SMWLevelGenerator

Generating Super Mario World Levels Using Deep Neural Networks

Jan Ebert

January 13, 2020

Outline

Managing Expectations

Fundamentals

- Super Mario World
- Julia

Framework

- Introduction
- Preprocessing Pipeline
- Models

Level Generation

Conclusions

Managing Expectations

Goals

- Create an open and optimized research framework for level generation in an arbitrarily complex environment
- Challenge/increase existing complexity (papers usually on much simpler Super Mario Bros.)
- Analyze usage of sequence prediction for generation
- Compare sequence generation capabilities of LSTMs vs. transformers

What *Not* to Expect: The Chair

Do not expect a “working” level generator producing great results:

- Decreasing loss does not imply great levels
- Levels generated by my models are barebones and unplayable
- Evaluating results is tedious and has to be done manually due to no objective function

What to Expect: The Hammer

Expect an accessible, extensible, efficient, cross-platform framework to implement, train and test models:

- Data (pre-)processing pipeline
- Highly compressed database
- Support for any desired data dimensionality
- Level generation using a combination of different methods
- Focus on simplicity for library user

With That Out of the Way...

- Source code, thesis and slides available at `github.com/janEbert/SMWLevelGenerator/tree/fzj`
- Clickable links are invisible but should be obvious

Fundamentals

Super Mario World (SMW)

- 1990 2D platformer by Nintendo for the Super Nintendo Entertainment System (SNES)
- Huge amount of tiles, enemies, interactions, . . .
- Running, jumping, flying, tossing, riding dinosaur, . . .
- Large amount of data due to active hacking scene

Let's keep it relatively simple!

Super Mario World (SMW)

- 1990 2D platformer by Nintendo for the Super Nintendo Entertainment System (SNES)
- Huge amount of tiles, enemies, interactions, ...
- Running, jumping, flying, tossing, riding dinosaur, ...
- Large amount of data due to active hacking scene

Let's keep it relatively simple!

Hacking

- Active hacking community around SMW
- Hacks available as patches to original ROM due to copyright
- *Very* heavy modifications possible; we keep it simple ("vanilla" hacks)
- SMWCentral.net as main resource
- Hacks filtered by following criteria:
  - Rating $\geq$ 3.0 (community average)
  - "vanilla" tag
- However, neither "vanilla"-ness nor quality of hacks is guaranteed

In the end: over 300 hacks with over 17 000 levels
(over 15 000 after filtering).

Lunar Magic

Lunar Magic (LM) is the main tool to modify SMW:

- GUI editing
- Advanced modifications via 65C816 assembly
- Many convenience modifications (e.g. extended level dimensions, placing secondary entrances anywhere, . . .)
- Author *FuSoYa* provided private build to support dumping and reading levels from and into ROMs (and answered many questions)

Short Lunar Magic Showcase

Live Demo

Super Mario World Levels In-Depth

- We will only focus on horizontal levels
- Levels are divided into screens (27×16-rectangles)
- Levels contain metadata having influence on tileset, water, ...
- Levels are 3D tensors consisting of the following layers:
  - Tiles (512)
  - Main and midway entrance (2)
  - Sprites (enemies, special effects) (241)
  - Level exits (screen and secondary) ($511 + 449$)
  - Secondary entrances (449)

2164 layers in total!

Super Mario World Levels In-Depth

- We will only focus on horizontal levels
- Levels are divided into screens (27×16-rectangles)
- Levels contain metadata having influence on tileset, water, ...
- Levels are 3D tensors consisting of the following layers:
  - Tiles (512)
  - Main and midway entrance (2)
  - Sprites (enemies, special effects) (241)
  - Level exits (screen and secondary) ($511 + 449$)
  - Secondary entrances (449)

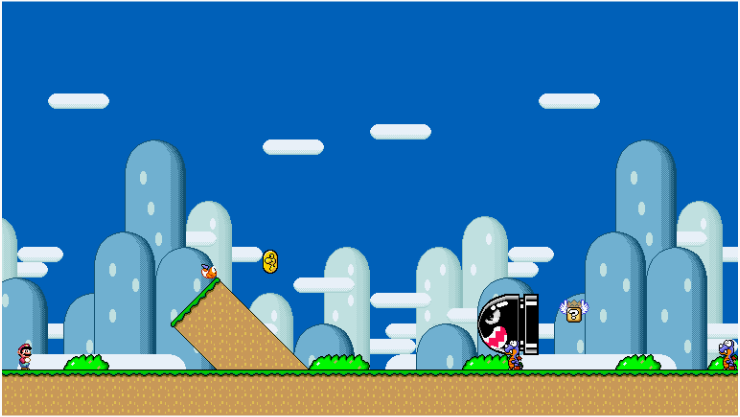
2164 layers in total! With 27 rows and 512 columns at maximum: $\approx 300\,000$ binary entries per level.

