Autoencoders for Level Generation, Repair, and Recognition

Rishabh Jain, Aaron Isaksen, Christoffer Holmgård, Julian Togelius
Tandon School of Engineering, New York University

Abstract

Autoencoders are neural networks for unsupervised learning and dimensionality reduction which have recently been used for generating and modeling images. In this paper we argue for the use of autoencoders in game content generation, recognition and repair, and describe proof-of-concept implementations of autoencoders for these tasks for Super Mario Bros levels. Concretely, we train autoencoders to reproduce levels from the original Super Mario Bros game, and then use these networks to discriminate generated levels from original levels, and to generate new levels as transformation from noise. We believe these methods will generalize to other types of two-dimensional game content.

Introduction

Procedural Content Generation has been used in games since the early 80's. In recent years, it has also become a topic of academic research (Togelius et al. 2011). The generation of content by machines is a practical application of *exploratory computational creativity* (Boden 2004; Wiggins 2006) whereby a design space is explored for new variations. Typically, human content is analyzed and a generator is created to generate content in a similar style.

Deep learning (Goodfellow, Bengio, and Courville 2016) has been used for various creative purposes including the recognition, generation, and transfer of visual style in relation to images (Mordvintsev, Olah, and Tyka 2015; Johnson 2015). Although trained on a subset of images, neural networks seem particularly capable of generating novel content that was not envisioned by the original content used to train the networks. We aim to use similar techniques to generate new levels for a 2D platformer.

In this paper, we use autoencoders (Hinton and Salakhutdinov 2006), a form of neural network, to generate levels, repair level content, and classify level examples by style. In a nutshell, an autoencoder learns to output its own input after first passing it through a channel (a neural layer) of much smaller bandwidth (size) than the input and output. Of course, it is impossible to learn a generic autoencoder that performs well on any input. By the nature of the narrowing hidden layers, we are forcing information to be lost. But well-performing autoencoders can still be learned for specific datasets. They can perform well because they have learned to encode and reproduce the type of data that can be found in the particular dataset they are trained on, to the extent of performing worse on data that is substantially different from what it was trained on. The autoencoder has implicitly encoded the regularities, or "style" of the training data.

One way of viewing what an autoencoder does is in terms of non-linear dimensionality reduction. Because one or more hidden layers have a lower number of units than the input and output, the number of the dimensions of the data needs to be reduced in order to fit in this layer. Training the autoencoder amounts to finding a way of doing this non-linear dimensionality reduction while losing as little information as possible, relative to the training data. Unlike linear methods such as Principal Component Analysis, the resulting non-linear dimensionality reduction can model far more complicated relationships.

Another way of viewing what an encoder does is that it learns the style of the input data, and can then be used to discriminate between data that adheres to the style of the training data and data that diverges from that style. This can be done because the reconstruction error of data from another style would (in most cases) be higher. Trained autoencoders can also be used to create new examples of the style they are trained on. Once an autoencoder has been trained, one can decide to use only the "encoder" part (the connections from the input to the hidden layer) or only the "decoder" part (connections from the hidden layer to the input). For example, one can input random data into the hidden layer and then observe the output from the decoder; a properly trained autoencoder would output data which corresponds to (some of) the characteristics of the training data. One can also search the hidden layer for patterns that are as different as possible from any patterns generated by feeding the training data to the encoder, and observe the results of feeding such unusual patterns to the decoder (Liapis et al. 2013). Neural Networks have previously been used in all those roles for image data (Denton et al. 2015).

What we propose here is to use autoencoders not on the level of pixel data, i.e. images, but on the level of meaning-bearing game elements. Specifically, we propose training autoencoders on game levels represented as tiles. In this paper we will train and test our networks with levels from Super Mario Bros, obtained through the Mario AI Framework (Horn et al. 2014) or Video Game Library Corpus (Summerville et al. 2016) – this very popular 2-D platformer provides a

similar benchmark for AI in games that MNIST does for Deep Learning research (Jarrett et al. 2009). We expect these methods will transfer to other examples of 2D tile-based game content as well.

