Semantic Hashing for Video Game Levels

Aaron Isaksen, New York University

Christoffer Holmgård, New York University

Julian Togelius, New York University

We use semantic hashing, an unsupervised machine learning technique based on a type of deep neural network called autoencoders, to categorise video game levels. We show how this technique can be used on dungeon room maps from The Legend of Zelda and segments of 2D-platformer levels from Super Mario Bros. We also discuss how this technique can be useful for game designers, AI-assisted game design, and procedural content generation.

1 Introduction

SEMANTIC hashing is an automated, non-linear method for finding similarities between instances of content using pretrained neural networks. The technique was originally developed for determining document similarity for searching through text documents to find related examples [1]. In this paper, we show how a variant of the semantic hashing technique [2] based on autoencoders [3, 4] can be used to find similarities in video game content.

Semantic hashing uses unsupervised learning to perform clustering of similar content. *Unsupervised* means the data does not need to be labelled by a game designer or player before processing. Instead, the system can take in level data – segmented into regions such as rooms in a dungeon or slices of a linear level – and determine which regions are similar given no additional knowledge of what the level data means.

Once a semantic hashing neural network is trained, it is simple to input new level data and have the network return the category which the new data is closest to. Thus, the network also allows us to classify unseen future data using the same structure.

We demonstrate the use of semantic hashing on two types of classic video game levels: top-down dungeon RPG maps from The Legend of Zelda [5] and side-scrolling platformer levels from Super Mario Bros. [6]. Level data is obtained from the Video Game Level Corpus [7], which takes level maps and preprocesses them to load into machine learning algorithms.

1.1 Application to Game Design

While this paper demonstrates a proof of concept to show that semantic hashing can effectively classify and cluster new game content automatically and quickly, we believe this method has a number of potentially useful applications in game design, procedural content generation, and user-generated content.

To begin, the statistical analysis of game content can help a designer understand which game content is common, unique, and under/over-represented in their design. For example, a designer may want to know how much variety and repetition there is in the game. Similarly, multiple designers working on the same game might wish to compare their content to see how much of it is similar. In a mixed-initiative system, an AI agent may offer suggestions to a human designer based on the type of work they are currently creating – to enable this, the machine must be able to identify what makes content similar or unique.

Procedural content generation allows a machine to generate new game assets [8], and a large category of such methods use machine learning to produce new content from existing game data [9]. This enables new methods in autonomous generation, co-creation, mixed-initiative design, and repair of content for games. With automated content classification, one can control the output of a procedural content generation system; e.g. if we wanted to generate levels with many bridges, we could use the classifier to check for this. Or, more generally, we could use an autoencoder to ensure that machine generated content has a certain typological distribution, ensuring variations or patterns in the output.

In games where players contribute content, semantic hashing could inform the game what the players are creating by assigning the new content to the most likely category already in the game. The autoencoder could learn these categories automatically from the existing game content, and can be retrained as more is created.

2 Semantic Hashing

Semantic hashing is based on autoencoders, which are a type of neural network. Neural networks are trainable non-linear mathematic models [10]; most neural networks take an input vector (Figure 1a) and generate an output vector (Figure 1g). By providing input data and matching output data, the back-propagation algorithm can learn the parameters of the neural network to create a non-linear function [11]. Neural networks are typically structured in layers, where each layer is a simple linear function with an input and output, and the layers are linked together to compose a deep neural network implementing a more complex non-linear function. Each layer can be of a different size, and the input and output vectors can have different sizes as well.

We show an autoencoder neural network in Figure. 1. An autoencoder neural network is trained to reproduce as output a vector identical to the input. The input data is fed into the first layer of the neural network (Figure 1b) which is fully connected to successive layers of smaller and smaller hidden layers (Figure 1c). By *fully connected*, we mean that the entire output vector from layer i is given as input to layer $i+1$.

