Machine Learning is Fun! Part 2

Using Machine Learning to generate Super Mario Maker levels

Adam Geitgey Follow
Jan 3, 2016 · 15 min read

Update: *This article is part of a series. Check out the full series: Part 1, Part 2, Part 3, Part 4, Part 5, Part 6, Part 7 and Part 8!* You can also read this article in *Italiano, Español, Français, Türkçe, Русский, 한국어 Português, فارسی, Tiê´ng Viêt or 普通话.*

Giant update: *I've written a new book based on these articles! It not only expands and updates all my articles, but it has tons of brand new content and lots of hands-on coding projects. Check it out now!*

In Part 1, we said that Machine Learning is using generic algorithms to tell you something interesting about your data without writing any code specific to the problem you are solving. (If you haven't already read part 1, read it now!).

This time, we are going to see one of these generic algorithms do something really cool—create video game levels that look like they were made by humans. We'll build a neural network, feed it existing Super Mario levels and watch new ones pop out!

One of the levels our algorithm will generate

Just like Part 1, this guide is for anyone who is curious about machine learning but has no idea where to start. The goal is be accessible to anyone—which means that there's a lot of generalizations and we skip lots of details. But who cares? If this gets anyone more interested in ML, then mission accomplished.

• • •

Making Smarter Guesses

Back in Part 1, we created a simple algorithm that estimated the value of a house based on its attributes. Given data about a house like this:

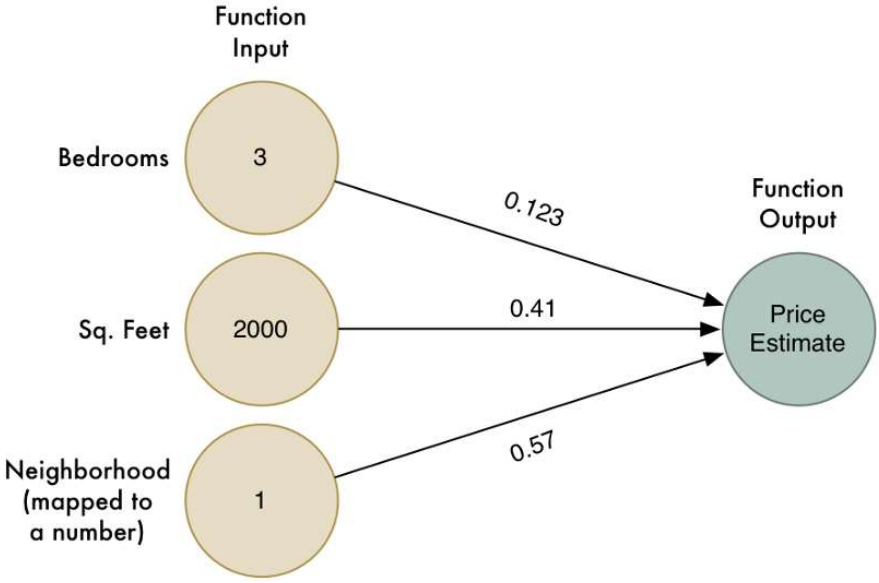
Bedrooms	Sq. feet	Neighborhood	Sale price
3	2000	Hipsterton	???

We ended up with this simple estimation function:

```
def estimate_house_sales_price(num_of_bedrooms, sqft,
neighborhood):
 price = 0

# a little pinch of this
 price += num_of_bedrooms * 0.123

# and a big pinch of that
 price += sqft * 0.41

# maybe a handful of this
 price += neighborhood * 0.57

return price
```

In other words, we estimated the value of the house by multiplying each of its attributes by a **weight**. Then we just added those numbers up to get the house's value.