We believe that autoencoders applied to games can be useful in variety of ways, opening up new possibilities for data-driven procedural content generation. Most of these techniques could be useful both for standalone online or offline content generators, or as part of AI-assisted game design tools:

- Content generation/transformation. An autoencoder can generate content that fits into a particular style from all kinds of input, including random noise, a picture, a geometric shape or a signature. This could be useful for data games: one can take a 2D representation of some real-world object or area and turn it into game content respecting style and gameplay constraints.
- Style recognition. We can detect whether content created by another content generator, or human creator, fits into a particular style. This is useful both for staying close to a style and for intentionally deviating from or breaking it.
- Content repair. An autoencoder could take an unplayable level and make it better fit the style it has been trained on, thereby repairing it and making it playable. In the context of an AI-assisted game design tool, this could be used to complete levels that have been partly designed.
- Data compression. The compression into fewer dimensions allows to store game content more efficiently. This might seem pointless given today's storage capacity, but on the other hand some games might benefit from very large game worlds, such as in PCG games like *No Man's Sky*.
- Design research. Similar to the work to explore the Flappy Bird game space (Isaksen, Gopstein, and Nealen 2015), examining the distribution of values in the hidden layers (Erhan, Courville, and Bengio 2010) can give some insight into what features are commonly shared among most levels and most designers. By moving outside of the normal regions defined by the game, it could be possible to generate new level ideas that have not been tried before.

In the following section, we briefly outline other approaches in the literature to level generation for 2D platformers in general, and Super Mario Bros in particular.

Level Generation for 2D Platformers

Many types of procedural content generation methods have been applied to generating platformer levels. Some of these are based on designer-defined top-down models, such as rule-based generation which was the basis of the original *Infinite Mario* version of the framework (Togelius, Karakovskiy, and Baumgarten 2010) or e.g. Smith et al.'s work on rhythm-based generation (Smith et al. 2009).

Other work, closer to the work presented in this paper, uses a bottom-up data-driven approach to generate the levels. Dahlskog, Togelius, and Nelson trained n-grams on sequences on of level "slices" (columns of tiles) cut from the original Super Mario Bros levels, and showed that with n=3 believable Mario levels can be generated; however the constraints of the slice-based representation together with the relatively small dataset leads to a lack of diversity in the generated content (Dahlskog, Togelius, and Nelson 2014). Similarly, Snodgrass and Ontanon trained two-dimensional Markov models on the level of individual tiles to generate levels (Snodgrass and Ontanón 2013). Baumgarten suggested the use of Linear Discriminant analysis to gauge player skill and then using these player models to adjust feature weights for level generation (Shaker et al. 2011). Other approaches have included search-based methods such grammatical evolution (Shaker et al. 2012), and learning from video replay data using computer vision and probabilistic modeling (Guzdial and Riedl 2015).

Recently, approaches using neural networks have been proposed and demonstrated by Hoover, Togelius, and Yannakis (2015), who use neuro-evolution, and Summerville and Mateas (2016), who use long short-term memory (LSTM) recurrent neural networks. Apart from these examples, work in using networks for level generation is still sparse. In this paper, we suggest using a third variant of neural networks, autoencoders, to learn, reproduce, and vary levels for Super Mario Bros.

Horn et al. (2014), in addition to providing a framework for evaluating the expressive range of various generation procedural level generation methods, also provides a grouping of these methods previously documented in the literature by control type. The paper lists *constructive (none)*, *indirect*, *parameterized*, and *knowledge representation* as control types for shaping the levels expressed by the cited level generators. It also notes that pattern-based approaches combine the *indirect* and *knowledge representation* based categories.

The approach we present here is novel in the sense that it provides a new combination of control types: Autoencoders allow us to to use existing levels as *indirect* representations of patterns, learned by the autoencoders, but also allow us to control the kinds of levels that are generated *parametrically*, by varying the noise inserted into the autoencoder, as explained below. In the following section, we provide an introduction to autoencoders and how we adapt them to the problem of generating Mario levels.

Autoencoders

Autoencoders are neural networks which encode data into a different number of dimensions. A simple autoencoder is a feed-forward multilayer neural network which has a hidden layer smaller than the input layer. The output layer is the same size as the input layer in a simple autoencoder since it tries to recreate the input and learn its lower dimensional representation in the hidden layer. Since the training data does not need additional metadata, like labels in a classification problem, it is considered to be an unsupervised algorithm.

Figure 1 presents a diagram of a simple autoencoder. We take a small window of the input level of size WxH and reshape it by appending each row of data into a vector of size WHx1. This new vector is fully connected to a hidden layer of size hx1. A HardTanh transfer function is placed after the fully connected layer so that our network is non-linear – this

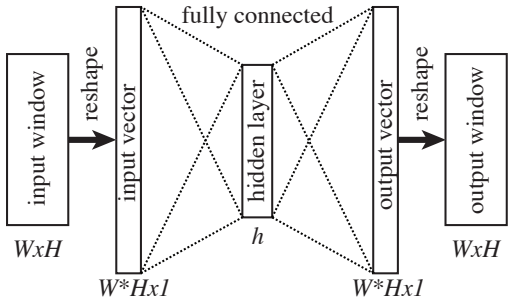


Figure 1: A simple autoencoder with 3 levels tries to replicate the output window from the input window. Because the hidden layer is smaller than the input and output, correlations in the data are discovered during the learning process.

