Machine Learning is Fun!

The world's easiest introduction to Machine Learning

Adam Geitgey
May 5, 2014 · 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 日本語, Português, Português (alternate), Türkçe, Français, 한국어, العَرَبِيَّة, Español (México), Español (España), Polski, Italiano, 普通话, Русский, 한국어, Tiê´ng Việt or* فارسی.

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!*

Have you heard people talking about machine learning but only have a fuzzy idea of what that means? Are you tired of nodding your way through conversations with co-workers? Let's change that!

. . .

This guide is for anyone who is curious about machine learning but has no idea where to start. I imagine there are a lot of people who tried reading the wikipedia article, got frustrated and gave up wishing someone would just give them a high-level explanation. That's what this is.

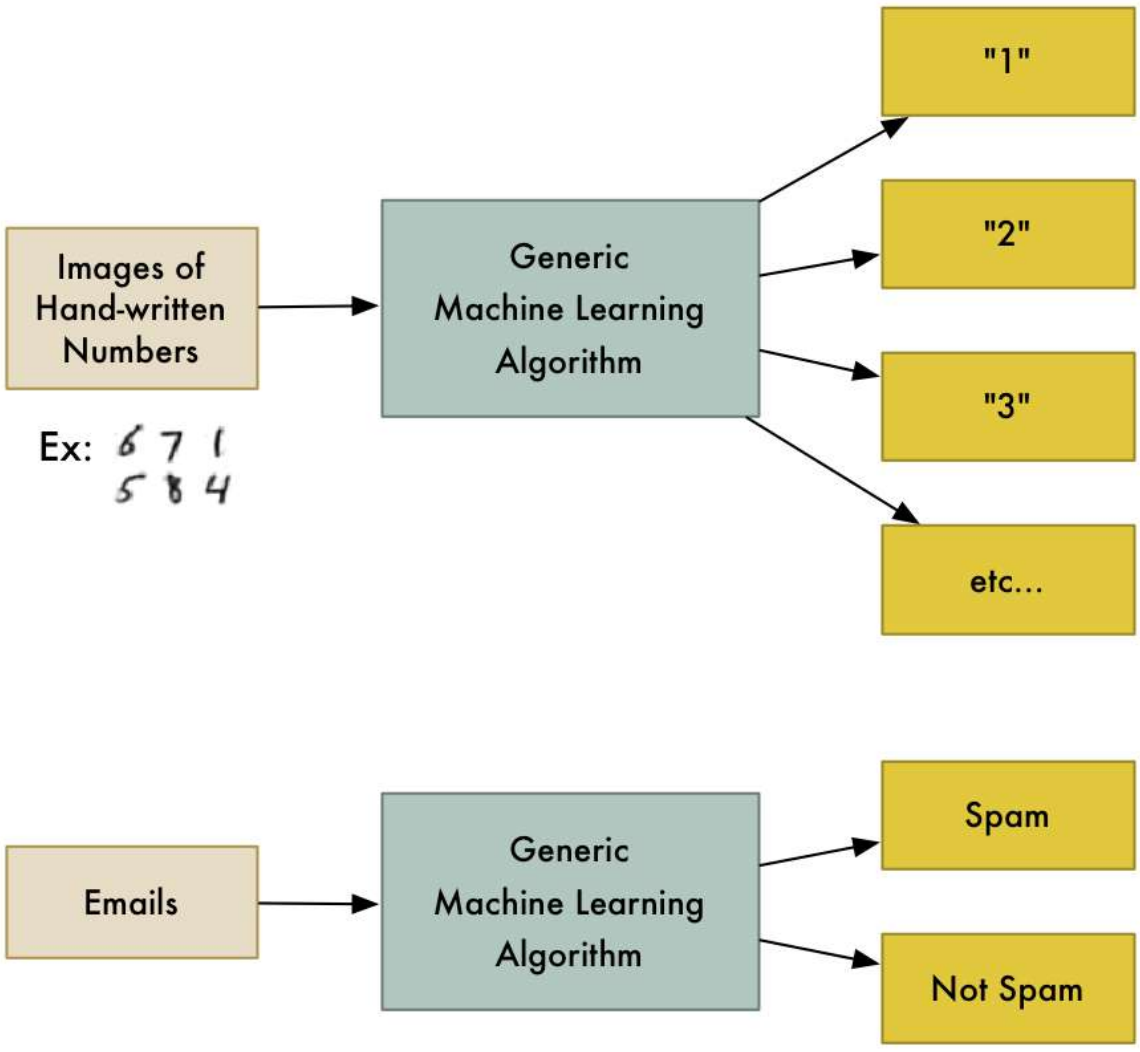
The goal is be accessible to anyone — which means that there's a lot of generalizations. But who cares? If this gets anyone more interested in ML, then mission accomplished.

. . .

What is machine learning?

Machine learning is the idea that there are generic algorithms that can tell you something interesting about a set of data without you having to write any custom code specific to the problem. Instead of writing code, you feed data to the generic algorithm and it builds its own logic based on the data.

For example, one kind of algorithm is a classification algorithm. It can put data into different groups. The same classification algorithm used to recognize handwritten numbers could also be used to classify emails into spam and not-spam without changing a line of code. It's the same algorithm but it's fed different training data so it comes up with different classification logic.



This machine learning algorithm is a black box that can be re-used for lots of different classification problems.

"Machine learning" is an umbrella term covering lots of these kinds of generic algorithms.

Two kinds of Machine Learning Algorithms

You can think of machine learning algorithms as falling into one of two main categories — **supervised learning** and **unsupervised learning**. The difference is simple, but really important.

Supervised Learning

Let's say you are a real estate agent. Your business is growing, so you hire a bunch of new trainee agents to help you out. But there's a problem — you can glance at a house and have a pretty good idea of what a house is worth, but your trainees don't have your experience so they don't know how to price their houses.

To help your trainees (and maybe free yourself up for a vacation), you decide to write a little app that can estimate the value of a house in your area based on it's size, neighborhood, etc, and what similar houses have sold for.

So you write down every time someone sells a house in your city for 3 months. For each house, you write down a bunch of details — number of bedrooms, size in square feet, neighborhood, etc. But most importantly, you write down the final sale price:

Bedrooms	Sq. feet	Neighborhood	Sale price
3	2000	Normaltown	$250,000
2	800	Hipsterton	$300,000
2	850	Normaltown	$150,000
1	550	Normaltown	$78,000
4	2000	Skid Row	$150,000

This is our "training data."

Using that training data, we want to create a program that can estimate how much any other house in your area is worth:

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

We want to use the training data to predict the prices of other houses.

This is called **supervised learning**. You knew how much each house sold for, so in other words, you knew the answer to the problem and could work backwards from there to figure out the logic.

To build your app, you feed your training data about each house into your machine learning algorithm. The algorithm is trying to figure out what kind of math needs to be done to make the numbers work out.

This kind of like having the answer key to a math test with all the arithmetic symbols erased:

Math Quiz #1 - Teacher's Answer Key

1) 2 4 5 = 3
2) 5 2 8 = 2
3) 2 2 1 = 3
4) 4 2 2 = 6
5) 6 2 2 = 10
6) 3 1 1 = 2
7) 5 3 4 = 11
8) 1 8 1 = 7

Oh no! A devious student erased the arithmetic symbols from the teacher's answer key!

From this, can you figure out what kind of math problems were on the test? You know you are supposed to "do something" with the numbers on the left to get each answer on the right.

In **supervised learning**, you are letting the computer work out that relationship for you. And once you know what math was required to solve this specific set of problems, you could answer to any other problem of the same type!

Unsupervised Learning

Let's go back to our original example with the real estate agent. What if you didn't know the sale price for each house? Even if all you know is the size, location, etc of each house, it turns out you can still do some really cool stuff. This is called **unsupervised** learning.

Bedrooms	Sq. feet	Neighborhood
3	2000	Normaltown
2	800	Hipsterton
2	850	Normaltown
1	550	Normaltown
4	2000	Skid Row

Even if you aren't trying to predict an unknown number (like price), you can still do interesting things with machine learning.

This is kind of like someone giving you a list of numbers on a sheet of paper and saying "I don't really know what these numbers mean but maybe you can figure out if there is a pattern or grouping or something — good luck!"

So what could do with this data? For starters, you could have an algorithm that automatically identified different market segments in your data. Maybe you'd find out that home buyers in the neighborhood near the local college really like small houses with lots of bedrooms, but home buyers in the suburbs prefer 3-bedroom houses with

lots of square footage. Knowing about these different kinds of customers could help direct your marketing efforts.

Another cool thing you could do is automatically identify any outlier houses that were way different than everything else. Maybe those outlier houses are giant mansions and you can focus your best sales people on those areas because they have bigger commissions.

Supervised learning is what we'll focus on for the rest of this post, but that's not because unsupervised learning is any less useful or interesting. In fact, unsupervised learning is becoming increasingly important as the algorithms get better because it can be used without having to label the data with the correct answer.

Side note: There are lots of other types of machine learning algorithms. But this is a pretty good place to start.

That's cool, but does being able to estimate the price of a house really count as "learning"?

As a human, your brain can approach most any situation and learn how to deal with that situation without any explicit instructions. If you sell houses for a long time, you will instinctively have a "feel" for the right price for a house, the best way to market that house, the kind of client who would be interested, etc. The goal of Strong AI research is to be able to replicate this ability with computers.

But current machine learning algorithms aren't that good yet — they only work when focused a very specific, limited problem. Maybe a better definition for "learning" in this case is "figuring out an equation to solve a specific problem based on some example data".

Unfortunately *"Machine Figuring out an equation to solve a specific problem based on some example data"* isn't really a great name. So we ended up with "Machine Learning" instead.

Of course if you are reading this 50 years in the future and we've figured out the algorithm for Strong AI, then this whole post will all seem a little quaint. Maybe stop reading and go tell your robot servant to go make you a sandwich, future human.

Let's write that program!

So, how would you write the program to estimate the value of a house like in our example above? Think about it for a second before you read further.

If you didn't know anything about machine learning, you'd probably try to write out some basic rules for estimating the price of a house like this:

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

  # In my area, the average house costs $200 per sqft
  price_per_sqft = 200

  if neighborhood == "hipsterton":
    # but some areas cost a bit more
    price_per_sqft = 400

  elif neighborhood == "skid row":
    # and some areas cost less
    price_per_sqft = 100

  # start with a base price estimate based on how big the place is
  price = price_per_sqft * sqft

  # now adjust our estimate based on the number of bedrooms
  if num_of_bedrooms == 0:
    # Studio apartments are cheap
    price = price—20000
  else:
    # places with more bedrooms are usually
    # more valuable
    price = price + (num_of_bedrooms * 1000)

 return price
```

If you fiddle with this for hours and hours, you might end up with something that sort of works. But your program will never be perfect and it will be hard to maintain as prices change.

Wouldn't it be better if the computer could just figure out how to implement this function for you? Who cares what exactly the function does as long is it returns the correct number:

```
def estimate_house_sales_price(num_of_bedrooms, sqft, neighborhood):
  price = <computer, plz do some math for me>
```

```
    return price
```

One way to think about this problem is that the **price** is a delicious stew and the ingredients are the **number of bedrooms**, the **square footage** and the **neighborhood**. If you could just figure out how much each ingredient impacts the final price, maybe there's an exact ratio of ingredients to stir in to make the final price.

That would reduce your original function (with all those crazy *if*'s and *else*'s) down to something really simple like this:

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

 # a little pinch of this
 price += num_of_bedrooms * .841231951398213

 # and a big pinch of that
 price += sqft * 1231.1231231

 # maybe a handful of this
 price += neighborhood * 2.3242341421

 # and finally, just a little extra salt for good measure
 price += 201.23432095

 return price
```

Notice the magic numbers in bold — **.841231951398213**, **1231.1231231**, **2.3242341421**, and **201.23432095**. These are our **weights**. If we could just figure out the perfect weights to use that work for every house, our function could predict house prices!

A dumb way to figure out the best weights would be something like this:

Step 1:

Start with each weight set to **1.0:**

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

  # a little pinch of this
  price += num_of_bedrooms * 1.0
```

```
    # and a big pinch of that
    price += sqft * 1.0

    # maybe a handful of this
    price += neighborhood * 1.0

    # and finally, just a little extra salt for good measure
    price += 1.0

    return price
```

Step 2:

Run every house you know about through your function and see how far off the function is at guessing the correct price for each house:

Bedrooms	Sq. feet	Neighborhood	Sale price	My Guess
3	2000	Normaltown	$250,000	$178,000
2	800	Hipsterton	$300,000	$371,000
2	850	Normaltown	$150,000	$148,000
1	550	Normaltown	$78,000	$101,000
4	2000	Skid Row	$150,000	$121,000

Use your function to predict a price for each house.

For example, if the first house really sold for $250,000, but your function guessed it sold for $178,000, you are off by $72,000 for that single house.

Now add up the squared amount you are off for each house you have in your data set. Let's say that you had 500 home sales in your data set and the square of how much your function was off for each house was a grand total of $86,123,373. That's how "wrong" your function currently is.

Now, take that sum total and divide it by 500 to get an average of how far off you are for each house. Call this average error amount the **cost** of your function.

If you could get this cost to be zero by playing with the weights, your function would be perfect. It would mean that in every case, your function perfectly guessed the price of the house based on the input data. So that's our goal — get this cost to be as low as possible by trying different weights.

Step 3:

Repeat Step 2 over and over with **every single possible combination of weights**. Whichever combination of weights makes the cost closest to zero is what you use. When you find the weights that work, you've solved the problem!

Mind Blowage Time

That's pretty simple, right? Well think about what you just did. You took some data, you fed it through three generic, really simple steps, and you ended up with a function that can guess the price of any house in your area. Watch out, Zillow!

But here's a few more facts that will blow your mind:

1. Research in many fields (like linguistics/translation) over the last 40 years has shown that these generic learning algorithms that "stir the number stew" (a phrase I just made up) out-perform approaches where real people try to come up with explicit rules themselves. The "dumb" approach of machine learning eventually beats human experts.

2. The function you ended up with is totally dumb. It doesn't even know what "square feet" or "bedrooms" are. All it knows is that it needs to stir in some amount of those numbers to get the correct answer.

3. It's very likely you'll have no idea ***why*** a particular set of weights will work. So you've just written a function that you don't really understand but that you can prove will work.

4. Imagine that instead of taking in parameters like "sqft" and "num_of_bedrooms", your prediction function took in an array of numbers. Let's say each number represented the brightness of one pixel in an image captured by camera mounted on top of your car. Now let's say that instead of outputting a prediction called "price", the function outputted a prediction called "degrees_to_turn_steering_wheel". **You've just made a function that can steer your car by itself!**

Pretty crazy, right?

What about that whole "try every number" bit in Step 3?

Ok, of course you can't just try every combination of all possible weights to find the combo that works the best. That would literally take forever since you'd never run out of numbers to try.

To avoid that, mathematicians have figured out lots of clever ways to quickly find good values for those weights without having to try very many. Here's one way:

First, write a simple equation that represents Step #2 above:

$$\text{Cost} = \frac{\sum_{i=1}^{500} \left(\text{MyGuess}(i) - \text{RealAnswer}(i)\right)^2}{500 \cdot 2}$$

This is your **cost function**.

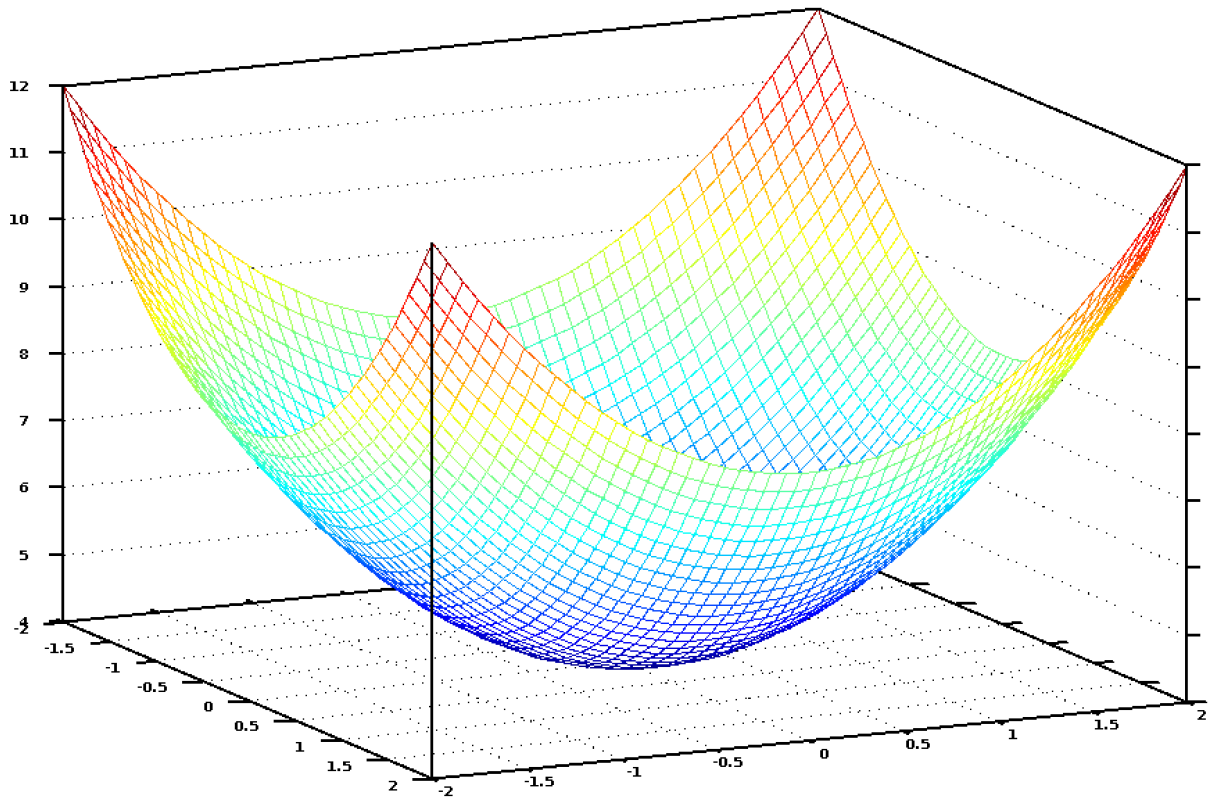
Now let's re-write exactly the same equation, but using a bunch of machine learning math jargon (that you can ignore for now):

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right)^2$$

θ is what represents your current weights. J(θ) means the 'cost for your current weights'.

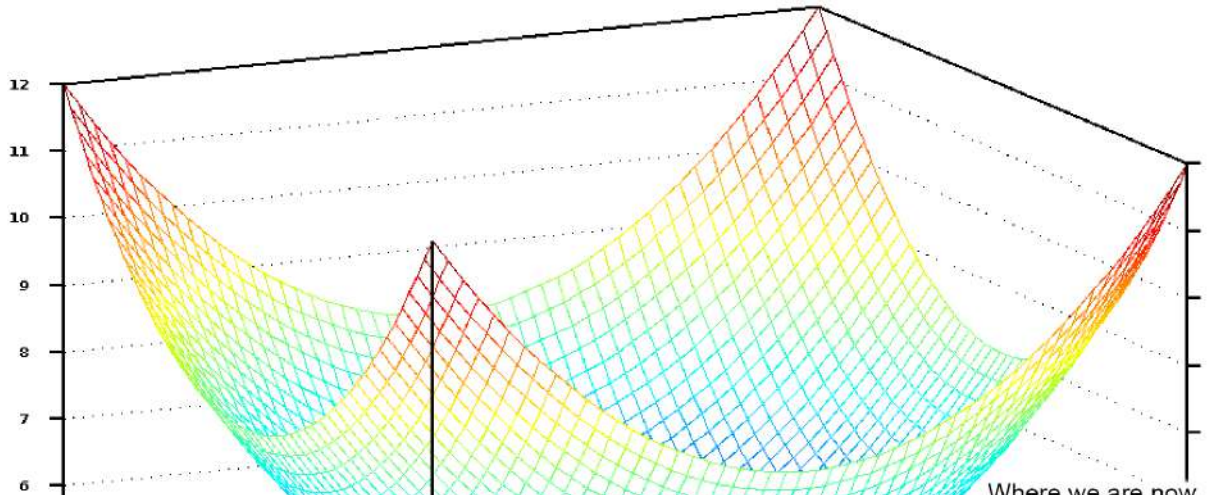
This equation represents how wrong our price estimating function is for the weights we currently have set.

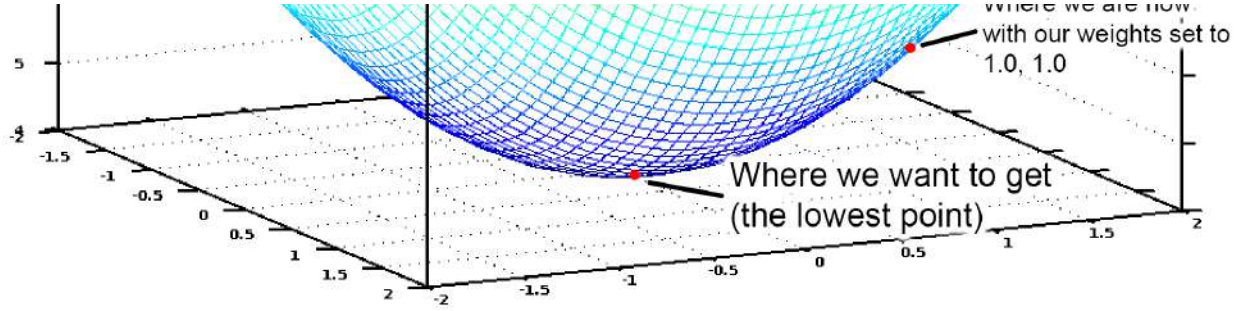
If we graph this cost equation for all possible values of our weights for **number_of_bedrooms** and **sqft**, we'd get a graph that might look something like this:



The graph of our cost function looks like a bowl. The vertical axis represents the cost.

In this graph, the lowest point in blue is where our cost is the lowest — thus our function is the least wrong. The highest points are where we are most wrong. So if we can find the weights that get us to the lowest point on this graph, we'll have our answer!





So we just need to adjust our weights so we are “walking down hill” on this graph towards the lowest point. If we keep making small adjustments to our weights that are always moving towards the lowest point, we’ll eventually get there without having to try too many different weights.

If you remember anything from Calculus, you might remember that if you take the derivative of a function, it tells you the slope of the function’s tangent at any point. In other words, it tells us which way is downhill for any given point on our graph. We can use that knowledge to walk downhill.

So if we calculate a partial derivative of our cost function with respect to each of our weights, then we can subtract that value from each weight. That will walk us one step closer to the bottom of the hill. Keep doing that and eventually we’ll reach the bottom of the hill and have the best possible values for our weights. (If that didn’t make sense, don’t worry and keep reading).

That’s a high level summary of one way to find the best weights for your function called **batch gradient descent**. Don’t be afraid to dig deeper if you are interested on learning the details.

When you use a machine learning library to solve a real problem, all of this will be done for you. But it’s still useful to have a good idea of what is happening.

What else did you conveniently skip over?

The three-step algorithm I described is called **multivariate linear regression**. You are estimating the equation for a line that fits through all of your house data points. Then you are using that equation to guess the sales price of houses you’ve never seen before based where that house would appear on your line. It’s a really powerful idea and you can solve “real” problems with it.

But while the approach I showed you might work in simple cases, it won't work in all cases. One reason is because house prices aren't always simple enough to follow a continuous line.

But luckily there are lots of ways to handle that. There are plenty of other machine learning algorithms that can handle non-linear data (like neural networks or SVMs with kernels). There are also ways to use linear regression more cleverly that allow for more complicated lines to be fit. In all cases, the same basic idea of needing to find the best weights still applies.

Also, I ignored the idea of **overfitting**. It's easy to come up with a set of weights that always works perfectly for predicting the prices of the houses in your original data set but never actually works for any new houses that weren't in your original data set. But there are ways to deal with this (like regularization and using a cross-validation data set). Learning how to deal with this issue is a key part of learning how to apply machine learning successfully.

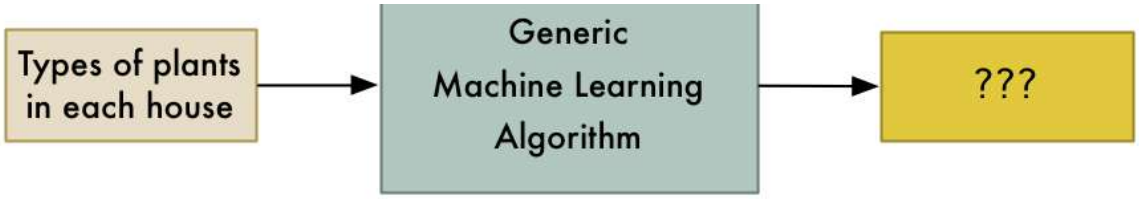
In other words, while the basic concept is pretty simple, it takes some skill and experience to apply machine learning and get useful results. But it's a skill that any developer can learn!

Is machine learning magic?

Once you start seeing how easily machine learning techniques can be applied to problems that seem really hard (like handwriting recognition), you start to get the feeling that you could use machine learning to solve any problem and get an answer as long as you have enough data. Just feed in the data and watch the computer magically figure out the equation that fits the data!

But it's important to remember that machine learning only works if the problem is actually solvable with the data that you have.

For example, if you build a model that predicts home prices based on the type of potted plants in each house, it's never going to work. There just isn't any kind of relationship between the potted plants in each house and the home's sale price. So no matter how hard it tries, the computer can never deduce a relationship between the two.



You can only model relationships that actually exist.

So remember, if a human expert couldn't use the data to solve the problem manually, a computer probably won't be able to either. Instead, focus on problems where a human could solve the problem, but where it would be great if a computer could solve it much more quickly.

How to learn more about Machine Learning

In my mind, the biggest problem with machine learning right now is that it mostly lives in the world of academia and commercial research groups. There isn't a lot of easy to understand material out there for people who would like to get a broad understanding without actually becoming experts. But it's getting a little better every day.

If you want to try out what you've learned in this article, I made a course that walks you through every step of this article, including writing all the code. Give it a try!

If you want to go deeper, Andrew Ng's free Machine Learning class on Coursera is pretty amazing as a next step. I highly recommend it. It should be accessible to anyone who has a Comp. Sci. degree and who remembers a very minimal amount of math.

Also, you can play around with tons of machine learning algorithms by downloading and installing SciKit-Learn. It's a python framework that has "black box" versions of all the standard algorithms.

. . .

If you liked this article, please consider signing up for my Machine Learning is Fun! Newsletter:

Also, please check out the **full-length course version of this article**. It covers everything in this article in more detail, including writing the actual code in Python. You can get a free 30-day trial to watch the course if you sign up with this link.

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 2!

Machine Learning

Machine Learning is Fun! Part 2

Using Machine Learning to generate Super Mario Maker levels

Adam Geitgey
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 Việt 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:

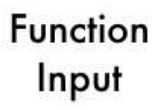
Bedrooms	Sq. feet	Neighborhood	Sale price
3	2000	Hipsterton	???

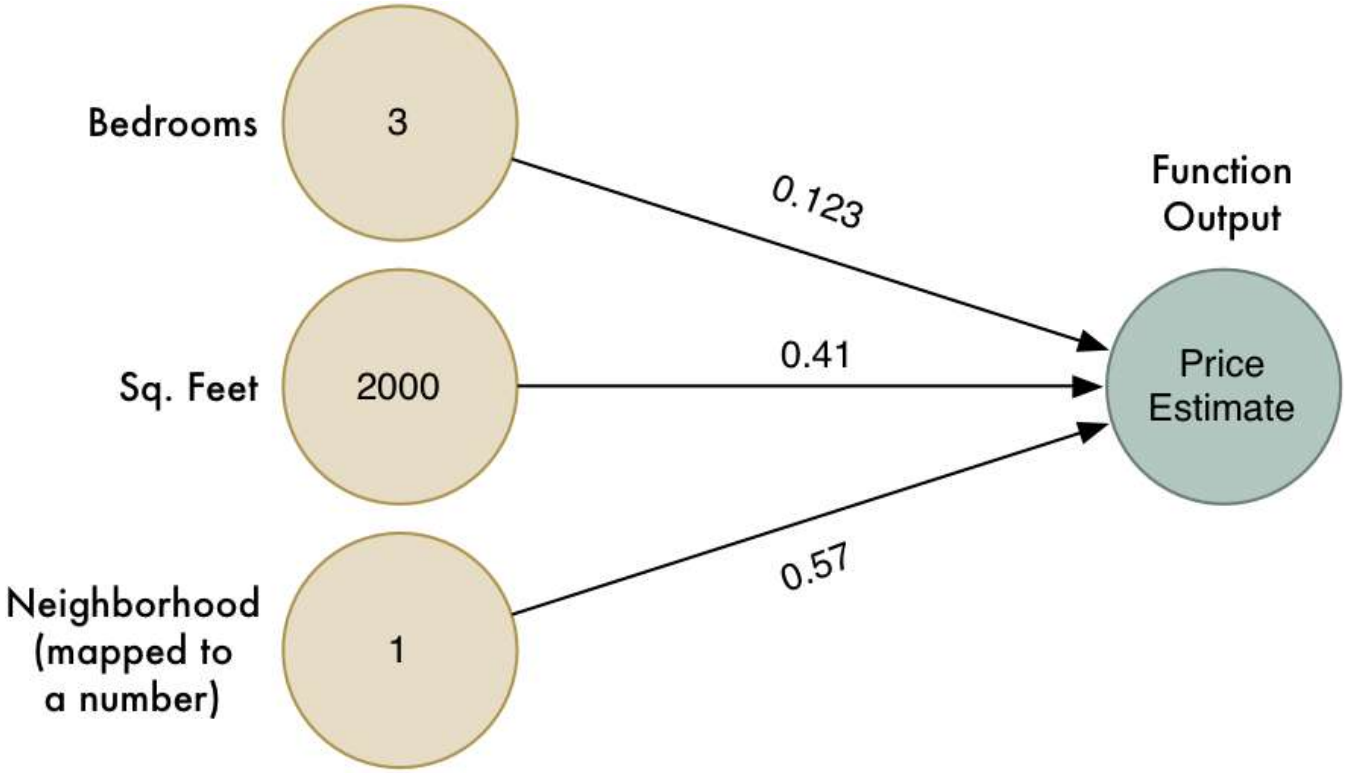
We ended up with this simple estimation function:

```
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:

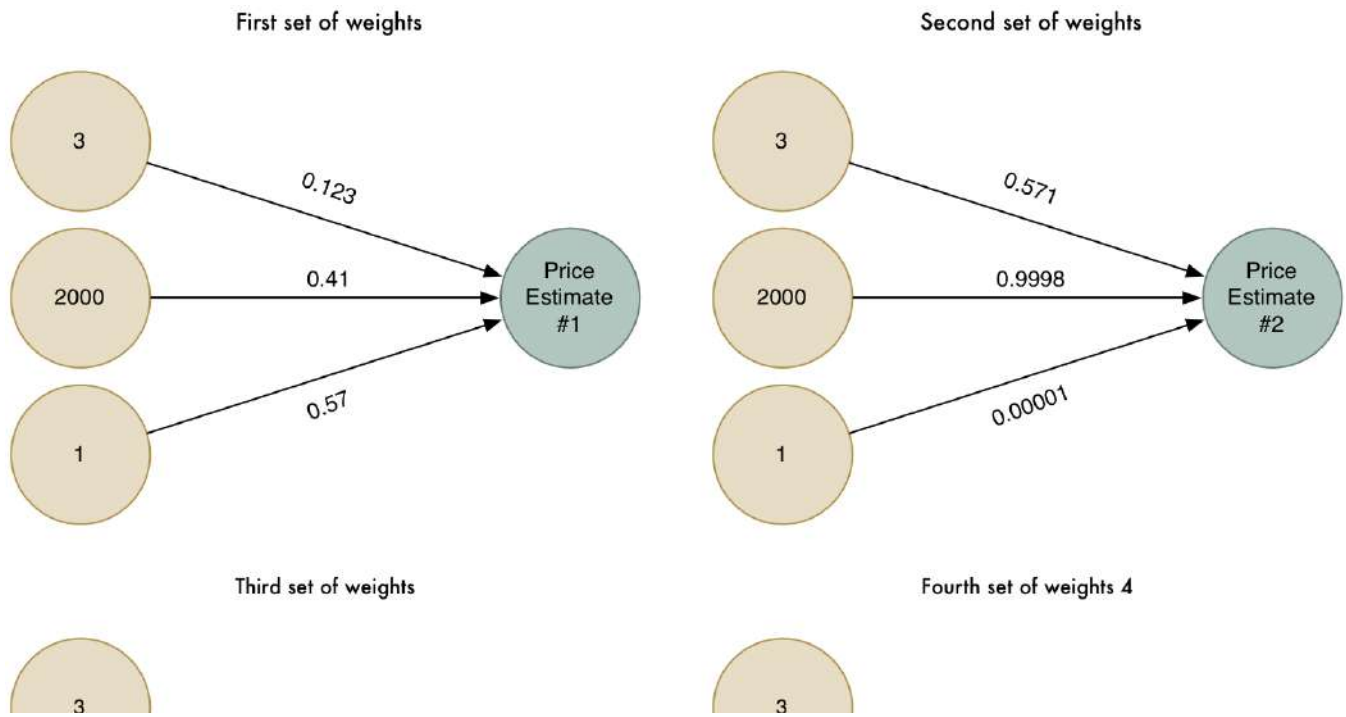


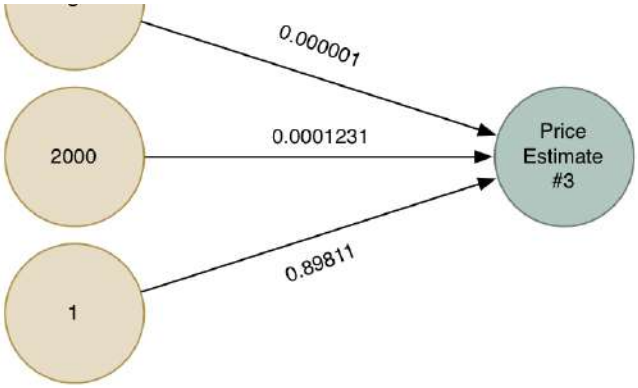


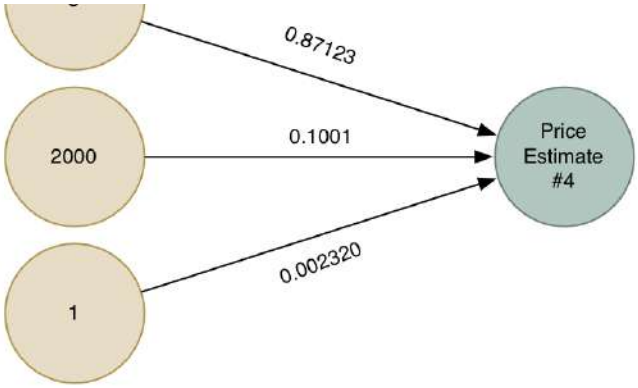
The arrows represent the weights in our function.