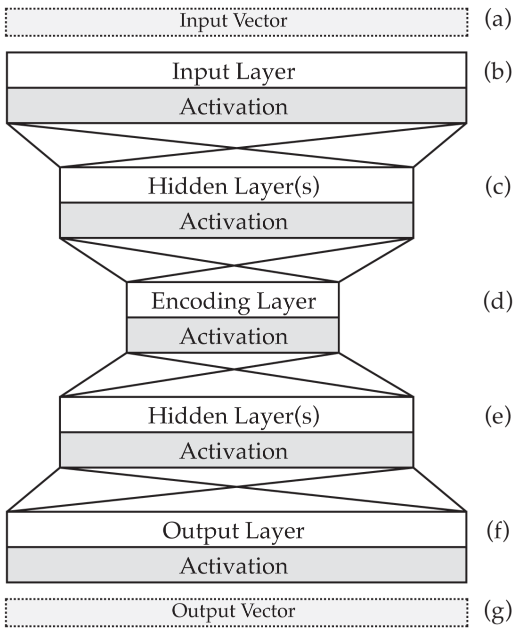


Figure 1. Neural network autoencoder structure.

In the middle of the network is a narrow choke-point, the binary encoding layer (Figure 1d). This narrow layer forces the network to represent the input in an abstract and compressed form, so that it can come up with a more general model of the data. From this point, the network expands again through more hidden layers (Figure 1e) to the full-sized output layer (Figure 1f).

The network tries to produce the same output as input, which is difficult because the data must pass through the lossy narrow encoding layer. If we keep the size of this layer small, and encourage the values that pass through this layer to be 0.0 or 1.0, we can treat this layer as a hash value that categorises the data. If the encoding layer is k bits wide, we can represent 2^k different categories. Because the training tries to reduce the difference between the input vector and output vector, the network will eventually produce a best-fit classification on the data it is trained on.

Between each layer, we also pass the data through what is called an *activation function*. It takes the output of each neuron and performs a simple non-linear transformation on it. We commonly use the "rectified linear unit" (ReLU) activation, which is simply $f(x) = \max(0, x)$, such that positive values are unmodified and negative values are clamped to 0. We also use a sigmoid function, which maps input values from $-\infty$ to ∞ to output values 0.0 to 1.0, in a smoothed step-like function. This allows us to model binary values in a way appropriate for neural networks.

By injecting Gaussian noise into the training in the output of the binary encoding layer, the network is forced to learn how to ignore noise and become less sensitive to small changes. This makes the data leaving the binary encoding layer either highly negative or highly positive. We then pass this through a sigmoid activation function, making the data either 0.0 or 1.0. This gives us the 0/1 bits we need for semantic hashing.

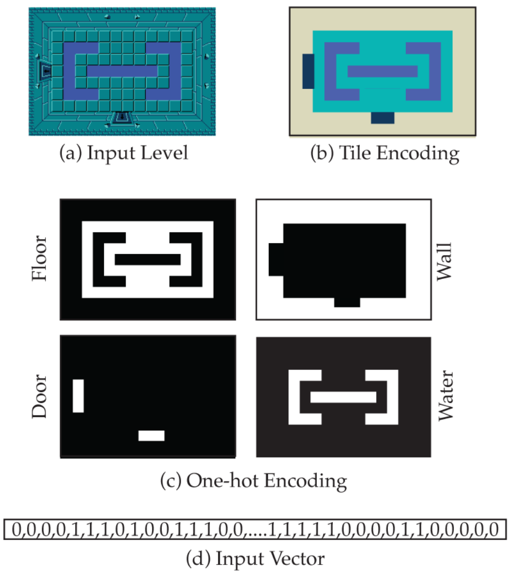


Figure 2. Level maps using one-hot encoding.

In order to represent different classes of tiles in the game maps, we use a *one-hot* encoding [12, p. 129], as shown in Figure 2. We start with an input level, such as the example in Figure 2a from The Legend of Zelda. Each tile is given a semantic tile value (e.g. floor, water, door, wall), with N different possible tile values on the $W \times H$ game map (Figure 2b). This encoding is split up into N binary planes each of size $W \times H$, where for each position x, y only one of the N planes has a 1 and all others have a 0 (Fig 2c). We then flatten these planes into a single vector of length $N \times W \times H$, with each element in the vector either a 0 or a 1 (Figure 2d).