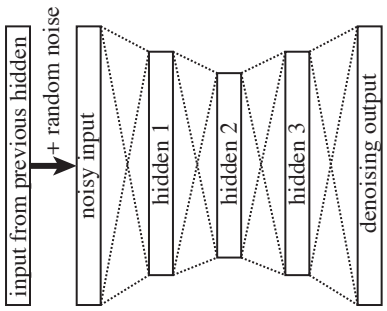


Figure 2: Our denoising autoencoder with 5 levels tries to replicate the values from the hidden layers of the original network. We add noise during the training process so that the network is forced to avoid overfitting.

is a simple function that clamps values < -1 to -1 and values > 1 to 1. However, this simple function combined with the linear transforms allows the network to learn non-linear behaviors. We then have another fully connected layer that takes the hidden layer to an output vector that is the same size as the input vector, again followed by a HardTanh for non-linearity. This is finally reshaped back into an output window of size WxH. We calculate the error between the input and output windows and use back-propagation to calculate the gradients in the fully connected layers that will minimize this error. After repeated training, the autoencoder learns to replicate the input data. Autoencoders can have more than one hidden layer, and in this paper we use autoencoders of different numbers of layers and layer sizes.

In order to prevent the autoencoder from overfitting and only learning the original data, *denoising autoencoders* attempt to force the network to learn useful features by adding noise to the original signal during training (Vincent et al. 2008). This noise "corrupts" the input data and therefore requires the autoencoder to learn how to reconstruct the original signal from redundancies in the source. When calculating the loss, the noise is not used so that the gradients and error determine how good the autoencoder is at reconstructing the original signal.

An example of a denoising autoencoder is presented in Figure 2. The input, which as we will explain later in the paper comes from another network, is corrupted with random noise. It then passes through three fully connected hidden layers, each followed by a HardTanh transfer function. The final output is of the same size as the input, and we use a loss function that measures the error between the denoised output and the original input without noise.

Dataset and Training

Since autoencoders are an unsupervised learning technique, the input data is not required to be labeled. Since autoencoders use the same data as both input and for error back propagation we are able to use windows constructed from the maps as our training set without any labeling. The maps used for the training set were the 22 overground as well as the underground levels from the original Mario world maps.

When designing an autoencoder to be trained on map data, we structure the training data in such a way that training and reconstruction is logically and computationally feasible. To make extraction of map characteristics easier and non redundant we encode the map with a unique character representing each kind of tile (LeCun 2012). We use this encoding, instead of e.g. the images representing the tiles in the game, to enable the autoencoder to learn the properties of the map without having to understand actual appearance of the tiles themselves - which would be included in the image format of the map. This ensures we only include relevant data and reduces the size of the network to something easier to train.

For initial analysis of the map we used a binary encoding where blank space, perceived as empty to a user, as 0 while obstacles are encoded as a 1. The resulting encoded maps are binary interpretations of the original maps, as shown in Figure 3.

Although we did not do so in this paper, it would be possible to represent the different types of objects (e.g. enemies, coin boxes, pipes) instead of just conflating them into terrain/obstacles. The accepted approach is to use additional channels for each type of object (including empty), and to still use a binary encoding. This is similar to representing colors in an image using 3 channels instead of using a 1 channel palette. The reason this is preferred is that it allows the network to learn better than using different values in a single channel. In a palette based approach, where for example 0 represents empty space, 1 represents a block, and 2 represents an enemy, etc. the mathematical average of an enemy (2) and empty space (0) is then a block ($(2+0)/2 = 1$) which obviously misrepresents the relations between the encoded values, since a block is not the average of an empty space and an enemy. Instead, with the multichannel approach, the channel with the highest value after using a softmax would be selected for the output, ensuring that all possible content types can be learnt.