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:







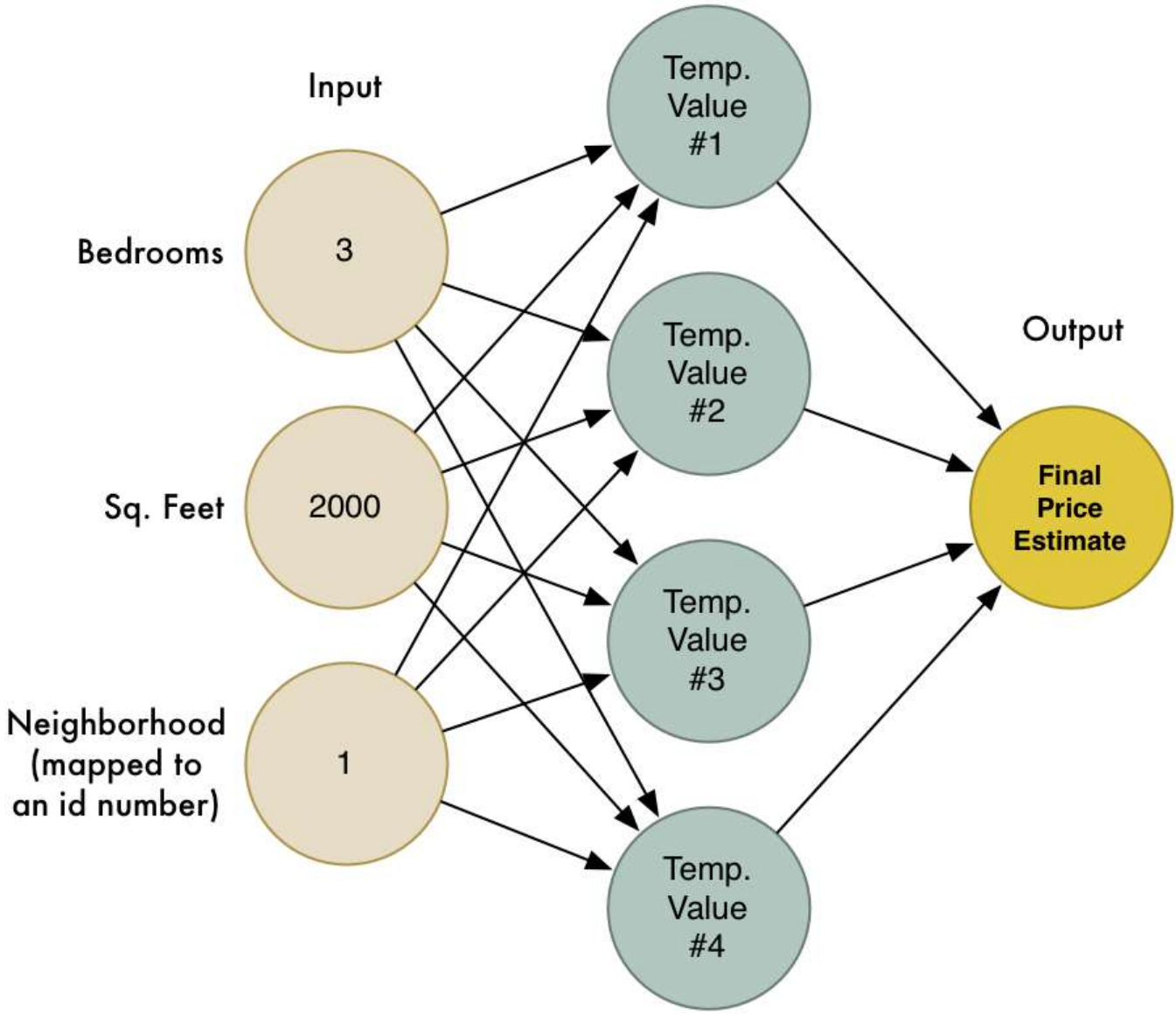
Let's try solving the problem four different ways

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:

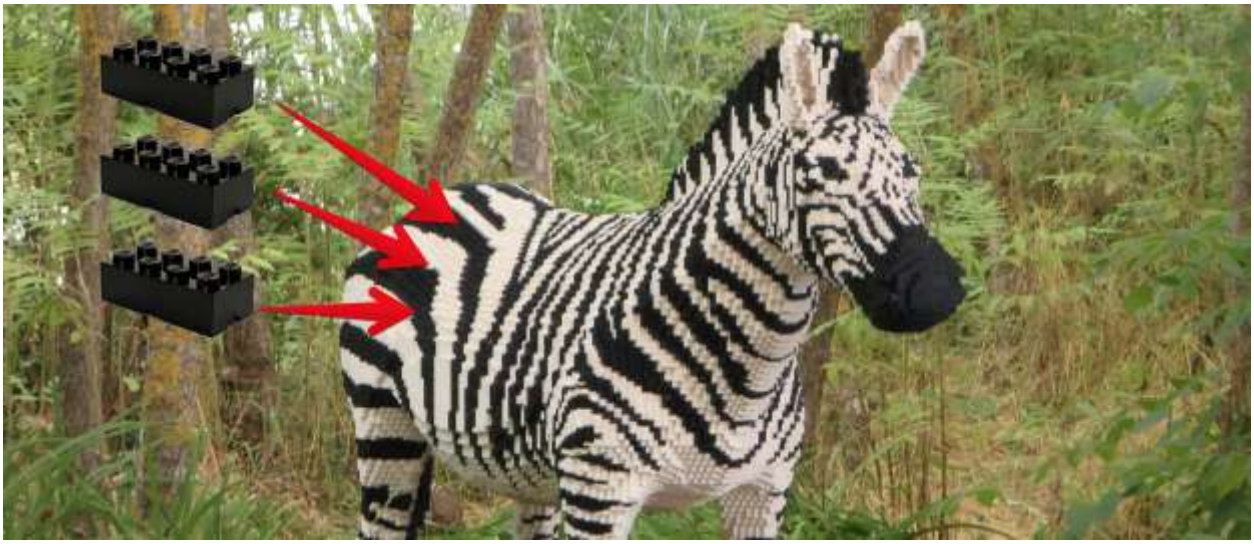


This is a neural network! Each node knows how to take in a set of inputs, apply weights to them, and calculate an output value. By chaining together lots of these nodes, we can model complex functions.

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

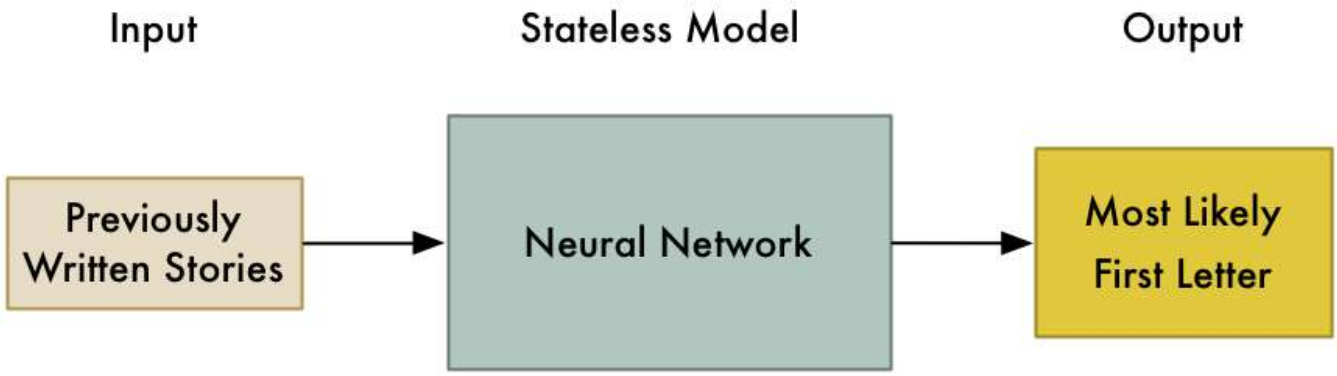
The neural network we've seen always returns the same answer when you give it the same inputs. It has no memory. In programming terms, it's a *stateless algorithm*.

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:

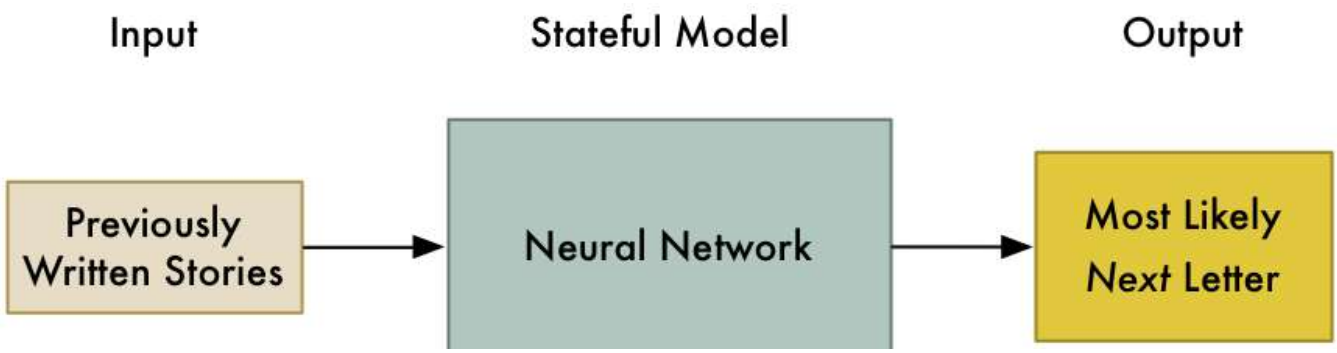
> *Robert Cohn was once middleweight boxi*

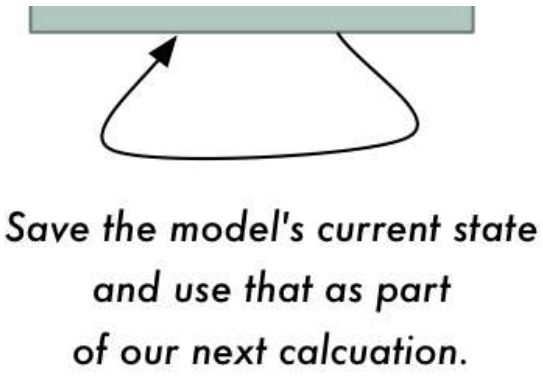
What letter is going to come next?

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.





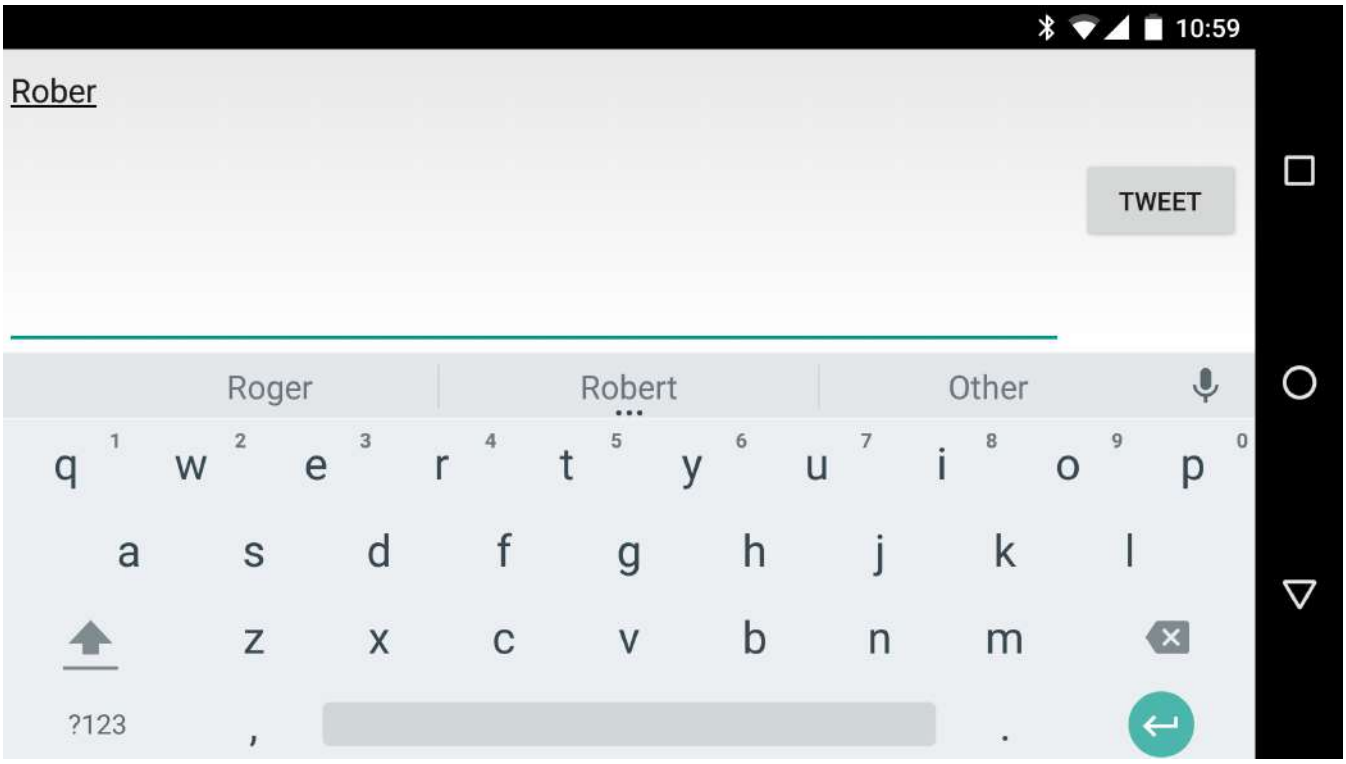
Keeping track of state in our model makes it possible to not just predict the most likely *first* letter in the story, but to predict the most likely *next* letter given all previous letters.

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.

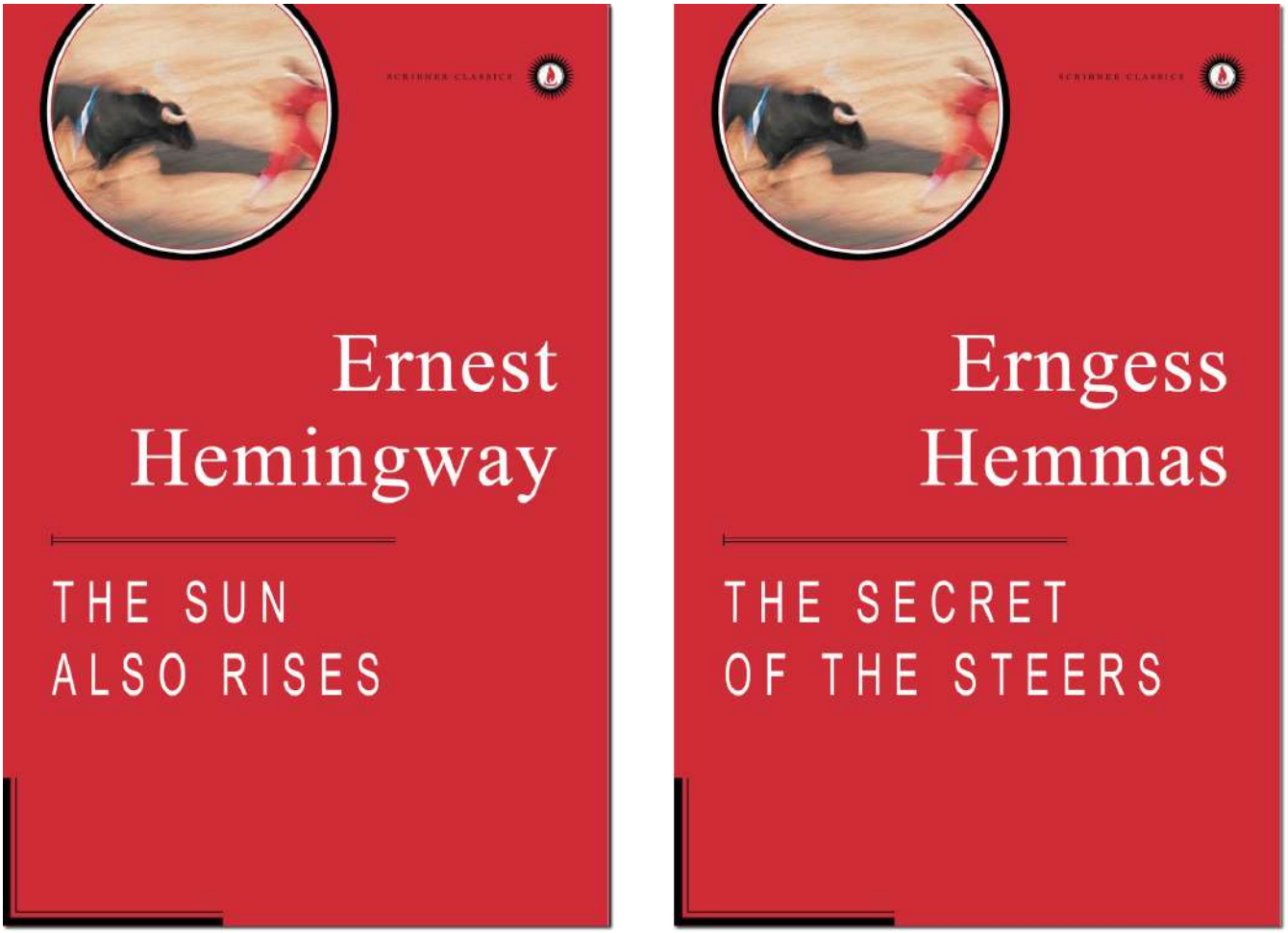
Compare that with some real text from the book:

> *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

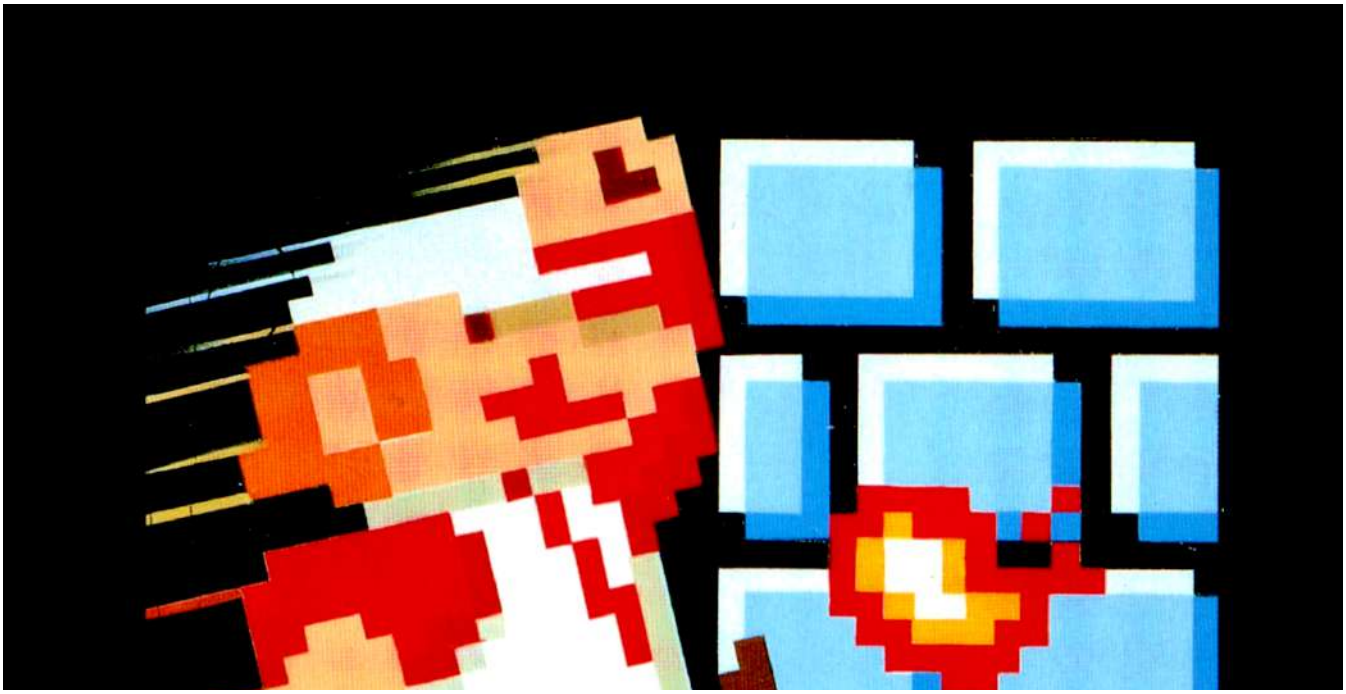
In 2015, Nintendo released Super Mario Maker™ for the Wii U gaming system.

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!

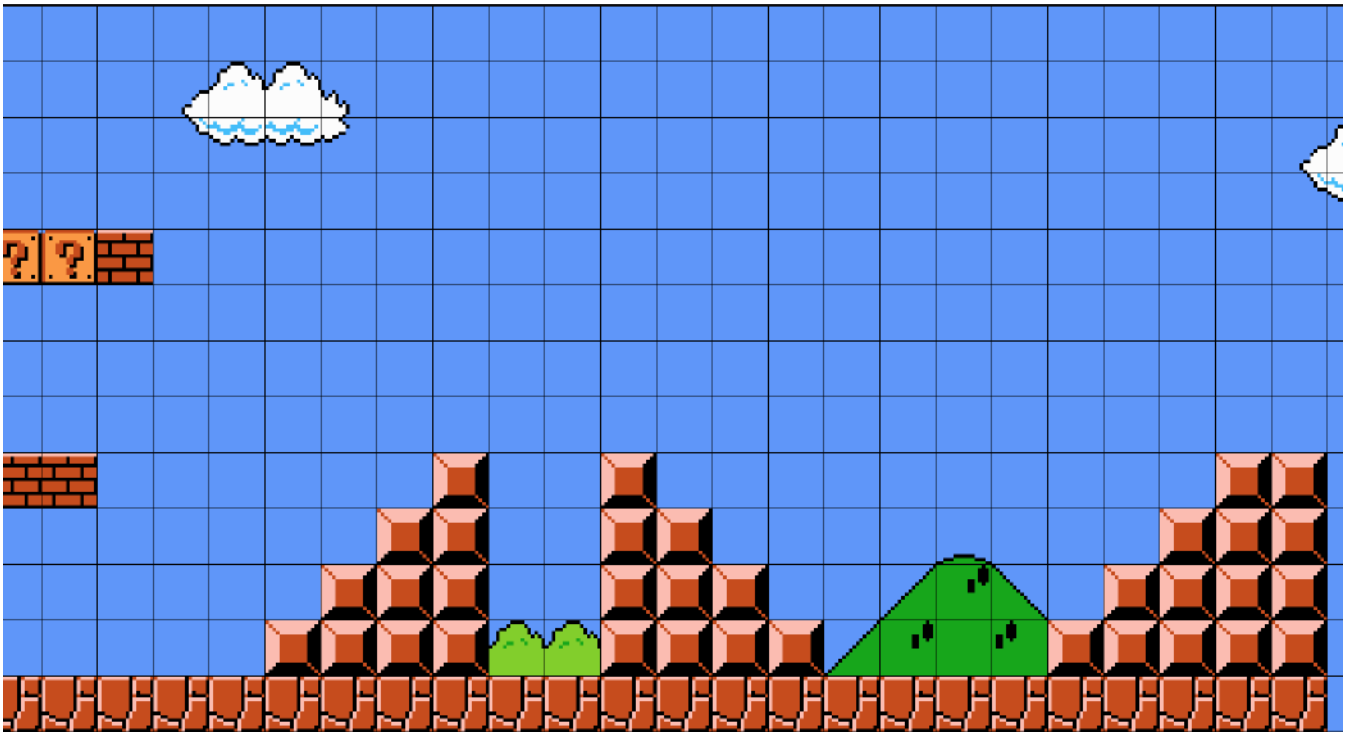
This game has 32 levels and about 70% of them have the same outdoor style. So we'll stick to those.

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:

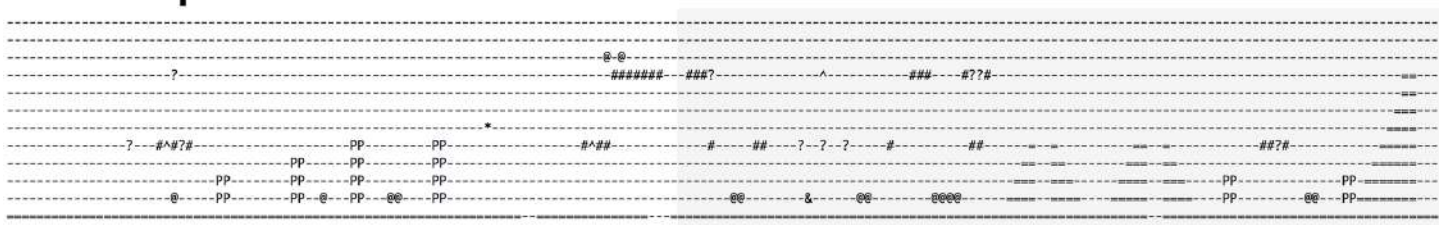
- '-' is a blank space
- '=' is a solid block
- '#' is a breakable brick
- '?' is a coin block

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

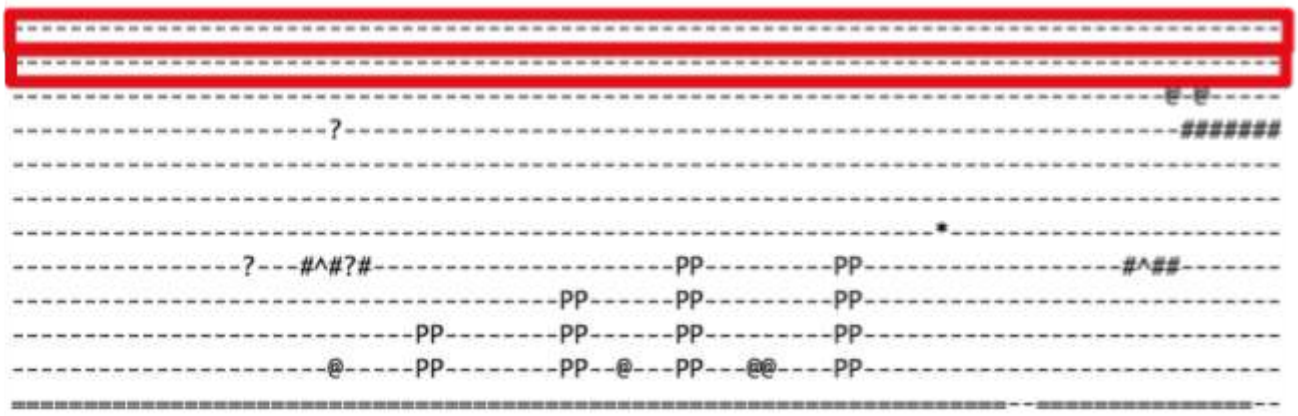
I ended up with text files that looked like this:

Original Level

Text Representation

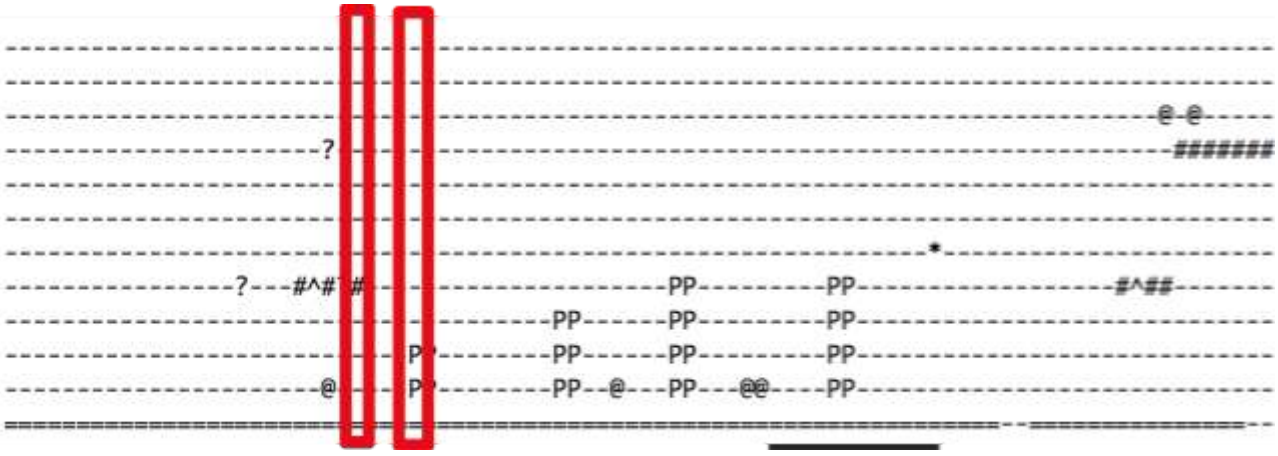


Looking at the text file, you can see that Mario levels don't really have much of a pattern if you read them line-by-line:



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:

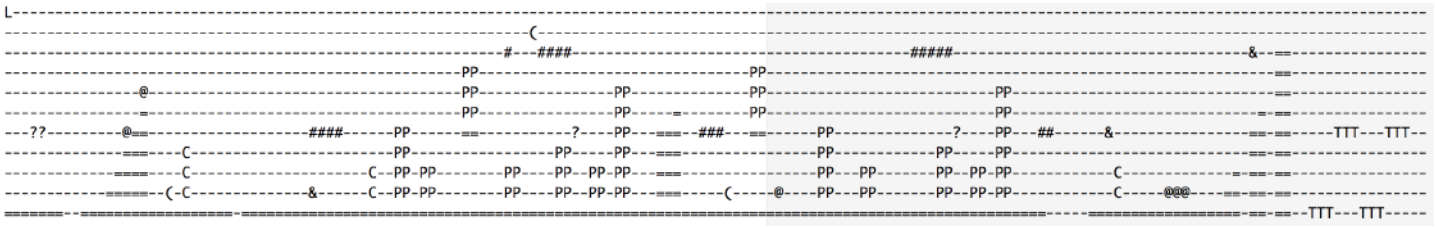
```
--
-----------=
----------=
--------PP=
--------PP=
-----------=
-----------=
-----------=
-------?---=
-----------=
-----------=
```

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

SHOW EMBED

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!

Artificial Intelligence Machine Learning Nintendo

Machine Learning is Fun! Part 3: Deep Learning and Convolutional Neural Networks

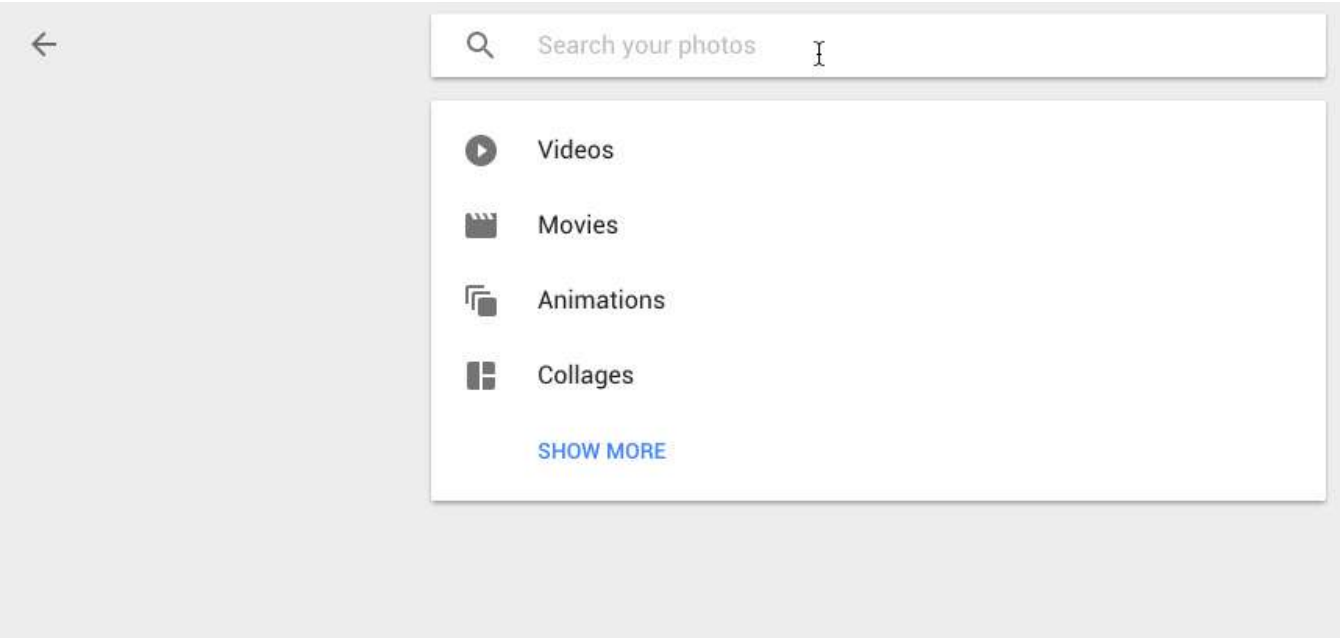
Adam Geitgey
Jun 13, 2016 · 16 min read

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 普通话, Русский, 한국어, Português, Tiê´ng Việt or Italiano.*

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!*

Are you tired of reading endless news stories about *deep learning* and not really knowing what that means? Let's change that!

This time, we are going to learn how to write programs that recognize objects in images using deep learning. In other words, we're going to explain the black magic that allows Google Photos to search your photos based on what is in the picture:



Google now lets you search your own photos by description — even if they aren't tagged! How does this work??

Just like Part 1 and Part 2, 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!

(If you haven't already read part 1 and part 2, read them now!)

. . .

Recognizing Objects with Deep Learning

xkcd #1425 (View original here)

You might have seen this famous xkcd comic before.

The goof is based on the idea that any 3-year-old child can recognize a photo of a bird, but figuring out how to make a computer recognize objects has puzzled the very best computer scientists for over 50 years.

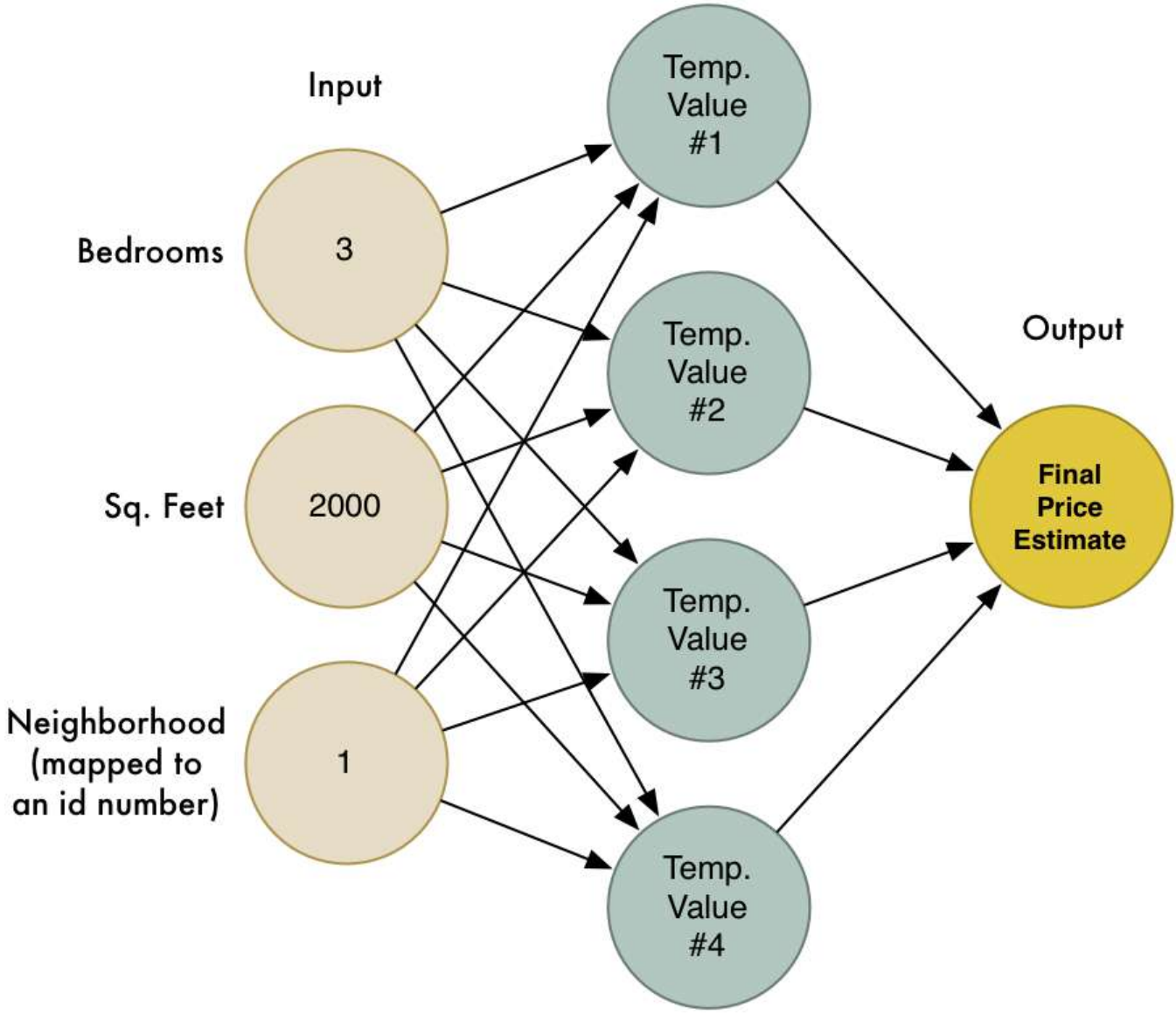
In the last few years, we've finally found a good approach to object recognition using *deep convolutional neural networks*. That sounds like a a bunch of made up words from a William Gibson Sci-Fi novel, but the ideas are totally understandable if you break them down one by one.

So let's do it — let's write a program that can recognize birds!