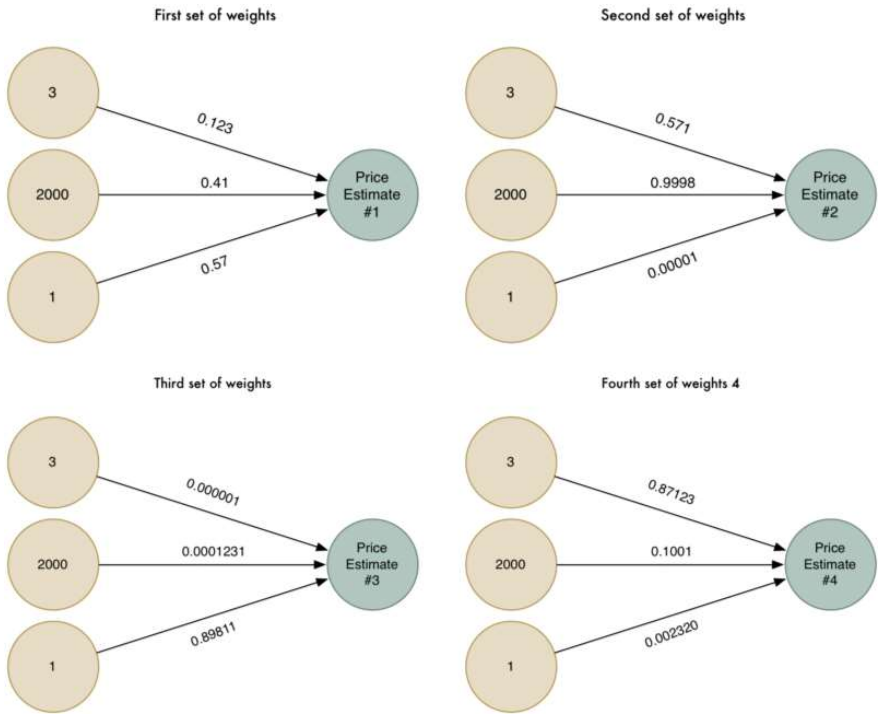
Instead of using code, let's represent that same function as a simple diagram:



The arrows represent the weights in our function.

However this algorithm only works for simple problems where the result has a *linear* relationship with the input. What if the truth behind house prices isn't so simple? For example, maybe the neighborhood matters a lot for big houses and small houses but doesn't matter at all for medium-sized houses. How could we capture that kind of complicated detail in our model?

To be more clever, we could run this algorithm multiple times with different of weights that each capture different edge cases:



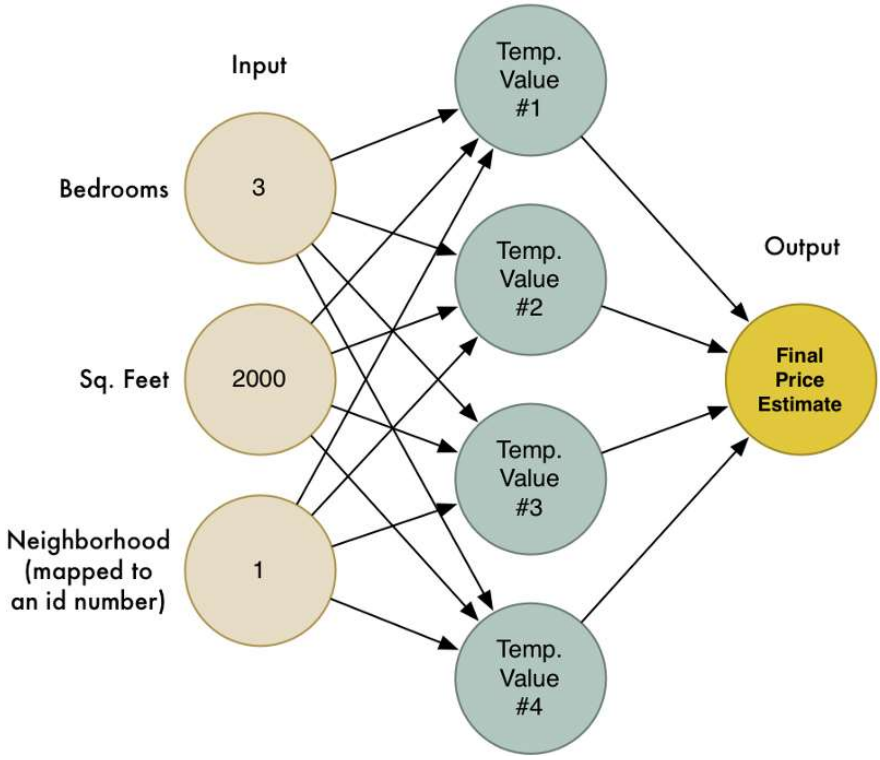
Let's try solving the problem four different ways

Now we have four different price estimates. Let's combine those four price estimates into one final estimate. We'll run them through the same algorithm again (but using another set of weights)!

Our new *Super Answer* combines the estimates from our four different attempts to solve the problem. Because of this, it can model more cases than we could capture in one simple model.

What is a Neural Network?

Let's combine our four attempts to guess into one big diagram:



This is a neural network! Each node knows how to take in a set of inputs, apply weights to them, and calculate an output value. By chaining together lots of these nodes, we can model complex functions.

There's a lot that I'm skipping over to keep this brief (including feature scaling and the activation function), but the most important part is that these basic ideas *click:*

- We made a simple estimation function that takes in a set of inputs and multiplies them by weights to get an output. Call this simple function a ***neuron***.
- By chaining lots of simple ***neurons*** together, we can model functions that are too complicated to be modeled by one single neuron.