Because of the sliding window nature of gameplay of Mario, where the player can only see 16x14 tiles at a time, we used a window striding over the horizontally aligned map for training samples. For example the World 1-1 of Mario displayed in Figure 3. The level map is encoded into a

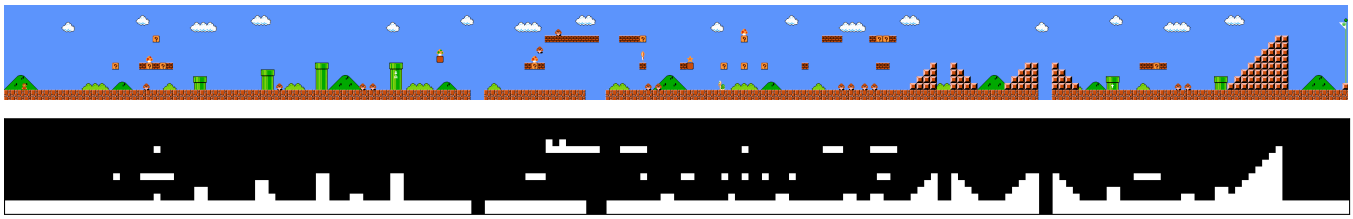


Figure 3: Mario World 1-1 and its corresponding encoding.

199x14 tensor representing the tiles from which we sample with sliding windows, overlapping to maintain the spatial relationship among the entities.

To construct a tensor which comprises of windows, we set a window size and extracted the first window from the map. We then took strides of one tile horizontally and concatenated the result to the tensor until the end of the map. Hence for each map we obtained a number of windows $n = M - W + 1$, where M is the horizontal size of the map in tiles and W is the horizontal size of the window to construct in tiles.

Once we had an encoded form of all the maps from the original Super Mario Bros, we sampled different windows sizes from the concatenated tensor. The window sizes that we obtained satisfactory results from were 1x14, 2x14, 3x14 and 4x14. We started out with window size 16x14 matching the player's view on the screen (and assuming that a squarish window would yield acceptable reconstruction due to symmetric reconstruction), however, the reconstruction tended towards a local minimum generating a heavily averaged reconstructed image. This lead to the intuition that smaller a window size would lead to a smaller hidden state as well as establish stronger spatial relationships. Another source of inspiration for the windows sizes is the fact that the average horizontal entity is less than 4 horizontal blocks in the encoded maps which would lead to continuity in the reconstruction of the map after the stitching process. The process of windows together into a map is described in the Level Reconstruction section.

After splitting 22 maps from the original Mario into windows we were able to generate a substantial input dataset of 4432, 4410, 4388, and 4366 windows respectively for window sizes 1x14, 2x14, 3x14, and 4x14. We use the entire training set for training purposes as we would cross validate the results for classification, repair and generation separately. We can however, observe the training accuracy of each architecture to check for convergence. We also notice that a trivial autoencoder (with a hidden layer larger or equal to the input) reconstructs with no error per tile which reinforces our assumption of characteristic learning by the autoencoders.

For training each of the autoencoders for the purpose of our experiments we used the training dataset and experimented with the loss criterion, network architecture and training parameters like learning rate, momentum and decay rate. During the experiments we found that the convergence for Stochastic Gradient Descent works best with a learning rate in the range of 0.05-0.175 and with a momentum of $1e-4$. This however would vary with different network architecture. RMSprop and Adadelta with the Keras framework (Chollet 2015) default settings are also acceptable for good convergence.

We trained simple autoencoders with a single hidden layer but experimented with varying sizes of the layer, each for 500 epochs. On using mean squared error, L2 error, and absolute difference criterion, we obtained the best reconstruction from absolute difference. We also found that the activation function most suited for map reconstruction to be HardTanh because of the behavioral pattern of the gradient as well as the activation function limit. All the experiments were performed either using the Torch framework, a Lua based scientific computing language for Neural Network based experiments (Collobert, Bengio, and Marithoz 2002), or the Keras framework, a Python-based neural network library (Chollet 2015). Both frameworks provide efficient processing of complex matrix computations attributed to its underlying GPU-based implementation for hardware accelerated parallel processing.

Level Reconstruction

To visualize the features learned by the autoencoder by training it is essential to reconstruct and determine the training quality. We could reconstruct single windows using the simple autoencoder and determine the reconstruction quality of the trained networks. However a better test of learning would be to try to generate the entire map over the windows to see how well each network configuration works.

In our experiments we were able to determine the difference in learning with each of the network configuration and by changing the window sizes. We observed that trivial autoencoders with the hidden layer of a single node was able to successfully learn the placement of the ground including the pits. However, the ground height was not reconstructed well by the trivial autoencoder. With increasing number of nodes more characteristics were learned by the simple autoencoder with a complete autoencoder being able to perfectly replicate the input map.