Starting Simple

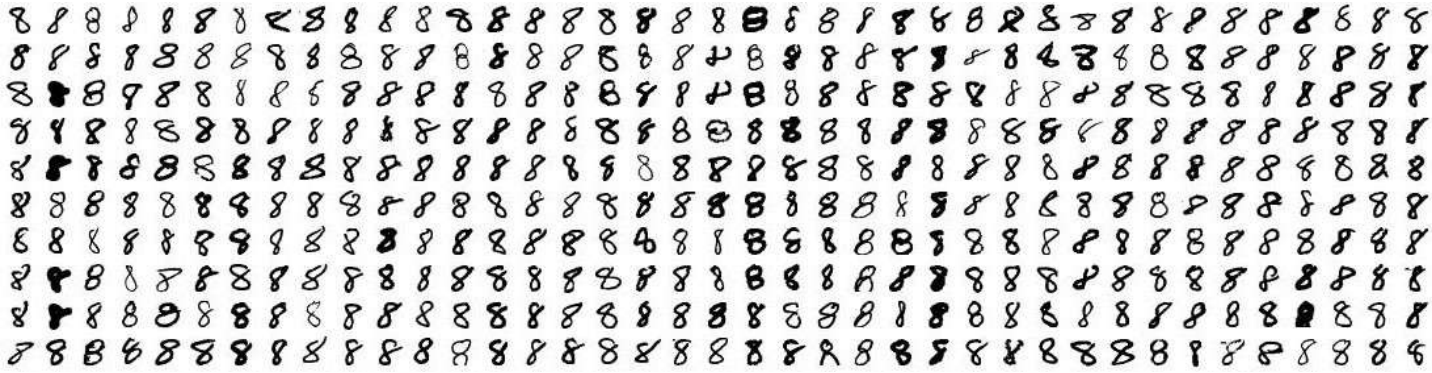
Before we learn how to recognize pictures of birds, let's learn how to recognize something much simpler — the handwritten number "8".

In Part 2, we learned about how neural networks can solve complex problems by chaining together lots of simple neurons. We created a small neural network to estimate the price of a house based on how many bedrooms it had, how big it was, and which neighborhood it was in:



We also know that the idea of machine learning is that the same generic algorithms can be reused with different data to solve different problems. So let's modify this same neural network to recognize handwritten text. But to make the job really simple, we'll only try to recognize one letter — the numeral "8".

Machine learning only works when you have data — preferably a lot of data. So we need lots and lots of handwritten "8"s to get started. Luckily, researchers created the MNIST data set of handwritten numbers for this very purpose. MNIST provides 60,000 images of handwritten digits, each as an 18x18 image. Here are some "8"s from the data set:

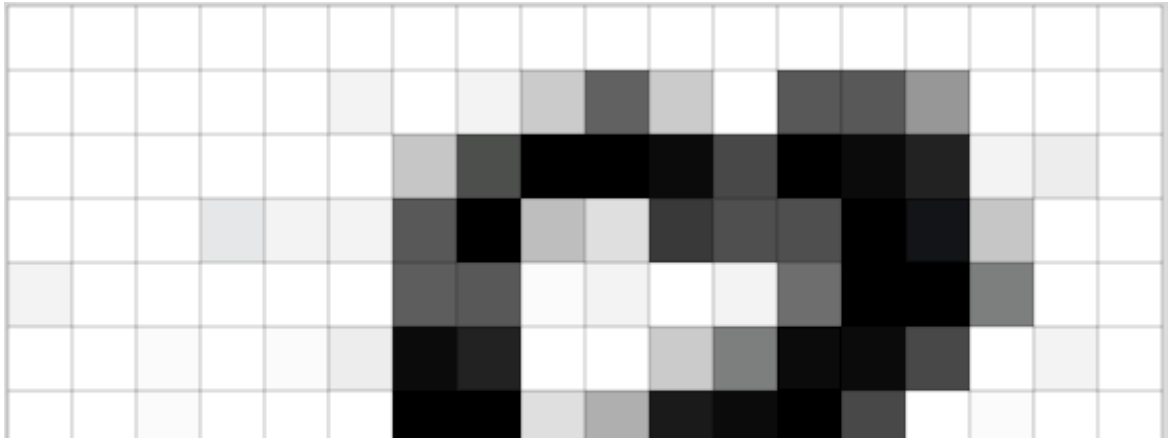


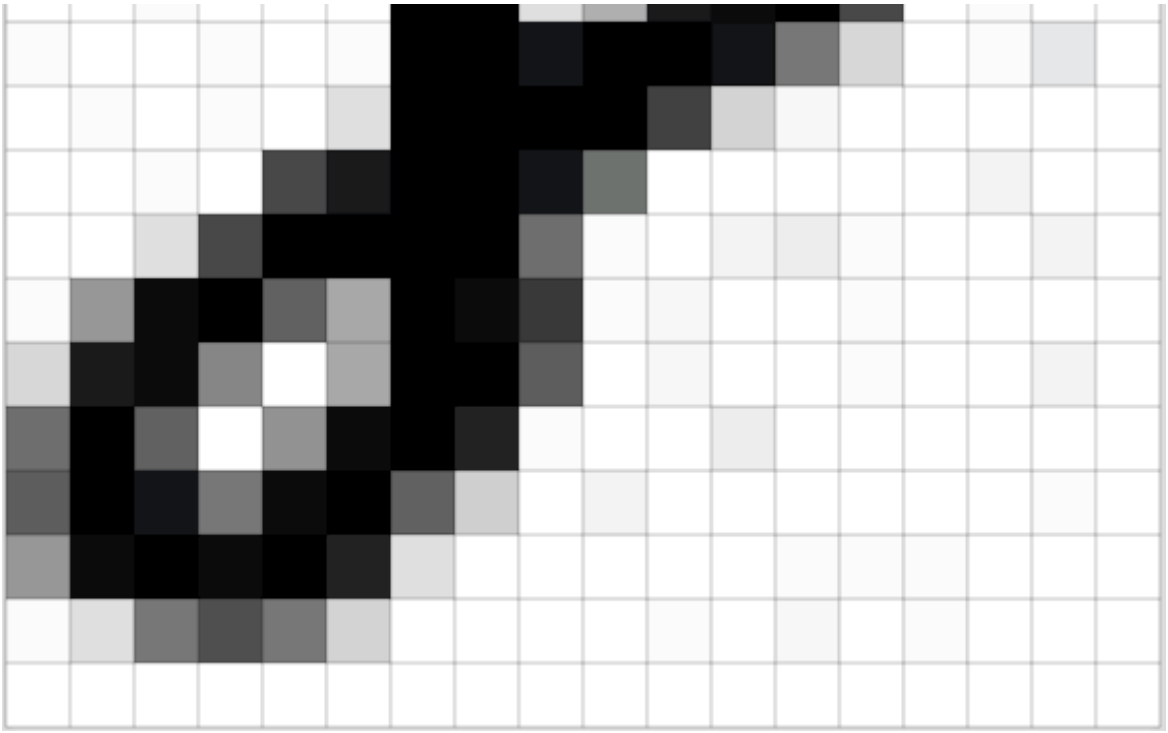
Some 8s from the MNIST data set

If you think about it, everything is just numbers

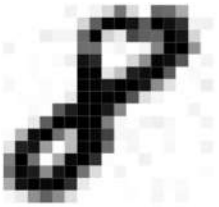
The neural network we made in Part 2 only took in a three numbers as the input ("3" bedrooms, "2000" sq. feet , etc.). But now we want to process images with our neural network. How in the world do we feed images into a neural network instead of just numbers?

The answer is incredible simple. A neural network takes numbers as input. To a computer, an image is really just a grid of numbers that represent how dark each pixel is:





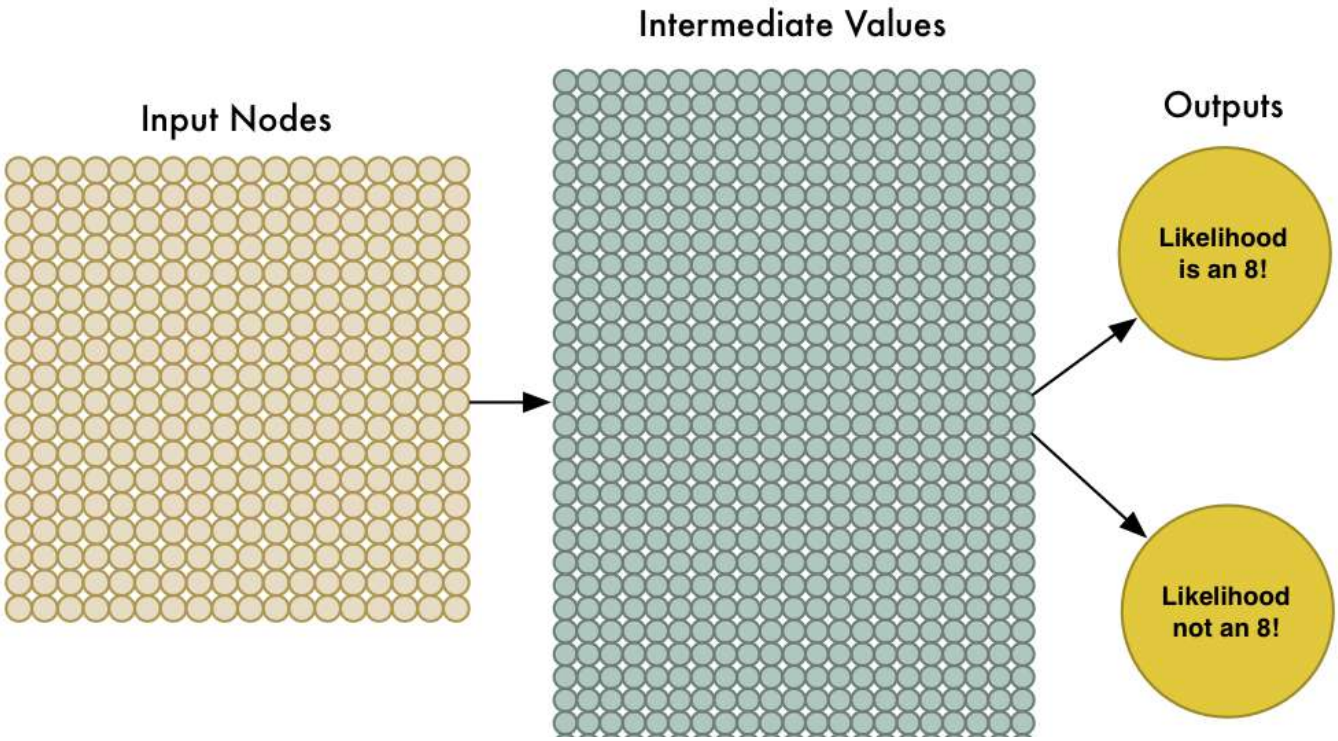
To feed an image into our neural network, we simply treat the 18x18 pixel image as an array of 324 numbers:



=

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 12, 0, 11, 39, 137, 37, 0, 152, 147, 84, 0, 0, 0, 0,
0, 1, 0, 0, 0, 41, 160, 250, 255, 235, 162, 255, 238, 206, 11, 13, 0, 0, 0, 0, 16, 9, 9, 150, 251, 45, 21, 184, 159, 154, 2
55, 233, 40, 0, 0, 10, 0, 0, 0, 0, 0, 145, 146, 3, 10, 0, 11, 124, 253, 255, 107, 0, 0, 0, 0, 3, 0, 4, 15, 236, 216, 0, 0,
38, 109, 247, 240, 169, 0, 11, 0, 1, 0, 2, 0, 0, 0, 253, 253, 23, 62, 224, 241, 255, 164, 0, 5, 0, 0, 6, 0, 0, 4, 0, 3, 252
, 250, 228, 255, 255, 234, 112, 28, 0, 2, 17, 0, 0, 2, 1, 4, 0, 21, 255, 253, 251, 255, 172, 31, 8, 0, 1, 0, 0, 0, 0, 0, 4,
0, 163, 225, 251, 255, 229, 120, 0, 0, 0, 0, 0, 0, 11, 0, 0, 0, 0, 21, 162, 255, 255, 254, 255, 126, 6, 0, 10, 14, 6, 0, 0, 9
, 0, 3, 79, 242, 255, 141, 66, 255, 245, 189, 7, 8, 0, 0, 5, 0, 0, 0, 0, 26, 221, 237, 98, 0, 67, 251, 255, 144, 0, 8, 0, 0
, 7, 0, 0, 11, 0, 125, 255, 141, 0, 87, 244, 255, 208, 3, 0, 0, 13, 0, 1, 0, 1, 0, 0, 145, 248, 228, 116, 235, 255, 141, 34
, 0, 11, 0, 1, 0, 0, 0, 1, 3, 0, 85, 237, 253, 246, 255, 210, 21, 1, 0, 1, 0, 0, 6, 2, 4, 0, 0, 0, 6, 23, 112, 157, 114, 32
, 0, 0, 0, 0, 2, 0, 8, 0, 7, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

The handle 324 inputs, we'll just enlarge our neural network to have 324 input nodes:



Notice that our neural network also has two outputs now (instead of just one). The first output will predict the likelihood that the image is an “8” and thee second output will predict the likelihood it isn’t an “8”. By having a separate output for each type of object we want to recognize, we can use a neural network to classify objects into groups.

Our neural network is a lot bigger than last time (324 inputs instead of 3!). But any modern computer can handle a neural network with a few hundred nodes without blinking. This would even work fine on your cell phone.

All that’s left is to train the neural network with images of “8”s and not-“8"s so it learns to tell them apart. When we feed in an “8”, we’ll tell it the probability the image is an “8” is 100% and the probability it’s not an “8” is 0%. Vice versa for the counter-example images.

Here’s some of our training data:

Mmm… sweet, sweet training data

We can train this kind of neural network in a few minutes on a modern laptop. When it’s done, we’ll have a neural network that can recognize pictures of “8”s with a pretty high accuracy. Welcome to the world of (late 1980’s-era) image recognition!

Tunnel Vision

It’s really neat that simply feeding pixels into a neural network actually worked to build image recognition! Machine learning is magic! …*right?*

Well, of course it’s not that simple.

First, the good news is that our “8” recognizer really does work well on simple images where the letter is right in the middle of the image:

But now the really bad news:

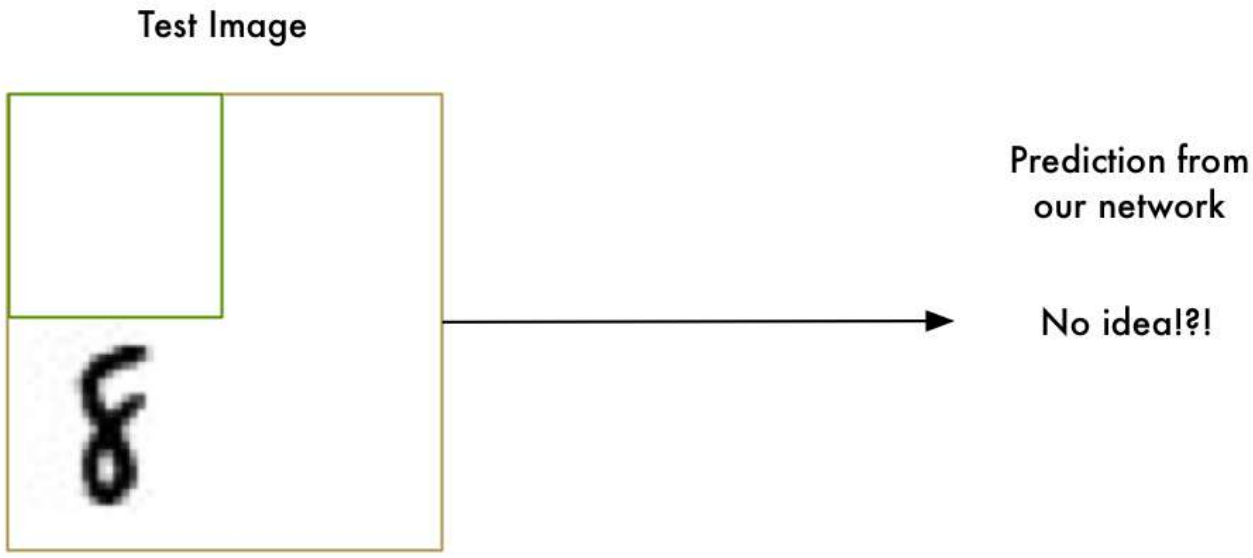
Our “8” recognizer *totally fails* to work when the letter isn’t perfectly centered in the image. Just the slightest position change ruins everything:

This is because our network only learned the pattern of a perfectly-centered “8”. It has absolutely no idea what an off-center “8” is. It knows exactly one pattern and one pattern only.

That’s not very useful in the real world. Real world problems are never that clean and simple. So we need to figure out how to make our neural network work in cases where the “8” isn’t perfectly centered.

Brute Force Idea #1: Searching with a Sliding Window

We already created a really good program for finding an “8” centered in an image. What if we just scan all around the image for possible “8”s in smaller sections, one section at a time, until we find one?

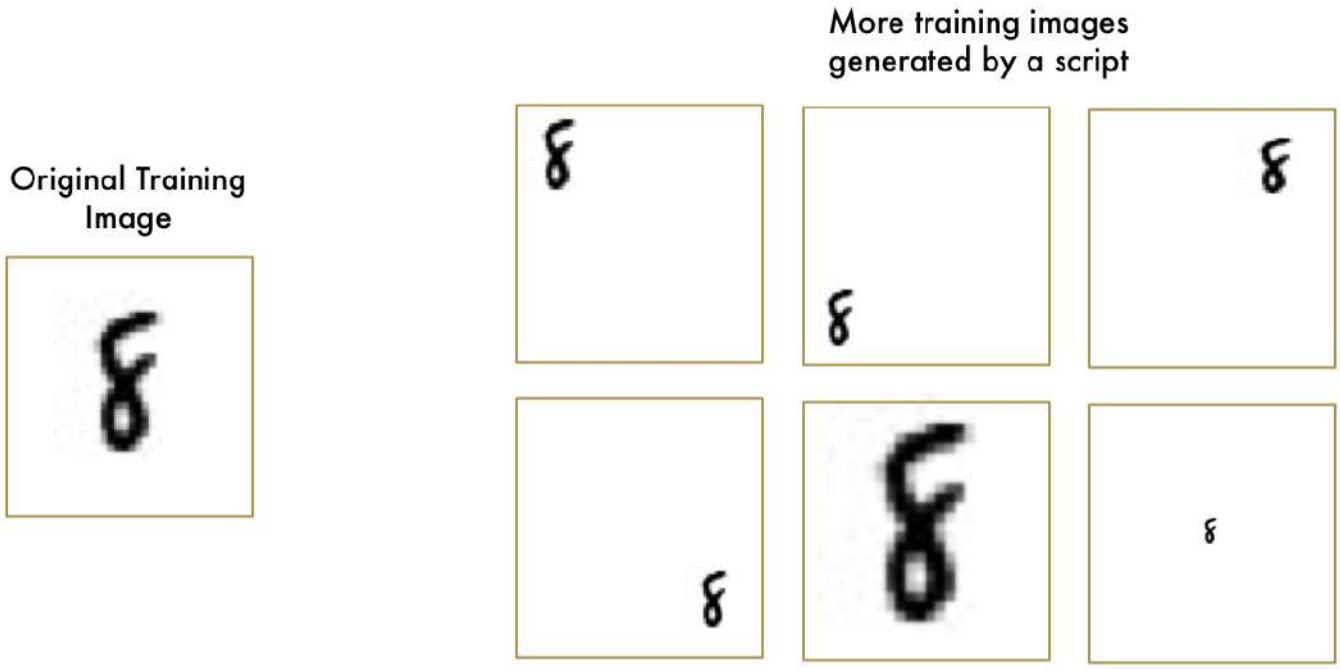


This approach called a sliding window. It's the brute force solution. It works well in some limited cases, but it's really inefficient. You have to check the same image over and over looking for objects of different sizes. We can do better than this!

Brute Force Idea #2: More data and a Deep Neural Net

When we trained our network, we only showed it "8"s that were perfectly centered. What if we train it with more data, including "8"s in all different positions and sizes all around the image?

We don't even need to collect new training data. We can just write a script to generate new images with the "8"s in all kinds of different positions in the image:

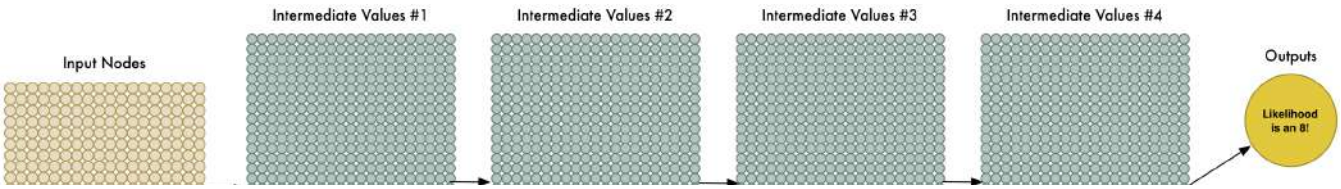


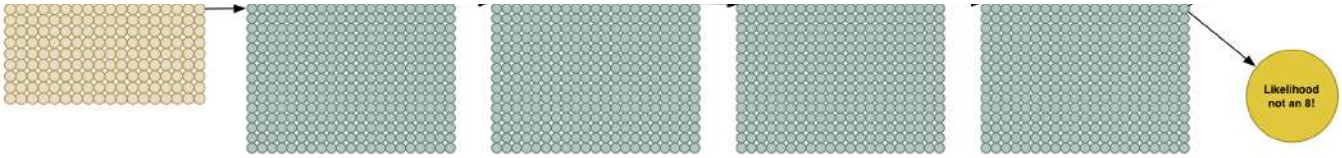
We created **Synthetic Training Data** by creating different versions of the training images we already had. This is a very useful technique!

Using this technique, we can easily create an endless supply of training data.

More data makes the problem harder for our neural network to solve, but we can compensate for that by making our network bigger and thus able to learn more complicated patterns.

To make the network bigger, we just stack up layer upon layer of nodes:





We call this a “deep neural network” because it has more layers than a traditional neural network.

This idea has been around since the late 1960s. But until recently, training this large of a neural network was just too slow to be useful. But once we figured out how to use 3d graphics cards (which were designed to do matrix multiplication really fast) instead of normal computer processors, working with large neural networks suddenly became practical. In fact, the exact same NVIDIA GeForce GTX 1080 video card that you use to play Overwatch can be used to train neural networks incredibly quickly.

But even though we can make our neural network really big and train it quickly with a 3d graphics card, that still isn’t going to get us all the way to a solution. We need to be smarter about how we process images into our neural network.

Think about it. It doesn’t make sense to train a network to recognize an “8” at the top of a picture separately from training it to recognize an “8” at the bottom of a picture as if those were two totally different objects.

There should be some way to make the neural network smart enough to know that an “8” anywhere in the picture is the same thing without all that extra training. Luckily… there is!

The Solution is Convolution

As a human, you intuitively know that pictures have a *hierarchy* or *conceptual structure*. Consider this picture:

Gratuitous picture of my son

As a human, you instantly recognize the hierarchy in this picture:

- The ground is covered in grass and concrete
- There is a child
- The child is sitting on a bouncy horse
- The bouncy horse is on top of the grass

Most importantly, we recognize the idea of a *child* no matter what surface the child is on. We don't have to re-learn the idea of *child* for every possible surface it could appear on.

But right now, our neural network can't do this. It thinks that an "8" in a different part of the image is an entirely different thing. It doesn't understand that moving an object around in the picture doesn't make it something different. This means it has to re-learn the identify of each object in every possible position. That sucks.

We need to give our neural network understanding of *translation invariance* — an "8" is an "8" no matter where in the picture it shows up.

We'll do this using a process called Convolution. The idea of convolution is inspired partly by computer science and partly by biology (i.e. mad scientists literally poking cat brains with weird probes to figure out how cats process images).

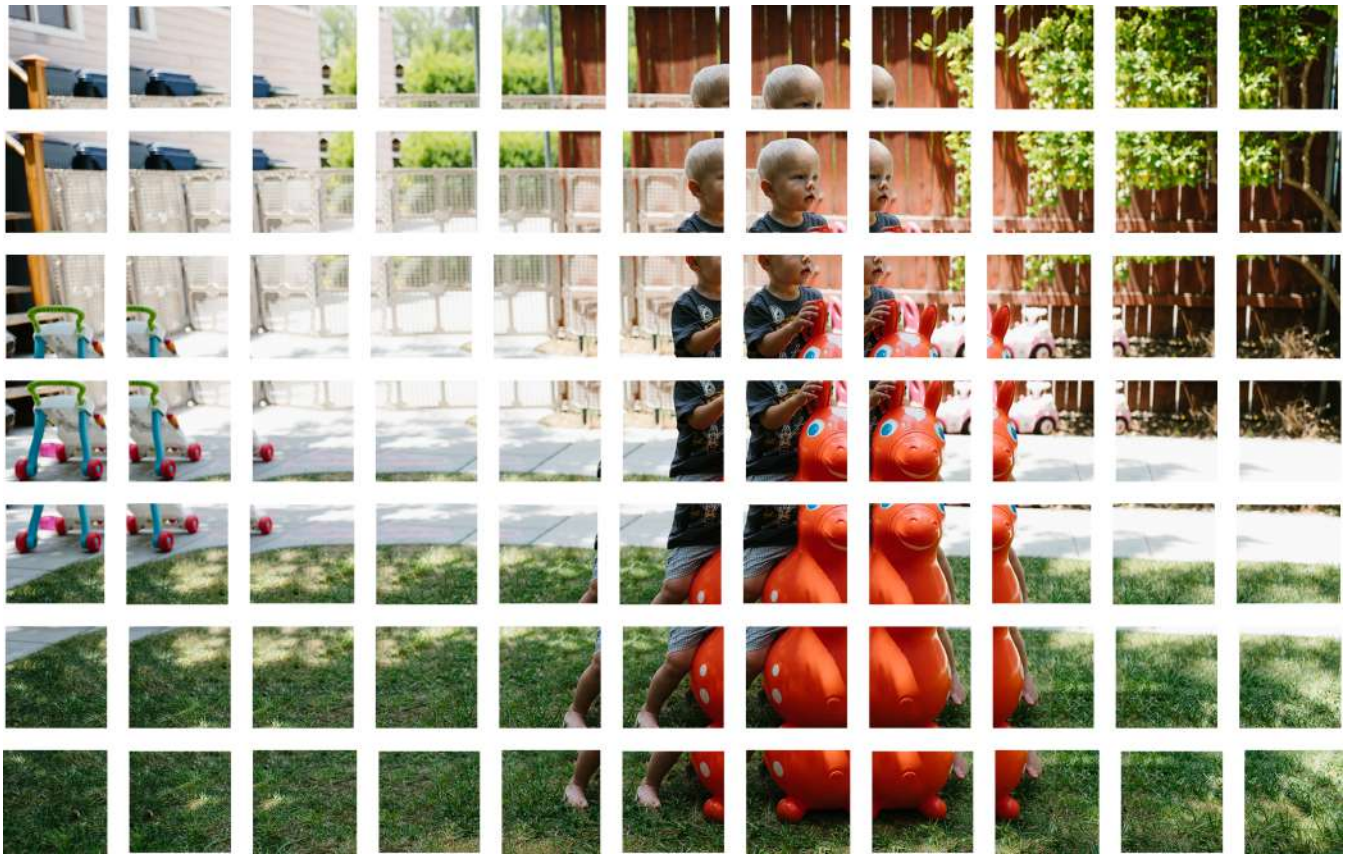
How Convolution Works

Instead of feeding entire images into our neural network as one grid of numbers, we're going to do something a lot smarter that takes advantage of the idea that an object is the same no matter where it appears in a picture.

Here's how it's going to work, step by step —

Step 1: Break the image into overlapping image tiles

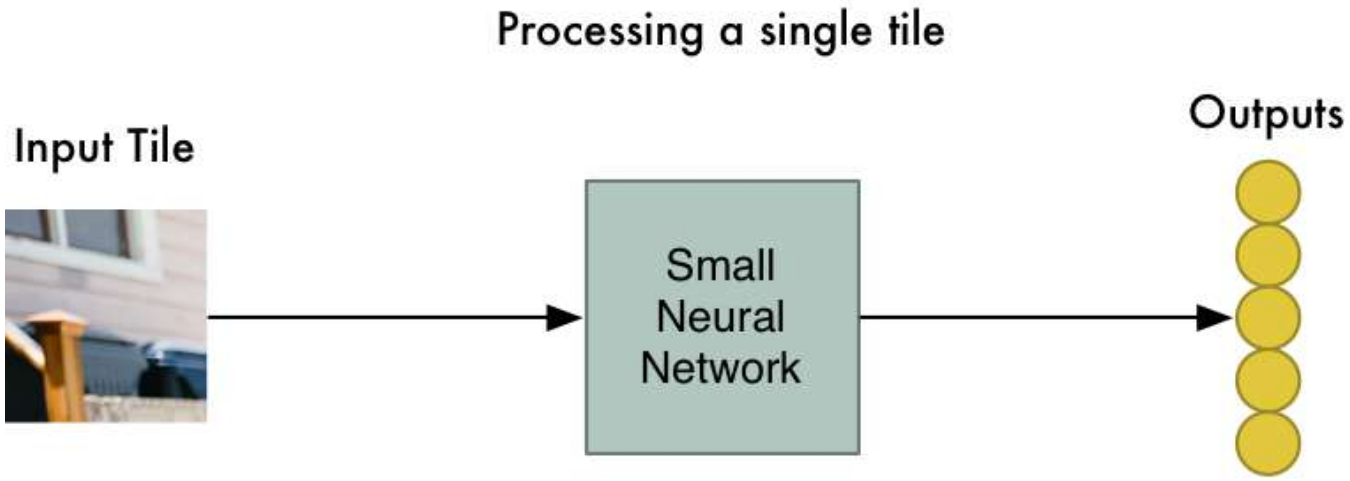
Similar to our sliding window search above, let's pass a sliding window over the entire original image and save each result as a separate, tiny picture tile:



By doing this, we turned our original image into 77 equally-sized tiny image tiles.

Step 2: Feed each image tile into a small neural network

Earlier, we fed a single image into a neural network to see if it was an “8”. We’ll do the exact same thing here, but we’ll do it for each individual image tile:

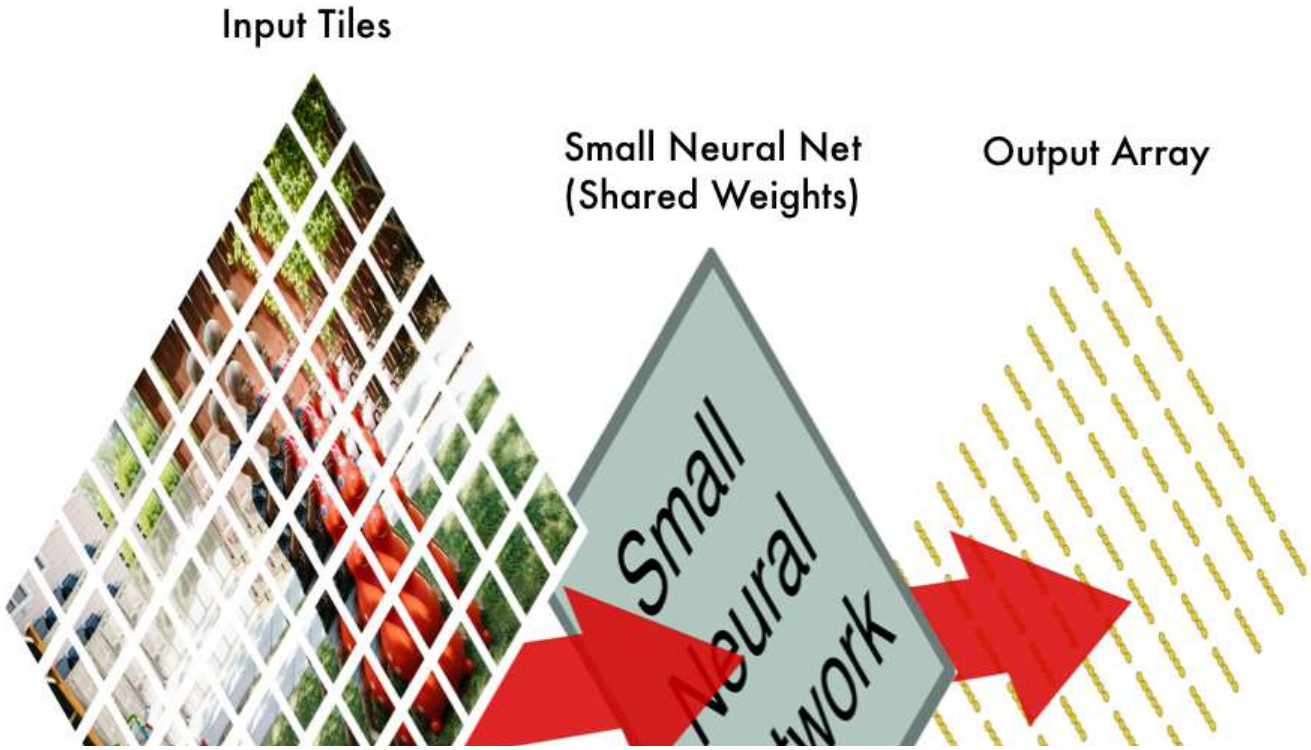


Repeat this 77 times, once for each tile.

However, **there’s one big twist**: We’ll keep the **same neural network weights** for every single tile in the same original image. In other words, we are treating every image tile equally. If something interesting appears in any given tile, we’ll mark that tile as interesting.

Step 3: Save the results from each tile into a new array

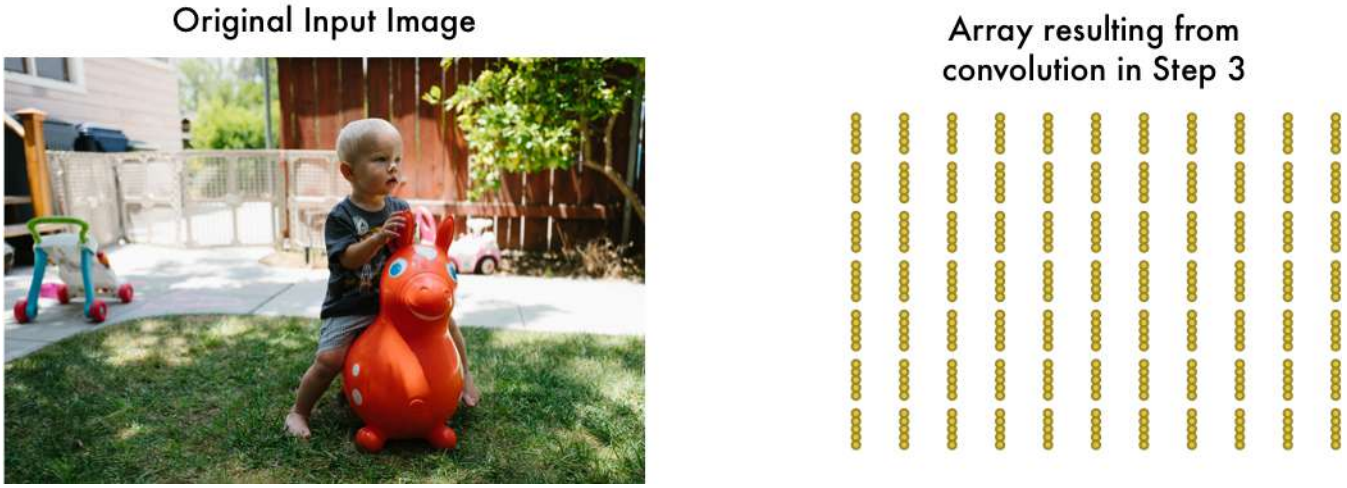
We don’t want to lose track of the arrangement of the original tiles. So we save the result from processing each tile into a grid in the same arrangement as the original image. It looks like this:



In other words, we've started with a large image and we ended with a slightly smaller array that records which sections of our original image were the most interesting.