Super Mario World Levels In-Depth

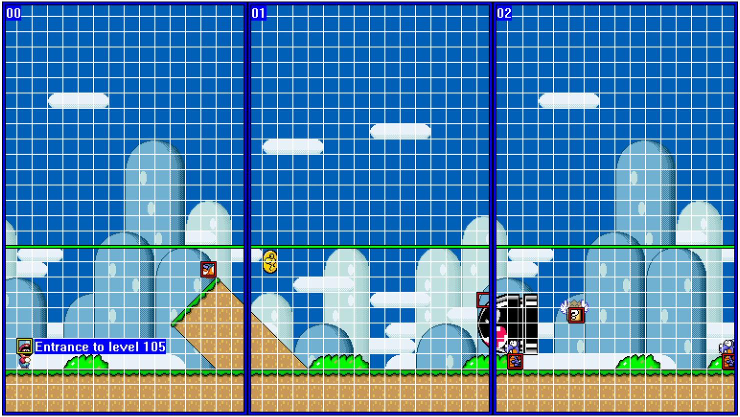
- We will only focus on horizontal levels
- Levels are divided into screens (27×16-rectangles)
- Levels contain metadata having influence on tileset, water, ...
- Levels are 3D tensors consisting of the following layers:
  - Tiles (512)
  - Main and midway entrance (2)
  - Sprites (enemies, special effects) (241)
  - Level exits (screen and secondary) ($511 + 449$)
  - Secondary entrances (449)

2164 layers in total! With 27 rows and 512 columns at maximum: $\approx 30\,000\,000$ binary entries per level.