As expected, adding a node to the hidden layer and retraining makes the neural network capable of gathering knowledge of the spatial placement of an increasing number of elements, as shown in Figure 4. Increasing the hidden layer size to just 2 nodes increased the learning to incorporate knowledge of pipe placements in the map reconstruction as well as partially reconstruct the staircases which was visible distinctly with hidden layer of size 4 nodes. Hidden layer size 7 and 14 (perfect reconstruction) are almost the same quality. We can understand the learning capability of the network configurations by observing the average reconstruction error.

Figure 4: Autoencoder reconstruction with different hidden layer sizes on windows of 1x14

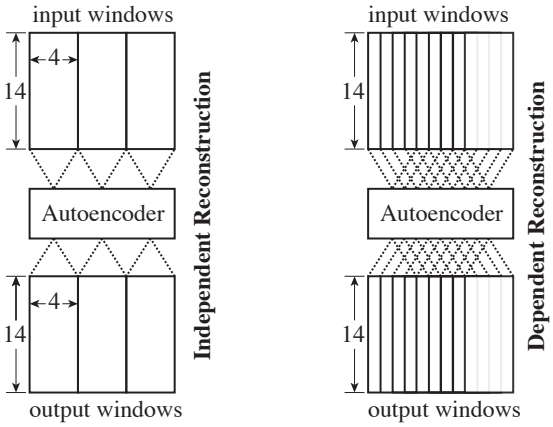


Figure 5: Reconstruction models used for generating maps from windows

To reconstruct the complete map from individual windows we used two different strategies, shown in Figure 5:

Independent Reconstruction In this strategy, instead of using the input samples from training, we used adjacent windows from the map with no overlap and concatenated the output from the autoencoder in the same way as we sample, i.e. take outputs of adjacent non-overlapping windows and concatenate to observe the reconstruction.

Dependent Reconstruction In this strategy we use the overlapping input from the sample we constructed during the training phase. In the input two adjacent windows have an overlapping area along with a slice of the map belonging to each of the windows uniquely. Hence after generating a new window from the autoencoder we only concatenate the unique overflowing slice from the generated window to the reconstructed map. This way, apart from the first window, the contribution of each window to the final reconstruction is a single slice irrespective of the window size.

For all reconstruction referred to later in the paper we use dependent reconstruction because of its superior performance in minimizing reconstruction as well to maintain associative flow between frames. This comes from the intuition that a new generated window should have continuity propagated from previous windows.

Level Style Recognition

The objective of training on the simple autoencoder is to learn the parameters which determine the characteristics of the map. The low level embedding of the hidden layer should be able to determine the reconstruction of the individual windows. By understanding the dynamics of the map we expect that the compressed state vector would be an accurate descriptor of the map type, which in turn affects the game play experience of the map. We should also be able to determine the windows of each map which are more indicative of each style.

With the same intuition, we trained the simple autoencoder on the Super Mario Bros maps from our dataset of 4366 windows of size 14x4. After training, we use the the lower level embedding matrix from forward propagated maps to be used for classification.

We trained the simple autoencoder with various hidden layer configurations and found that we could use the autoencoder with a compression ratio of 0.5535 for classification because of its stable reconstruction and low absolute error of 0.29 per tile. We observed the classification for three types of levels namely the underground level World-1-2, above ground level World-1-1, and an above ground level generated by the Parameterized Notch algorithm (Horn et al. 2014). Due to the distinct style variation in the maps, we assume that the classification should be deterministic.

We calculated the hidden layer embedding from the three maps resulting in 154, 194 and 198 vectors for each map respectively. To classify this data, we construct a neural net-

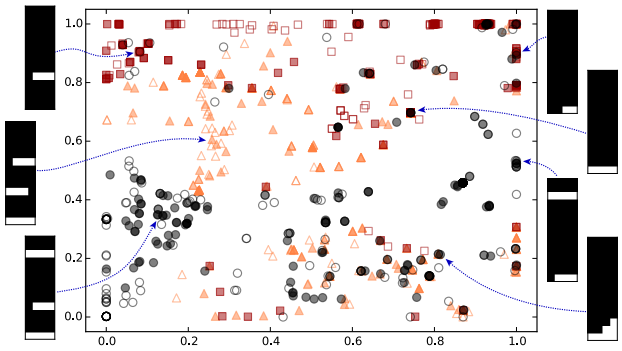


Figure 6: Autoencoder with 2 nodes on the hidden layer for segmenting styles. Orange Triangles: Above Ground Levels (Solid: World 1-1, Hollow: 2-1). Black Circles: Underground Levels (Solid: World 1-2, Hollow: 4-2). Red Squares: Tree Levels (Solid: World 1-3, Hollow: 3-3). Example windows give a sense of what is located in each section of the graph.