It's just like LEGO! We can't model much with one single LEGO block, but we can model anything if we have enough basic LEGO blocks to stick together:

A grim preview of our plastic animal future? Only time can tell...

Giving our Neural Network a Memory

The neural network we've seen always returns the same answer when you give it the same inputs. It has no memory. In programming terms, it's a *stateless algorithm*.

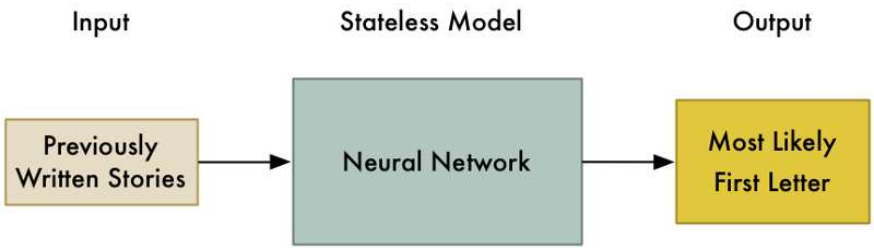
In many cases (like estimating the price of house), that's exactly what you want. But the one thing this kind of model can't do is respond to patterns in data over time.

Imagine I handed you a keyboard and asked you to write a story. But before you start, my job is to guess the very first letter that you will type. What letter should I guess?

I can use my knowledge of English to increase my odds of guessing the right letter. For example, you will probably type a letter that is common

at the beginning of words. If I looked at stories you wrote in the past, I could narrow it down further based on the words you usually use at the beginning of your stories. Once I had all that data, I could use it to build a neural network to model how likely it is that you would start with any given letter.

Our model might look like this:



But let's make the problem harder. Let's say I need to guess the *next* letter you are going to type at any point in your story. This is a much more interesting problem.

Let's use the first few words of Ernest Hemingway's *The Sun Also Rises* as an example:

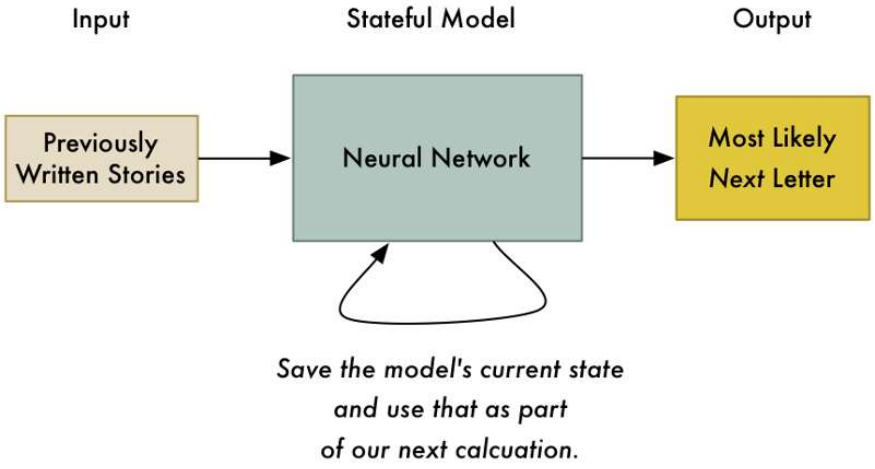
> *Robert Cohn was once middleweight boxi*

What letter is going to come next?

You probably guessed 'n'—the word is probably going to be *boxing*. We know this based on the letters we've already seen in the sentence and our knowledge of common words in English. Also, the word 'middleweight' gives us an extra clue that we are talking about boxing.

In other words, it's easy to guess the next letter if we take into account the sequence of letters that came right before it and combine that with our knowledge of the rules of English.

To solve this problem with a neural network, we need to add *state* to our model. Each time we ask our neural network for an answer, we also save a set of our intermediate calculations and re-use them the next time as part of our input. That way, our model will adjust its predictions based on the input that it has seen recently.