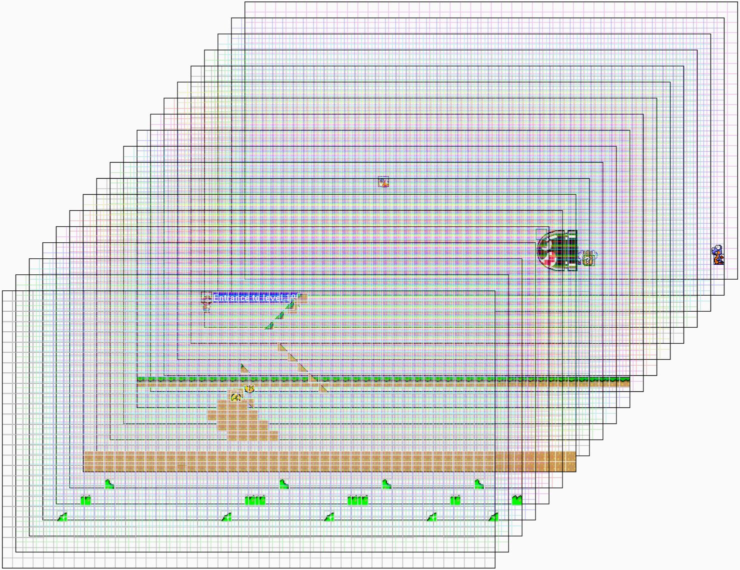
Level Example



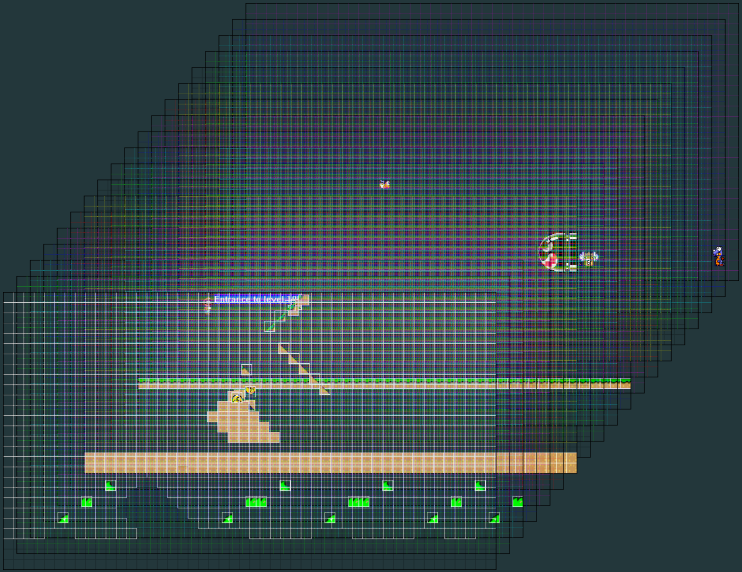
Level Detailed



Level Layers



Level Layers



Vanilla and Custom Tiles

- Lunar Magic allows new tiles with custom behavior to be implemented
- Vanilla game features 512 unique tiles which may reference each other
- Custom tiles may use different graphics but reference vanilla behavior
  $\rightarrow$ Not all custom tiles are non-vanilla! Most people do not program new tiles but want custom graphics
- Lunar Magic rejects cyclical references
- Resolve custom tiles to vanilla tiles by following references

Julia

- Implemented in Julia 1.3 (and 1.2)
- Modern dynamically typed language; combination of Lisp, Python and Octave with C-level performance
- JIT-compiled via LLVM
- Simple GPU usage and extensibility
- User-friendly multi threading and distributed programming
- Great REPL and package manager
- Easy to use: type stability and care with caching $\rightarrow$ speed

Julia and CUDAnative.jl

Wasserstein GANs need clamping of parameters for convergence properties.
Method in Julia stdlib: `clamp!(array, low, high)`

Wasserstein GANs need clamping of parameters for convergence properties.
Method in Julia stdlib: `clamp!(array, low, high)`

However! Slow on GPUs due to scalar indexing. :(

Julia and CUDAnative.jl

Wasserstein GANs need clamping of parameters for convergence properties.
Method in Julia stdlib: `clamp!(array, low, high)`

However! Slow on GPUs due to scalar indexing. :(
Solution: write it yourself – in high-level Julia thanks to CUDAnative.jl

Julia and CUDAnative.jl

Wasserstein GANs need clamping of parameters for convergence properties.
Method in Julia stdlib: `clamp!(array, low, high)`

However! Slow on GPUs due to scalar indexing. :(
Solution: write it yourself – in high-level Julia thanks to CUDAnative.jl (and make a pull request later).

DIY GPU Kernel

We simply define a new `clamp!` method on GPU arrays:

```
function Base.clamp!(a::CuArray, low, high)
    function kernel(a, low, high)
        I = CuArrays.@cuindex a
        a[I...] = clamp(a[I...], low, high)
        return
    end

    blocks, threads = CuArrays.cudims(a)
    @cuda(blocks=blocks, threads=threads,
          kernel(a, low, high))
    return a
end
```

Julia Profits

- Thesis resulted in multiple PRs all over the Julia ecosystem
- Due to combination of readability and efficiency, it was both easy and satisfying for me to contribute
- Writing in Julia made adding new features and functionality a breeze (sparse GPU array support in a few lines)
- Since submission: countless improvements in the ecosystem such as using optimized math via Torch.jl

There is still a lot of work ahead but it is getting there.

Julia Ecosystem – What to Look Out For

Hot at the moment and interesting for us:

- SciML toolbox for physics-informed ML with focus on differential equations
- Source-to-source automatic differentiation via Zygote.jl (was too unstable for me; became default AD engine for Flux.jl after submission) or soon Diffractor.jl (WIP)
- TPU compilation via FluxML/XLA.jl (using JAX build) or JuliaTPU/XLA.jl (with TensorFlow.jl)
- Many others including classical ML frameworks, toolkits and algorithms

Framework

SMWLevelGenerator

The framework can be roughly divided into these modules:

- Data preprocessing and database generation
- Data iterators
- Model interface
- Training loops
- Level generation pipeline

Setup (Quick Version)

1. Get dependencies (Julia 1.3, TensorBoard, Lunar Magic[1], Floating IPS, Wine if not on Windows, Super Mario World ROM[2])