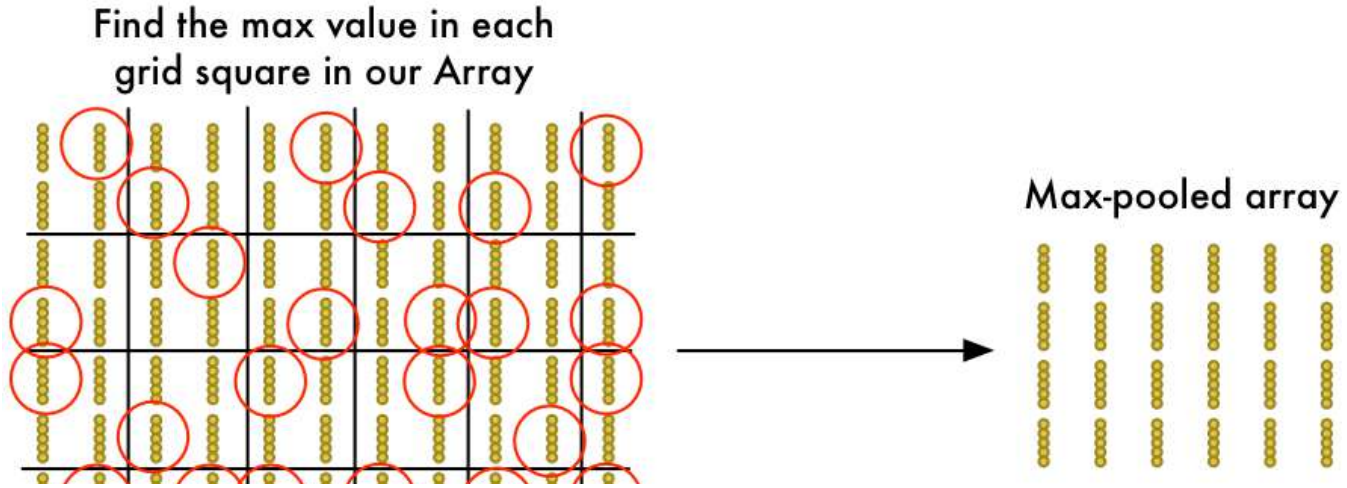
Step 4: Downsampling

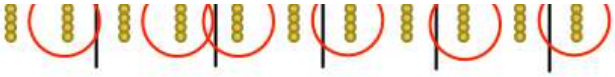
The result of Step 3 was an array that maps out which parts of the original image are the most interesting. But that array is still pretty big:



To reduce the size of the array, we *downsample* it using an algorithm called max pooling. It sounds fancy, but it isn't at all!

We'll just look at each 2x2 square of the array and keep the biggest number:





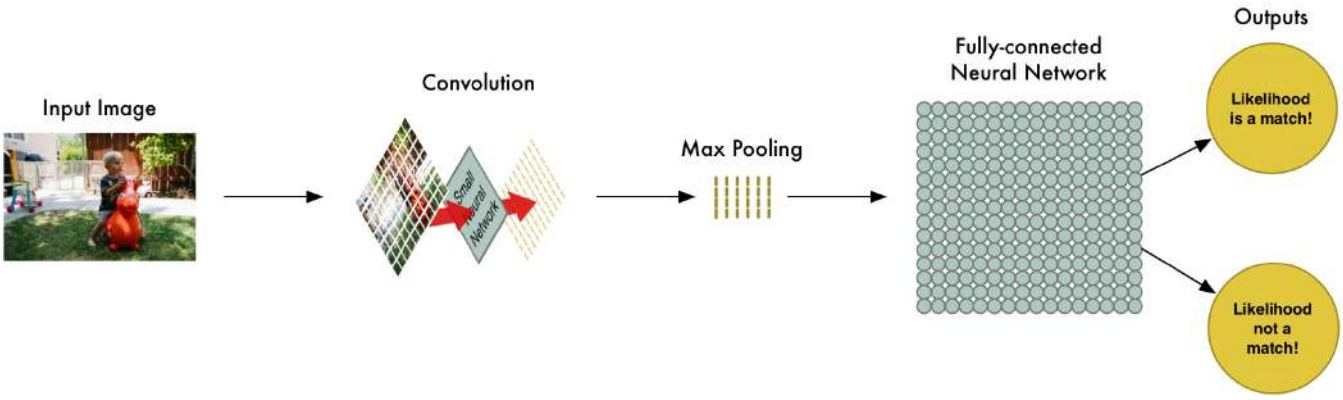
The idea here is that if we found something interesting in any of the four input tiles that makes up each 2x2 grid square, we'll just keep the most interesting bit. This reduces the size of our array while keeping the most important bits.

Final step: Make a prediction

So far, we've reduced a giant image down into a fairly small array.

Guess what? That array is just a bunch of numbers, so we can use that small array as input into *another neural network*. This final neural network will decide if the image is or isn't a match. To differentiate it from the convolution step, we call it a "fully connected" network.

So from start to finish, our whole five-step pipeline looks like this:



Adding Even More Steps

Our image processing pipeline is a series of steps: convolution, max-pooling, and finally a fully-connected network.

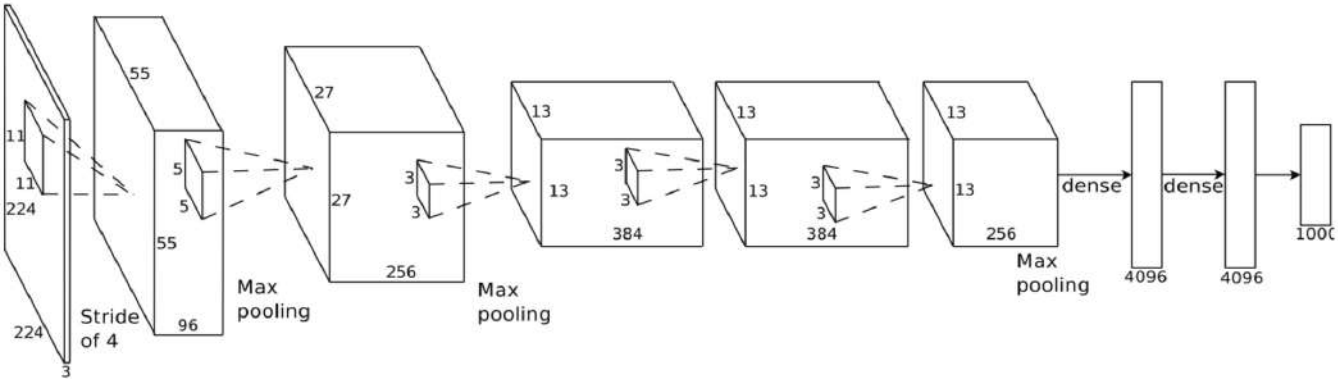
When solving problems in the real world, these steps can be combined and stacked as many times as you want! You can have two, three or even ten convolution layers. You can throw in max pooling wherever you want to reduce the size of your data.

The basic idea is to start with a large image and continually boil it down, step-by-step, until you finally have a single result. The more convolution steps you have, the more complicated features your network will be able to learn to recognize.

For example, the first convolution step might learn to recognize sharp edges, the second convolution step might recognize beaks using it's knowledge of sharp edges, the

third step might recognize entire birds using it's knowledge of beaks, etc.

Here's what a more realistic deep convolutional network (like you would find in a research paper) looks like:



In this case, they start a 224 x 224 pixel image, apply convolution and max pooling twice, apply convolution 3 more times, apply max pooling and then have two fully-connected layers. The end result is that the image is classified into one of 1000 categories!

Constructing the Right Network

So how do you know which steps you need to combine to make your image classifier work?

Honestly, you have to answer this by doing a lot of experimentation and testing. You might have to train 100 networks before you find the optimal structure and parameters for the problem you are solving. Machine learning involves a lot of trial and error!

Building our Bird Classifier

Now finally we know enough to write a program that can decide if a picture is a bird or not.

As always, we need some data to get started. The free CIFAR10 data set contains 6,000 pictures of birds and 52,000 pictures of things that are not birds. But to get even more data we'll also add in the Caltech-UCSD Birds-200–2011 data set that has another 12,000 bird pics.

Here's a few of the birds from our combined data set:

And here's some of the 52,000 non-bird images:

This data set will work fine for our purposes, but 72,000 low-res images is still pretty small for real-world applications. If you want Google-level performance, you need *millions* of large images. In machine learning, having more data is almost always more important that having better algorithms. Now you know why Google is so happy to offer you unlimited photo storage. They want your sweet, sweet data!

To build our classifier, we'll use TFLearn. TFlearn is a wrapper around Google's TensorFlow deep learning library that exposes a simplified API. It makes building convolutional neural networks as easy as writing a few lines of code to define the layers of our network.

Here's the code to define and train the network:

```
# -*- coding: utf-8 -*-

"""
Based on the tflearn example located here:
https://github.com/tflearn/tflearn/blob/master/examples/images/convnet_cifar10.py
"""
from __future__ import division, print_function, absolute_import

# Import tflearn and some helpers
import tflearn
from tflearn.data_utils import shuffle
```

```
from tflearn.layers.core import input_data, dropout, fully_connected
from tflearn.layers.conv import conv_2d, max_pool_2d
from tflearn.layers.estimator import regression
from tflearn.data_preprocessing import ImagePreprocessing
from tflearn.data_augmentation import ImageAugmentation
import pickle

# Load the data set
X, Y, X_test, Y_test = pickle.load(open("full_dataset.pkl", "rb"))

# Shuffle the data
X, Y = shuffle(X, Y)

# Make sure the data is normalized
img_prep = ImagePreprocessing()
img_prep.add_featurewise_zero_center()
img_prep.add_featurewise_stdnorm()

# Create extra synthetic training data by flipping, rotating and blurring the
# images on our data set.
img_aug = ImageAugmentation()
img_aug.add_random_flip_leftright()
img_aug.add_random_rotation(max_angle=25.)
img_aug.add_random_blur(sigma_max=3.)

# Define our network architecture:

# Input is a 32x32 image with 3 color channels (red, green and blue)
network = input_data(shape=[None, 32, 32, 3],
                     data_preprocessing=img_prep,
                     data_augmentation=img_aug)

# Step 1: Convolution
network = conv_2d(network, 32, 3, activation='relu')

# Step 2: Max pooling
network = max_pool_2d(network, 2)

# Step 3: Convolution again
network = conv_2d(network, 64, 3, activation='relu')

# Step 4: Convolution yet again
network = conv_2d(network, 64, 3, activation='relu')

# Step 5: Max pooling again
network = max_pool_2d(network, 2)

# Step 6: Fully connected 512 node neural network
```

```
# Step 6: Fully-connected 512 node neural network
network = fully_connected(network, 512, activation='relu')

# Step 7: Dropout - throw away some data randomly during training to prevent over-fitting
network = dropout(network, 0.5)

# Step 8: Fully-connected neural network with two outputs (0=isn't a bird, 1=is a bird) to make
network = fully_connected(network, 2, activation='softmax')

# Tell tflearn how we want to train the network
network = regression(network, optimizer='adam',
                     loss='categorical_crossentropy',
                     learning_rate=0.001)

# Wrap the network in a model object
model = tflearn.DNN(network, tensorboard_verbose=0, checkpoint_path='bird-classifier.tfl.ckpt')

# Train it! We'll do 100 training passes and monitor it as it goes.
model.fit(X, Y, n_epoch=100, shuffle=True, validation_set=(X_test, Y_test),
          show_metric=True, batch_size=96,
          snapshot_epoch=True,
          run_id='bird-classifier')

# Save model when training is complete to a file
model.save("bird-classifier.tfl")
print("Network trained and saved as bird-classifier.tfl!")
```

GTX 980 Ti or better), this will be done in less than an hour. If you are training with a normal cpu, it might take a lot longer.

As it trains, the accuracy will increase. After the first pass, I got 75.4% accuracy. After just 10 passes, it was already up to 91.7%. After 50 or so passes, it capped out around 95.5% accuracy and additional training didn't help, so I stopped it there.

Congrats! Our program can now recognize birds in images!

Testing our Network

Now that we have a trained neural network, we can use it! Here's a simple script that takes in a single image file and predicts if it is a bird or not.

But to really see how effective our network is, we need to test it with lots of images. The data set I created held back 15,000 images for validation. When I ran those 15,000 images through the network, it predicted the correct answer 95% of the time.

That seems pretty good, right? Well… it depends!

How accurate is 95% accurate?

Our network claims to be 95% accurate. But the devil is in the details. That could mean all sorts of different things.

For example, what if 5% of our training images were birds and the other 95% were not birds? A program that guessed "not a bird" every single time would be 95% accurate! But it would also be 100% useless.

We need to look more closely at the numbers than just the overall accuracy. To judge how good a classification system really is, we need to look closely at *how* it failed, not just the percentage of the time that it failed.

Instead of thinking about our predictions as "right" and "wrong", let's break them down into four separate categories —

- First, here are some of the birds that our network correctly identified as birds. Let's call these **True Positives:**

Wow! Our network can recognize lots of different kinds of birds successfully!

- Second, here are images that our network correctly identified as "not a bird". These are called **True Negatives:**

Horses and trucks don't fool us!

- Third, here are some images that we thought were birds but were not really birds at all. These are our **False Positives**:

Lots of planes were mistaken for birds! That makes sense.

- And finally, here are some images of birds that we didn't correctly recognize as birds. These are our **False Negatives**:

These birds fooled us! Stupid ostriches! Do they even count as birds?

Using our validation set of 15,000 images, here's how many times our predictions fell into each category:

Results for 15,000 Validation Images

(6000 images are birds, 9000 images are not birds)

	Predicted 'bird'	Predicted 'not a bird'
Bird	**5,450** *True Positves*	**550** *False Negatives*
Not a Bird	**162** *False Positives*	**8,838** *True Negatives*

Why do we break our results down like this? Because not all mistakes are created equal.

Imagine if we were writing a program to detect cancer from an MRI image. If we were detecting cancer, we'd rather have false positives than false negatives. False negatives would be the worse possible case — that's when the program told someone they definitely didn't have cancer but they actually did.

Instead of just looking at overall accuracy, we calculate Precision and Recall metrics. Precision and Recall metrics give us a clearer picture of how well we did:

Precision *If we predicted 'bird', how often was it really a bird?*	**97.11%** *(True Positives ÷ All Positive Guesses)*
Recall *What percentage of the actual birds did we find?*	**90.83%** *(True Positives ÷ Total Birds in Dataset)*

This tells us that 97% of the time we guessed “Bird”, we were right! But it also tells us that we only found 90% of the actual birds in the data set. In other words, we might not find every bird but we are pretty sure about it when we do find one!

Where to go from here

Now that you know the basics of deep convolutional networks, you can try out some of the examples that come with tflearn to get your hands dirty with different neural network architectures. It even comes with built-in data sets so you don’t even have to find your own images.

You also know enough now to start branching and learning about other areas of machine learning. Why not learn how to use algorithms to train computers how to play Atari games next?

. . .

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 4, Part 5 and Part 6!

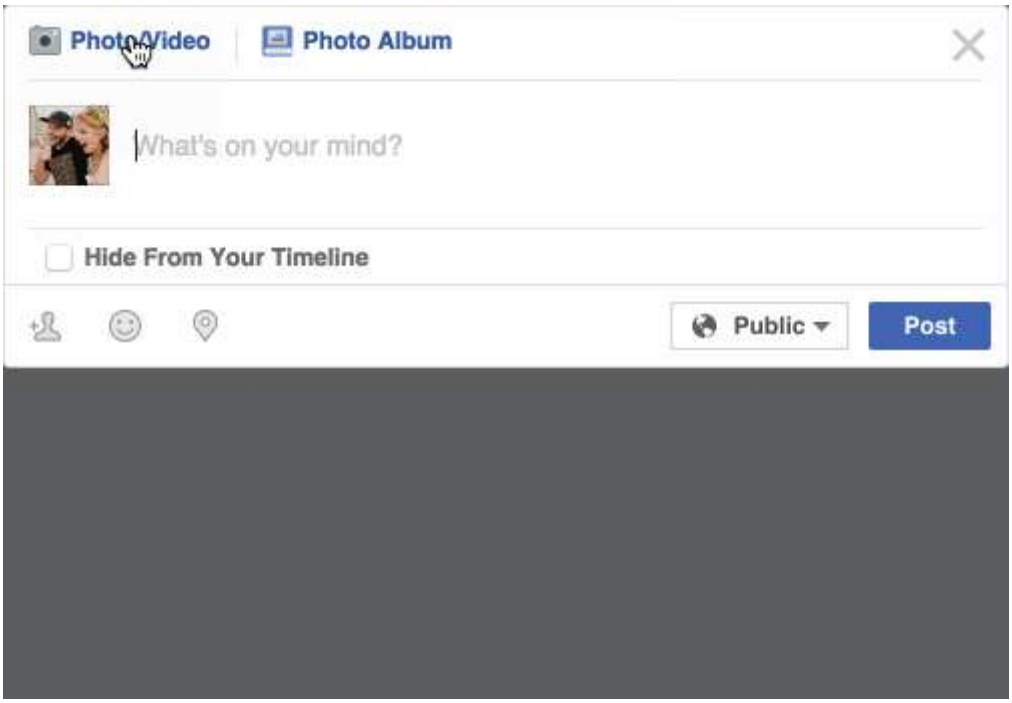
Machine Learning is Fun! Part 4: Modern Face Recognition with Deep Learning

Adam Geitgey
Jul 24, 2016 · 13 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 普通话, Русский, 한국어, Português, Tiê´ng Việt or Italiano.*

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!*

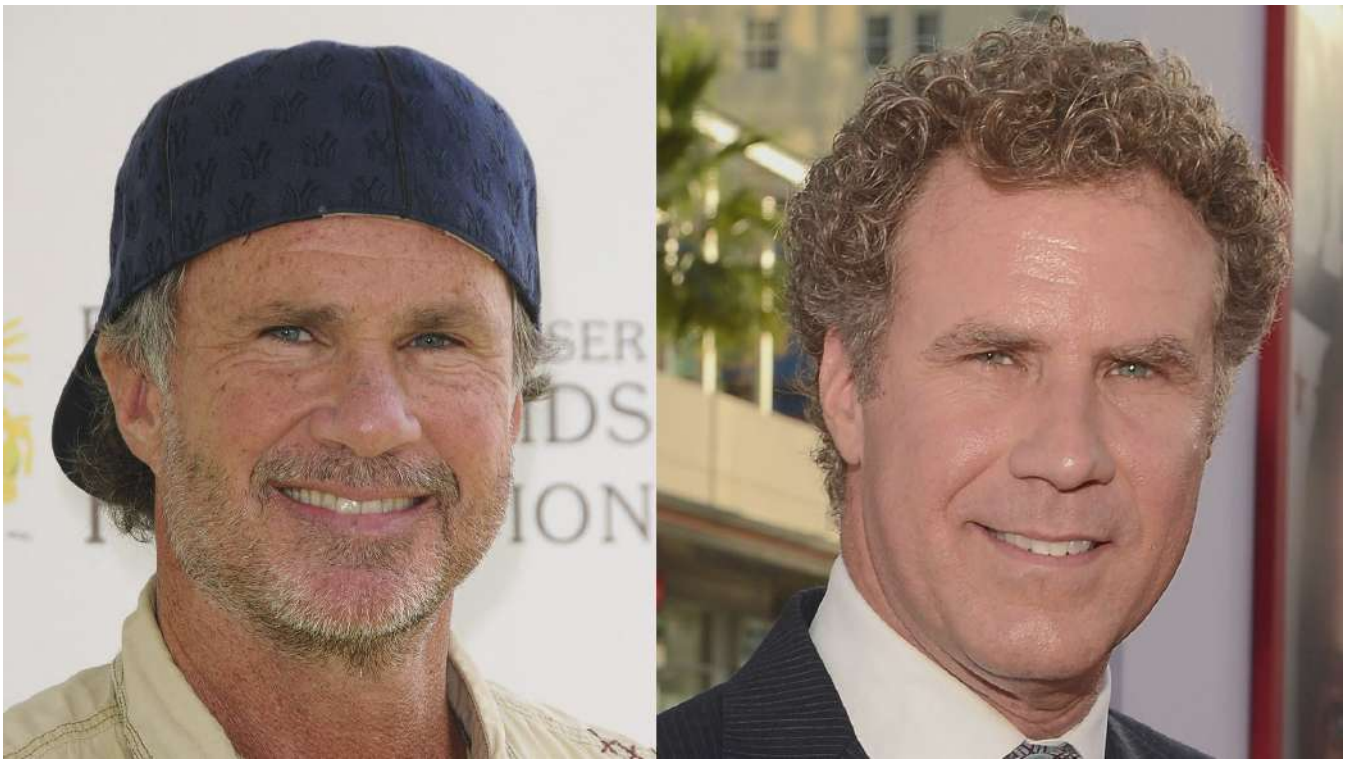
Have you noticed that Facebook has developed an uncanny ability to recognize your friends in your photographs? In the old days, Facebook used to make you to tag your friends in photos by clicking on them and typing in their name. Now as soon as you upload a photo, Facebook tags everyone for you *like magic*:



Facebook automatically tags people in your photos that you have tagged before. I'm not sure if this is helpful or creepy!

This technology is called face recognition. Facebook's algorithms are able to recognize your friends' faces after they have been tagged only a few times. It's pretty amazing technology — Facebook can recognize faces with 98% accuracy which is pretty much as good as humans can do!

Let's learn how modern face recognition works! But just recognizing your friends would be too easy. We can push this tech to the limit to solve a more challenging problem — telling Will Ferrell (famous actor) apart from Chad Smith (famous rock musician)!



One of these people is Will Farrell. The other is Chad Smith. I swear they are different people!

. . .

How to use Machine Learning on a Very Complicated Problem

So far in Part 1, 2 and 3, we've used machine learning to solve isolated problems that have only one step — estimating the price of a house, generating new data based on existing data and telling if an image contains a certain object. All of those problems can

be solved by choosing one machine learning algorithm, feeding in data, and getting the result.

But face recognition is really a series of several related problems:

1. First, look at a picture and find all the faces in it
2. Second, focus on each face and be able to understand that even if a face is turned in a weird direction or in bad lighting, it is still the same person.
3. Third, be able to pick out unique features of the face that you can use to tell it apart from other people— like how big the eyes are, how long the face is, etc.
4. Finally, compare the unique features of that face to all the people you already know to determine the person's name.

As a human, your brain is wired to do all of this automatically and instantly. In fact, humans are *too good* at recognizing faces and end up seeing faces in everyday objects:

Computers are not capable of this kind of high-level generalization (*at least not yet*…), so we have to teach them how to do each step in this process separately.

We need to build a *pipeline* where we solve each step of face recognition separately and pass the result of the current step to the next step. In other words, we will chain together several machine learning algorithms:

How a basic pipeline for detecting faces might work

. . .

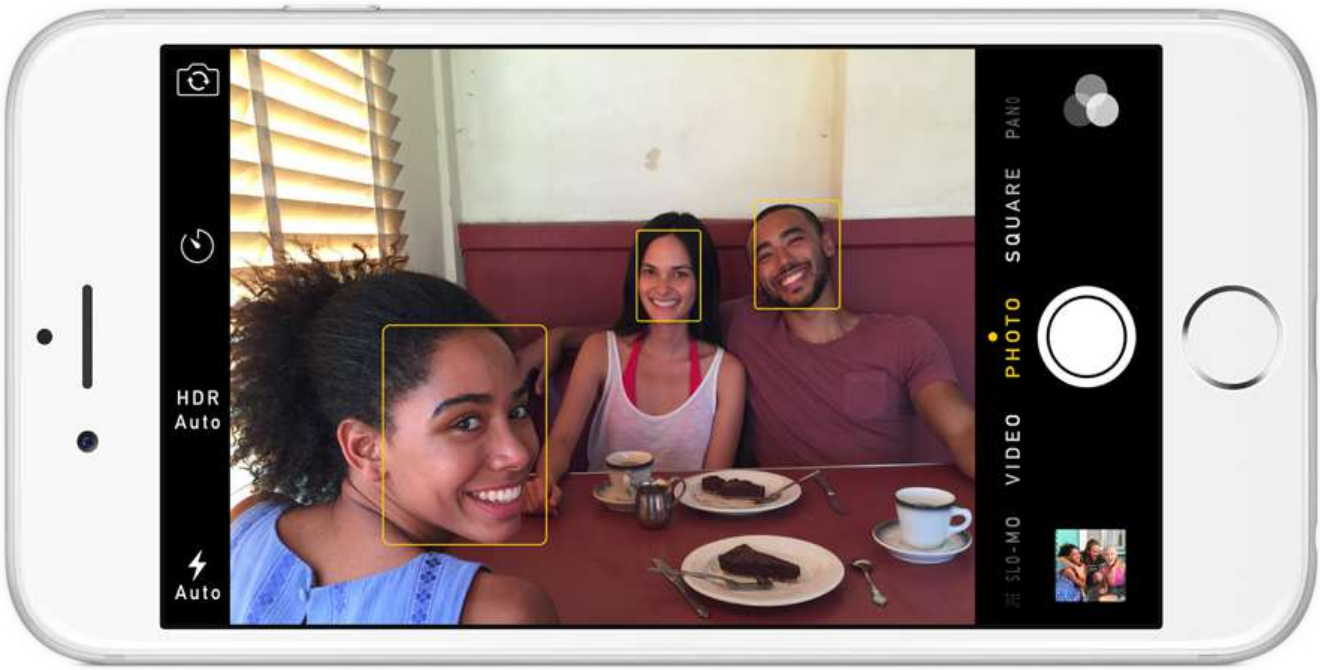
Face Recognition — Step by Step

Let's tackle this problem one step at a time. For each step, we'll learn about a different machine learning algorithm. I'm not going to explain every single algorithm completely to keep this from turning into a book, but you'll learn the main ideas behind each one and you'll learn how you can build your own facial recognition system in Python using OpenFace and dlib.

Step 1: Finding all the Faces

The first step in our pipeline is *face detection*. Obviously we need to locate the faces in a photograph before we can try to tell them apart!

If you've used any camera in the last 10 years, you've probably seen face detection in action:

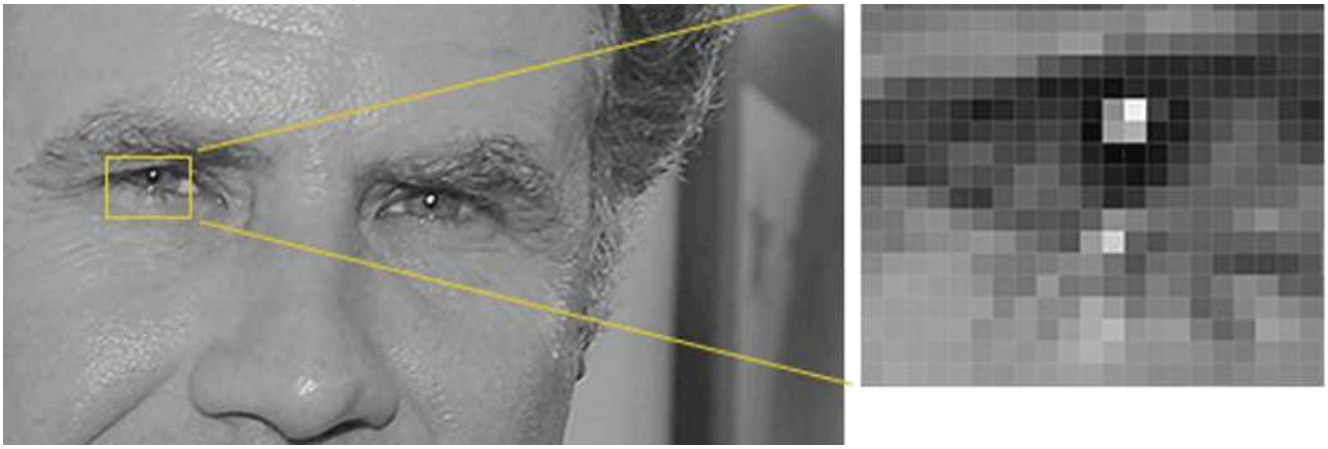


Face detection is a great feature for cameras. When the camera can automatically pick out faces, it can make sure that all the faces are in focus before it takes the picture. But we'll use it for a different purpose — finding the areas of the image we want to pass on to the next step in our pipeline.

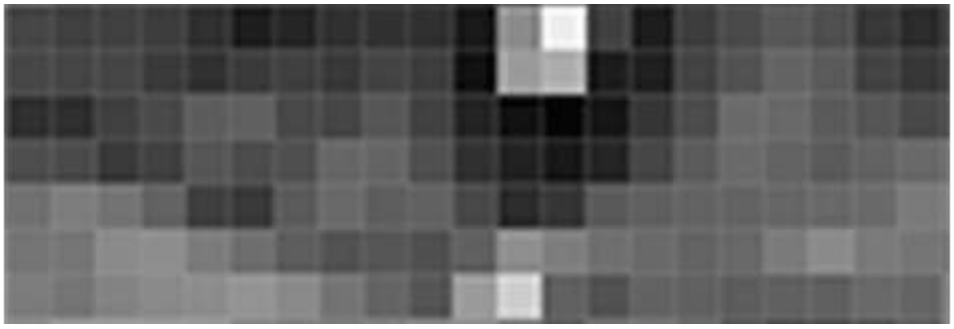
Face detection went mainstream in the early 2000's when Paul Viola and Michael Jones invented a way to detect faces that was fast enough to run on cheap cameras. However, much more reliable solutions exist now. We're going to use a method invented in 2005 called Histogram of Oriented Gradients — or just ***HOG*** for short.

To find faces in an image, we'll start by making our image black and white because we don't need color data to find faces:

Then we'll look at every single pixel in our image one at a time. For every single pixel, we want to look at the pixels that directly surrounding it:



Our goal is to figure out how dark the current pixel is compared to the pixels directly surrounding it. Then we want to draw an arrow showing in which direction the image is getting darker:



Looking at just this one pixel and the pixels touching it, the image is getting darker towards the upper right.

If you repeat that process for **every single pixel** in the image, you end up with every pixel being replaced by an arrow. These arrows are called *gradients* and they show the flow from light to dark across the entire image:

This might seem like a random thing to do, but there's a really good reason for replacing the pixels with gradients. If we analyze pixels directly, really dark images and really light images of the same person will have totally different pixel values. But by only considering the *direction* that brightness changes, both really dark images and really bright images will end up with the same exact representation. That makes the problem a lot easier to solve!

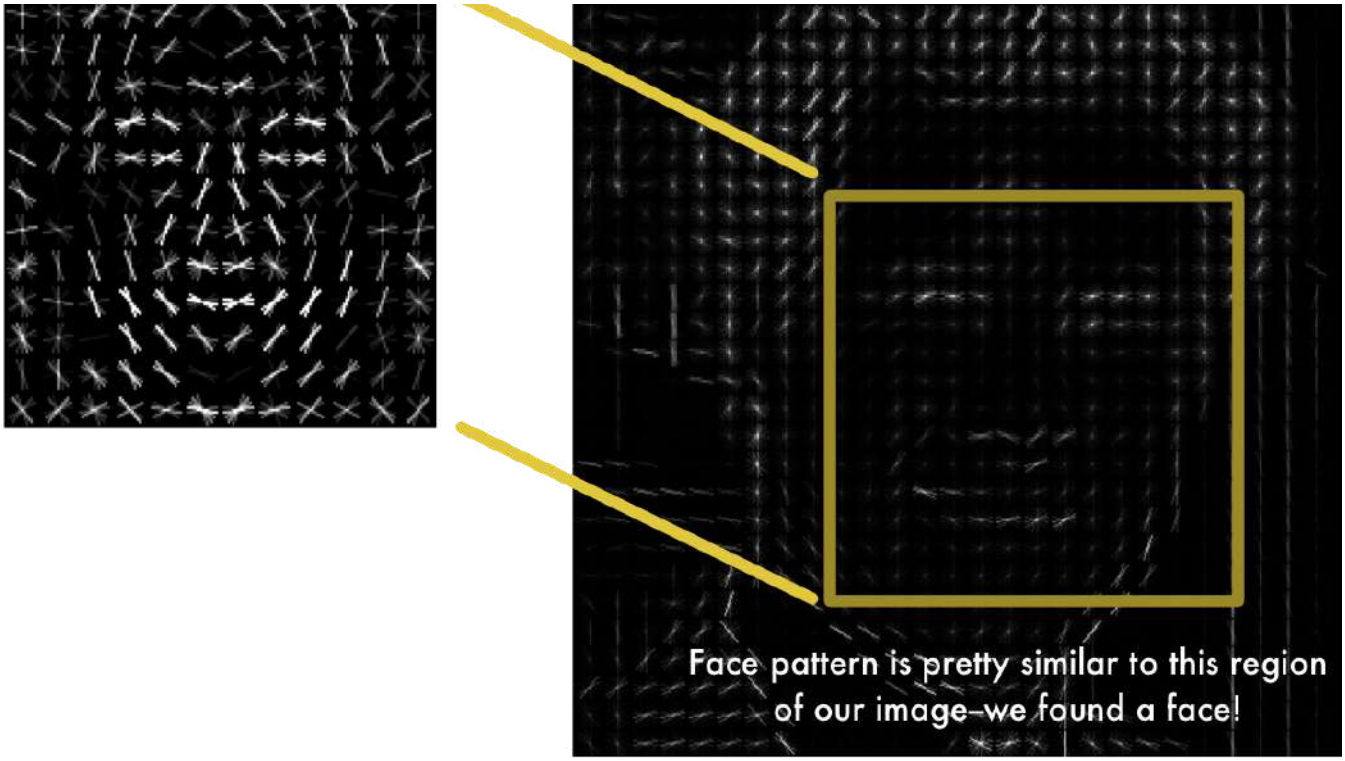
But saving the gradient for every single pixel gives us way too much detail. We end up missing the forest for the trees. It would be better if we could just see the basic flow of lightness/darkness at a higher level so we could see the basic pattern of the image.

To do this, we'll break up the image into small squares of 16x16 pixels each. In each square, we'll count up how many gradients point in each major direction (how many point up, point up-right, point right, etc…). Then we'll replace that square in the image with the arrow directions that were the strongest.

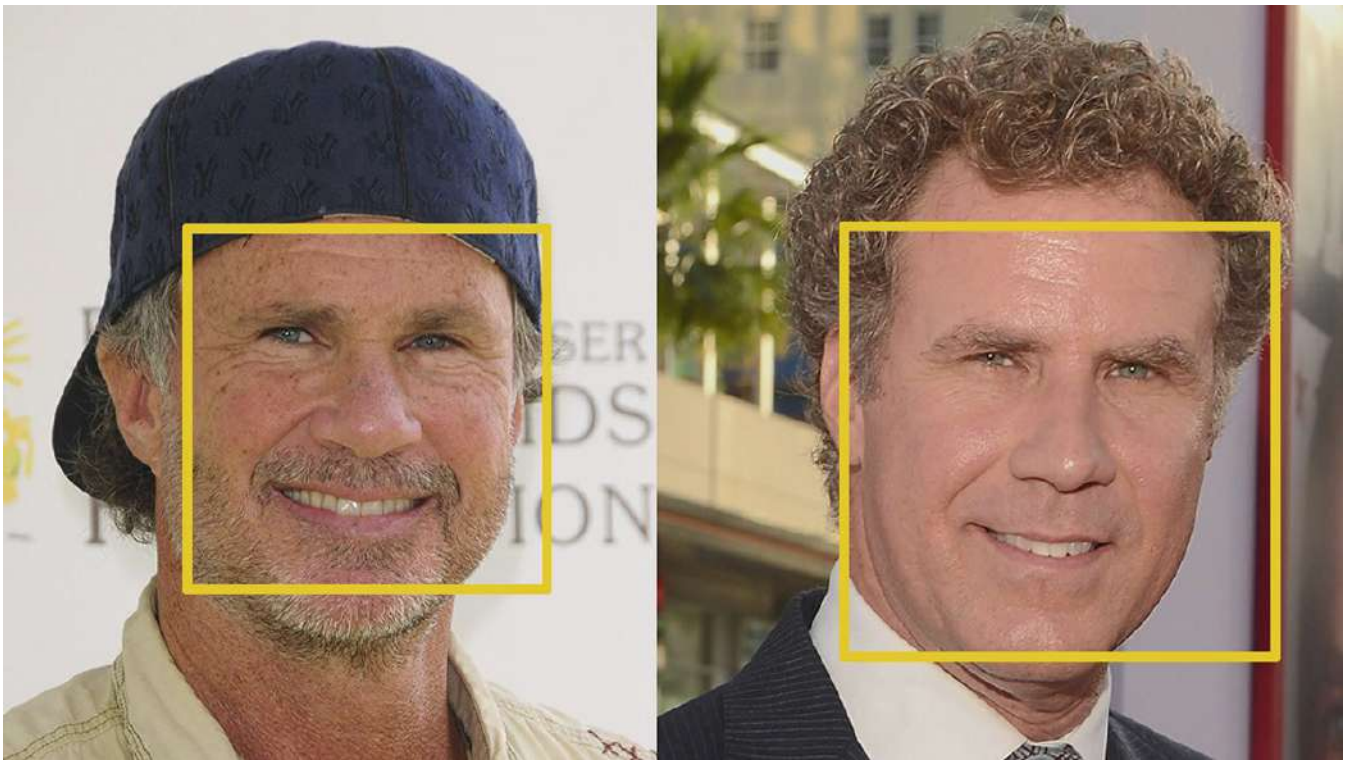
The end result is we turn the original image into a very simple representation that captures the basic structure of a face in a simple way:

The original image is turned into a HOG representation that captures the major features of the image regardless of image brightnesss.