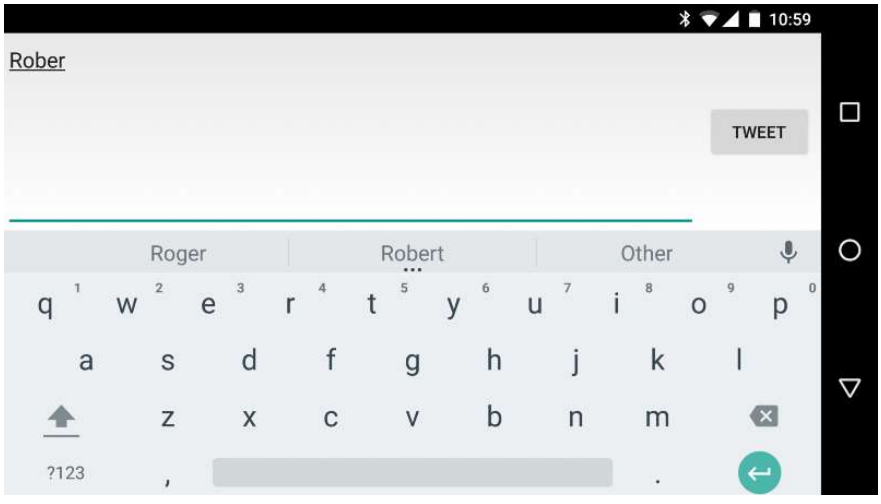
Keeping track of state in our model makes it possible to not just predict the most likely *first* letter in the story, but to predict the most likely *next* letter given all previous letters.

This is the basic idea of a *Recurrent* Neural Network. We are updating the network each time we use it. This allows it to update its predictions based on what it saw most recently. It can even model patterns over time as long as we give it enough of a memory.

What's a single letter good for?

Predicting the next letter in a story might seem pretty useless. What's the point?

One cool use might be auto-predict for a mobile phone keyboard:



The next most likely letter is "t".

But what if we took this idea to the extreme? What if we asked the model to predict the next most likely character over and over—forever? We'd be asking it to write a complete story for us!

Generating a story

We saw how we could guess the next letter in Hemingway's sentence. Let's try generating a whole story in the style of Hemingway.

To do this, we are going to use the Recurrent Neural Network implementation that Andrej Karpathy wrote. Andrej is a Deep-Learning researcher at Stanford and he wrote an excellent introduction to generating text with RNNs, You can view all the code for the model on github.

We'll create our model from the complete text of *The Sun Also Rises*—362,239 characters using 84 unique letters (including punctuation, uppercase/lowercase, etc). This data set is actually really small compared to typical real-world applications. To generate a really good model of Hemingway's style, it would be much better to have at several times as much sample text. But this is good enough to play around with as an example.

As we just start to train the RNN, it's not very good at predicting letters. Here's what it generates after a 100 loops of training:

> *hjCTCnhoofeoxelif edElobe negnk e iohehasenoldndAmdaI ayio pe e h'e*
> *btentmuhgehi bcgdltt. gey heho grpiahe.*
> *Ddelnss.eelaishaner" cot AAfhB ht ltny*
> *ehbih a"on bhnte ectrsnae abeahngy*
> *amo k ns aeo?cdse nh a taei.rairrhelardr er deffijha*

You can see that it has figured out that sometimes words have spaces between them, but that's about it.

After about 1000 iterations, things are looking more promising:

> *hing soor ither. And the caraos, and the crowebel for figttier and ale the room of me? Streat was not to him Bill-stook of the momansbed mig out ust on the bull, out here. I been soms*
> *inick stalling that aid.*
> *"Hon't me and acrained on .Hw's don't you for the roed," In's pair."*
> *"Alough marith him."*

The model has started to identify the patterns in basic sentence structure. It's adding periods at the ends of sentences and even quoting dialog. A few words are recognizable, but there's also still a lot of nonsense.

But after several thousand more training iterations, it looks pretty good:

> *He went over to the gate of the café. It was like a country bed.*
> *"Do you know it's been me."*

> *“Damned us,” Bill said.*
> *“I was dangerous,” I said. “You were she did it and think I would a fine cape you,” I said.*
> *“I can’t look strange in the cab.”*
> *“You know I was this is though,” Brett said.*
> *“It’s a fights no matter?”*
> *“It makes to do it.”*
> *“You make it?”*
> *“Sit down,” I said. “I wish I wasn’t do a little with the man.”*
> *“You found it.”*
> *“I don’t know.”*
> *“You see, I’m sorry of chatches,” Bill said. “You think it’s a friend off back and make you really drunk.”*

At this point, the algorithm has captured the basic pattern of Hemingway’s short, direct dialog. A few sentences even sort of make sense.

