learning," *Oregon State University, Corvallis*, vol. 18, pp. 1–25, 2010.

[56] A. Summerville, "Expanding expressive range: Evaluation methodologies for procedural content generation," in *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 14, 2018.

[57] A. Summerville, M. Behrooz, M. Mateas, and A. Jhala, "What does that?-block do? learning latent causal affordances from mario play traces," in *Workshops at the Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[58] A. Summerville, J. R. H. Mariño, S. Snodgrass, S. Ontañón, and L. H. S. Lelis, "Understanding mario: An evaluation of design metrics for platformers," in *Proceedings of the 12th International Conference on the Foundations of Digital Games*, ser. FDG '17, Hyannis, Massachusetts: Association for Computing Machinery, 2017, ISBN: 9781450353199. DOI: 10.1145/3102071.3102080. [Online]. Available: https://doi.org/10.1145/3102071.3102080.

[59] A. Summerville, S. Snodgrass, M. Guzdial, *et al.*, "Procedural content generation via machine learning (pcgml)," *IEEE Transactions on Games*, vol. 10, no. 3, pp. 257–270, 2018.

[60] A. J. Summerville and M. Mateas, "Super mario as a string: Platformer level generation via lstms," in *Proceedings of the First Joint International Conference of Digital Games Research Association and Foundation of Digital Games, DiGRA/FDG 2016, Dundee, Scotland, UK, August 1-6, 2016*, S. Björk, C. O'Donnell, and R. Bidarra, Eds., Digital Games Research Association/Society for the Advancement of the Science of Digital Games, 2016. [Online]. Available: http://www.digra.org/digital-library/publications/super-mario-as-a-string-platformer-level-generation-via-lstms/.

[61] A. J. Summerville, S. Snodgrass, M. Mateas, and S. Ontanón, "The vglc: The video game level corpus," in *Proceedings of the 7th Workshop on Procedural Content Generation*, 2016.

[62] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," *Advances in neural information processing systems*, vol. 27, 2014.

[63] S. Thakkar, C. Cao, L. Wang, T. J. Choi, and J. Togelius, "Autoencoder and evolutionary algorithm for level generation in lode runner," in *2019 IEEE Conference on Games (CoG)*, IEEE, 2019, pp. 1–4.

[64] L. Van der Maaten and G. Hinton, "Visualizing data using t-sne.," *Journal of machine learning research*, vol. 9, no. 11, 2008.

[65] P. Vincent, H. Larochelle, Y. Bengio, and P.-A. Manzagol, "Extracting and composing robust features with denoising autoencoders," in *Proceedings of the 25th international conference on Machine learning*, 2008, pp. 1096–1103.

[66] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan, "Show and tell: Lessons learned from the 2015 mscoco image captioning challenge," *IEEE transactions on pattern analysis and machine intelligence*, vol. 39, no. 4, pp. 652–663, 2016.

[67] X. Yang, D. Lo, X. Xia, L. Bao, and J. Sun, "Combining word embedding with information retrieval to recommend similar bug reports," in *2016 IEEE 27Th international symposium on software reliability engineering (ISSRE)*, IEEE, 2016, pp. 127–137.

[68] Z. Yang, A. Sarkar, and S. Cooper, "Game level clustering and generation using gaussian mixture vaes," in *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 16, 2020, pp. 137–143.

Appendix A

Additional Outputs and Discussions

Game	Train	Test
SMB	768.92 ± 140.98	743.06 ± 101.67
Genghis Khan	39.32 ± 3.93	40.98 ± 5.18
BBCC	192.39 ± 22.30	201.84 ± 19.24

Table A.1: The edit distances observed between the generated cluster representations and the training and test data suggests that the model is not overfitting.