work classifier with 31 input nodes and 3 output classes and 2 fully-connected hidden layers. We then perform training with this dataset using 80% for training and 20% for cross validation. Upon 20 epochs of training we were able to get a classification accuracy of 82%. The loss criterion that we used was negative log likelihood. We do however note that since the windows are overlapping, there would be a bias introduced towards lower loss due to similar data in the training set since adjacent windows have common data.

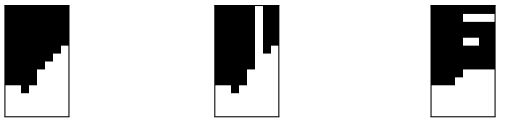
In Figure 6 we create a 2D visualization by using an autoencoder with 2 values in the middle hidden layer. Training on the entire data set, we can then visualize the 2D value of the hidden layer using windows from two above ground, two underground, and two tree levels. We can see that there are regions in the 2D map that are unique for each type of level: e.g. the lower left corner has almost exclusively windows from underground levels, indicated by these levels having overhead blocks; the upper left corner has windows with no ground, indicative of tree levels. This shows the effectiveness of autoencoders at finding unique style traits in the level data without human guidance.

Level Repair

The simple autoencoder is able to learn the style characteristics of windows in the map. It is thus expected to learn the essentials of a playable map. Since the integrity of the map is an essential part of the design, we assume that the autoencoder is able to mold an unplayable map or window to a more realistic playable map by enforcing style rules learned from the training data.

To implement an unplayable section of map we sampled a window from one of the maps (Figure 7a) and added features which are presumed to be unplayable. Here we used a window which included a staircase often found in Mario and added an impassable wall expanding the entire height of the window as seen in Figure 7b.

We show in Figure 7c that the simple autoencoder was



(a) Original (b) Unplayable (c) Repaired

Figure 7: The original window is overwritten with a vertical wall making the game unplayable. The autoencoder is able to repair the window to make it playable again, although it chooses a different solution to the problem.

(a) Simulated window with impassable pit (b) Unsuccessfully repaired window

Figure 8: Unsuccessful Repairs – due to the window size, the algorithm is unable to bridge the gap.

able to construct a window given the unusable input which is playable as well as embraces the style of the training data. We obtain a window which is nontrivial and with element placement features consistent with subsequent windows.

However we also found that not all irregularities in the map integrity could be repaired. We experimented on windows which contained large trenches, in one experiment spanning about 24 pixels, and found that the repair was not able to affect the visible playability of the windows as shown in Figure 8. We expect an approach that incorporates a simple AI path into the map data before learning will help with this problem (Summerville and Mateas 2016).

With this result, we believe that spatial association is limited by the individual windows of reconstruction which determines the extent of repair. To be able to repair such large valued irregularities it would take larger windows for the map to learn the patterns in construction associated with larger sections of the map.

Level Generation through Transformation

Autoencoders can also be used to generate new content, by injecting randomness into the hidden layer. We will use this general approach to generate new levels from the autoencoders trained on existing map data, something we can call *data-driven procedural content generation*.

To generate playable Mario levels we will first study the distribution of values in the middle hidden layer when passing through the original map data. We then create new levels by generating random values from a similar distribution as the observed values from the original maps. For the purpose of experiments with generation we used the autoencoder with 31 hidden layer nodes trained on 14x4 windows because of visibly better performance on independent reconstruction.

The hidden layer vector is a 31 element vector with each element between -1 and 1 due to the hardTanh activation used in the previous layers in the simple autoencoder. We observed