[1] Private build

[2] American version; CRC32 checksum `a31bead4`

Setup (Quick Version)

1. Get dependencies (Julia 1.3, TensorBoard, Lunar Magic[1], Floating IPS, Wine if not on Windows, Super Mario World ROM[2])
2. Instantiate Julia project:
   `julia --project -e "using Pkg; Pkg.instantiate()"`

[1] Private build

[2] American version; CRC32 checksum `a31bead4`

Setup (Quick Version)

1. Get dependencies (Julia 1.3, TensorBoard, Lunar Magic[1], Floating IPS, Wine if not on Windows, Super Mario World ROM[2])
2. Instantiate Julia project:
   `julia --project -e "using Pkg; Pkg.instantiate()"`
3. Download databases and decompress

[1] Private build
[2] American version; CRC32 checksum `a31bead4`

Setup (Quick Version)

1. Get dependencies (Julia 1.3, TensorBoard, Lunar Magic[1], Floating IPS, Wine if not on Windows, Super Mario World ROM[2])
2. Instantiate Julia project:
   `julia --project -e "using Pkg; Pkg.instantiate()"`
3. Download databases and decompress

You're done; train and generate to your heart's content.

[1] Private build

[2] American version; CRC32 checksum `a31bead4`

Setup (Manual/BTS Version)

Most of these are done in a single line; still, this gives an overview of what happens behind the scenes.

1. Get dependencies and instantiate project (see previous slide)

Setup (Manual/BTS Version)

Most of these are done in a single line; still, this gives an overview of what happens behind the scenes.

1. Get dependencies and instantiate project (see previous slide)
2. Download desired hacks

Setup (Manual/BTS Version)

Most of these are done in a single line; still, this gives an overview of what happens behind the scenes.

1. Get dependencies and instantiate project (see previous slide)
2. Download desired hacks
3. Unzip hacks, patch ROMs and dump levels via scripts

Setup (Manual/BTS Version)

Most of these are done in a single line; still, this gives an overview of what happens behind the scenes.

1. Get dependencies and instantiate project (see previous slide)
2. Download desired hacks
3. Unzip hacks, patch ROMs and dump levels via scripts
4. Remove test, duplicate and "dirty" levels via scripts

Setup (Manual/BTS Version)

Most of these are done in a single line; still, this gives an overview of what happens behind the scenes.

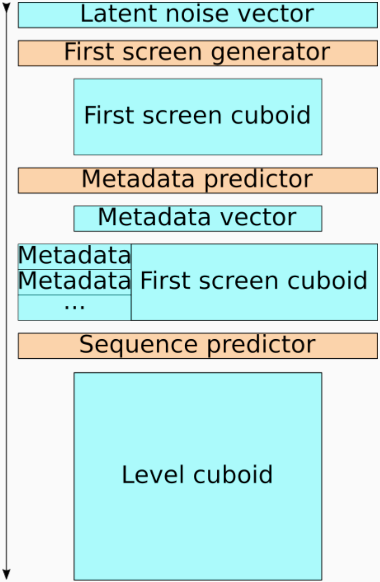
1. Get dependencies and instantiate project (see previous slide)
2. Download desired hacks
3. Unzip hacks, patch ROMs and dump levels via scripts
4. Remove test, duplicate and "dirty" levels via scripts
5. Generate level statistics via Julia REPL

Setup (Manual/BTS Version)

Most of these are done in a single line; still, this gives an overview of what happens behind the scenes.

1. Get dependencies and instantiate project (see previous slide)
2. Download desired hacks
3. Unzip hacks, patch ROMs and dump levels via scripts
4. Remove test, duplicate and “dirty” levels via scripts
5. Generate level statistics via Julia REPL
6. Generate database(s)

Pipeline Overview



How Does It Work?

- Combination of different methods:
  1. Generative methods to generate initial inputs (first screen)
  2. Image processing to predict metadata from initial input
  3. Natural language processing to sequentially generate the rest of the level from the initial inputs
- What we will focus on: read level column by column, predict next column (tile by tile also possible)
- Each column contains constant metadata and bit whether level has *not* ended
- Levels end with column of zeros (during generation, only the one bit matters)
- Loss: summed MSE of each predicted column in relation to target column

Why Regression and Not Classification?