To find faces in this HOG image, all we have to do is find the part of our image that looks the most similar to a known HOG pattern that was extracted from a bunch of other training faces:



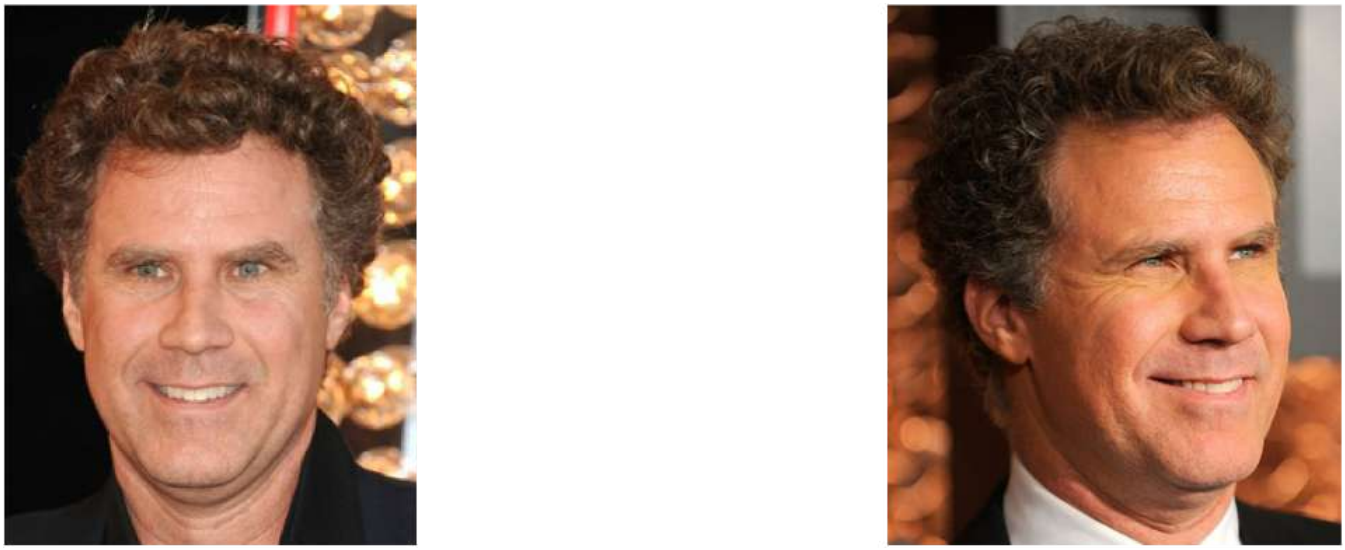
Using this technique, we can now easily find faces in any image:



If you want to try this step out yourself using Python and dlib, here's code showing how to generate and view HOG representations of images.

Step 2: Posing and Projecting Faces

Whew, we isolated the faces in our image. But now we have to deal with the problem that faces turned different directions look totally different to a computer:



Humans can easily recognize that both images are of Will Ferrell, but computers would see these pictures as two completely different people.

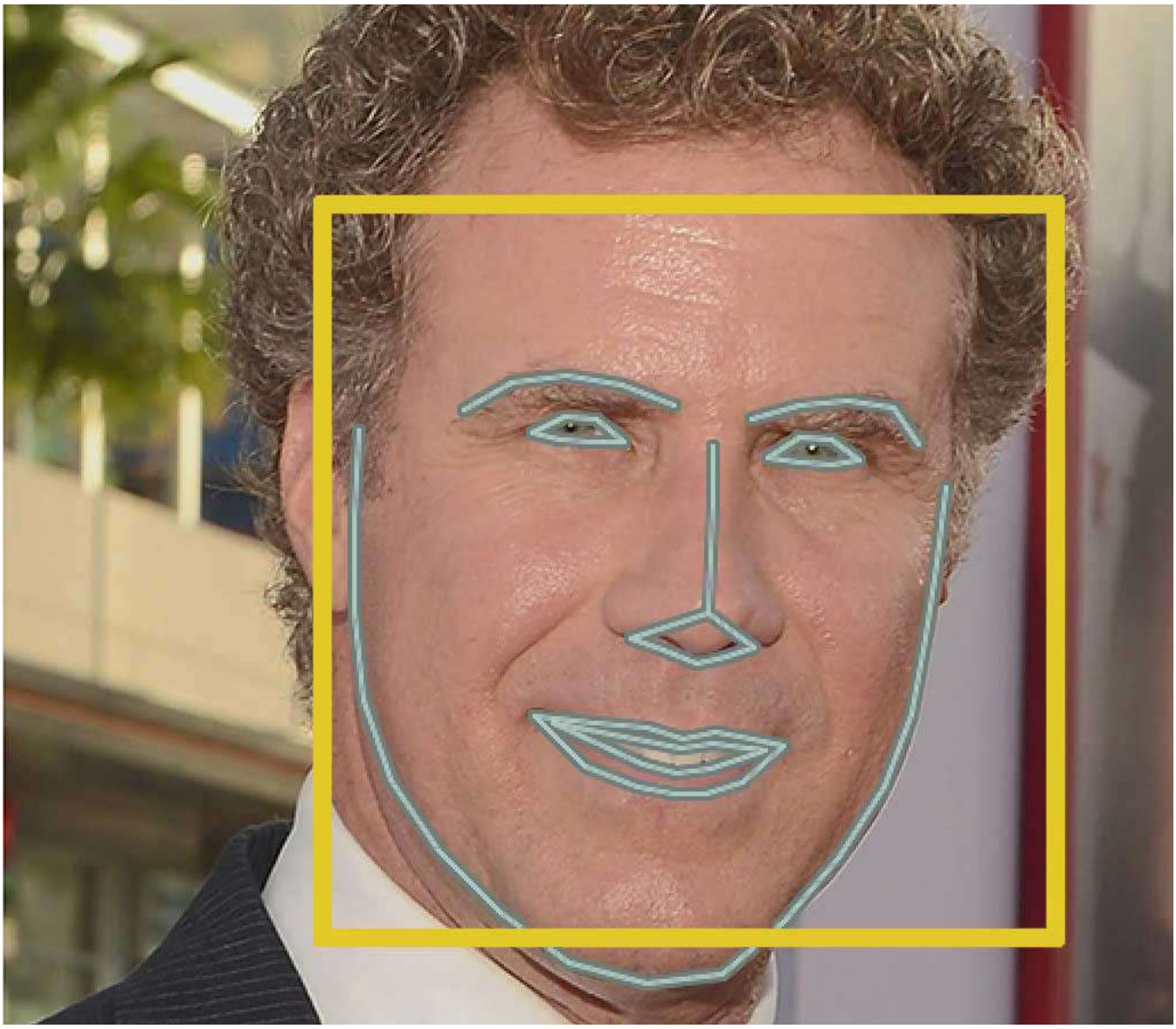
To account for this, we will try to warp each picture so that the eyes and lips are always in the sample place in the image. This will make it a lot easier for us to compare faces in the next steps.

To do this, we are going to use an algorithm called **face landmark estimation**. There are lots of ways to do this, but we are going to use the approach invented in 2014 by Vahid Kazemi and Josephine Sullivan.

The basic idea is we will come up with 68 specific points (called *landmarks*) that exist on every face — the top of the chin, the outside edge of each eye, the inner edge of each eyebrow, etc. Then we will train a machine learning algorithm to be able to find these 68 specific points on any face:

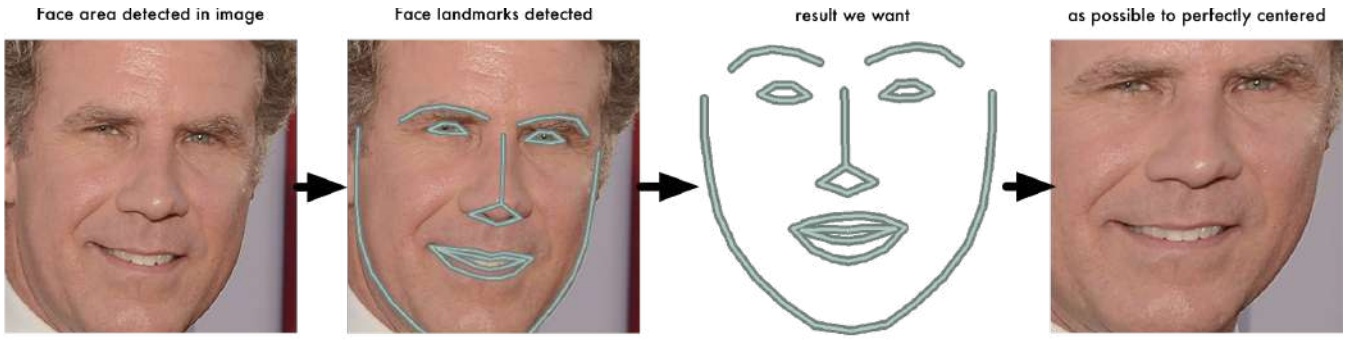
The 68 landmarks we will locate on every face. This image was created by Brandon Amos of CMU who works on OpenFace.

Here's the result of locating the 68 face landmarks on our test image:



PROTIP: You can also use this same technique to implement your own version of Snapchat's real-time 3d face filters!

Now that we know were the eyes and mouth are, we'll simply rotate, scale and shear the image so that the eyes and mouth are centered as best as possible. We won't do any fancy 3d warps because that would introduce distortions into the image. We are only going to use basic image transformations like rotation and scale that preserve parallel lines (called affine transformations):



Now no matter how the face is turned, we are able to center the eyes and mouth are in roughly the same position in the image. This will make our next step a lot more accurate.

If you want to try this step out yourself using Python and dlib, here's the code for finding face landmarks and here's the code for transforming the image using those landmarks.

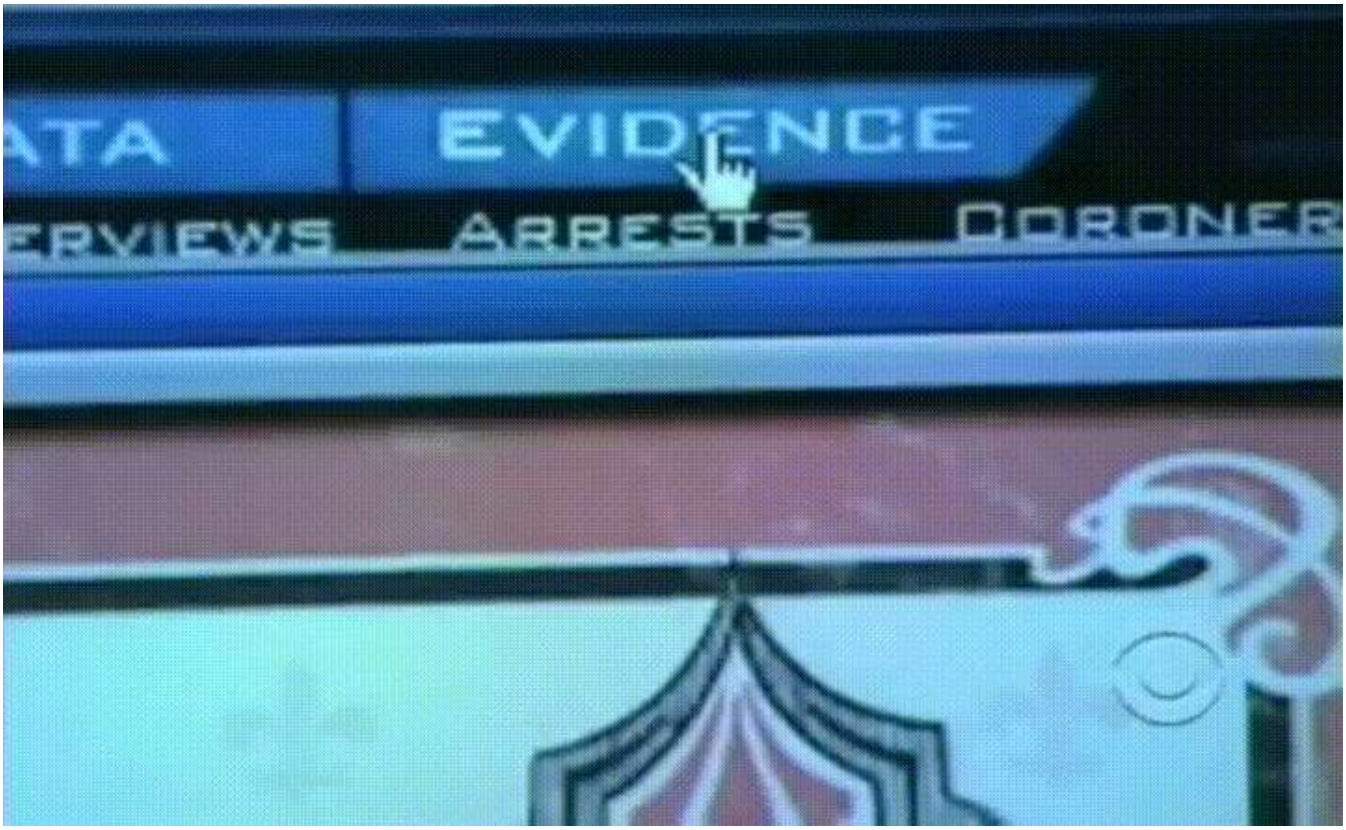
Step 3: Encoding Faces

Now we are to the meat of the problem — actually telling faces apart. This is where things get really interesting!

The simplest approach to face recognition is to directly compare the unknown face we found in Step 2 with all the pictures we have of people that have already been tagged. When we find a previously tagged face that looks very similar to our unknown face, it must be the same person. Seems like a pretty good idea, right?

There's actually a huge problem with that approach. A site like Facebook with billions of users and a trillion photos can't possibly loop through every previous-tagged face to compare it to every newly uploaded picture. That would take way too long. They need to be able to recognize faces in milliseconds, not hours.

What we need is a way to extract a few basic measurements from each face. Then we could measure our unknown face the same way and find the known face with the closest measurements. For example, we might measure the size of each ear, the spacing between the eyes, the length of the nose, etc. If you've ever watched a bad crime show like CSI, you know what I am talking about:



Just like TV! So real! #science

The most reliable way to measure a face

Ok, so which measurements should we collect from each face to build our known face database? Ear size? Nose length? Eye color? Something else?

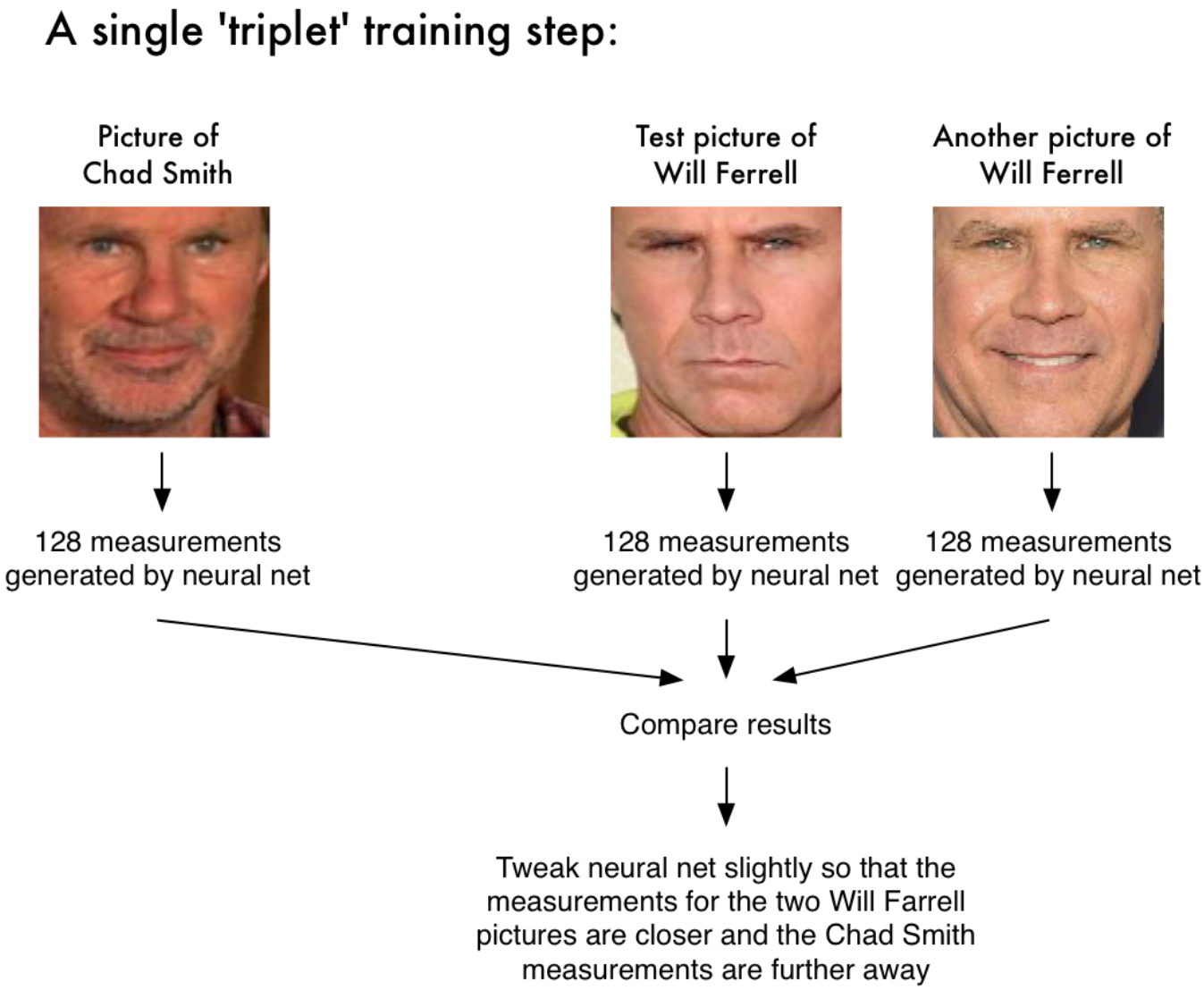
It turns out that the measurements that seem obvious to us humans (like eye color) don't really make sense to a computer looking at individual pixels in an image. Researchers have discovered that the most accurate approach is to let the computer figure out the measurements to collect itself. Deep learning does a better job than humans at figuring out which parts of a face are important to measure.

The solution is to train a Deep Convolutional Neural Network (just like we did in Part 3). But instead of training the network to recognize pictures objects like we did last time, we are going to train it to generate 128 measurements for each face.

The training process works by looking at 3 face images at a time:

1. Load a training face image of a known person
2. Load another picture of the same known person
3. Load a picture of a totally different person

Then the algorithm looks at the measurements it is currently generating for each of those three images. It then tweaks the neural network slightly so that it makes sure the measurements it generates for #1 and #2 are slightly closer while making sure the measurements for #2 and #3 are slightly further apart:



After repeating this step millions of times for millions of images of thousands of different people, the neural network learns to reliably generate 128 measurements for each person. Any ten different pictures of the same person should give roughly the same measurements.

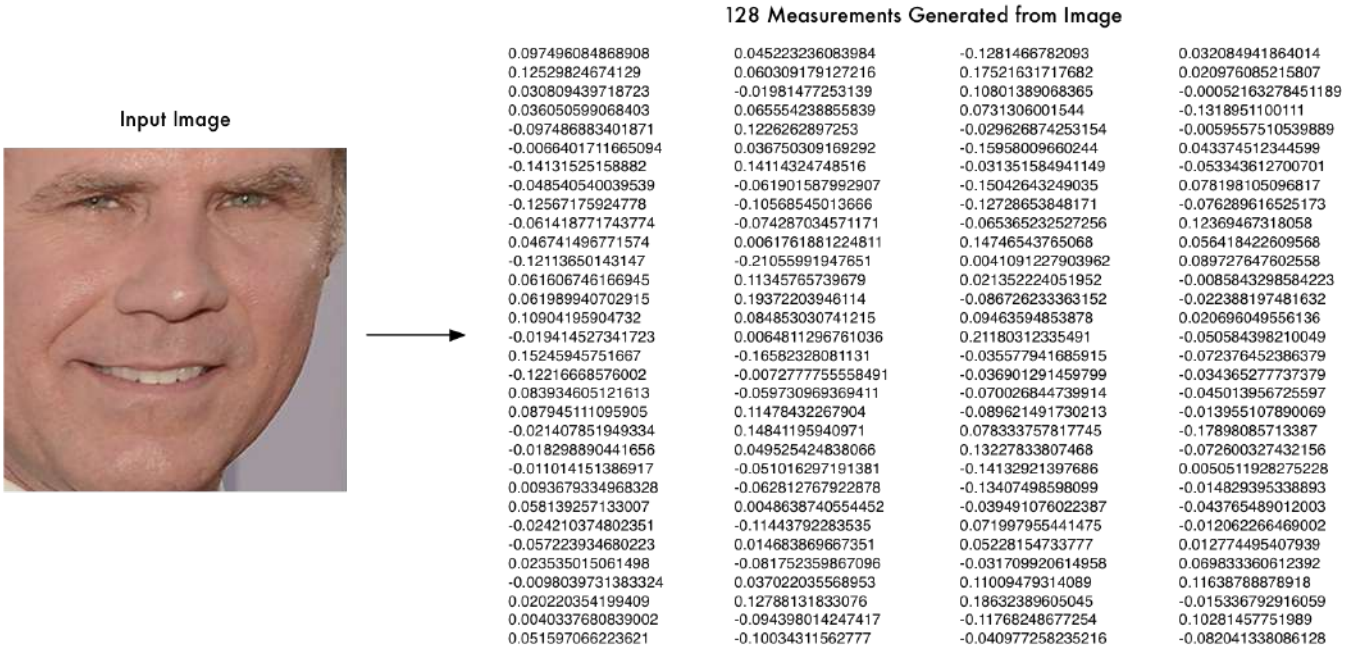
Machine learning people call the 128 measurements of each face an **embedding**. The idea of reducing complicated raw data like a picture into a list of computer-generated numbers comes up a lot in machine learning (especially in language translation). The exact approach for faces we are using was invented in 2015 by researchers at Google but many similar approaches exist.

Encoding our face image

This process of training a convolutional neural network to output face embeddings requires a lot of data and computer power. Even with an expensive NVidia Telsa video card, it takes about 24 hours of continuous training to get good accuracy.

But once the network has been trained, it can generate measurements for any face, even ones it has never seen before! So this step only needs to be done once. Lucky for us, the fine folks at OpenFace already did this and they published several trained networks which we can directly use. Thanks Brandon Amos and team!

So all we need to do ourselves is run our face images through their pre-trained network to get the 128 measurements for each face. Here's the measurements for our test image:



So what parts of the face are these 128 numbers measuring exactly? It turns out that we have no idea. It doesn't really matter to us. All that we care is that the network generates nearly the same numbers when looking at two different pictures of the same person.

If you want to try this step yourself, OpenFace provides a lua script that will generate embeddings all images in a folder and write them to a csv file. You run it like this.

Step 4: Finding the person's name from the encoding

This last step is actually the easiest step in the whole process. All we have to do is find the person in our database of known people who has the closest measurements to our

test image.

You can do that by using any basic machine learning classification algorithm. No fancy deep learning tricks are needed. We'll use a simple linear SVM classifier, but lots of classification algorithms could work.

All we need to do is train a classifier that can take in the measurements from a new test image and tells which known person is the closest match. Running this classifier takes milliseconds. The result of the classifier is the name of the person!

So let's try out our system. First, I trained a classifier with the embeddings of about 20 pictures each of Will Ferrell, Chad Smith and Jimmy Falon:

Sweet, sweet training data!

Then I ran the classifier on every frame of the famous youtube video of Will Ferrell and Chad Smith pretending to be each other on the Jimmy Fallon show:

It works! And look how well it works for faces in different poses — even sideways faces!

Running this Yourself

Let's review the steps we followed:

1. Encode a picture using the HOG algorithm to create a simplified version of the image. Using this simplified image, find the part of the image that most looks like a generic HOG encoding of a face.
2. Figure out the pose of the face by finding the main landmarks in the face. Once we find those landmarks, use them to warp the image so that the eyes and mouth are centered.
3. Pass the centered face image through a neural network that knows how to measure features of the face. Save those 128 measurements.
4. Looking at all the faces we've measured in the past, see which person has the closest measurements to our face's measurements. That's our match!

Now that you know how this all works, here's instructions from start-to-finish of how run this entire face recognition pipeline on your own computer:

UPDATE 4/9/2017: You can still follow the steps below to use OpenFace. However, I've released a new Python-based face recognition library called face_recognition that is much easier to install and use. So I'd recommend trying out face_recognition first instead of continuing below!

I even put together a pre-configured virtual machine with face_recognition, OpenCV, TensorFlow and lots of other deep learning tools pre-installed. You can download and run it on your computer very easily. Give the virtual machine a shot if you don't want to install all these libraries yourself!

Original OpenFace instructions:

Before you start

Make sure you have python, OpenFace and dlib installed. You can either install them manually or use a preconfigured docker image that has everying already installed:

```
docker pull bamos/openface
```

```
docker run -p 9000:9000 -p 8000:8000 -t -i bamos/openface /bin/bash
cd /root/openface
```

Pro-tip: If you are using Docker on OSX, you can make your OSX /Users/ folder visible inside a docker image like this:

```
docker run -v /Users:/host/Users -p 9000:9000 -p 8000:8000 -t -i bamos/openface /bin/b
cd /root/openface
```

Then you can access all your OSX files inside of the docker image at `/host/Users/...`

```
ls /host/Users/
```

Step 1

Make a folder called `./training-images/` inside the openface folder.

```
mkdir training-images
```

Step 2

Make a subfolder for each person you want to recognize. For example:

```
mkdir ./training-images/will-ferrell/
mkdir ./training-images/chad-smith/
mkdir ./training-images/jimmy-fallon/
```

Step 3

Copy all your images of each person into the correct sub-folders. Make sure only one face appears in each image. There's no need to crop the image around the face. OpenFace will do that automatically.

Step 4

Run the openface scripts from inside the openface root directory:

First, do pose detection and alignment:

```
./util/align-dlib.py ./training-images/ align outerEyesAndNose ./aligned-images/ --size
```

This will create a new `./aligned-images/` subfolder with a cropped and aligned version of each of your test images.

Second, generate the representations from the aligned images:

```
./batch-represent/main.lua -outDir ./generated-embeddings/ -data ./aligned-images/
```

After you run this, the `./generated-embeddings/` sub-folder will contain a csv file with the embeddings for each image.

Third, train your face detection model:

```
./demos/classifier.py train ./generated-embeddings/
```

This will generate a new file called `./generated-embeddings/classifier.pkl`. This file has the SVM model you'll use to recognize new faces.

At this point, you should have a working face recognizer!

Step 5: Recognize faces!

Get a new picture with an unknown face. Pass it to the classifier script like this:

```
./demos/classifier.py infer ./generated-embeddings/classifier.pkl your_test_image.jpg
```

You should get a prediction that looks like this:

```
=== /test-images/will-ferrel-1.jpg ===
Predict will-ferrell with 0.73 confidence.
```

From here it's up to you to adapt the `./demos/classifier.py` python script to work however you want.

Important notes:

- If you get bad results, try adding a few more pictures of each person in Step 3 (especially picures in different poses).
- This script will *always* make a prediction even if the face isn't one it knows. In a real application, you would look at the confidence score and throw away predictions with a low confidence since they are most likely wrong.

• • •

If you liked this article, please consider signing up for my Machine Learning is Fun! newsletter:

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 5!

Machine Learning Artificial Intelligence Deep Learning

Machine Learning is Fun Part 5: Language Translation with Deep Learning and the Magic of Sequences

Adam Geitgey
Aug 22, 2016 · 16 min read

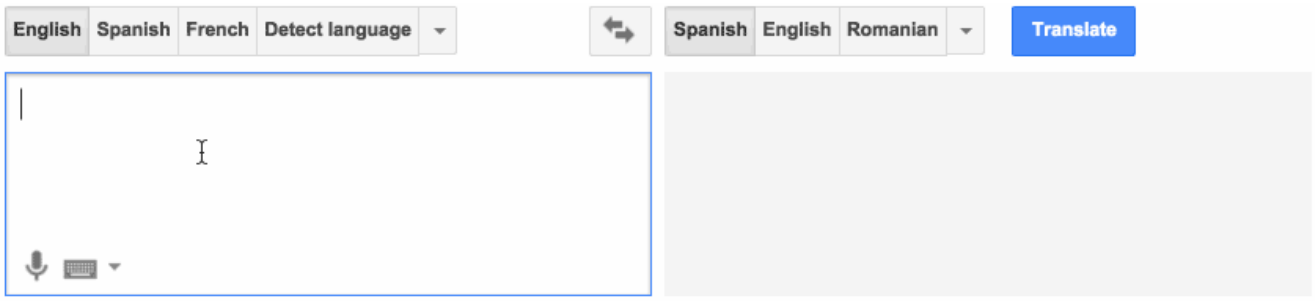
Update: *This article is part of a series. Check out the full series: Part 1, Part 2, Part 3, Part 4, Part 5, Part 6, Part 7 and Part 8! You can also read this article in 普通话, Русский, 한국어, Tiê´ng Việt or Italiano.*

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!*

We all know and love Google Translate, the website that can instantly translate between 100 different human languages as if by magic. It is even available on our phones and smartwatches:

The technology behind Google Translate is called Machine Translation. It has changed the world by allowing people to communicate when it wouldn't otherwise be possible.

But we all know that high school students have been using Google Translate to… umm… *assist* with their Spanish homework for 15 years. Isn't this old news?



It turns out that over the past two years, deep learning has totally rewritten our approach to machine translation. Deep learning researchers who know almost nothing about language translation are throwing together relatively simple machine learning solutions that are beating the best expert-built language translation systems in the world.

The technology behind this breakthrough is called **sequence-to-sequence learning**. It's very powerful technique that be used to solve many kinds problems. After we see how it is used for translation, we'll also learn how the exact same algorithm can be used to write AI chat bots and describe pictures.

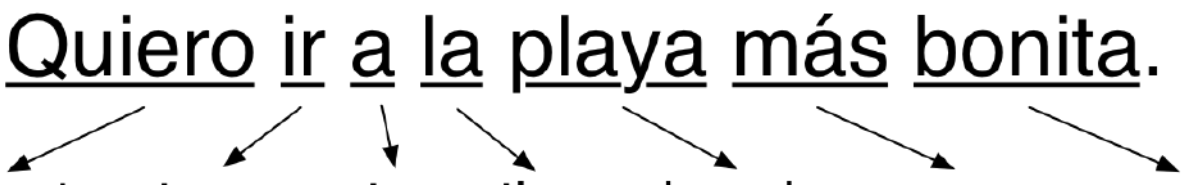
Let's go!

. . .

Making Computers Translate

So how do we program a computer to translate human language?

The simplest approach is to replace every word in a sentence with the translated word in the target language. Here's a simple example of translating from Spanish to English word-by-word:

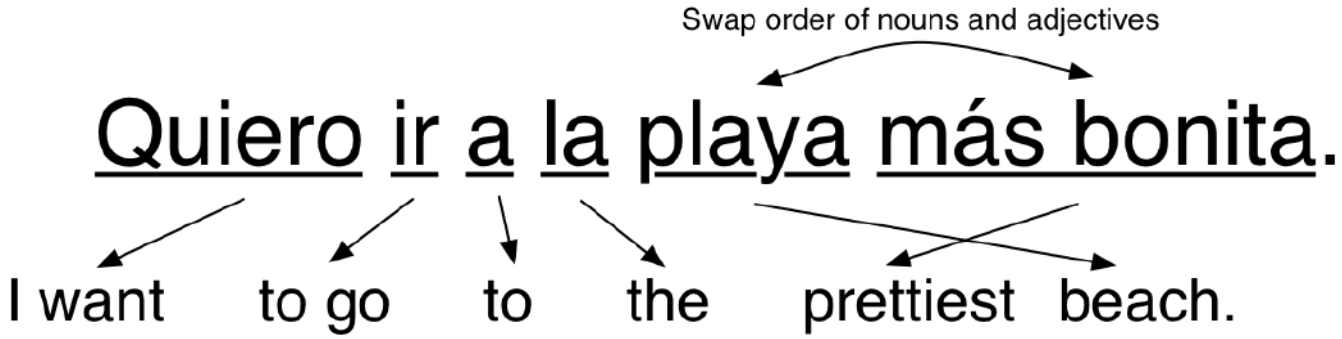


I want to go to the beach more pretty.

We just replace each Spanish word with the matching English word.

This is easy to implement because all you need is a dictionary to look up each word's translation. But the results are bad because it ignores grammar and context.

So the next thing you might do is start adding language-specific rules to improve the results. For example, you might translate common two-word phrases as a single group. And you might swap the order nouns and adjectives since they usually appear in reverse order in Spanish from how they appear in English:



That worked! If we just keep adding more rules until we can handle every part of grammar, our program should be able to translate any sentence, right?

This is how the earliest machine translation systems worked. Linguists came up with complicated rules and programmed them in one-by-one. Some of the smartest linguists in the world labored for years during the Cold War to create translation systems as a way to interpret Russian communications more easily.

Unfortunately this only worked for simple, plainly-structured documents like weather reports. It didn't work reliably for real-world documents.

The problem is that human language doesn't follow a fixed set of rules. Human languages are full of special cases, regional variations, and just flat out rule-breaking. The way we speak English more influenced by who invaded who hundreds of years ago than it is by someone sitting down and defining grammar rules.

Making Computers Translate Better Using Statistics

After the failure of rule-based systems, new translation approaches were developed using models based on probability and statistics instead of grammar rules.

Building a statistics-based translation system requires lots of training data where the exact same text is translated into at least two languages. This double-translated text is called *parallel corpora*. In the same way that the Rosetta Stone was used by scientists in the 1800s to figure out Egyptian hieroglyphs from Greek, computers can use parallel corpora to guess how to convert text from one language to another.

Luckily, there's lots of double-translated text already sitting around in strange places. For example, the European Parliament translates their proceedings into 21 languages. So researchers often use that data to help build translation systems.

English	Spanish
Resumption of the session I declare resumed the session of the European Parliament adjourned on Friday 17 December 1999, and I would like once again to wish you a happy new year in the hope that you enjoyed a pleasant festive period.	Reanudación del período de sesiones Declaro reanudado el período de sesiones del Parlamento Europeo, interrumpido el viernes 17 de diciembre pasado, y reitero a Sus Señorías mi deseo de que hayan tenido unas buenas vacaciones.

Training data is usually exciting! But this is just millions and millions of lines of dry government documents...

Thinking in Probabilities

The fundamental difference with statistical translation systems is that they don't try to generate one exact translation. Instead, they generate thousands of possible translations and then they rank those translations by likely each is to be correct. They estimate how "correct" something is by how similar it is to the training data. Here's how it works:

Step 1: Break original sentence into chunks

First, we break up our sentence into simple chunks that can each be easily translated:

Quiero ir a la playa más bonita.

Step 2: Find all possible translations for each chunk

Next, we will translate each of these chunks by finding all the ways humans have translated those same chunks of words in our training data.

It's important to note that we are not just looking up these chunks in a simple translation dictionary. Instead, we are seeing how actual people translated these same

chunks of words in real-world sentences. This helps us capture all of the different ways they can be used in different contexts:

Even the most common phrases have lots of possible translations.

Some of these possible translations are used more frequently than others. Based on how frequently each translation appears in our training data, we can give it a score.

For example, it's much more common for someone to say "Quiero" to mean "I want" than to mean "I try." So we can use how frequently "Quiero" was translated to "I want" in our training data to give that translation more weight than a less frequent translation.

Step 3: Generate all possible sentences and find the most likely one

Next, we will use every possible combination of these chunks to generate a bunch of possible sentences.

Just from the chunk translations we listed in Step 2, we can already generate nearly 2,500 different variations of our sentence by combining the chunks in different ways. Here are some examples:

> *I love | to leave | at | the seaside | more tidy.*
> *I mean | to be on | to | the open space | most lovely.*
> *I like | to be |on | per the seaside | more lovely.*
> *I mean | to go | to | the open space | most tidy.*

But in a real-world system, there will be even more possible chunk combinations because we'll also try different orderings of words and different ways of chunking the sentence:

> *I try* | *to run* | *at* | *the prettiest* | *open space.*
> *I want* | *to run* | *per* | *the more tidy* | *open space.*
> *I mean* | *to forget* | *at* | *the tidiest* | *beach.*
> *I try* | *to go* | *per* | *the more tidy* | *seaside.*

Now need to scan through all of these generated sentences to find the one that is that sounds the "most human."