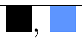
Super Mario Bros			Lode Runner		
Tile	Example Tile Sprite	Median	Tile	Example Tile Sprite	Median
-		88.33%	.		58.09%
E		7.26%	E		21.59%
S		0.99%	G		8.52%
X		0.55%	b		4.11%
<		0.51%	#		3.26%

Table A.2: Median percentages of top five tiles occurring in a level. This table illustrates skewed tile distribution in Super Mario Bros and comparatively balanced tile distribution in lode runner tiles.

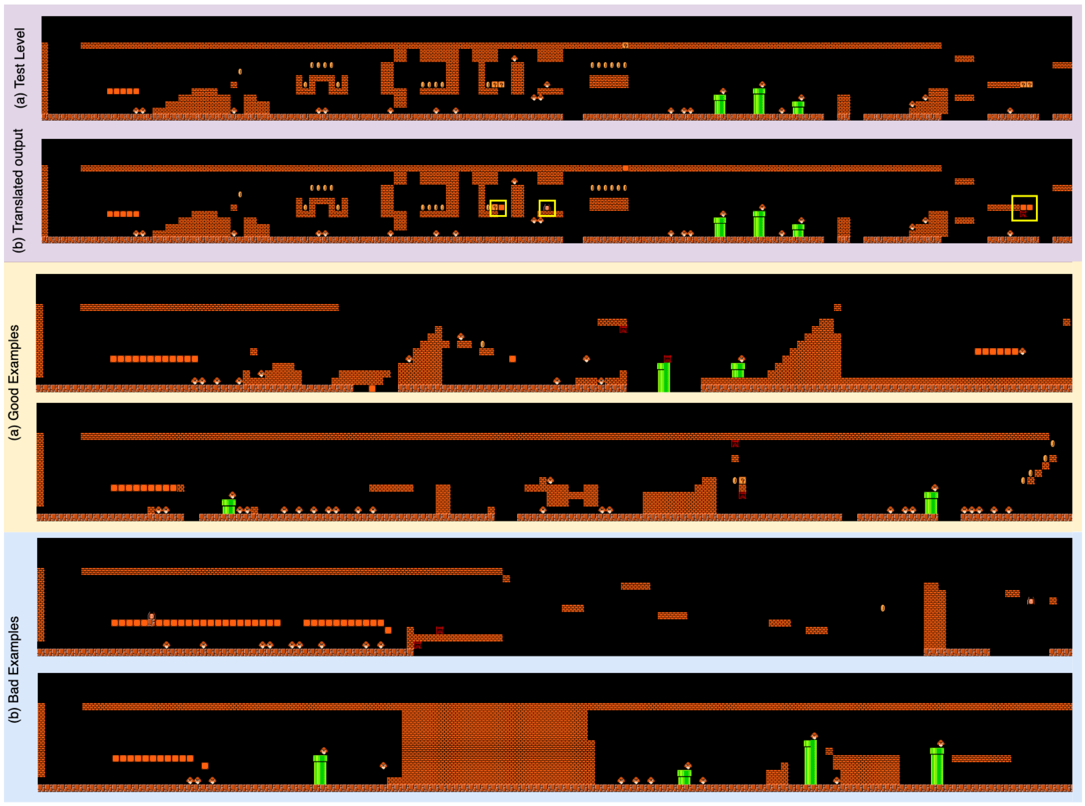


Figure A.1: Figure (a) shows a test SMB dataset level and Figure (b) shows its translated output obtained using the second step of our two-step generator. The differences between the two are highlighted in yellow. To get this translated version we convert the dataset levels of a game to: 1) their cluster representation using the DBSCAN and 2) their CTE representation using our newly trained autoencoder. We use these cluster representation and their correspoding CTE representation of dataset levels to train the translation model as discussed in the two-step level generation process. Figure (c) and (d) show more examples of SMB level generation output with the two-step level generator trained on our CTE representation.