For these kinds of tasks: usually use one-hot-encoding to predict the "class" of the next tile/column.
I wanted to predict column by column for speed and more local correlation.

Number of classes when reading...

- ... column by column: $\approx 17\,600$
- ... tile by tile: 662

Why Regression and Not Classification?

For these kinds of tasks: usually use one-hot-encoding to predict the "class" of the next tile/column.
I wanted to predict column by column for speed and more local correlation.

Number of classes when reading...

- ... column by column: $\approx 17\,600$ digit number
- ... tile by tile: 662 digit number

No way I could train a model on that many classes even if memory problems were solved.

Dimensionalities

Different complexities reflected on the highest level via dimensionality:

- 1D: Level is seen as single row of one type of tile.
- 2D: Level is seen as matrix of one type of tile.
- 3D: Level is seen as cuboid with chosen tile layers.

The default type of tile for 1D and 2D is the ground tile of the level.

Simplifications

Too many to list (check out the thesis!), here are important ones:

- Levels are observed independently (connections by exits/entrances are ignored)
- A lot of metadata is omitted (unrelated to level generation)
- Test levels or unfinished levels are left in the dataset
- Levels are assumed to always go from left to right
- Several types of levels are excluded (vertical, boss, with layer 2 interaction[3])

[3] Usually a background layer but may be made interactive with sprite commands.

Data Preprocessing

- Lunar Magic dumps five different files for each level corresponding to different parts of the level
- Remove encrypted hacks and those throwing errors
- Filter duplicate and vanilla game test levels by checksums
- Remove levels not adhering to vanilla behavior
- Format levels according to user-specified configuration (dimensionality, which layers, output type, . . .)

Database Compression

- Maximum storage required per level if storing bits compactly[4]: $30 \cdot 10^6$ bits $\div 8 = 3.75$ MB
- With 17 000 levels: 3.75 MB $\cdot 17\,000 \approx 63.75$ GB
- Usable but too much for me; database should fit into 8 GB of RAM[5]

1. Use sparse arrays! Additional speed benefits for free (0.046 % of data are assigned in full levels)

[4] One bit per bit, 8 bits per byte
[5] Free space on my laptop with 2 000 browser tabs open

Database Compression

- Maximum storage required per level if storing bits compactly[4]: $30 \cdot 10^6$ bits $\div 8 = 3.75$ MB
- With 17 000 levels: 3.75 MB $\cdot 17\,000 \approx 63.75$ GB
- Usable but too much for me; database should fit into 8 GB of RAM[5]

1. Use sparse arrays! Additional speed benefits for free (0.046 % of data are assigned in full levels)
2. Use sparse arrays with smallest integer sizes covering full range (`Int64` → `UInt16`)

[4] One bit per bit, 8 bits per byte

[5] Free space on my laptop with 2 000 browser tabs open

Database Compression

- Maximum storage required per level if storing bits compactly[4]: $30 \cdot 10^6$ bits $\div 8 = 3.75$ MB
- With 17 000 levels: $3.75 \text{ MB} \cdot 17\,000 \approx 63.75$ GB
- Usable but too much for me; database should fit into 8 GB of RAM[5]

1. Use sparse arrays! Additional speed benefits for free (0.046 % of data are assigned in full levels)
2. Use sparse arrays with smallest integer sizes covering full range (`Int64` → `UInt16`)
3. Most layers are empty: do not save these either

[4] One bit per bit, 8 bits per byte

[5] Free space on my laptop with 2 000 browser tabs open

Database Compression Results

Maximum calculated required storage[6]: 63.75 GB
After highest compression: 430 MB

Due to recurring values, `tar` and `gzip` compress this further to 28 MB.
7-Zip compresses to only 16 MB! Now *that's* portable.

[6] 17 000 levels · 30 000 000 entries = 510 000 000 000 entries in total

Data Iterator

Optimizations:

- Sparse arrays already optimize our data iterator for large dimensionalities
- Sparse GPU arrays commented due to previously missing functionality in external package (now fixed, but needs testing)
- Arbitrarily many threads or single coroutine for data iterator

Different models require different data layouts; there are several implementations that cover most cases (as matrix or as list of columns/tiles, both with optional padding).