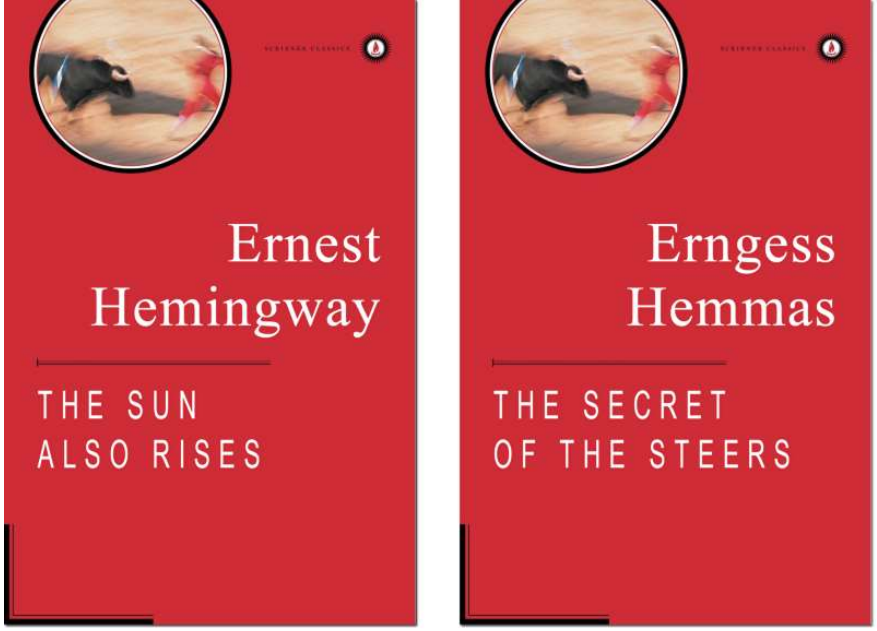
Compare that with some real text from the book:

> *There were a few people inside at the bar, and outside, alone, sat Harvey Stone. He had a pile of saucers in front of him, and he needed a shave.*
> *“Sit down,” said Harvey, “I’ve been looking for you.”*
> *“What’s the matter?”*
> *“Nothing. Just looking for you.”*
> *“Been out to the races?”*
> *“No. Not since Sunday.”*
> *“What do you hear from the States?”*
> *“Nothing. Absolutely nothing.”*
> *“What’s the matter?”*

Even by only looking for patterns *one character at a time*, our algorithm has reproduced plausible-looking prose with proper formatting. That is kind of amazing!

We don’t have to generate text completely from scratch, either. We can seed the algorithm by supplying the first few letters and just let it find the next few letters.

For fun, let’s make a fake book cover for our imaginary book by generating a new author name and a new title using the seed text of “Er”, “He”, and “The S”:



The real book is on the left and our silly computer-generated nonsense book is on the right.

Not bad!

But the **really mind-blowing part** is that this algorithm can figure out patterns in any sequence of data. It can easily generate real-looking recipes or fake Obama speeches. But why limit ourselves human language? We can apply this same idea to any kind of sequential data that has a pattern.

Making Mario without actually Making Mario

In 2015, Nintendo released Super Mario Maker™ for the Wii U gaming system.

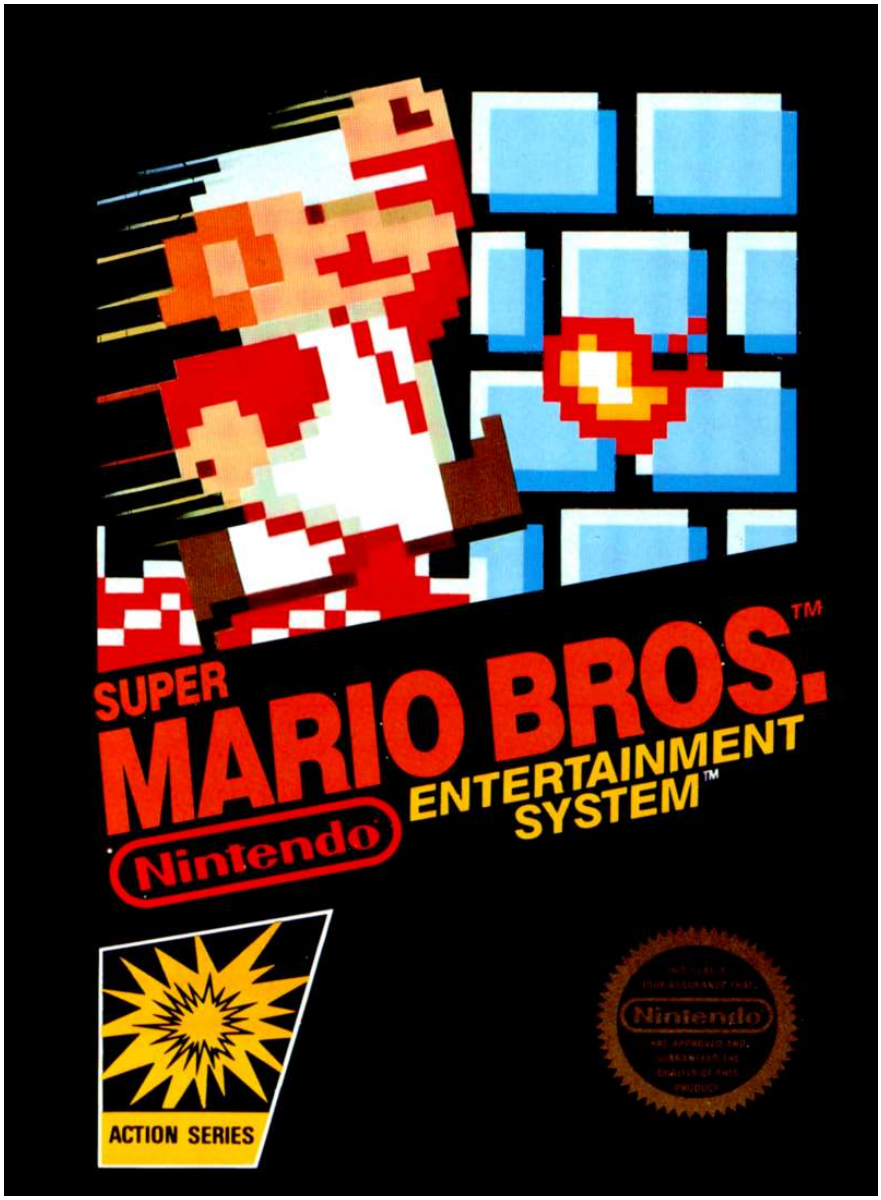
Every kid's dream!