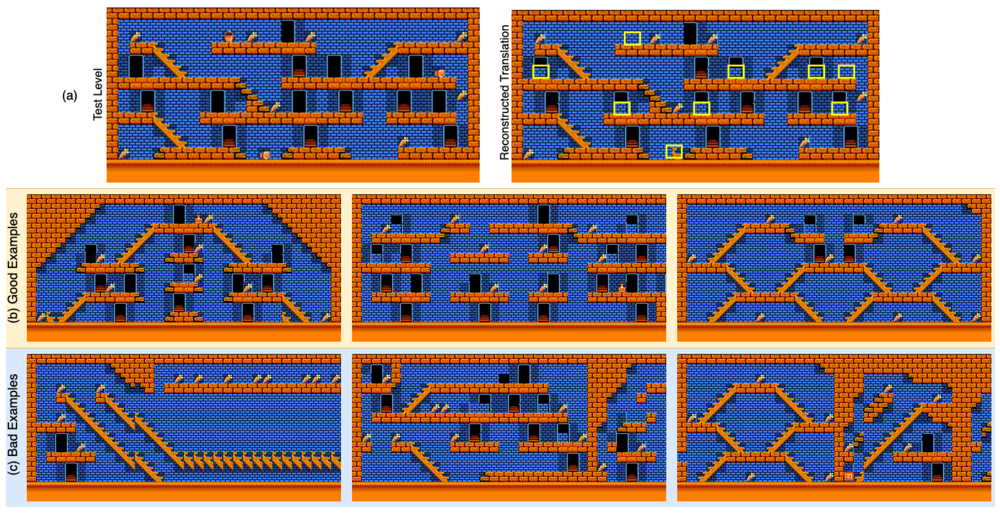


Figure A.2: Level Generation for Bugs Bunny Crazy Castle : (a) Test dataset level (left) and its corresponding translated output (right) with differences highlighted in yellow (b) Examples of good generation output (c) Examples of bad generation output. Unlike good examples as in shown in (b), bad examples in (c) show the presence of unreachable level sections due to the lack of portals/doors, and inconsistency in level structure.

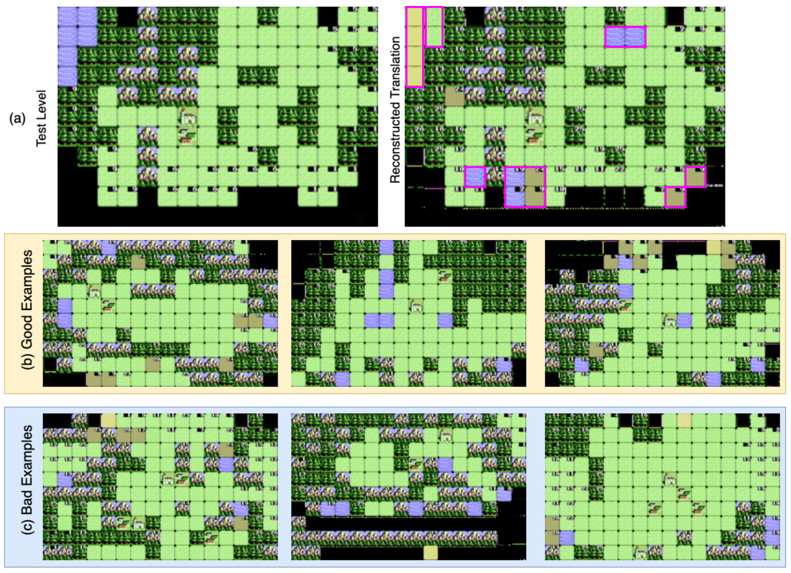


Figure A.3: Level Generation for Genghis Khan : (a) Test dataset level (left) and its corresponding translated output (right) with differences highlighted in red (b) Example of good generation output(c) Example of bad generation output. The dataset levels of Genghis Khan only have one pair of town and castle tiles each whereas examples of bad generation (c), have multiple pairs. The bad levels also contain randomly placed mountain and forest tiles, instead of the clustered appearance found in (b) and in the original dataset.