Data Iterator

Optimizations:

- Sparse arrays already optimize our data iterator for large dimensionalities
- Sparse GPU arrays commented due to previously missing functionality in external package (now fixed, but needs testing)
- Arbitrarily many threads or single coroutine for data iterator

Different models require different data layouts; there are several implementations that cover most cases (as matrix or as list of columns/tiles, both with optional padding).
All of this behind the scenes due to…

Model Interface

Abstract type requiring minimal implementation to work with the framework; adding new models requires the following:

1. Defined as `Flux.@treelike` (`Flux.@functor` in recent versions)
2. Field `hyperparams` of type `Dict{Symbol, Any}`
3. Required key in `hyperparams`[7]: `:dimensionality` of model (a `Symbol`)[8]
4. 5 required functions for sequence predictors, one less for GANs/metadata predictors

Any model implementing this interface works with the framework.

[7] GAN generators require one more key `:inputsize`

[8] 1D, 2D, 3D only tiles, 3D, . . . (very easily extensible)

Sequence Predictors

- LSTM (stack)
- Transformer (GPT-2) (required data layout is not optimal for us)
- Random predictor (optimal activation chance by default)

Models may implement "soft" loss that penalizes incorrect predictions less if the prior two elements were the same.

Predictions not done on predicted data $\rightarrow$ error accumulation not observed/reduced during training.

Inputs are levels from beginning to (current) end, outputs are the predicted next column for each input column. Remember each input also has a metadata vector attached; outputs do not.

GANs

- DCGAN
- Wasserstein DCGAN
- Dense Wasserstein GAN

1D GANs automatically adjust layers to input; 2D and above have manually chosen stride, padding and dilation so output size matches first screen size.

Discriminators: Inputs are first screen tensors (vector in 1D, matrix in 2D, cuboid in 3D), outputs are scalars whether input is real.
Generators: Inputs are noise vectors, outputs are first screen tensors.

Image Processing Models

- Convolutional
- Dense (MLP)

Inputs are first screen tensors, outputs are the constant metadata vectors and “level has not ended”-bit also supplied to the sequence predictor.

Training Loops

- Checkpointing and resuming training
- Handling experiments (storing all parameters, logging, ...)
- Logging via TensorBoard
- Early stopping
- Overfitting on batch for debugging

Many other settings allow the exact modifications you want.

Level Generation

Pipeline

The models are trained, what next?

1. Feed input data (generated or from database) and subsequent generations into sequence predictor until the "level has not ended"-bit is *not* set or until the maximum level length is reached
2. Post-process
3. Revert all pre-processing
4. Write back to Lunar Magic-processable files
5. Write back to ROM

Level Example

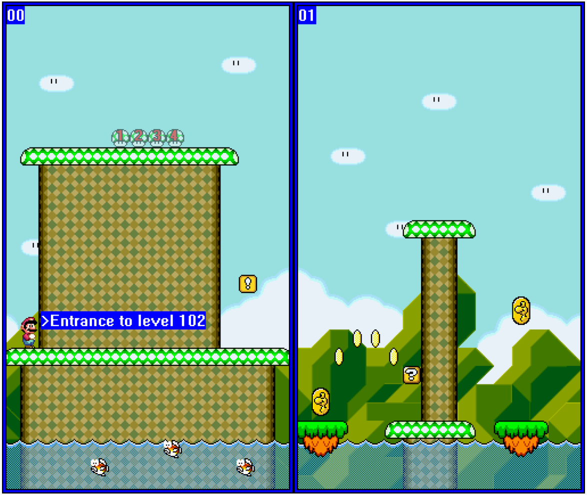


Figure 1: First two screens of level 258

Results 1D: Sequence Prediction Only

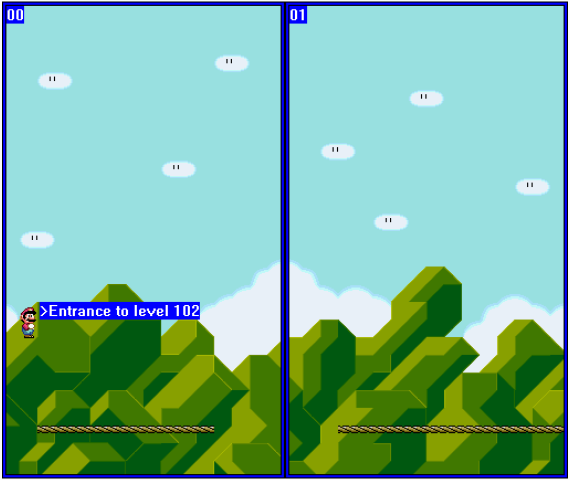


Figure 2: First two generated screens for level 258 via transformer sequence prediction only