One interesting and useful feature of semantic hashing is that the number of bit flips between two categories gives us a sense of distance between two categories. That is, the category representing 0000 is closer to 0001, 0010, 0100, and 1000 than it is to 1111. Thus not only do we automatically categorise and cluster the level maps, but we can also use the hash value to get a sense of which maps are closer to others.

3 Experiment and Results

In this section, we present our experimental results of using semantic hashing on dungeon maps from The Legend of Zelda and platformer levels from Super Mario Bros. Our networks are implemented in Keras[1] using the TensorFlow[2] backend running with 2×NVIDIA 1080 GPUs.

We use the same network topology for both games, as shown in Figure 3. The input and output layers are of size $n = N \times W \times H$, where N is the number of tile types, W is the width of the content, and H is the height of the content. These values are defined in the subsections for each game.

When training, we inject Gaussian noise of standard deviation 1.0 into the encoding layer and output layer, followed by a sigmoid activation that encourages output values of 0.0 and 1.0. While full details are beyond the scope of this paper, for completeness we use a binary-cross entropy to determine error between input and output, and RMSprop for gradient descent.

3.1 The Legend of Zelda

The Legend of Zelda is the first in a series of popular RPG video games for Nintendo game systems. The game contains two quests, each of which has nine multi-room dungeons. In total, there are 459 rooms, each of size 11×16, but only 264 of these are unique.

[1]https://keras.io/

[2]https://www.tensorflow.org/

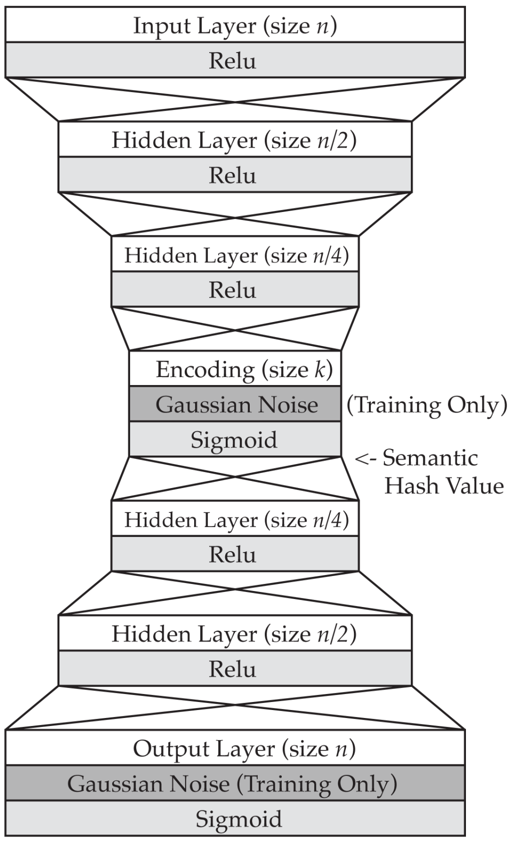


Figure 3. The training network.

We put each tile into one of six classes: WALL/BLOCK, MONSTER, LAVA/WATER, DOOR/STAIR, FLOOR, or VOID. Using one-hot encoding, this gives us an input vector of size $11 \times 16 \times 6 = 1,056$ for each room, where each element in the vector is either a 0 or 1.

Because we only have 264 samples, there is a danger of overfitting instead of generalising. We therefore keep the number of training iterations to 1,000 epochs, and use $k = 5$ for a maximum of 32 categories. We use a mini-batch of size 32, which means for each training epoch we only examine 32 randomly chosen rooms. This only takes a couple of minutes to train on our system.

In Figure 4, we show the result of allowing the semantic hashing algorithm with $k = 5$ hashing bits, giving up to $2^5 = 32$ categories. Each row is a different category, and five categories are unused, meaning the system found 27 unique categories (the autoencoder is not forced to use all possible values). The number above each map indicates how often that room exists in the Zelda dungeon corpus. Note that each category can have a different number of maps. Many of these categorisations are accurate and encode shape and content, not just colour of tiles.