This game lets you draw out your own Super Mario Brothers levels on the gamepad and then upload them to the internet so you friends can

play through them. You can include all the classic power-ups and enemies from the original Mario games in your levels. It's like a virtual LEGO set for people who grew up playing Super Mario Brothers.

Can we use the same model that generated fake Hemingway text to generate fake Super Mario Brothers levels?

First, we need a data set for training our model. Let's take all the outdoor levels from the original Super Mario Brothers game released in 1985:



Best Christmas Ever. Thanks Mom and Dad!

This game has 32 levels and about 70% of them have the same outdoor style. So we'll stick to those.

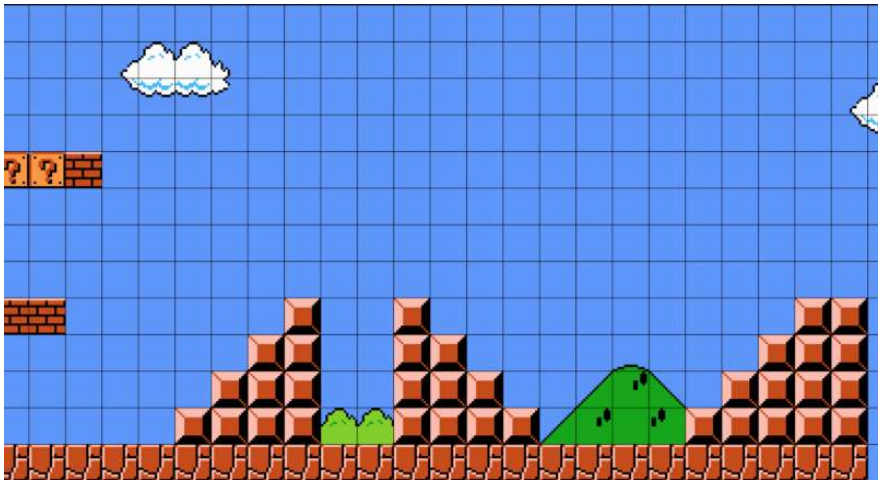
To get the designs for each level, I took an original copy of the game and wrote a program to pull the level designs out of the game's memory. Super Mario Bros. is a 30-year-old game and there are lots of resources online that help you figure out how the levels were stored in

the game's memory. Extracting level data from an old video game is a fun programming exercise that you should try sometime.

Here's the first level from the game (which you probably remember if you ever played it):

Super Mario Bros. Level 1–1

If we look closely, we can see the level is made of a simple grid of objects:



We could just as easily represent this grid as a sequence of characters with one character representing each object:

```
--------------------------
--------------------------
--------------------------
#??#----------------------
--------------------------
--------------------------
--------------------------
--------------------------
-##------=--=----------==-
--------==--==---------===-
-------===--=========----=-
------====--========-----=-
==========================-
```

We've replaced each object in the level with a letter:

- '-' is a blank space

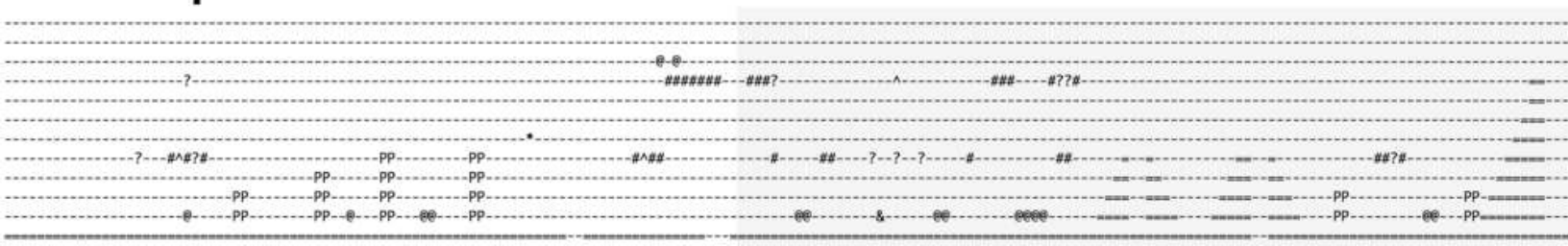
- '=' is a solid block
- '#' is a breakable brick
- '?' is a coin block

…and so on, using a different letter for each different kind of object in the level.

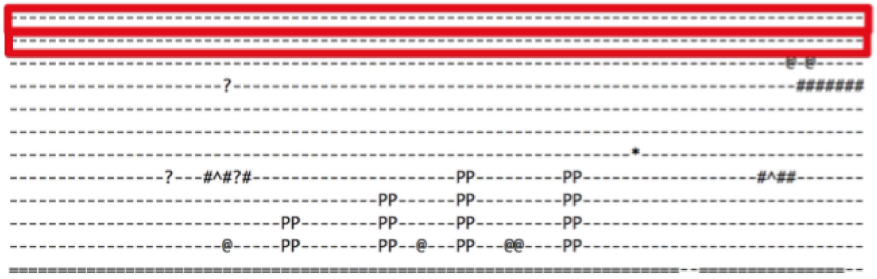
I ended up with text files that looked like this:

Original Level

Text Representation

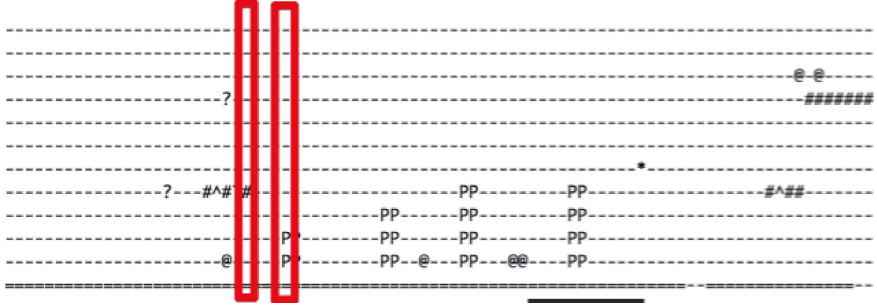


Looking at the text file, you can see that Mario levels don't really have much of a pattern if you read them line-by-line:



Reading line-by-line, there's not really a pattern to capture. Lots of lines are completely blank.

The patterns in a level really emerge when you think of the level as a series of columns:



Looking column-by-column, there's a real pattern. Each column ends in a '=' for example.

So in order for the algorithm to find the patterns in our data, we need to feed the data in column-by-column. Figuring out the most effective representation of your input data (called feature selection) is one of the keys of using machine learning algorithms well.

To train the model, I needed to rotate my text files by 90 degrees. This made sure the characters were fed into the model in an order where a pattern would more easily show up:

```
-----------=
-------#---=
-------#---=
-------?---=
-------#---=
-----------=
-----------=
----------@=
----------@=
-----------=
-----------=
-----------=
---------PP=
---------PP=
----------==
---------===
--------====
-------=====
------======
-----=======
---=========
----------
```

Training Our Model

Just like we saw when creating the model of Hemingway's prose, a model improves as we train it.

After a little training, our model is generating junk:

```
-------------------------
LL+<&=-------P-------------
--------
```

```
--------------------T--#--
-----
-===-=-=------------=-&--T--------------
--------------------
--=------$-=#-=-_
---------------=----=<----
-------b
-
```

It sort of has an idea that '-'s and '='s should show up a lot, but that's about it. It hasn't figured out the pattern yet.