Results 1D: Full Pipeline

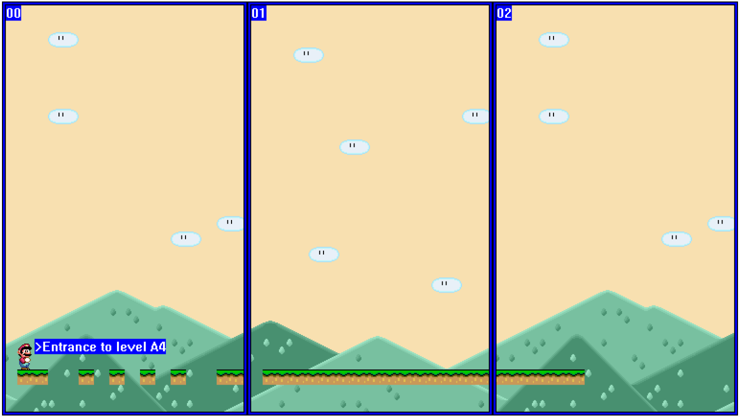


Figure 3: Complete level generated by pipeline with LSTM

Level Example

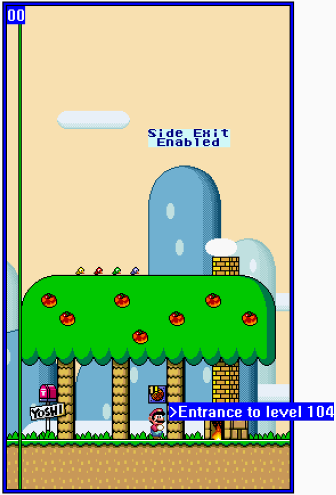


Figure 4: Level 260 as a different kind of "level"

Results 2D: Sequence Prediction Only

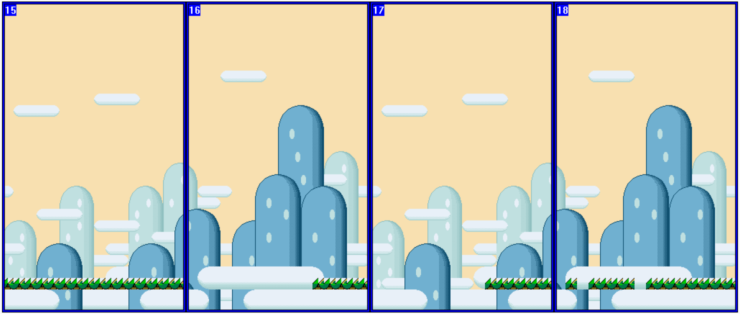


Figure 5: Selected screens generated for level 260 via transformer sequence prediction only

Conclusions

Current Problems

- Not using one-hot-encoding per tile complicates the problem way too much
- Sequence predictors do not work well enough as generators (no overfitting)
- GANs are GANs (training is hard; interpreting losses is harder)
- Evaluating results takes too long and is annoying (would need more command line scripting capabilities in Lunar Magic[9] or some objective function)
- Generating with transformers too slow

[9] Or we could roll our own...

Future Improvements

What can be done to further improve the framework?

- General hyperparameter tuning
- NAS/random search would be amazing.
  Idea: apply a macro to all models and list each parameter's value ranges. These value ranges may be read by a new random search module.
- Make sequence predictors more noisy and/or train them on their own predictions to improve sequential prediction
- More models, especially generative ones
- More modular pipeline; maybe you don't want to use a sequence predictor (good choice)
- More features (e.g. learning rate warmup)

My Takeaways

- Test models *in practice* regularly
- Stacking models is way too complex for this kind of task
- Julia keeps all its promises (although the language and ecosystem are still very young)
- Read even more task-specific machine learning papers prior to working on it

Questions

Any questions?

The End

Thank you for your attention!