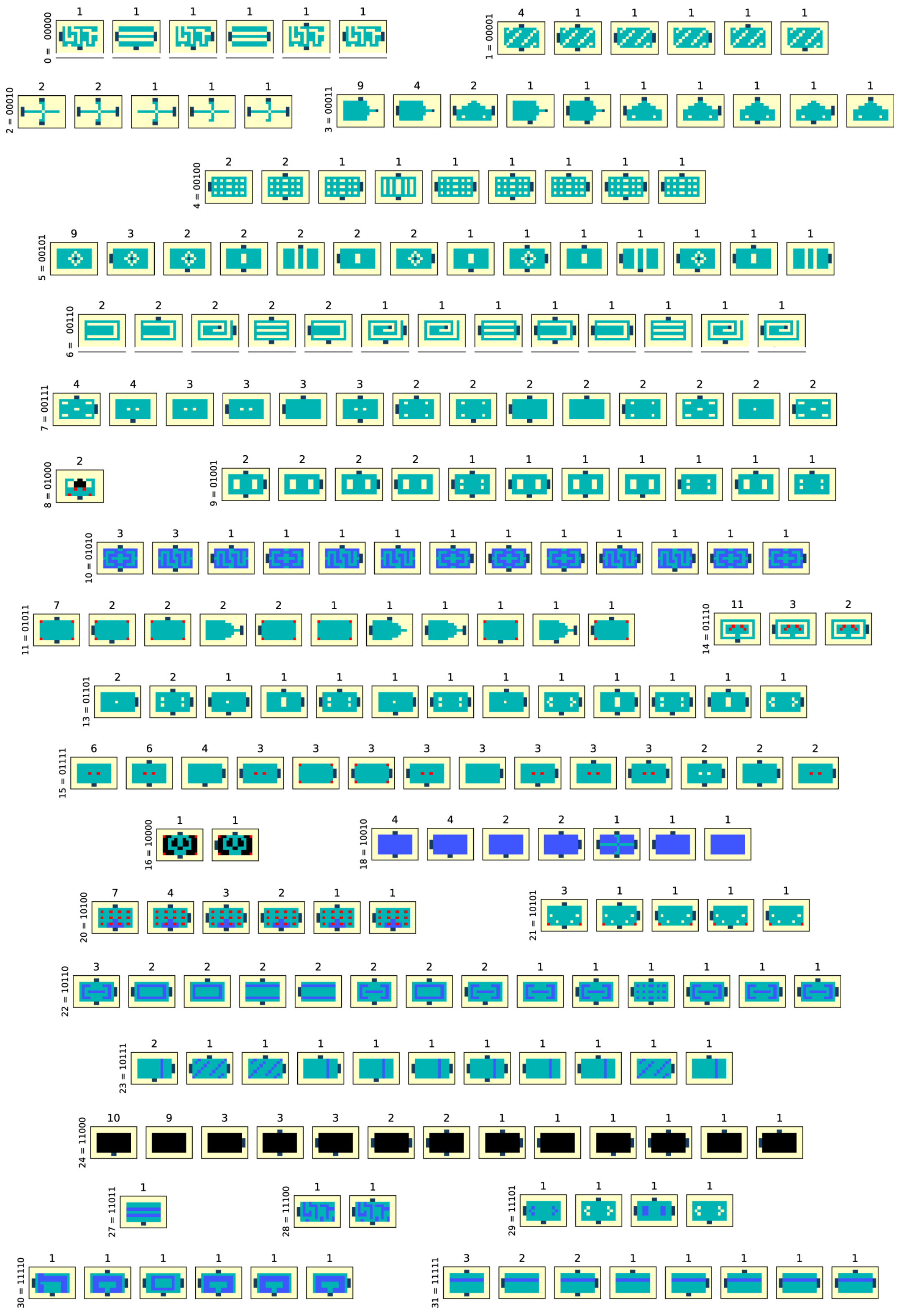


Figure 4. Semantic hashing of Zelda maps into $27 \leq 2^5$ categories.