To do this, we compare each generated sentence to millions of real sentences from books and news stories written in English. The more English text we can get our hands on, the better.

Take this possible translation:

> *I try* | *to leave* | *per* | *the most lovely* | *open space.*

It's likely that no one has ever written a sentence like this in English, so it would not be very similar to any sentences in our data set. We'll give this possible translation a low probability score.

But look at this possible translation:

> *I want* | *to go* | *to* | *the prettiest* | *beach.*

This sentence will be similar to something in our training set, so it will get a high probability score.

After trying all possible sentences, we'll pick the sentence that has the most likely chunk translations while also being the most similar overall to real English sentences.

Our final translation would be "I want to go to the prettiest beach." Not bad!

Statistical Machine Translation was a Huge Milestone

Statistical machine translation systems perform much better than rule-based systems if you give them enough training data. Franz Josef Och improved on these ideas and used

them to build Google Translate in the early 2000s. Machine Translation was finally available to the world.

In the early days, it was surprising to everyone that the "dumb" approach to translating based on probability worked better than rule-based systems designed by linguists. This led to a (somewhat mean) saying among researchers in the 80s:

> *"Every time I fire a linguist, my accuracy goes up."*
>
> *— Frederick Jelinek*

The Limitations of Statistical Machine Translation

Statistical machine translation systems work well, but they are complicated to build and maintain. Every new pair of languages you want to translate requires experts to tweak and tune a new multi-step translation pipeline.

Because it is so much work to build these different pipelines, trade-offs have to be made. If you are asking Google to translate Georgian to Telegu, it has to internally translate it into English as an intermediate step because there's not enough Georgain-to-Telegu translations happening to justify investing heavily in that language pair. And it might do that translation using a less advanced translation pipeline than if you had asked it for the more common choice of French-to-English.

Wouldn't it be cool if we could have the computer do all that annoying development work for us?

Making Computers Translate Better — Without all those Expensive People

The holy grail of machine translation is a black box system that learns how to translate by itself— just by looking at training data. With Statistical Machine Translation, humans are still needed to build and tweak the multi-step statistical models.

In 2014, KyungHyun Cho's team made a breakthrough. They found a way to apply deep learning to build this black box system. Their deep learning model takes in a *parallel corpora* and and uses it to learn how to translate between those two languages without any human intervention.

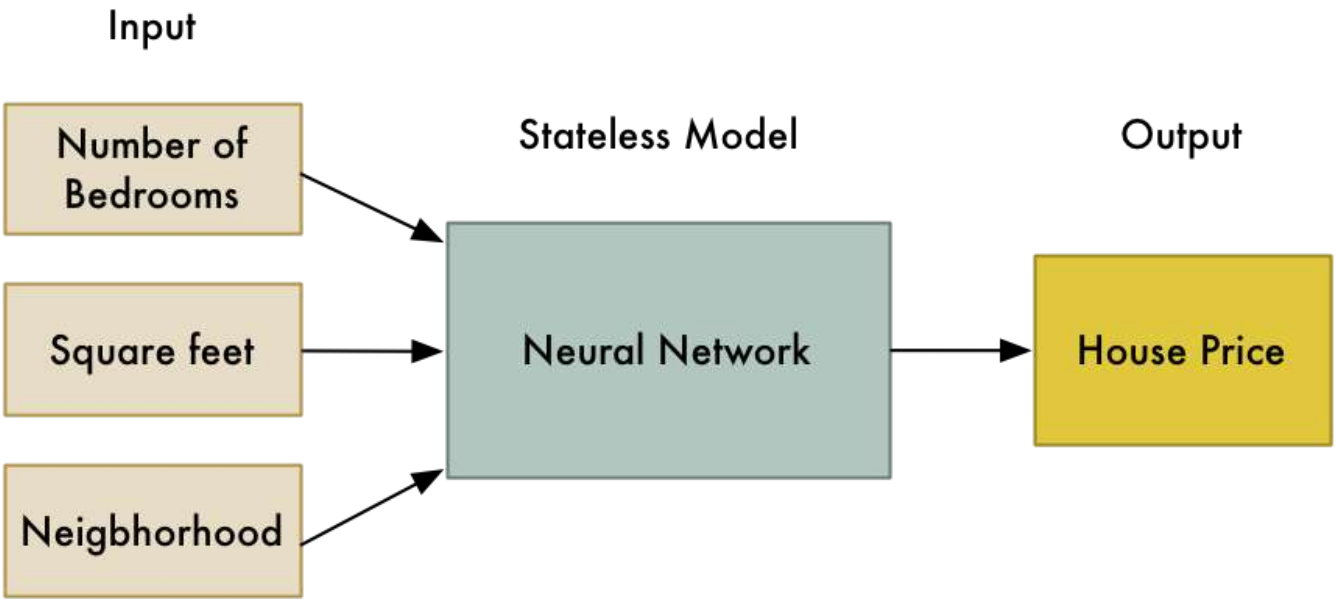
Two big ideas make this possible — *recurrent neural networks* and *encodings*. By combining these two ideas in a clever way, we can build a self-learning translation

system.

Recurrent Neural Networks

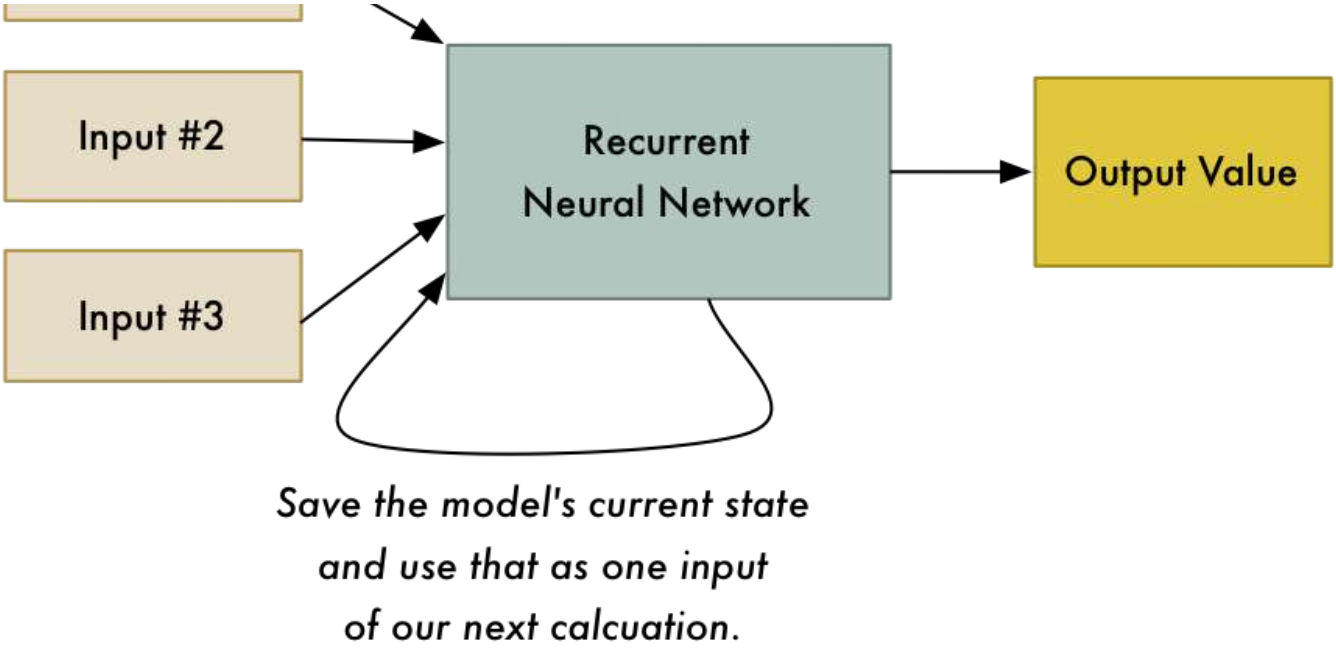
We've already talked about recurrent neural networks in Part 2, but let's quickly review.

A regular (non-recurrent) neural network is a generic machine learning algorithm that takes in a list of numbers and calculates a result (based on previous training). Neural networks can be used as a black box to solve lots of problems. For example, we can use a neural network to calculate the approximate value of a house based on attributes of that house:



But like most machine learning algorithms, neural networks are *stateless*. You pass in a list of numbers and the neural network calculates a result. If you pass in those same numbers again, it will always calculate the same result. It has no memory of past calculations. In other words, 2 + 2 always equals 4.

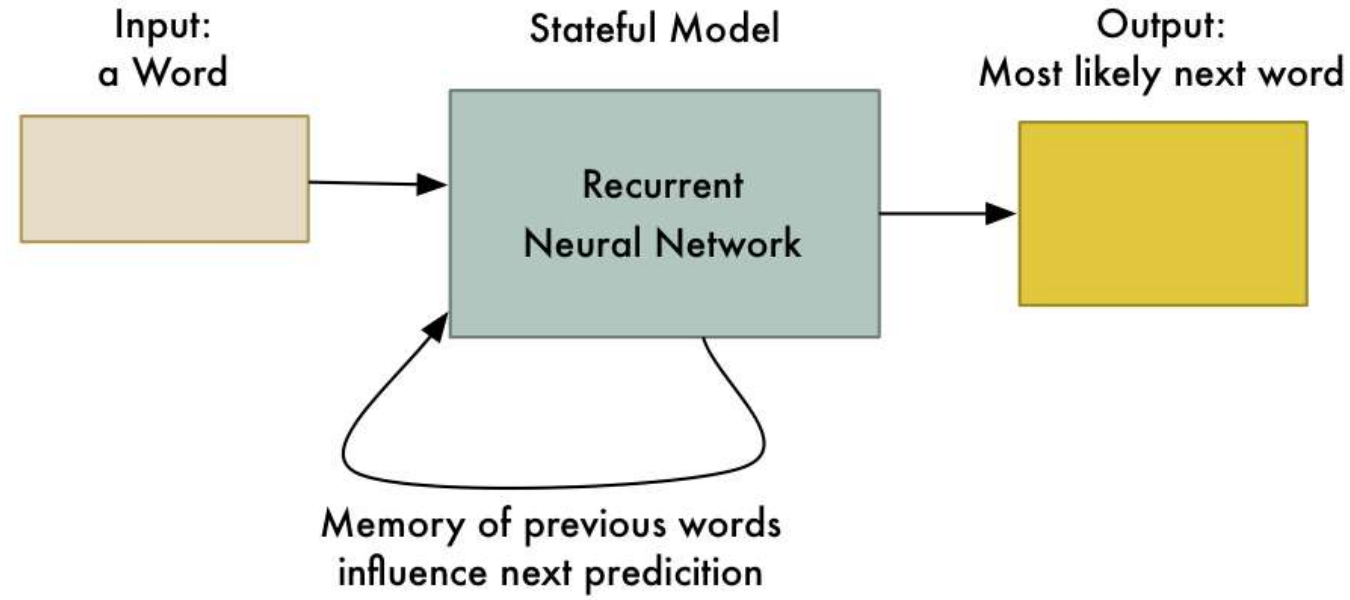
A *recurrent neural network* (or *RNN* for short) is a slightly tweaked version of a neural network where the previous state of the neural network is one of the inputs to the next calculation. This means that previous calculations change the results of future calculations!



Humans hate him: 1 weird trick that makes machines smarter!

Why in the world would we want to do this? Shouldn't 2 + 2 always equal 4 no matter what we last calculated?

This trick allows neural networks to learn patterns in a sequence of data. For example, you can use it to predict the next most likely word in a sentence based on the first few words:



Output so far:

Machine

This is one way you could implement "autocorrect" for a smart phone's keyboard app

RNNs are useful any time you want to learn patterns in data. Because human language is just one big, complicated pattern, RNNs are increasingly used in many areas of natural language processing.

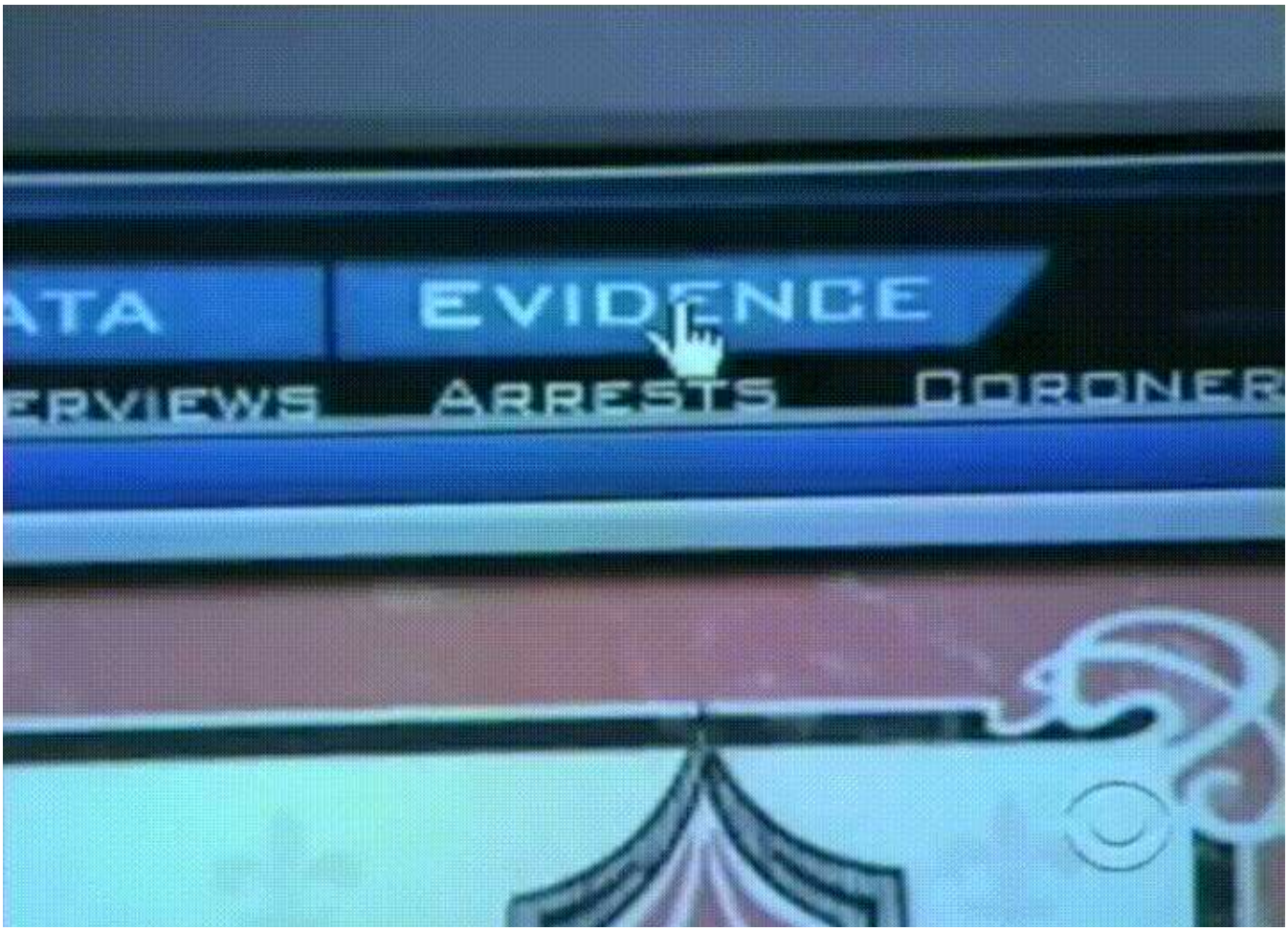
If you want to learn more about RNNs, you can read Part 2 where we used one to generate a fake Ernest Hemingway book and then used another one to generate fake Super Mario Brothers levels.

Encodings

The other idea we need to review is *Encodings*. We talked about encodings in Part 4 as part of face recognition. To explain encodings, let's take a slight detour into how we can tell two different people apart with a computer.

When you are trying to tell two faces apart with a computer, you collect different measurements from each face and use those measurements to compare faces. For example, we might measure the size of each ear or the spacing between the eyes and compare those measurements from two pictures to see if they are the same person.

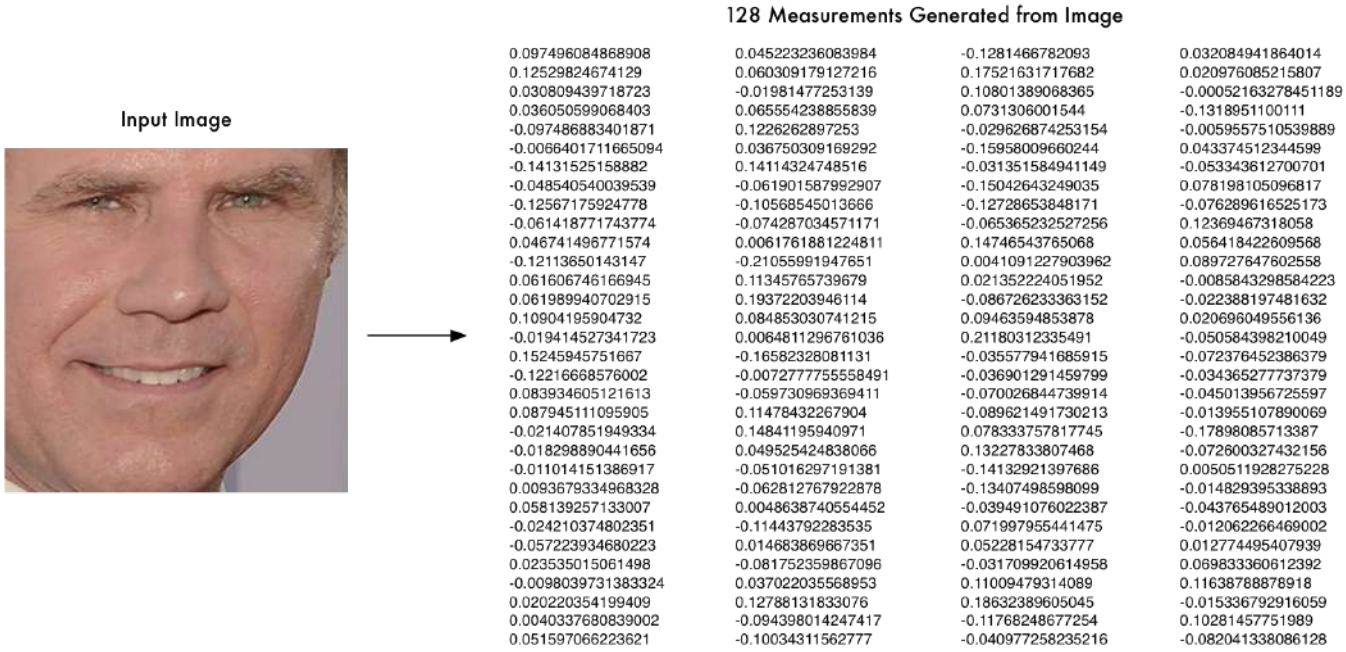
You're probably already familiar with this idea from watching any primetime detective show like CSI:



I love this dumb gif from CSI so much that I'll use it again — because it is somehow manages to demonstrate this idea clearly while also being total nonsense.

The idea of turning a face into a list of measurements is an example of an *encoding*. We are taking raw data (a picture of a face) and turning it into a list of measurements that represent it (the encoding).

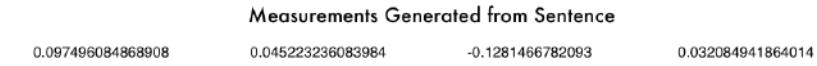
But like we saw in Part 4, we don't have to come up with a specific list of facial features to measure ourselves. Instead, we can use a neural network to generate measurements from a face. The computer can do a better job than us in figuring out which measurements are best able to differentiate two similar people:



These facial feature measurements are generated by a neural net that was trained to make sure different people's faces resulted in different numbers.

This is our *encoding*. It lets us represent something very complicated (a picture of a face) with something simple (128 numbers). Now comparing two different faces is much easier because we only have to compare these 128 numbers for each face instead of comparing full images.

Guess what? We can do the same thing with sentences! We can come up with an encoding that represents every possible different sentence as a series of unique numbers:



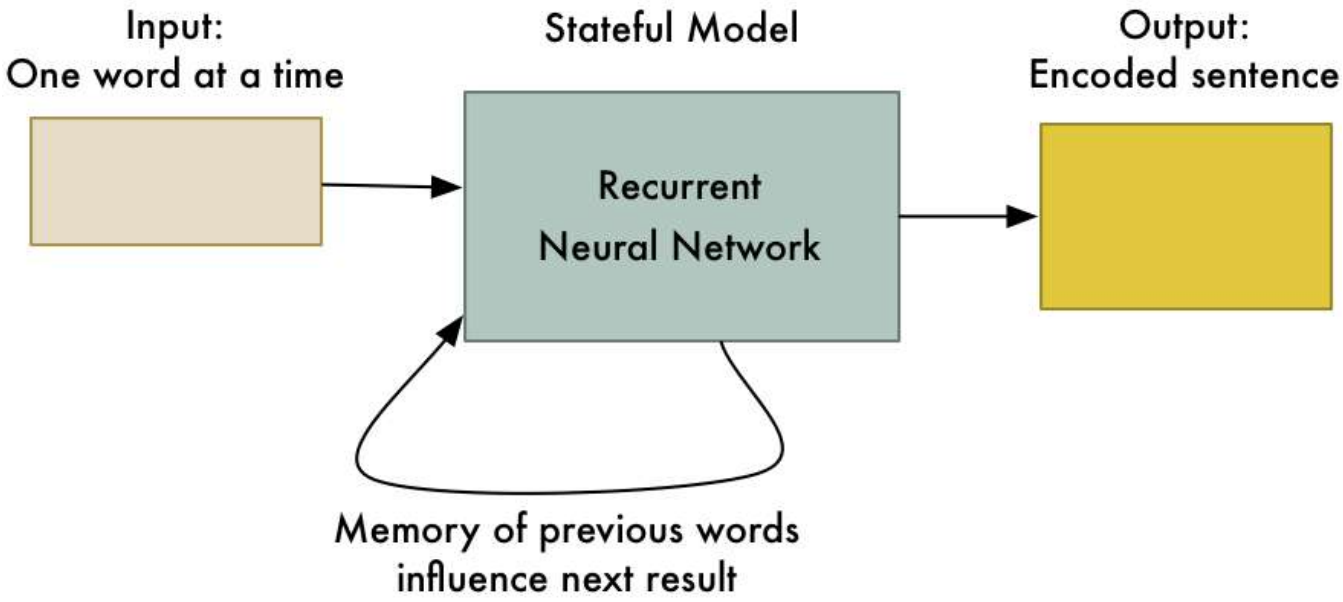
Input Sentence

"Machine Learning is Fun!" →

0.12529824674129	0.060309179127216	0.17521631717682	0.020976085215807
0.030809439718723	-0.01981477253139	0.10801389068365	-0.00052163278451189
0.036050599068403	0.065554238855839	0.0731306001544	-0.1318951100111
-0.097486883401871	0.1226262897253	-0.029626874253154	-0.0059557510539889
-0.0066401711665094	0.036750309169292	-0.15958009660244	0.043374512344599
-0.14131525158882	0.14114324748516	-0.031351584941149	-0.053343612700701
-0.048540540039539	-0.061901587992907	-0.15042643249035	0.078198105096817
-0.12567175924778	-0.10568545013666	-0.12728653848171	-0.076289616525173
-0.061418771743774	-0.074287034571171	-0.065365232527256	0.12369467318058
0.046741496771574	0.0061761881224811	0.14746543765068	0.056418422609568
-0.12113650143147	-0.21055991947651	0.0041091227903962	0.089727647602558
0.061606746166945	0.11345765739679	0.021352224051952	-0.0085843298584223
0.061989940702915	0.19372203946114	-0.086726233363152	-0.022388197481632
0.10904195904732	0.084853030741215	0.09463594853878	0.020696049556136
-0.019414527341723	0.0064811296761036	0.21180312335491	-0.050584398210049
0.15245945751667	-0.16582328081131	-0.035577941685915	-0.072376452386379
-0.12216668576002	-0.0072777755558491	-0.036901291459799	-0.034365277737379
0.083934605121613	-0.059730969369411	-0.070026844739914	-0.045013956725597
0.087945111095905	0.11478432267904	-0.089621491730213	-0.013955107890069
-0.021407851949334	0.14841195940971	0.078333757817745	-0.17898085713387
-0.018298890441656	0.049525424838066	0.13227833807468	-0.072600327432156
-0.011014151386917	-0.051016297191381	-0.14132921397686	0.0050511928275228
0.0093679334968328	-0.062812767922878	-0.13407498598099	-0.014829395338893
0.058139257133007	0.0048638740554452	-0.039491076022387	-0.043765489012003
-0.024210374802351	-0.11443792283535	0.071997955441475	-0.012062266469002
-0.057223934680223	0.014683869667351	0.05228154733777	0.012774495407939
0.023535015061498	-0.081752359867096	-0.031709920614958	0.069833360612392
-0.0098039731383324	0.037022035568953	0.11009479314089	0.11638788878918
0.020220354199409	0.12788131833076	0.18632389605045	-0.015336792916059
0.0040337680839002	-0.094398014247417	-0.11768248677254	0.10281457751989
0.051597066223621	-0.10034311562777	-0.040977258235216	-0.082041338086128

This list of numbers represents the English sentence "Machine Learning is Fun!". A different sentence would be represented by a different set of numbers.

To generate this encoding, we'll feed the sentence into the RNN, one word at time. The final result after the last word is processed will be the values that represent the entire sentence:



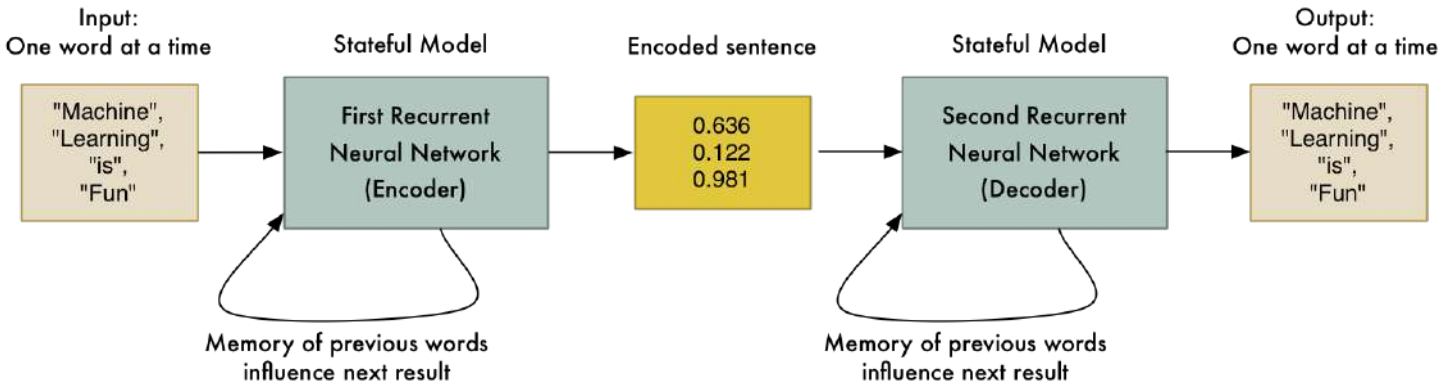
Because the RNN has a "memory" of each word that passed through it, the final encoding it calculates represents all the words in the sentence.

Great, so now we have a way to represent an entire sentence as a set of unique numbers! We don't know what each number in the encoding means, but it doesn't really matter. As long as each sentence is uniquely identified by it's own set of numbers, we don't need to know exactly how those numbers were generated.

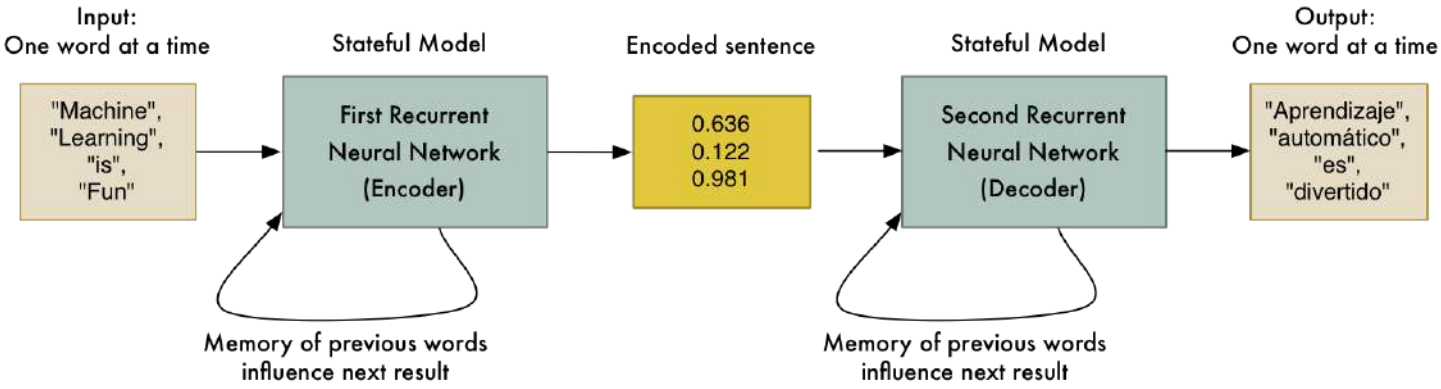
Let's Translate!

Ok, so we know how to use an RNN to encode a sentence into a set of unique numbers. How does that help us? Here's where things get really cool!

What if we took two RNNs and hooked them up end-to-end? The first RNN could generate the encoding that represents a sentence. Then the second RNN could take that encoding and just do the same logic in reverse to decode the original sentence again:



Of course being able to encode and then decode the original sentence again isn't very useful. But what if (*and here's the big idea!*) we could train the second RNN to decode the sentence into Spanish instead of English? We could use our *parallel corpora* training data to train it to do that:



And just like that, we have a generic way of converting a sequence of English words into an equivalent sequence of Spanish words!

This is a powerful idea:

- This approach is mostly limited by the amount of training data you have and the amount of computer power you can throw at it. Machine learning researchers only invented this **two years ago**, but it's already performing as well as statistical machine translation systems that took **20 years** to develop.

- This doesn't depend on knowing any rules about human language. The algorithm figures out those rules itself. This means you don't need experts to tune every step of your translation pipeline. The computer does that for you.

- ***This approach works for almost any kind of sequence-to-sequence problem!*** And it turns out that lots of interesting problems are sequence-to-sequence problems. Read on for other cool things you can do!

Note that we glossed over some things that are required to make this work with real-world data. For example, there's additional work you have to do to deal with different lengths of input and output sentences (see bucketing and padding). There's also issues with translating rare words correctly.

Building your own Sequence-to-Sequence Translation System

If you want to build your own language translation system, there's a working demo included with TensorFlow that will translate between English and French. However, this is not for the faint of heart or for those with limited budgets. This technology is still new and very resource intensive. Even if you have a fast computer with a high-end video card, it might take about a month of continuous processing time to train your own language translation system.

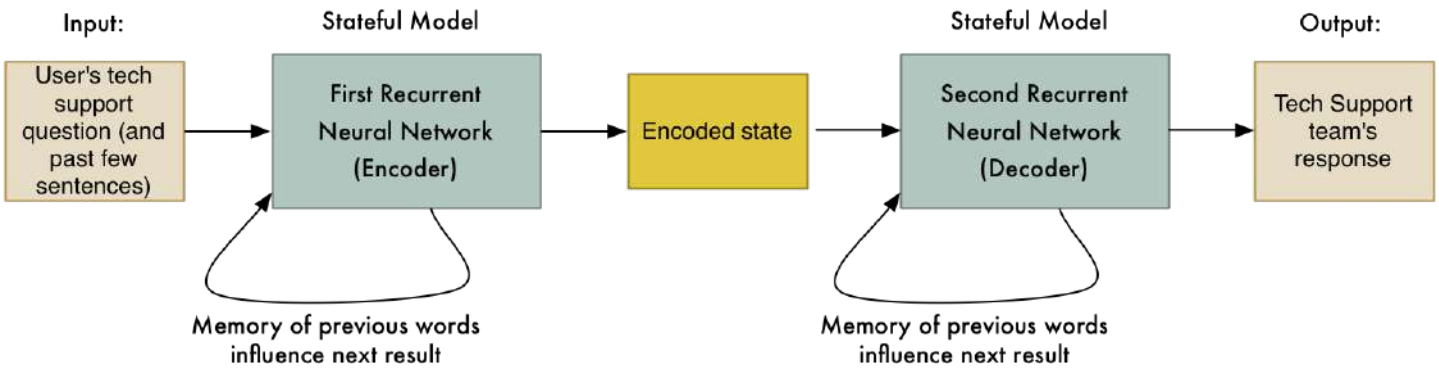
Also, Sequence-to-sequence language translation techniques are improving so rapidly that it's hard to keep up. Many recent improvements (like adding an attention mechanism or tracking context) are significantly improving results but these developments are so new that there aren't even wikipedia pages for them yet. If you want to do anything serious with sequence-to-sequence learning, you'll need to keep with new developments as they occur.

The Ridiculous Power of Sequence-to-Sequence Models

So what else can we do with sequence-to-sequence models?

About a year ago, researchers at Google showed that you can use sequence-to-sequence models to build AI bots. The idea is so simple that it's amazing it works at all.

First, they captured chat logs between Google employees and Google's Tech Support team. Then they trained a sequence-to-sequence model where the employee's question was the input sentence and the Tech Support team's response was the "translation" of that sentence.

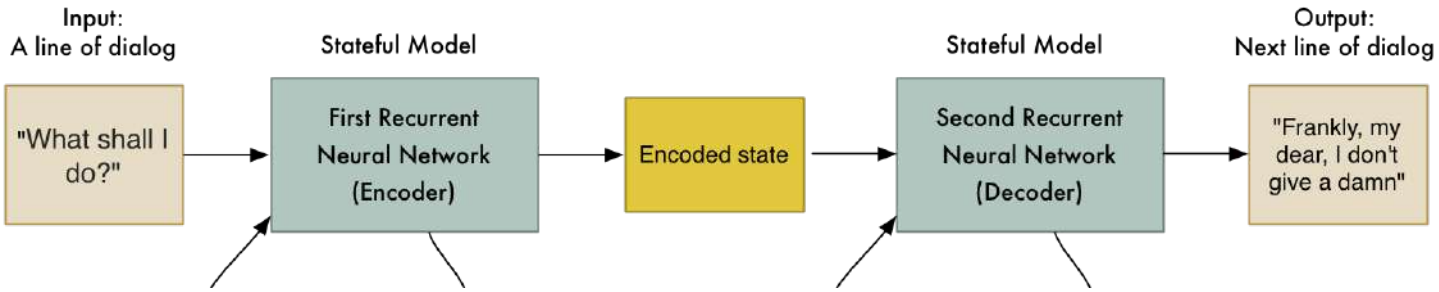


When a user interacted with the bot, they would "translate" each of the user's messages with this system to get the bot's response.