After several thousand iterations, it's starting to look like something:

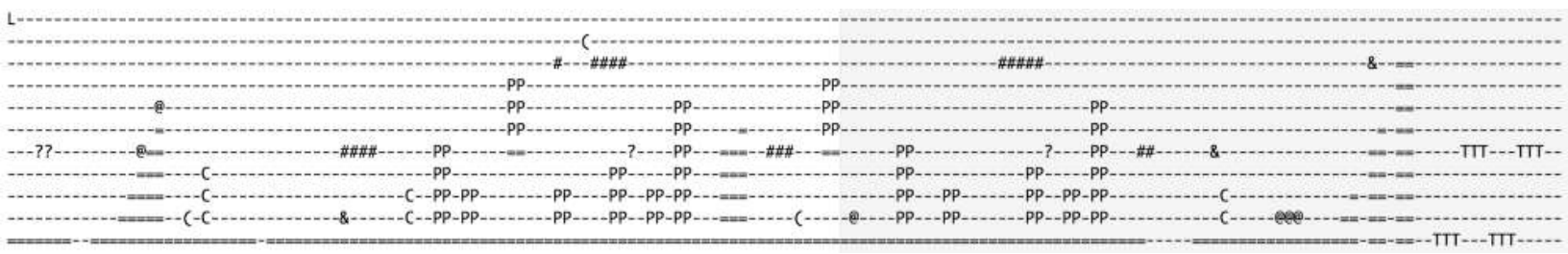
```
--
-----------=
----------=
--------PP=
--------PP=
-----------=
-----------=
-----------=
-------?---=
-----------=
-----------=
```

The model has almost figured out that each line should be the same length. It has even started to figure out some of the logic of Mario: The pipes in mario are always two blocks wide and at least two blocks high, so the "P"s in the data should appear in 2x2 clusters. That's pretty cool!

With a lot more training, the model gets to the point where it generates perfectly valid data:

```
--------PP=
--------PP=
----------=
----------=
----------=
---PPP=---=
---PPP=---=
----------=
```

Let's sample an entire level's worth of data from our model and rotate it back horizontal:



A whole level, generated from our model!

This data looks great! There are several awesome things to notice:

- It put a Lakitu (the monster that floats on a cloud) up in the sky at the beginning of the level—just like in a real Mario level.
- It knows that pipes floating in the air should be resting on top of solid blocks and not just hanging in the air.
- It places enemies in logical places.
- It doesn't create anything that would block a player from moving forward.
- It *feels* like a real level from Super Mario Bros. 1 because it's based off the style of the original levels that existed in that game.

Finally, let's take this level and recreate it in Super Mario Maker:

Our level data after being entered into Super Mario Maker

Play it yourself!

If you have Super Mario Maker, you can play this level by bookmarking it online or by looking it up using level code 4AC9–0000–0157-F3C3.

Toys vs. Real World Applications

The recurrent neural network algorithm we used to train our model is the same kind of algorithm used by real-world companies to solve hard problems like speech detection and language translation. What makes our model a 'toy' instead of cutting-edge is that our model is generated from very little data. There just aren't enough levels in the original Super Mario Brothers game to provide enough data for a really good model.

If we could get access to the hundreds of thousands of user-created Super Mario Maker levels that Nintendo has, we could make an amazing model. But we can't—because Nintendo won't let us have them. Big companies don't give away their data for free.

As machine learning becomes more important in more industries, the difference between a good program and a bad program will be how much data you have to train your models. That's why companies like Google and Facebook need your data so badly!

For example, Google recently open sourced TensorFlow, its software toolkit for building large-scale machine learning applications. It was a pretty big deal that Google gave away such important, capable technology for free. This is the same stuff that powers Google Translate.

But without Google's massive trove of data in every language, you can't create a competitor to Google Translate. Data is what gives Google its edge. Think about that the next time you open up your Google Maps Location History or Facebook Location History and notice that it stores every place you've ever been.

Further Reading

In machine learning, there's never a single way to solve a problem. You have limitless options when deciding how to pre-process your data and which algorithms to use. Often combining multiple approaches will give you better results than any single approach.

Readers have sent me links to other interesting approaches to generating Super Mario levels:

- Justin Michaud expanded on the approach I used here to generate levels and figured out how to hack his generated levels back into the original NES rom file (code written over 30 years ago)! You can even play his hacked rom online.

- Amy K. Hoover's team used an approach that represents each type of level object (pipes, ground, platforms, etc) as if it were single voice in an overall symphony. Using a process called functional scaffolding, the system can augment levels with blocks of any given object type. For example, you could sketch out the basic shape of a level and it could add in pipes and question blocks to complete your design.
- Steve Dahlskog's team showed that modeling each column of level data as a series of n-gram "words" makes it possible to generate levels with a much simpler algorithm than a large RNN.

. . .

If you liked this article, please consider **signing up for my Machine Learning is Fun! email list**. I'll only email you when I have something new and awesome to share. It's the best way to find out when I write more articles like this.

You can also follow me on Twitter at @ageitgey, email me directly or find me on linkedin. I'd love to hear from you if I can help you or your team with machine learning.

Now continue on to Machine Learning is Fun Part 3!