3.2 Super Mario Bros

Super Mario Bros is the first in a long series of popular 2D side-scrolling platformers for Nintendo game systems. The VGLC level corpus for *Super Mario Bros.* contains 15 platforming levels of varying length, each 14 tiles high and between 149 and 373 wide (average of approx. 195 width). The corpus contains the above ground, below ground, and tree levels. Instead of processing the entire level at once, we split up each level into sliding 4-tile wide vertical windows. This gives us 2,923 windows of size 14×4, which we reduce to 1,317 windows by eliminating duplicate rows for training the autoencoder, so that common windows are not overrepresented. Each tile is put into one of four classes: BLOCK, MONSTER, COIN, or VOID. Therefore, the input vector is of size $14 \times 4 \times 4 = 224$ for each window. We keep the number of training iterations to 1,000 epochs, with a mini-batch size of 32, and $k = 3$ bits of encoding for eight possible categories. As with the Zelda example, this only takes a couple of minutes to train on our system.

The results of the semantic hashing are shown in Figure 5. Yellow represents coins, red represents monsters, black represents empty space, and grey represents blocks, bricks, or ground. Some categories stand out to us: Category 0 generally represents underground levels without tunnels to climb through; Category 1 contains high obstacles to climb; Category 2 represents regions where something is over the player's head (coins or ceiling); Category 3 represents drops; Category 4 includes level segments without a ground plane; Category 5 indicates small obstacles to hop over; Category 6 consists of gaps without monsters or coins; and Category 7 includes open spaces.

4 Conclusion

This paper has shown how machine learning and semantic hashing can be used to categorise game content without human knowledge. The focus of this paper has been to explain how semantic hashing works (with a minimum of mathematics) and to demonstrate the results of classification on two popular video games. While it can categorise individual pieces of game content, there are many additional aspects of level design that we do not focus on here, such as connectivity between rooms, dynamics, or narrative. The examples here are also all in the realm of two-dimensional levels, and we do not know how much additional work will be needed to make this method work well for other types of content.

While the training and clustering proceeds without human intervention, hyperparameters such as the number of training epochs and the structure of the neural network have a large impact on the effectiveness of semantic hashing. The proof-of-concept method we used here does require us to adjust and tune the hyperparameters to find the best results for a particular game, but there exist methods for using random search [13] and genetic algorithms to discover hyperparameters for neural networks [14].

While we have not yet applied this algorithm to an actual game design process, we believe the semantic hashing technique can someday prove to be valuable for applications of automated and AI-assisted game design. These potential applications include: understanding how much of each type of content is present in a game; generating new levels in a procedural content generation system; suggesting similar or different designs in a mixed-initiative system; and recognising what an end user might be creating for user-generated content. Iterative design requires a solid analysis and understanding of what has been created in each iteration in order to improve it, and these computer-assisted techniques are a promising way to improve our understanding of design and our ability to execute good design.

Acknowledgements

Thanks to Philip Bontrager, Daniel Gopstein, Scott Lee, Rishabh Jain, and Andy Nealen for their discussions and training on neural networks, deep learning, and autoencoders. Thank you to NVIDIA for providing GPUs resources for this research project.

Aaron Isaksen received his PhD from New York University, is a long-time game designer, co-founder of Indie Fund, and Chairman of IndieBox. **Address:** New York City, New York, USA. **Email:** aisaksen@appabove.com

Christoffer Holmgård received his PhD from the IT University of Copenhagen, did his postdoc at New York University, and is a co-owner of Die Gute Fabrik. **Address:** Olfert Fischers Gade 45 2, 1311 Copenhagen K, Denmark. **Email:** christoffer@holmgard.org

Julian Togelius is an Associate Professor at New York University. He co-directs the Game Innovation Lab and researches AI and games. **Address:** New York University, 2 MetroTech Center, 11201 Brooklyn, NY, USA. **Email:** julian@togelius.com