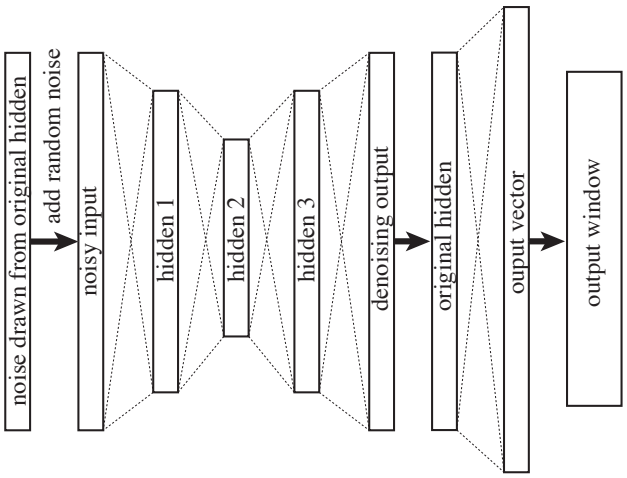


Figure 9: Our complete generation pipeline combines the two networks to generate new content. The input is noise generated from a similar distribution to the original levels.

the distribution of each of the element of the vector obtained by forward propagating the windows of World-1-1 to obtain a 31x50 matrix. The simple autoencoder learns the lower level embedding of the map properties, however the nodes in the hidden layer are not expected to be independent of each other. Therefore only some configurations of parameter values result in a map-like window (Kingma and Welling 2013). This was corroborated by reconstructing a map by feeding random noise into the hidden layer, which resulted in an image which does not look like a map.

One problem is that the autoencoder has not been trained to be robust to small variations, so we train a denoising autoencoder to observe the hidden layer and reproduce its values in an output layer (Poultney et al. 2006). The denoising autoencoder is expected to reconstruct a vector from input noise post-training which is then decoded by the original autoencoder to a valid playable map. We used several different configurations on the denoising autoencoder with different levels of compression and noise distribution, each resulting in different characteristics on the reconstructed levels. With a small set of training samples we are able to generate any number of maps because of the noisy source of creativity.

The two most effective denoising encoders that we formulated consisted of 31 inputs and outputs as well as 13 and 15 hidden layer nodes respectively. Apparently, the lower number of hidden layers results in a more conservative approach to map regeneration with more sparse objects; this can be attributed to higher loss during compression.

Observing the denoising encoder dataset we gathered that each of the elements of the hidden layer activation follows a sampling distribution which we fit each element to a normal distribution with mean and standard deviation. We used this property in the generation of new data from the denoising autoencoder using random values that follow this same distribution. After training the denoising autoencoder using the hidden layer activation dataset from World-1-1, we constructed a complete generative neural network consisting of the decoder module from the simple autoencoder appended to the denoising autoencoder, as shown in Figure 9. This neural network can generate levels from modeled noise. Although we can now theoretically generate new map windows with noise input, we found that the results were better when we calculated the element-wise normal distribution from the matrix generated from the autoencoder's hidden layer. We can then perform element-wise addition of noise to samples from the element wise distribution to generate newer windows (Ozair and Bengio 2014). This way we ensure that the noise loosely adheres to a certain range yet involves enough randomness to generate interesting windows.

We were able to use this method of generation to generate different types of maps, as shown in Figure 10, each with varying characteristics which changed over the course over the map. The generated maps were able to retain some essential characteristics of Mario maps. Properties of the map like the density of element placement, holes, etc. can be adjusted by a change in the noise distribution in the input. This can also be a time-varying function leading to different sections of the level possessing different characteristics.

Conclusion

The work included in this paper uses autoencoders to perform style recognition and also proposes a novel method for content generation using multiple autoencoders. We also experimented with the level repair abilities of autoencoders and their limitations. While most of the proposals in this work as well as the experiments are promising, this work functions mainly as an initial study in using autoencoders for platformer level content generation and analysis. We believe that these methods can be used to generate more complex content, especially when combined with multi-channel maps. We plan to use the generator to more deeply examine creation of different styles of content in this and other games.

References

Boden, M. A. 2004. *The Creative Mind: Myths and Mechanisms*. Psychology Press.

Chollet, F. 2015. keras. https://github.com/fchollet/keras.

Collobert, R.; Bengio, S.; and Marithoz, J. 2002. Torch: A modular machine learning software library.

Dahlskog, S.; Togelius, J.; and Nelson, M. J. 2014. Linear levels through n-grams. In *International Academic MindTrek Conference: Media Business, Management, Content & Services*, 200–206. ACM.

Denton, E. L.; Chintala, S.; Fergus, R.; et al. 2015. Deep generative image models using a laplacian pyramid of adversarial networks. In *Advances in Neural Information Processing Systems*, 1486–1494.

Erhan, D.; Courville, A.; and Bengio, Y. 2010. Understanding representations learned in deep architectures. *Department d'Informatique et Recherche Operationnelle, University of Montreal, QC, Canada, Tech. Rep* 1355.