The end result was a semi-intelligent bot that could (sometimes) answer real tech support questions. Here's part of a sample conversation between a user and the bot from their paper:

> ***Human:*** *connection refused or something like that*
> ***Machine:*** *may i know the version of network connect you connect?*
> ***Human:*** *i am not sure i know that*
> ***Machine:*** *is the network connect prompting for any user certificate?*
> ***Human:*** *yes*
> ***Machine:*** *are you at home or at work at the moment? office?*
> ***Human:*** *i am at home*
> ***Machine:*** *try this. goto <URL redacted> page and enter your name and password*
> ***Human:*** *done*
> ***Machine:*** *try logging in with <NAME redacted> and check if its working fine now*
> ***Human:*** *yes, now it works!*
> ***Machine:*** *great. anything else that i can help?*

They also tried building a chat bot based on millions of movie subtitles. The idea was to use conversations between movie characters as a way to train a bot to talk like a human. The input sentence is a line of dialog said by one character and the "translation" is what the next character said in response:

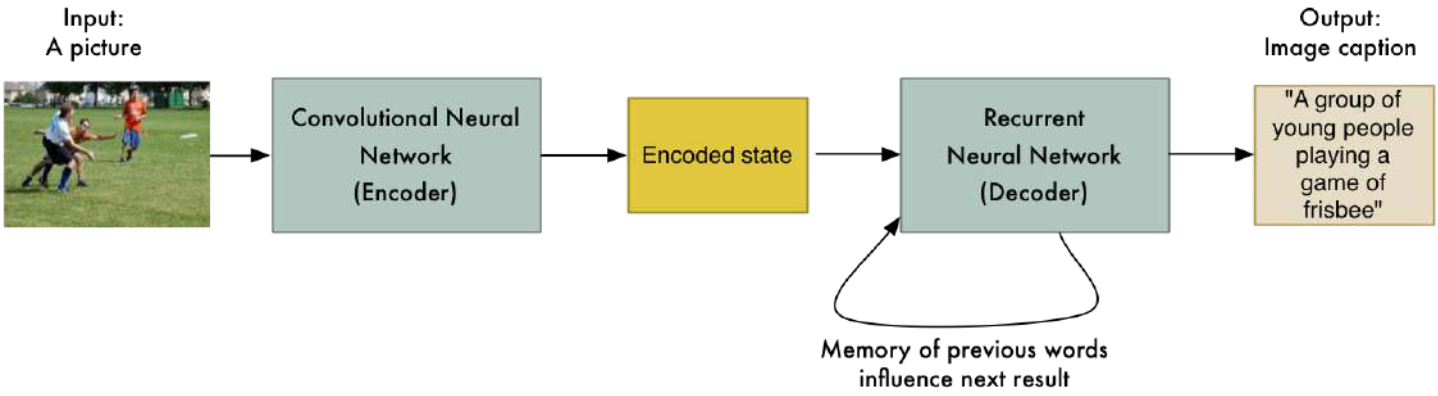


This produced really interesting results. Not only did the bot converse like a human, but it displayed a small bit of intelligence:

> ***Human:*** *my name is david. what is my name?*
> ***Machine:*** *david.*
> ***Human:*** *my name is john. what is my name?*
> ***Machine:*** *john.*
> ***Human:*** *are you a leader or a follower?*
> ***Machine:*** *i'm a leader.*
> ***Human:*** *are you a follower or a leader?*
> ***Machine:*** *i'm a leader.*

This is only the beginning of the possibilities. We aren't limited to converting one sentence into another sentence. It's also possible to make an image-to-sequence model that can turn an image into text!

A different team at Google did this by replacing the first RNN with a Convolutional Neural Network (like we learned about in Part 3). This allows the input to be a picture instead of a sentence. The rest works basically the same way:



And just like that, we can turn pictures into words (as long as we have lots and lots of training data)!

Andrej Karpathy expanded on these ideas to build a system capable of describing images in great detail by processing multiple regions of an image separately:

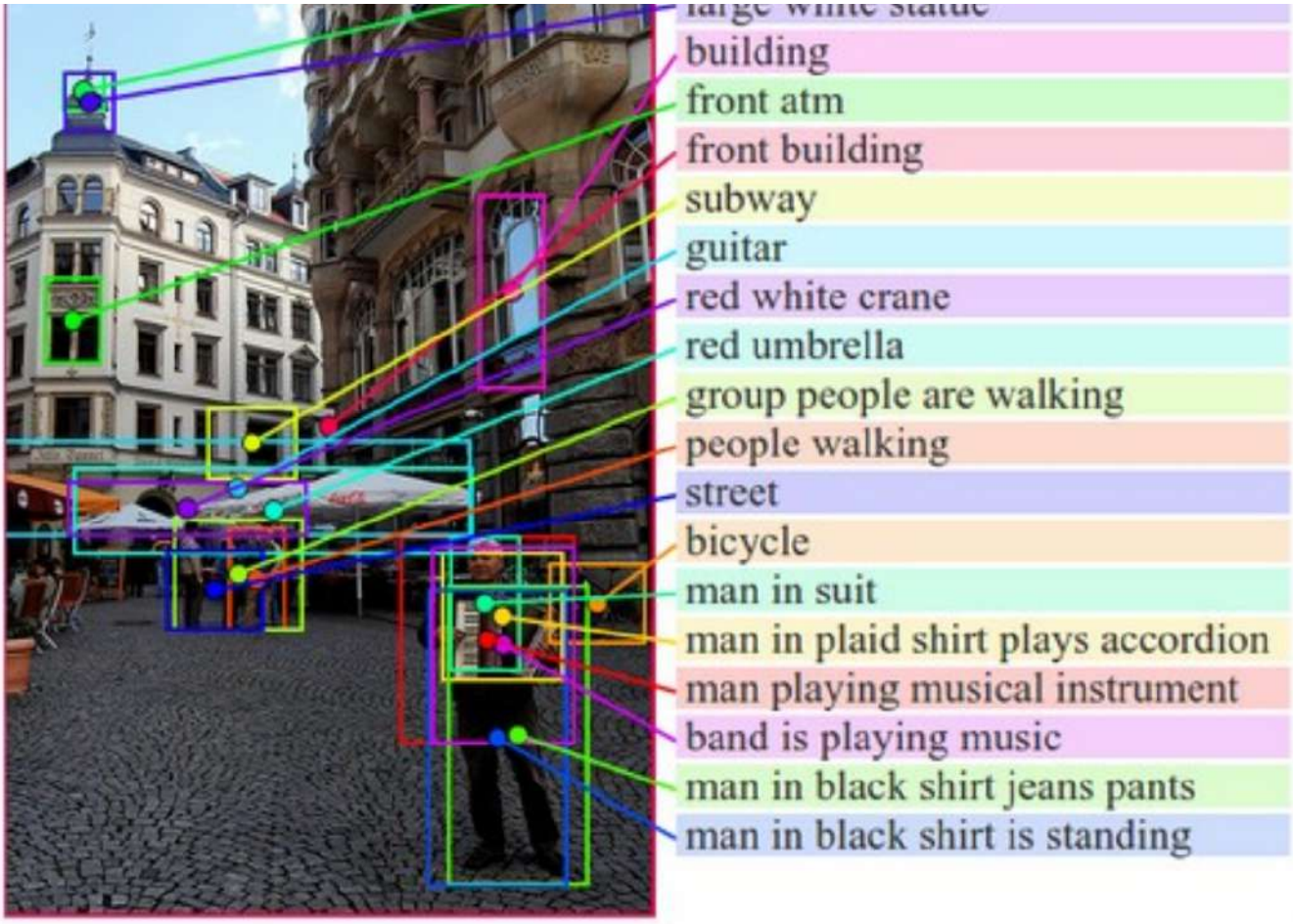
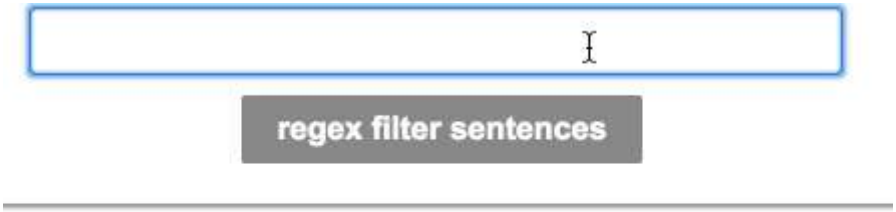


Image from this paper by Andrej Karpathy

This makes it possible to build image search engines that are capable of finding images that match oddly specific search queries:



Example from http://cs.stanford.edu/people/karpathy/deepimagesent/rankingdemo/

There's even researchers working on the reverse problem, generating an entire picture based on just a text description!

Just from these examples, you can start to imagine the possibilities. So far, there have been sequence-to-sequence applications in everything from speech recognition to computer vision. I bet there will be a lot more over the next year.

If you want to learn more in depth about sequence-to-sequence models and translation, here's some recommended resources:

- Richard Socher's CS224D Lecture— Fancy Recurrent Neural Networks for Machine Translation (video)
- Thang Luong's CS224D Lecture — Neural Machine Transation (PDF)
- TensorFlow's description of Seq2Seq modeling
- The Deep Learning Book's chapter on Sequence to Sequence Learning (PDF)

. . .

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 6!

Machine Learning Artificial Intelligence Translation

Machine Learning is Fun Part 6: How to do Speech Recognition with Deep Learning

Adam Geitgey
Dec 24, 2016 · 11 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 普通话, 한국어, Tiê´ng Việt or Русский.*

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!*

Speech recognition is invading our lives. It's built into our phones, our game consoles and our smart watches. It's even automating our homes. For just $50, you can get an Amazon Echo Dot — a magic box that allows you to order pizza, get a weather report or even buy trash bags — just by speaking out loud:

Alexa, order a large pizza!

The Echo Dot has been so popular this holiday season that Amazon can't seem to keep them in stock!

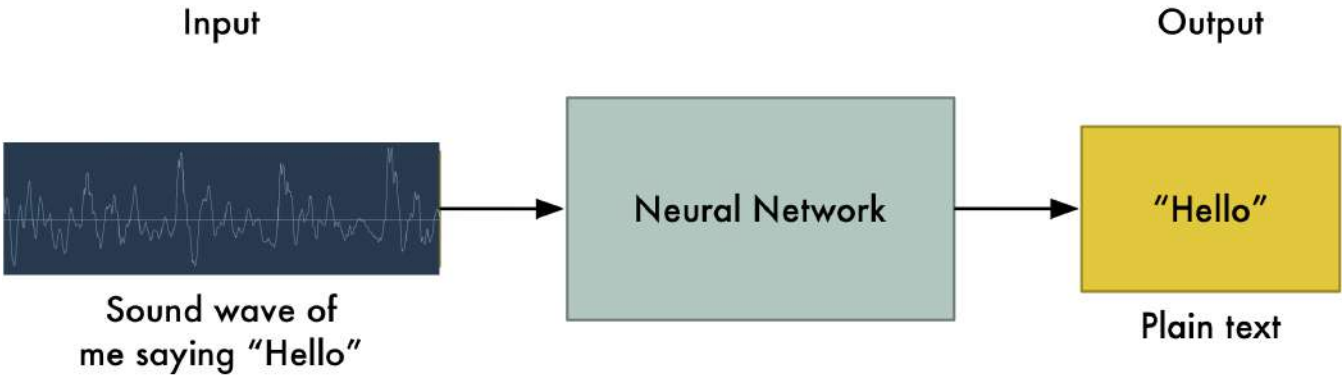
But speech recognition has been around for decades, so why is it just now hitting the mainstream? The reason is that deep learning finally made speech recognition accurate enough to be useful outside of carefully controlled environments.

Andrew Ng has long predicted that as speech recognition goes from 95% accurate to 99% accurate, it will become a primary way that we interact with computers. The idea is that this 4% accuracy gap is the difference between *annoyingly unreliable* and *incredibly useful*. Thanks to Deep Learning, we're finally cresting that peak.

Let's learn how to do speech recognition with deep learning!

Machine Learning isn't always a Black Box

If you know how neural machine translation works, you might guess that we could simply feed sound recordings into a neural network and train it to produce text:



That's the holy grail of speech recognition with deep learning, but we aren't quite there yet (at least at the time that I wrote this — I bet that we will be in a couple of years).

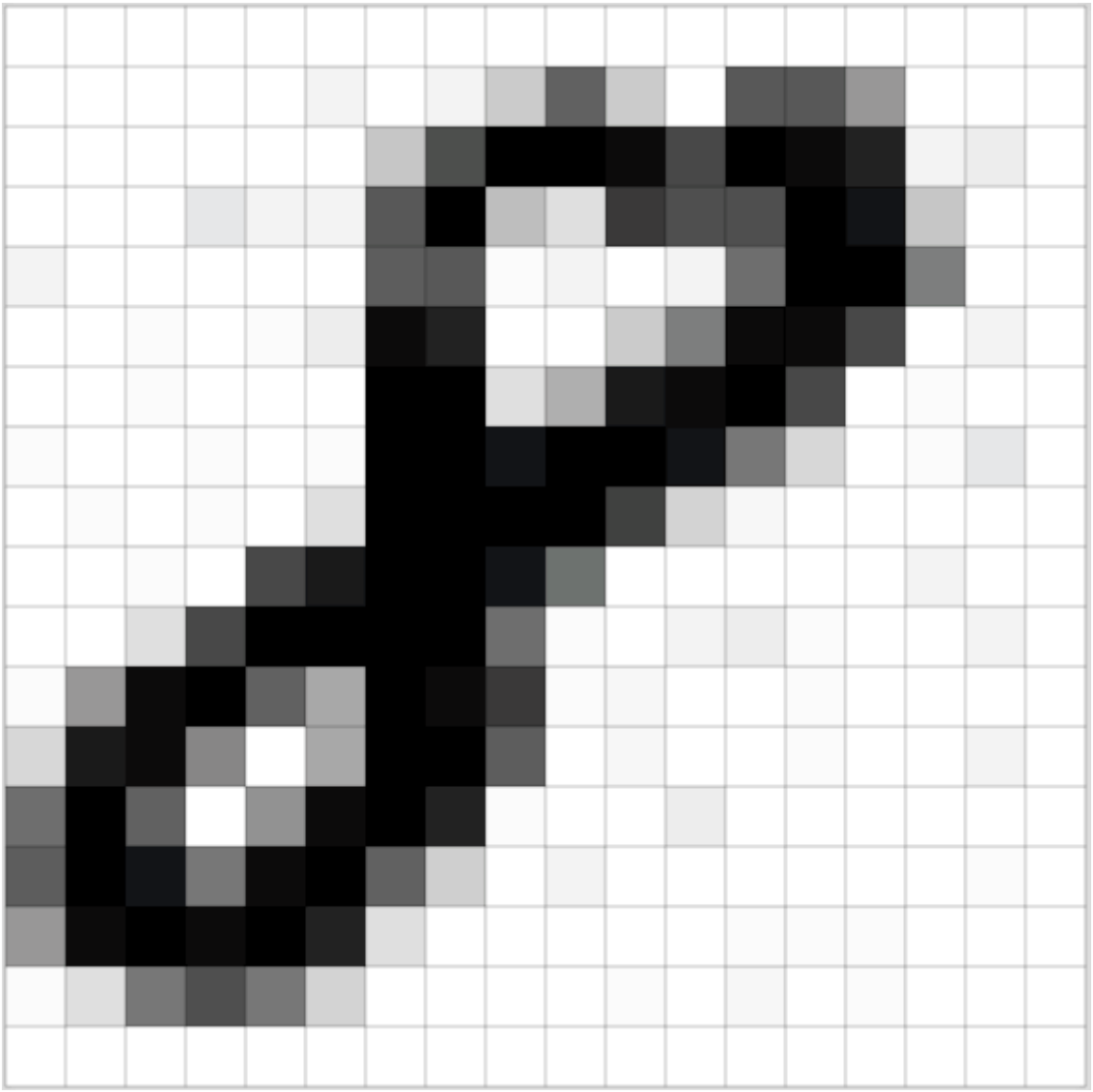
The big problem is that speech varies in speed. One person might say "hello!" very quickly and another person might say "heeeellllllllllllooooo!" very slowly, producing a much longer sound file with much more data. Both both sound files should be recognized as exactly the same text — "hello!" Automatically aligning audio files of various lengths to a fixed-length piece of text turns out to be pretty hard.

To work around this, we have to use some special tricks and extra precessing in addition to a deep neural network. Let's see how it works!

Turning Sounds into Bits

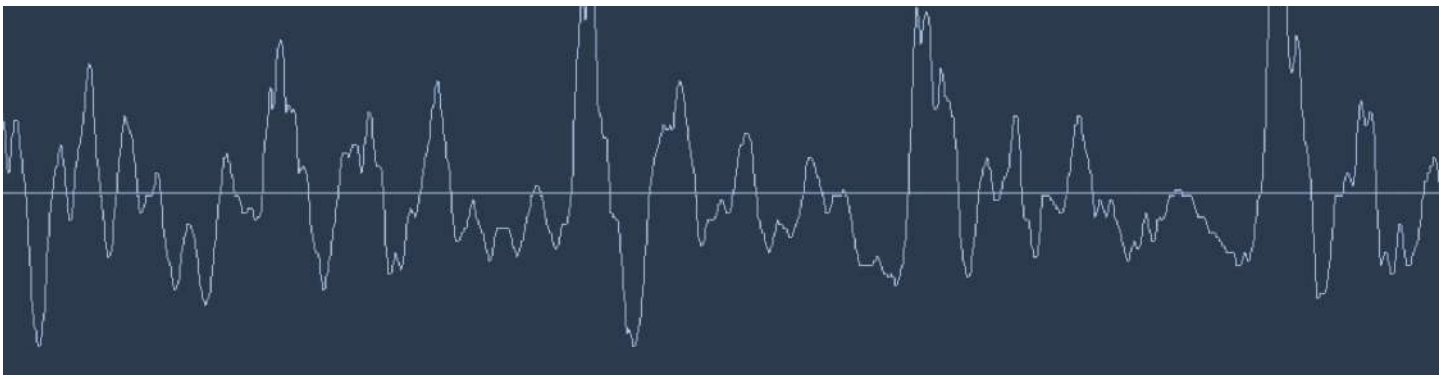
The first step in speech recognition is obvious — we need to feed sound waves into a computer.

In Part 3, we learned how to take an image and treat it as an array of numbers so that we can feed directly into a neural network for image recognition:



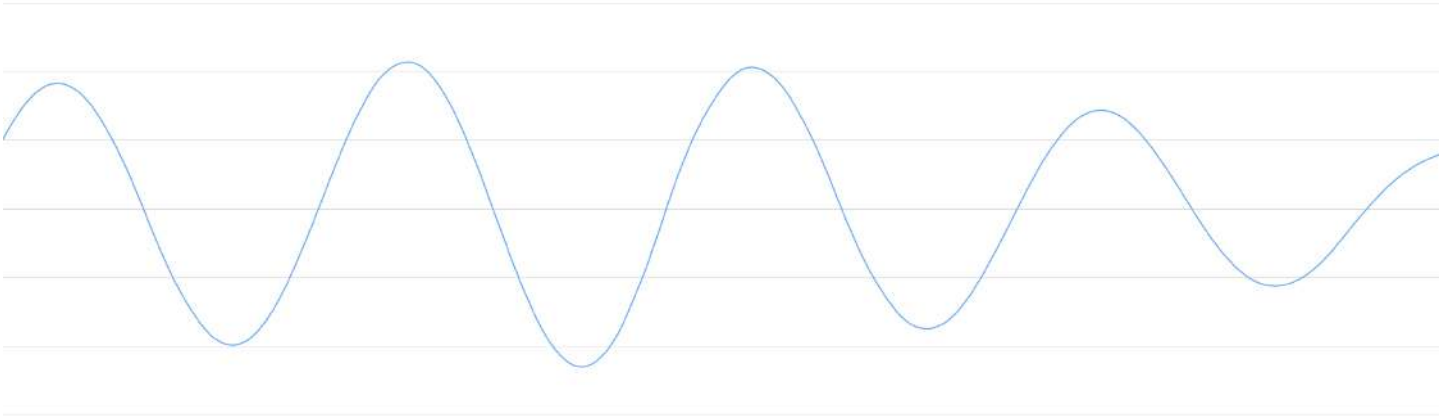
Images are just arrays of numbers that encode the intensity of each pixel

But sound is transmitted as *waves*. How do we turn sound waves into numbers? Let's use this sound clip of me saying "Hello":

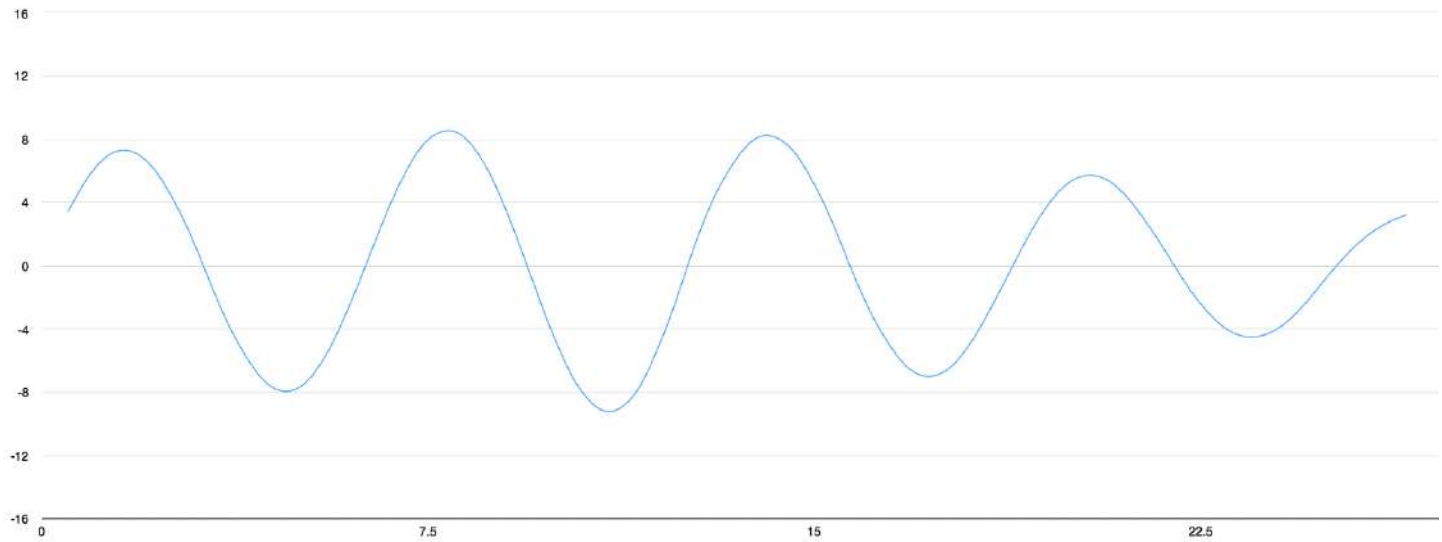


A waveform of me saying "Hello"

Sound waves are one-dimensional. At every moment in time, they have a single value based on the height of the wave. Let's zoom in on one tiny part of the sound wave and take a look:



To turn this sound wave into numbers, we just record of the height of the wave at equally-spaced points:



Sampling a sound wave

This is called *sampling*. We are taking a reading thousands of times a second and recording a number representing the height of the sound wave at that point in time. That's basically all an uncompressed .wav audio file is.

"CD Quality" audio is sampled at 44.1khz (44,100 readings per second). But for speech recognition, a sampling rate of 16khz (16,000 samples per second) is enough to cover the frequency range of human speech.

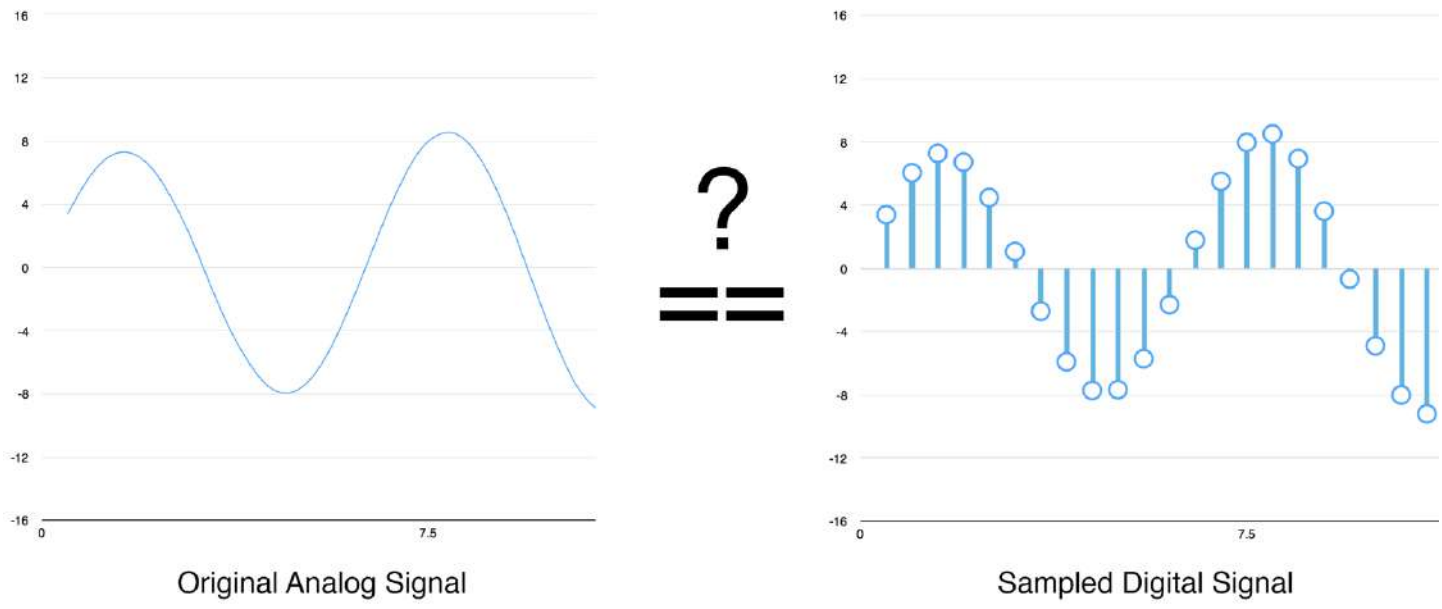
Lets sample our "Hello" sound wave 16,000 times per second. Here's the first 100 samples:

```
[-1274, -1252, -1160, -986, -792, -692, -614, -429, -286, -134, -57, -41, -169, -456, -450, -541, -761, -1067, -1231, -1047, -952, -645, -489, -448
, -397, -212, 193, 114, -17, -110, 128, 261, 198, 390, 461, 772, 948, 1451, 1974, 2624, 3793, 4968, 5939, 6057, 6581, 7302, 7640, 7223, 6119, 5461,
 4820, 4353, 3611, 2740, 2004, 1349, 1178, 1085, 901, 301, -262, -499, -488, -707, -1406, -1997, -2377, -2494, -2605, -2675, -2627, -2500, -2148, -
1648, -970, -364, 13, 260, 494, 788, 1011, 938, 717, 507, 323, 324, 325, 350, 103, -113, 64, 176, 93, -249, -461, -606, -909, -1159, -1307, -1544]
```

Each number represents the amplitude of the sound wave at 1/16000th of a second intervals

A Quick Sidebar on Digital Sampling

You might be thinking that sampling is only creating a rough approximation of the original sound wave because it's only taking occasional readings. There's gaps in between our readings so we must be losing data, right?



Can digital samples perfectly recreate the original analog sound wave? What about those gaps?

But thanks to the Nyquist theorem, we know that we can use math to perfectly reconstruct the original sound wave from the spaced-out samples — as long as we sample at least twice as fast as the highest frequency we want to record.

I mention this only because nearly everyone gets this wrong and assumes that using higher sampling rates always leads to better audio quality. It doesn't.

</end rant>

Pre-processing our Sampled Sound Data

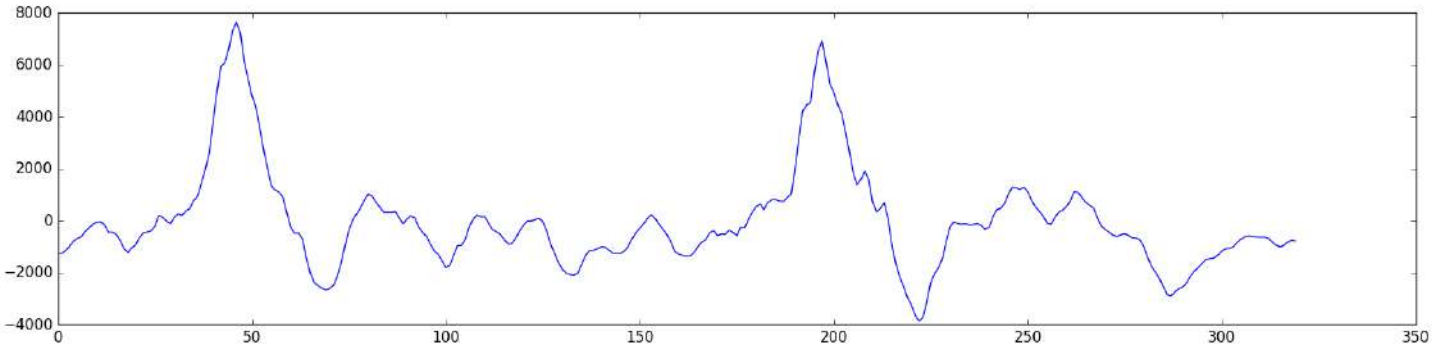
We now have an array of numbers with each number representing the sound wave's amplitude at 1/16,000th of a second intervals.

We *could* feed these numbers right into a neural network. But trying to recognize speech patterns by processing these samples directly is difficult. Instead, we can make the problem easier by doing some pre-processing on the audio data.

Let's start by grouping our sampled audio into 20-millisecond-long chunks. Here's our first 20 milliseconds of audio (i.e., our first 320 samples):

```
[-1274, -1252, -1160, -986, -792, -692, -614, -429, -286, -134, -57, -41, -169, -456, -450, -541, -761, -1067, -1231, -1047, -952, -645, -489, -448
, -397, -212, 193, 114, -17, -110, 128, 261, 198, 390, 461, 772, 948, 1451, 1974, 2624, 3793, 4968, 5939, 6057, 6581, 7302, 7640, 7223, 6119, 5461,
 4820, 4353, 3611, 2740, 2004, 1349, 1178, 1085, 901, 301, -262, -499, -488, -707, -1406, -1997, -2377, -2494, -2605, -2675, -2627, -2500, -2148, -
1648, -970, -364, 13, 260, 494, 788, 1011, 938, 717, 507, 323, 324, 325, 350, 103, -113, 64, 176, 93, -249, -461, -606, -909, -1159, -1307, -1544,
-1815, -1725, -1341, -971, -959, -723, -261, 51, 210, 142, 152, -92, -345, -439, -529, -710, -907, -887, -693, -403, -180, -14, -12, 29, 89, -47, -
398, -896, -1262, -1610, -1862, -2021, -2077, -2105, -2023, -1697, -1360, -1150, -1148, -1091, -1013, -1018, -1126, -1255, -1270, -1266, -1174, -10
03, -707, -468, -300, -116, 92, 224, 72, -150, -336, -541, -820, -1178, -1289, -1345, -1385, -1365, -1223, -1004, -839, -734, -481, -396, -580, -52
7, -531, -376, -458, -581, -254, -277, 50, 331, 531, 641, 416, 697, 810, 812, 759, 739, 888, 1008, 1977, 3145, 4219, 4454, 4521, 5691, 6563, 6909,
6117, 5244, 4951, 4462, 4124, 3435, 2671, 1847, 1370, 1591, 1900, 1586, 713, 341, 462, 673, 60, -938, -1664, -2185, -2527, -2967, -3253, -3636, -38
59, -3723, -3134, -2380, -2032, -1831, -1457, -804, -241, -51, -113, -136, -122, -158, -147, -114, -181, -338, -266, 131, 418, 471, 651, 994, 1295,
 1267, 1197, 1291, 1110, 793, 514, 370, 174, -90, -139, 104, 334, 407, 524, 771, 1106, 1087, 878, 703, 591, 471, 91, -199, -357, -454, -561, -605,
-552, -512, -575, -669, -672, -763, -1022, -1435, -1791, -1999, -2242, -2563, -2853, -2893, -2740, -2625, -2556, -2385, -2138, -1936, -1803, -1649,
 -1495, -1460, -1446, -1345, -1177, -1088, -1072, -1003, -856, -719, -621, -585, -613, -634, -638, -636, -683, -819, -946, -1012, -964, -836, -762,
 -788]
```

Plotting those numbers as a simple line graph gives us a rough approximation of the original sound wave for that 20 millisecond period of time:



This recording is only *1/50th of a second long*. But even this short recording is a complex mish-mash of different frequencies of sound. There's some low sounds, some mid-range sounds, and even some high-pitched sounds sprinkled in. But taken all

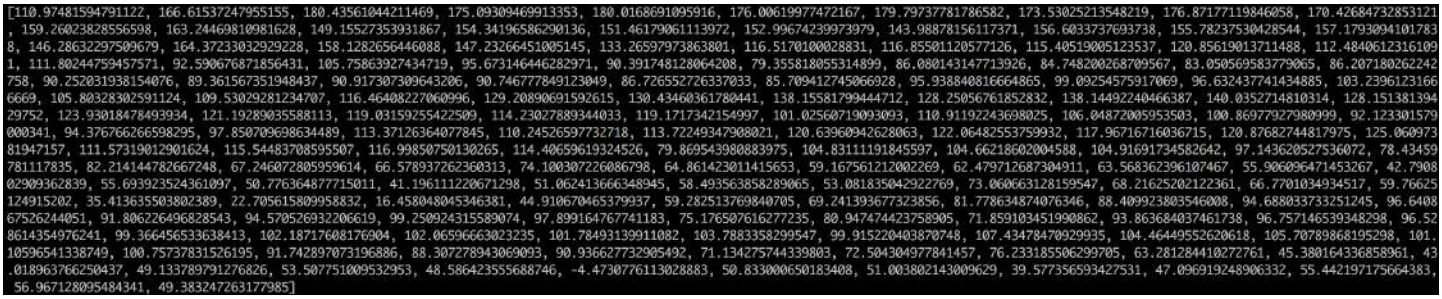
together, these different frequencies mix together to make up the complex sound of human speech.

To make this data easier for a neural network to process, we are going to break apart this complex sound wave into it's component parts. We'll break out the low-pitched parts, the next-lowest-pitched-parts, and so on. Then by adding up how much energy is in each of those frequency bands (from low to high), we create a *fingerprint* of sorts for this audio snippet.

Imagine you had a recording of someone playing a C Major chord on a piano. That sound is the combination of three musical notes— C, E and G — all mixed together into one complex sound. We want to break apart that complex sound into the individual notes to discover that they were C, E and G. This is the exact same idea.

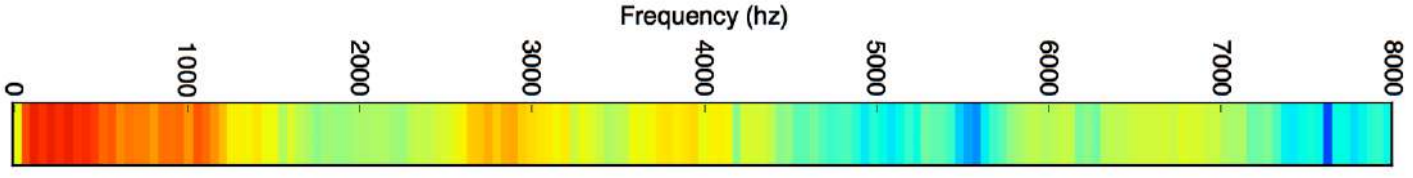
We do this using a mathematic operation called a *Fourier transform*. It breaks apart the complex sound wave into the simple sound waves that make it up. Once we have those individual sound waves, we add up how much energy is contained in each one.

The end result is a score of how important each frequency range is, from low pitch (i.e. bass notes) to high pitch. Each number below represents how much energy was in each 50hz band of our 20 millisecond audio clip:



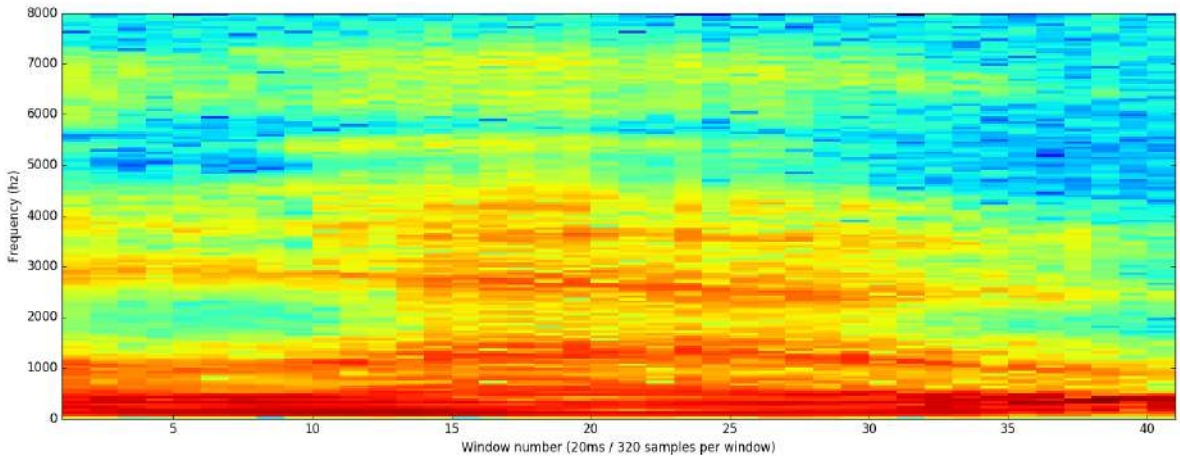
Each number in the list represents how much energy was in that 50hz frequency band

But this is a lot easier to see when you draw this as a chart:



You can see that our 20 millisecond sound snippet has a lot of low-frequency energy and not much energy in the higher frequencies. That's typical of "male" voices.