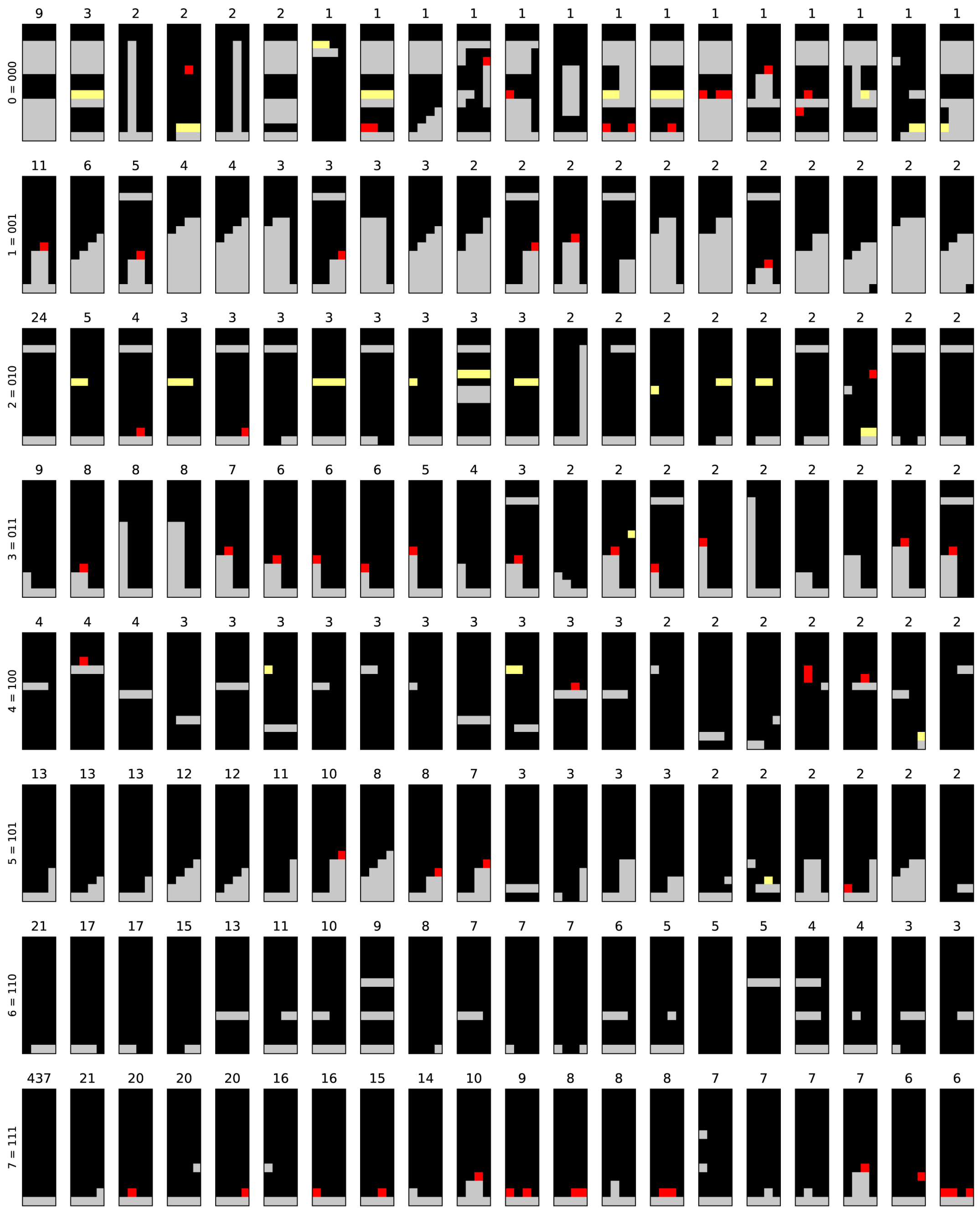


Figure 5. Semantic hashing of Mario level windows into $8 = 2^3$ categories.

References

[1] Salakhutdinov, R. and Hinton, G., 'Semantic Hashing', *International Journal of Approximate Reasoning*, vol. 50, no. 7, 2009, pp. 969–978.