Goodfellow, I.; Bengio, Y.; and Courville, A. 2016. Deep learning. Book in preparation for MIT Press.

Figure 10: Map generation with noise variance = 0.01, 0.1, 0.3, and 0.6

Guzdial, M., and Riedl, M. O. 2015. Toward game level generation from gameplay videos. In *Proceedings of the FDG workshop on Procedural Content Generation in Games*.

Hinton, G. E., and Salakhutdinov, R. R. 2006. Reducing the dimensionality of data with neural networks. *Science* 313(5786):504–507.

Hoover, A. K.; Togelius, J.; and Yannakis, G. N. 2015. Composing video game levels with music metaphors through functional scaffolding. *ICCC Workshop on Computational Creativity and Games*.

Horn, B.; Dahlskog, S.; Shaker, N.; Smith, G.; and Togelius, J. 2014. A comparative evaluation of procedural level generators in the mario ai framework. *Proceedings of Foundations of Digital Games*.

Isaksen, A.; Gopstein, D.; and Nealen, A. 2015. Exploring game space using survival analysis. In *Foundations of Digital Games*.

Jarrett, K.; Kavukcuoglu, K.; Ranzato, M.; and LeCun, Y. 2009. What is the best multi-stage architecture for object recognition? In *Computer Vision, 2009 IEEE 12th International Conference on*, 2146–2153. IEEE.

Johnson, J. 2015. Neural-style github repository. https://github.com/jcjohnson/neural-style.

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

LeCun, Y. 2012. Learning invariant feature hierarchies. In *Computer vision–ECCV 2012. Workshops and demonstrations*, 496–505. Springer.

Liapis, A.; Martınez, H. P.; Togelius, J.; and Yannakakis, G. N. 2013. Transforming exploratory creativity with delenox. In *International Conference on Computational Creativity*, 56–63. AAAI Press.

Mordvintsev, A.; Olah, C.; and Tyka, M. 2015. Inceptionism: Going deeper into neural networks. *Google Research Blog. Retrieved June* 20.

Ozair, S., and Bengio, Y. 2014. Deep directed generative autoencoders. *arXiv preprint arXiv:1410.0630*.

Poultney, C.; Chopra, S.; Cun, Y. L.; et al. 2006. Efficient learning of sparse representations with an energy-based model. In *Advances in neural information processing systems*, 1137–1144.

Shaker, N.; Togelius, J.; Yannakakis, G. N.; Weber, B.; Shimizu, T.; Hashiyama, T.; Sorenson, N.; Pasquier, P.; Mawhorter, P.; Takahashi, G.; et al. 2011. The 2010 mario ai championship: Level generation track. *Computational Intelligence and AI in Games, IEEE Transactions on* 3(4):332–347.

Shaker, N.; Nicolau, M.; Yannakakis, G. N.; Togelius, J.; and Neill, M. O. 2012. Evolving levels for super mario bros using grammatical evolution. In *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*, 304–311. IEEE.

Smith, G.; Treanor, M.; Whitehead, J.; and Mateas, M. 2009. Rhythm-based level generation for 2d platformers. In *Foundations of Digital Games*, 175–182. ACM.

Snodgrass, S., and Ontanón, S. 2013. Generating maps using markov chains. In *Ninth Artificial Intelligence and Interactive Digital Entertainment Conference*.

Summerville, A., and Mateas, M. 2016. Super mario as a string: Platformer level generation via LSTMs. *To appear in DiGRA/FDG*.

Summerville, A.; Snodgrass, S.; Mateas, M.; and Ontanón, S. 2016. The vglc: The video game level corpus. In *Procedural Content Generation Workshop at DiGRA/FDG*.

Togelius, J.; Yannakakis, G. N.; Stanley, K. O.; and Browne, C. 2011. Search-based procedural content generation: A taxonomy and survey. *Computational Intelligence and AI in Games, IEEE Transactions on* 3(3):172–186.

Togelius, J.; Karakovskiy, S.; and Baumgarten, R. 2010. The 2009 mario ai competition. In *Evolutionary Computation (CEC), 2010 IEEE Congress on*, 1–8. IEEE.

Vincent, P.; Larochelle, H.; Bengio, Y.; and Manzagol, P.-A. 2008. Extracting and composing robust features with denoising autoencoders. In *Intl. Conf. on Machine Learning*, 1096–1103. ACM.

Wiggins, G. A. 2006. A preliminary framework for

description, analysis and comparison of creative systems. *Knowledge-Based Systems* 19(7):449–458.