[2] Hinton, G., 'Lecture 15.4 – Semantic Hashing', *Coursera* course on *Neural Networks for Machine Learning*, 2012. https://www.coursera.org/learn/neural-networks

[3] Vincent, P., Larochelle, H., Bengio, Y. and Manzagol, P.-A., 'Extracting and Composing Robust Features with Denoising Autoencoders', in *Proceedings of the 25th International Conference on Machine Learning*, (ICML 2008), Helsinki, ACM, 2008, pp. 1096–1103.

[4] Jain, R., Isaksen, A., Holmgård, C. and Togelius, J., 'Autoencoders for Level Generation and Style Identification', in *The 2nd Computational Creativity and Games Workshop* (CCGW 2016), Paris, 2016.

[5] Miyamoto, S. and Tezuka, T., *The Legend of Zelda*, Nintendo, 1986.

[6] Miyamoto, S. and Tezuka, T., *Super Mario Bros*, Nintendo, 1985.

[7] Summerville, A., Snodgrass, S., Mateas, M. and Ontañón, S., 'The VGLC: The Video Game Level Corpus', in *The 7th Workshop on Procedural Content Generation* (PCG2016), Dundee, 2016.

[8] Togelius, J., Yannakakis, G. N., Stanley, K. O. and Browne, C., 'Search-Based Procedural Content Generation: A Taxonomy and Survey', *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 3, 2011, pp. 172–186.

[9] Summerville, A., Snodgrass, S., Guzdial, M., Holmgård, C., Hoover, A. K., Isaksen, A., Nealen, A. and Togelius, J., 'Procedural Content Generation via Machine Learning (PCGML)', *arXiv*, arXiv:1702.00539, 2017.

[10] Goodfellow, I., Bengio, Y. and Courville, A., *Deep Learning*, Massachusetts, MIT Press, 2016.

[11] Rumelhart, D. E., Hinton, G. and Williams, R. J., 'Learning Representations by Back-Propagating Errors', *Nature*, Vol. 323, 1986, pp. 533–536.

[12] Harris, D. and Harris, S., *Digital Design and Computer Architecture*, second edition, San Francisco, Morgan Kaufmann, 2012.

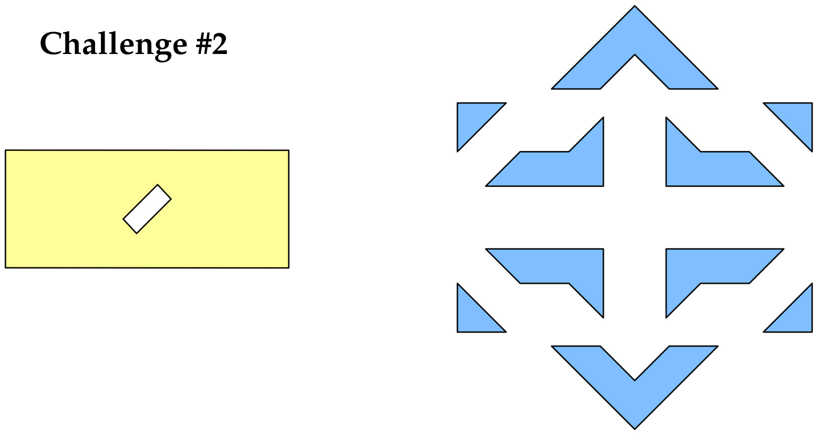
[13] Bergstra, J. and Bengio, Y., 'Random Search for Hyper-Parameter Optimization', *Journal of Machine Learning Research*, vol. 13, no. 1, 2012, pp. 281–305.

[14] Miikkulainen, R., Liang, J., Meyerson, E., Rawal, A., Fink, D., Francon, O., Raju, B., Navruzyan, A., Duffy, N. Hodjat, B., 'Evolving Deep Neural Networks', *arXiv*, arXiv:1703.00548, 2017.

Gadeiro Challenges #2 and #3

Pack the pieces on the right to fill the shapes on the left. Gadeiro is described on pages 39–41.

Challenge #2



Challenge #3