If we repeat this process on every 20 millisecond chunk of audio, we end up with a spectrogram (each column from left-to-right is one 20ms chunk):

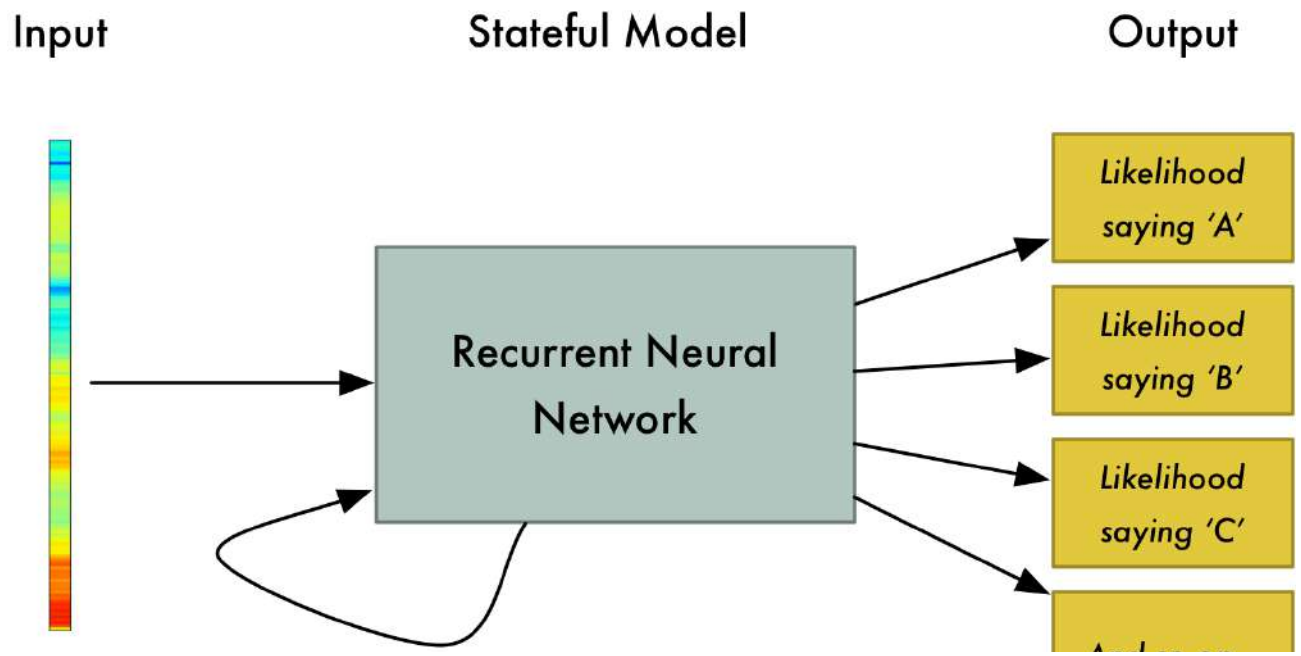


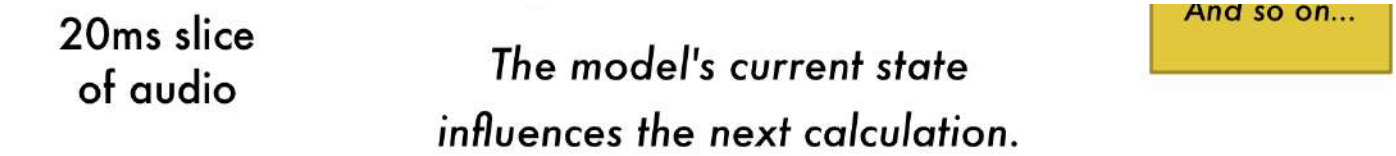
The full spectrogram of the "hello" sound clip

A spectrogram is cool because you can actually *see* musical notes and other pitch patterns in audio data. A neural network can find patterns in this kind of data more easily than raw sound waves. So this is the data representation we'll actually feed into our neural network.

Recognizing Characters from Short Sounds

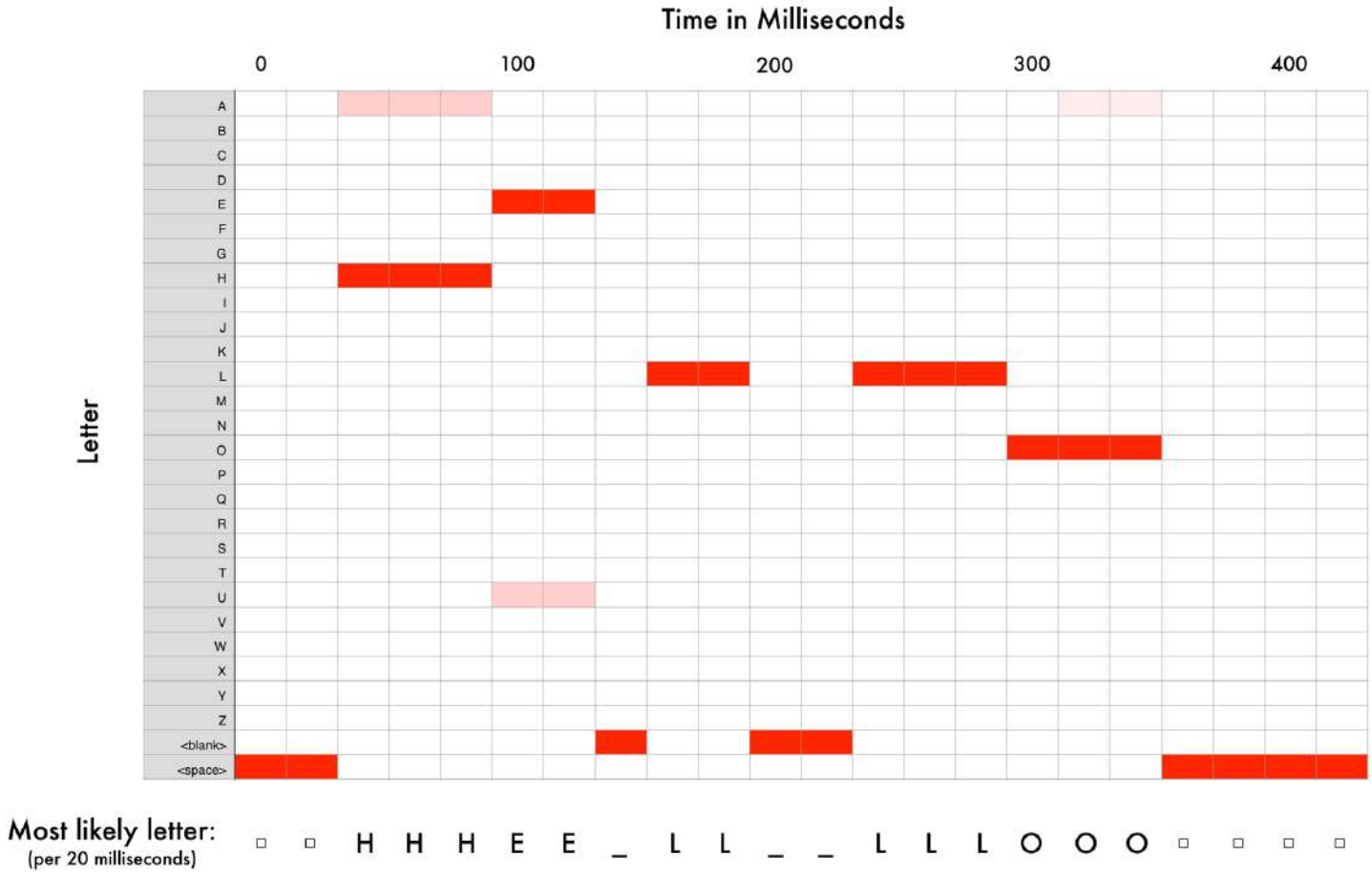
Now that we have our audio in a format that's easy to process, we will feed it into a deep neural network. The input to the neural network will be 20 millisecond audio chunks. For each little audio slice, it will try to figure out the *letter* that corresponds the sound currently being spoken.





We'll use a recurrent neural network — that is, a neural network that has a memory that influences future predictions. That's because each letter it predicts should affect the likelihood of the next letter it will predict too. For example, if we have said "HEL" so far, it's very likely we will say "LO" next to finish out the word "Hello". It's much less likely that we will say something unpronounceable next like "XYZ". So having that memory of previous predictions helps the neural network make more accurate predictions going forward.

After we run our entire audio clip through the neural network (one chunk at a time), we'll end up with a mapping of each audio chunk to the letters most likely spoken during that chunk. Here's what that mapping looks like for me saying "Hello":



Our neural net is predicting that one likely thing I said was "HHHEE_LL_LLLOOO". But it also thinks that it was possible that I said "HHHUU_LL_LLLOOO" or even "AAAUU_LL_LLLOOO".

We have some steps we follow to clean up this output. First, we'll replace any repeated characters a single character:

- HHHEE_LL_LLLOOO becomes HE_L_LO
- HHHUU_LL_LLLOOO becomes HU_L_LO
- AAAUU_LL_LLLOOO becomes AU_L_LO

Then we'll remove any blanks:

- HE_L_LO becomes HELLO
- HU_L_LO becomes HULLO
- AU_L_LO becomes AULLO

That leaves us with three possible transcriptions — "Hello", "Hullo" and "Aullo". If you say them out loud, all of these sound similar to "Hello". Because it's predicting one character at a time, the neural network will come up with these very *sounded-out* transcriptions. For example if you say "He would not go", it might give one possible transcription as "He wud net go".

The trick is to combine these pronunciation-based predictions with likelihood scores based on large database of written text (books, news articles, etc). You throw out transcriptions that seem the least likely to be real and keep the transcription that seems the most realistic.

Of our possible transcriptions "Hello", "Hullo" and "Aullo", obviously "Hello" will appear more frequently in a database of text (not to mention in our original audio-based training data) and thus is probably correct. So we'll pick "Hello" as our final transcription instead of the others. Done!

Wait a second!

You might be thinking *"But what if someone says 'Hullo'? It's a valid word. Maybe 'Hello' is the wrong transcription!"*

"Hullo! Who dis?"

Of course it is possible that someone actually said "Hullo" instead of "Hello". But a speech recognition system like this (trained on American English) will basically never produce "Hullo" as the transcription. It's just such an unlikely thing for a user to say compared to "Hello" that it will always think you are saying "Hello" no matter how much you emphasize the 'U' sound.

Try it out! If your phone is set to American English, try to get your phone's digital assistant to recognize the world "Hullo." You can't! It refuses! It will always understand it as "Hello."

Not recognizing "Hullo" is a reasonable behavior, but sometimes you'll find annoying cases where your phone just refuses to understand something valid you are saying. That's why these speech recognition models are always being retrained with more data to fix these edge cases.

Can I Build My Own Speech Recognition System?

One of the coolest things about machine learning is how simple it sometimes seems. You get a bunch of data, feed it into a machine learning algorithm, and then magically you have a world-class AI system running on your gaming laptop's video card… *Right*?

That sort of true in some cases, but not for speech. Recognizing speech is a hard problem. You have to overcome almost limitless challenges: bad quality microphones, background noise, reverb and echo, accent variations, and on and on. All of these

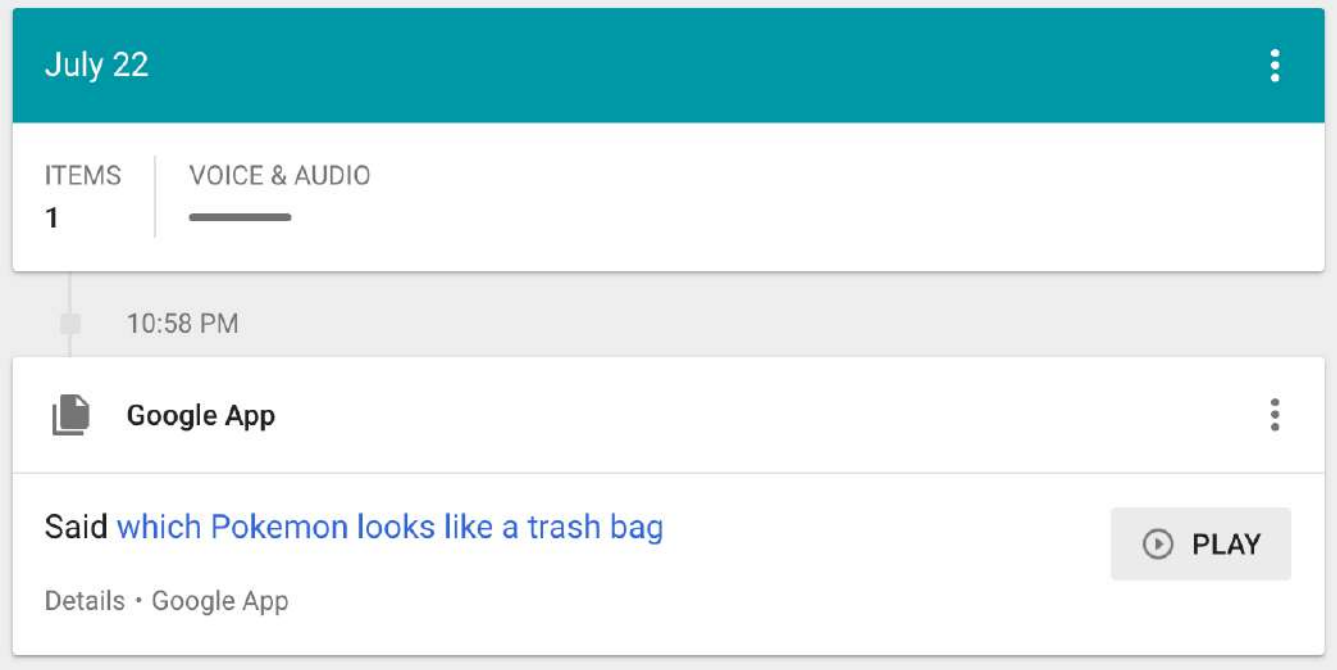
issues need to be present in your training data to make sure the neural network can deal with them.

Here's another example: Did you know that when you speak in a loud room you unconsciously raise the pitch of your voice to be able to talk over the noise? Humans have no problem understanding you either way, but neural networks need to be trained to handle this special case. So you need training data with people yelling over noise!

To build a voice recognition system that performs on the level of Siri, Google Now!, or Alexa, you will need a *lot* of training data — far more data than you can likely get without hiring hundreds of people to record it for you. And since users have low tolerance for poor quality voice recognition systems, you can't skimp on this. No one wants a voice recognition system that works 80% of the time.

For a company like Google or Amazon, hundreds of thousands of hours of spoken audio recorded in real-life situations is *gold*. That's the single biggest thing that separates their world-class speech recognition system from your hobby system. The whole point of putting *Google Now!* and *Siri* on every cell phone for free or selling $50 *Alexa* units that have no subscription fee is to get you to **use them as much as possible**. Every single thing you say into one of these systems is ***recorded forever*** and used as training data for future versions of speech recognition algorithms. That's the whole game!

Don't believe me? If you have an Android phone with *Google Now!*, click here to listen to actual recordings of yourself saying every dumb thing you've ever said into it:



You can access the same thing for Amazon via your Alexa app. Apple unfortunately doesn't let you access your Siri voice data.

So if you are looking for a start-up idea, I wouldn't recommend trying to build your own speech recognition system to compete with Google. Instead, figure out a way to get people to give you recordings of themselves talking for hours. The data can be your product instead.

Where to Learn More

- The algorithm (roughly) described here to deal with variable-length audio is called Connectionist Temporal Classification or CTC. You can read the original paper from 2006.
- Adam Coates of Baidu gave a great presentation on Deep Learning for Speech Recognition at the Bay Area Deep Learning School. You can watch the video on YouTube (his talk starts at 3:51:00). Highly recommended.

. . .

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 7!

Machine Learning Artificial Intelligence Speech Recognition

Machine Learning is Fun Part 7: Abusing Generative Adversarial Networks to Make 8-bit Pixel Art

Adam Geitgey
Feb 12, 2017 · 12 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 Русский, Tiê´ng Việt or 한국어.*

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!*

Generative models allow a computer to create data — like photos, movies or music — by itself.

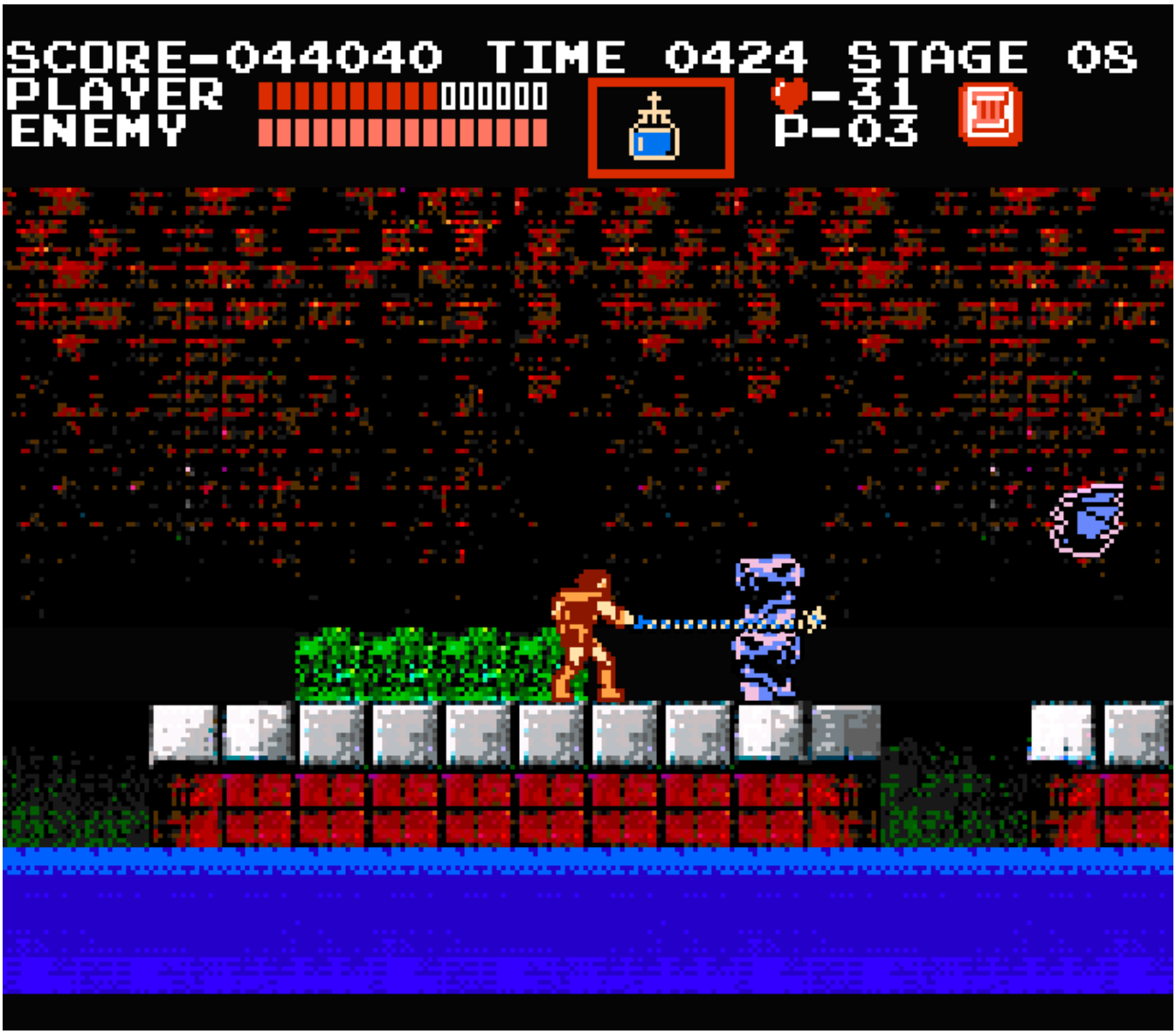
A little over a year ago, Alec Radford (building on the work of Ian Goodfellow) published a paper that changed how everyone thought about building generative models with machine learning. The new system is called Deep Convolutional Generative Adversarial Networks (or DCGANs for short).

DCGANs are able to hallucinate original photo-realistic pictures by using a clever combination of two deep neural networks that compete with each other. All of these pictures of bedrooms were dreamt up by a DCGAN:

Picture from Alec Radford's original DCGAN paper

AI researchers care about generative models because they seem to be a stepping stone towards building AI systems that can consume raw data from the world and automatically build understanding from it.

But let's use generative models to do something a bit more silly — make artwork for 8-bit video games!



All the art in this game level is machine-generated.

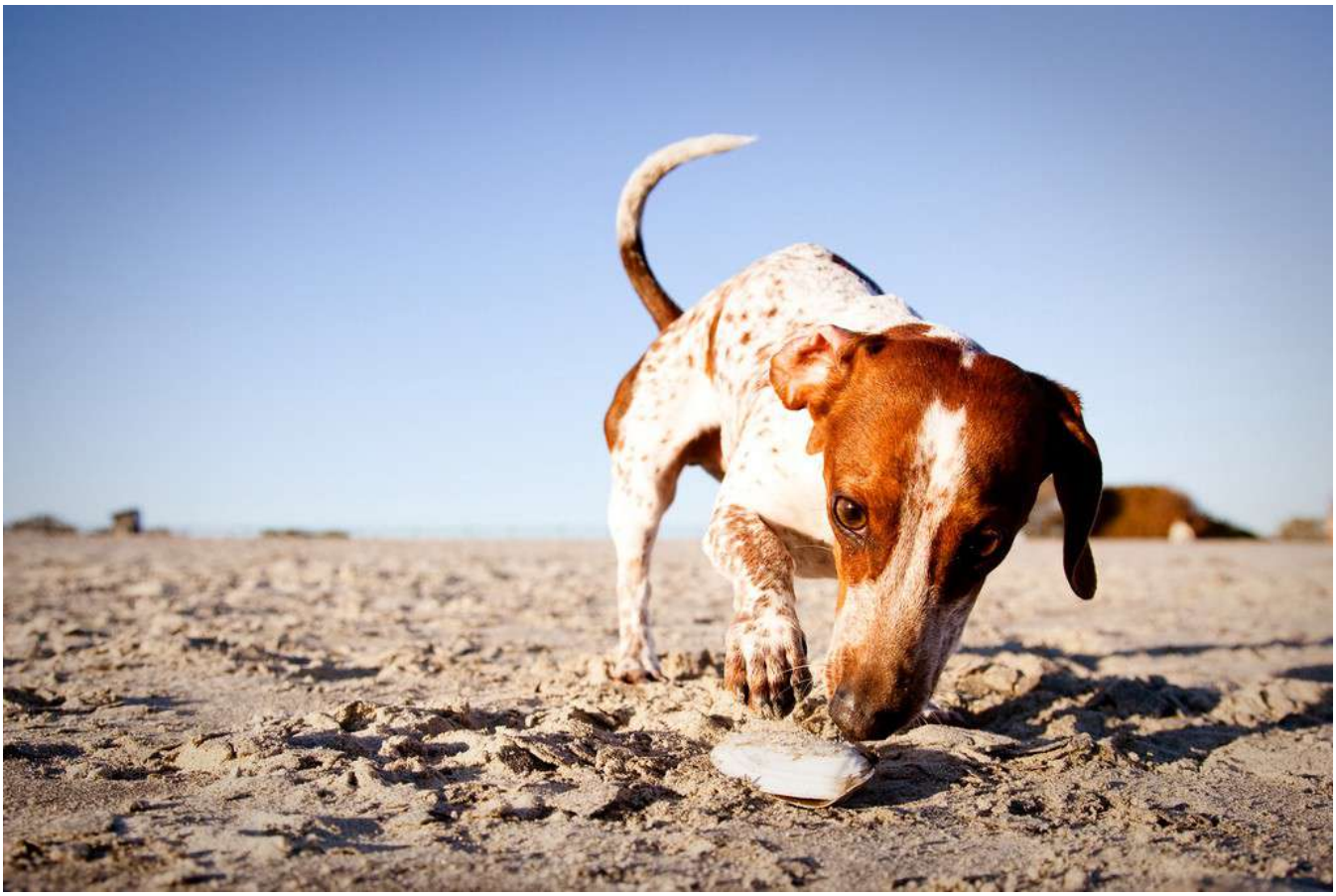
. . .

The goal of Generative Models

So why exactly are AI researchers building complex systems to generate slightly wonky-looking pictures of bedrooms?

The idea is that if you can generate pictures of something, you must have an understanding of it.

Look at this picture:



A dog. More specifically, my dog.

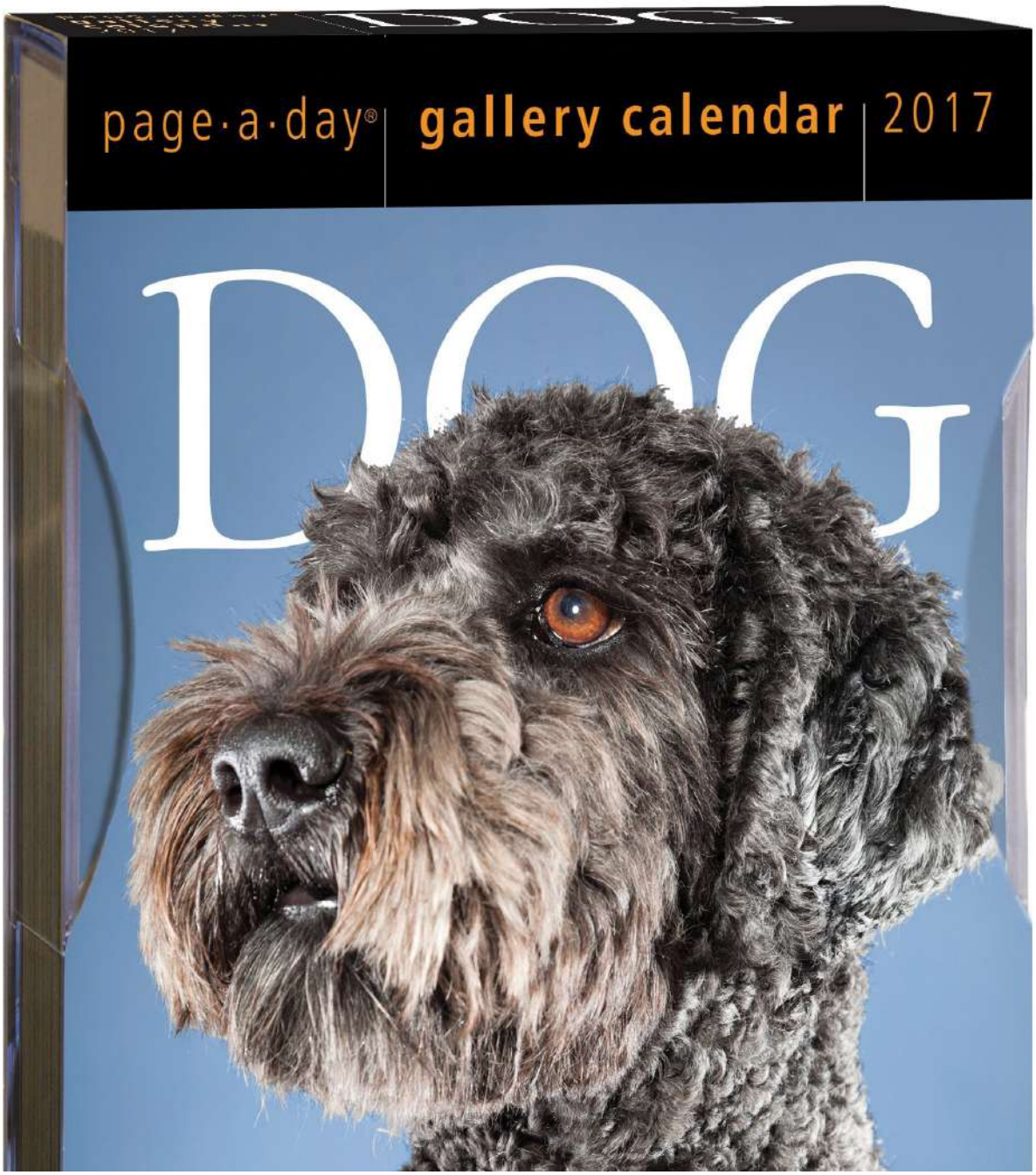
You instantly know this is a picture of a dog — a furry thing with four legs and a tail. But to a computer, the picture is just a grid of numbers representing the color of each pixel. The computer has no understanding that the picture represents a concept.

But now imagine that we showed a computer thousands of pictures of dogs and after seeing those pictures, the computer was able to generate new pictures of dogs on its own — including different dog breeds and pictures from different angles. Maybe we could even ask it for certain types of pictures, like “a side view of a beagle”.

If the computer was able to do this and the pictures it produced had the right number of legs, tails, and ears, it would prove that the computer knows what parts go into making up a “dog” even though no one told it explicitly. So in a sense, a good generative model is proof of basic understanding — at least on a toddler-level.

That’s why researchers are so excited about building generative models. They seem to be a way to train computers to understand concepts without being explicitly taught the meaning of those concepts. That’s a big step over current systems that can only learn from training data that has been painstakingly pre-labeled by humans.

But if all this research results in programs that generate pictures of dogs, how many years until we get the first computer-generated Dog-A-Day calendar as a side effect?



Yes, the robots are coming for everyone's jobs. Eventually.

And if you can build a program that understands dogs, why not a program that understands anything else? What about a program that could generate an unlimited number of stock photos of people shaking hands? I'm sure someone would pay for that.

I mean.. sure, that's a terrible idea for an AI start-up. But I've definitely heard worse start-up ideas, so.... maybe?

Ok, maybe a program that generates bad stock photos wouldn't be that interesting. But given the rate of progress in generative models over just the past year, who knows where we'll be in 5 or 10 years. What happens if someone invents a system to generate entire movies? Or music? Or video games?

If you look forward 20–30 years and squint, you can already imagine a world where entertainment could be 100% machine generated:

The video game industry is the first area of entertainment to start seriously experimenting with using AI to generate raw content. Aside from the obvious Venn diagram overlap between computer gaming and machine learning engineers, there's a huge cost incentive to invest in video game development automation given the $300+ million budgets of modern AAA video games.

We are still in the earliest days of machine-learning-based generative models and their practical uses are currently pretty narrow, but they are a lot of fun to play around with. Let's see what we can do with one.

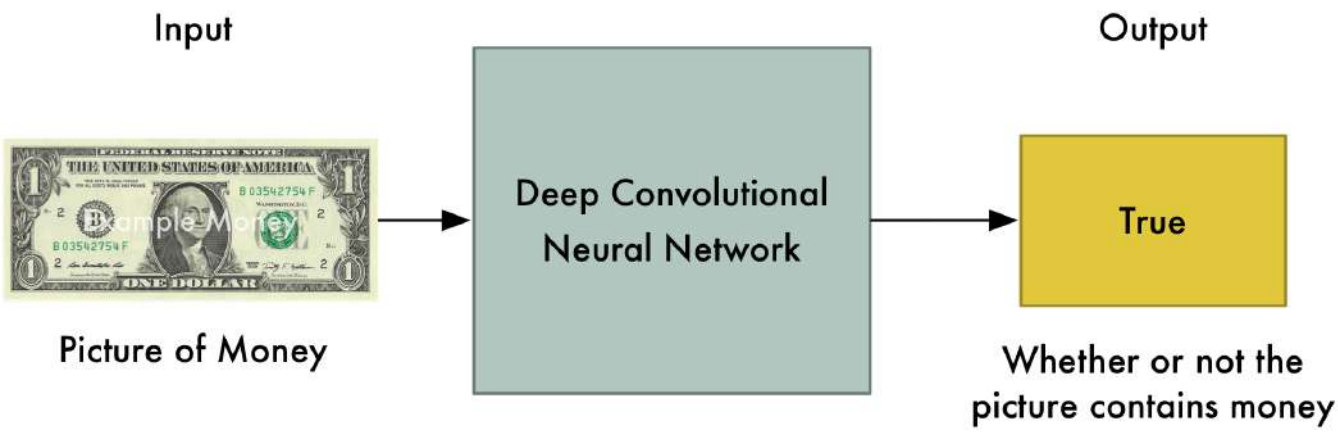
How DCGANs work

To build a DCGAN, we create two deep neural networks. Then we make them fight against each other, endlessly attempting to out-do one another. In the process, they both become stronger.

Let's pretend that the first deep neural network is a brand new police officer who is being trained to spot counterfeit money. It's job is to look at a picture and tell us if the picture contains real money.

Since we are looking for objects in pictures, we can use a standard Convolutional Neural Network for this job. If you aren't familiar with ConvNets, you can read my earlier post. But the basic idea is that the neural network that takes in an image, processes it through several layers that recognize increasingly complex features in the image and then it outputs a single value—in this case, whether or not the image contains a picture of real money.

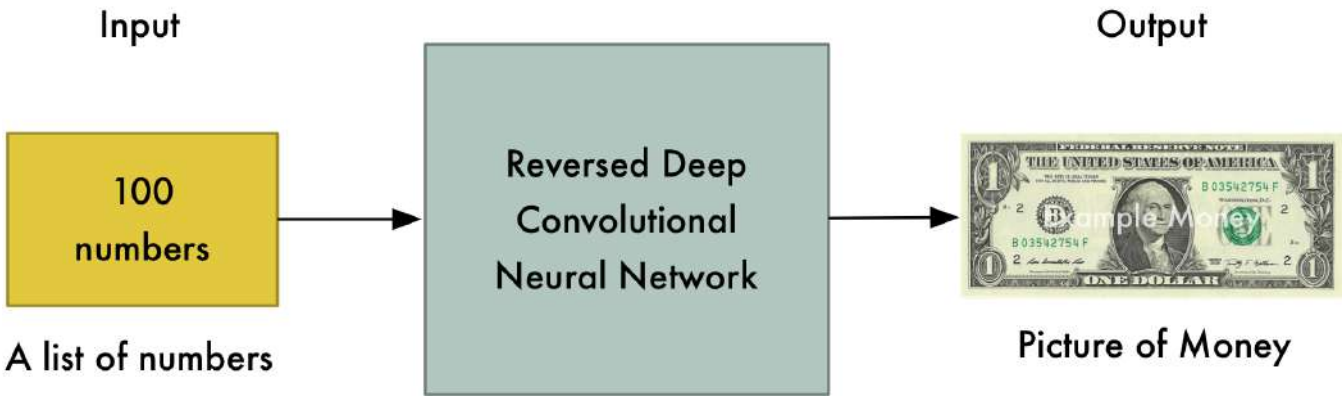
This first neural network is called the **Discriminator**:



The Discriminator Network

Now let's pretend the second neural network is a brand new counterfeiter who is just learning how to create fake money. For this second neural network, we'll reverse the layers in a normal ConvNet so that everything runs backwards. So instead of taking in a picture and outputting a value, it takes in a list of values and outputs a picture.

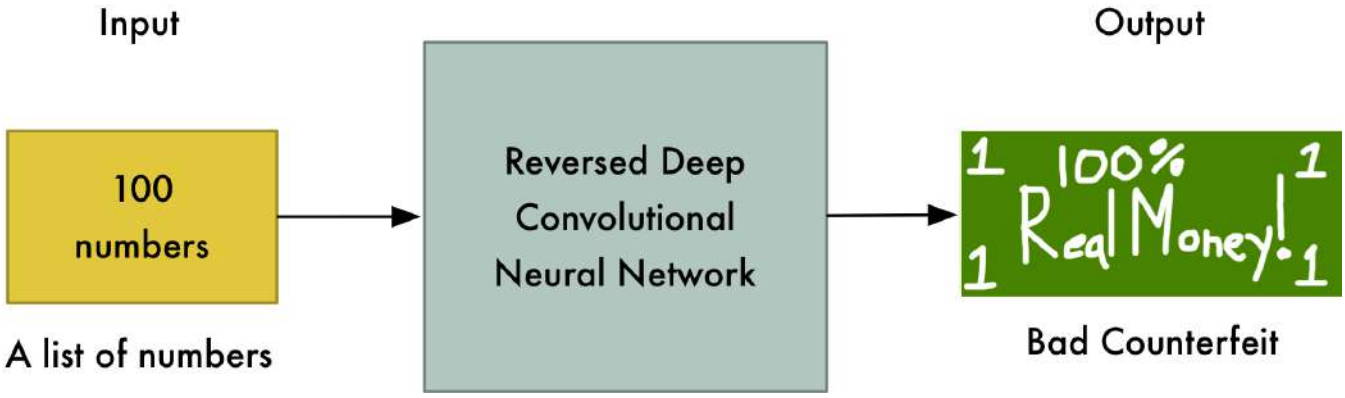
This second neural network is called the **Generator**:



The Generator Network

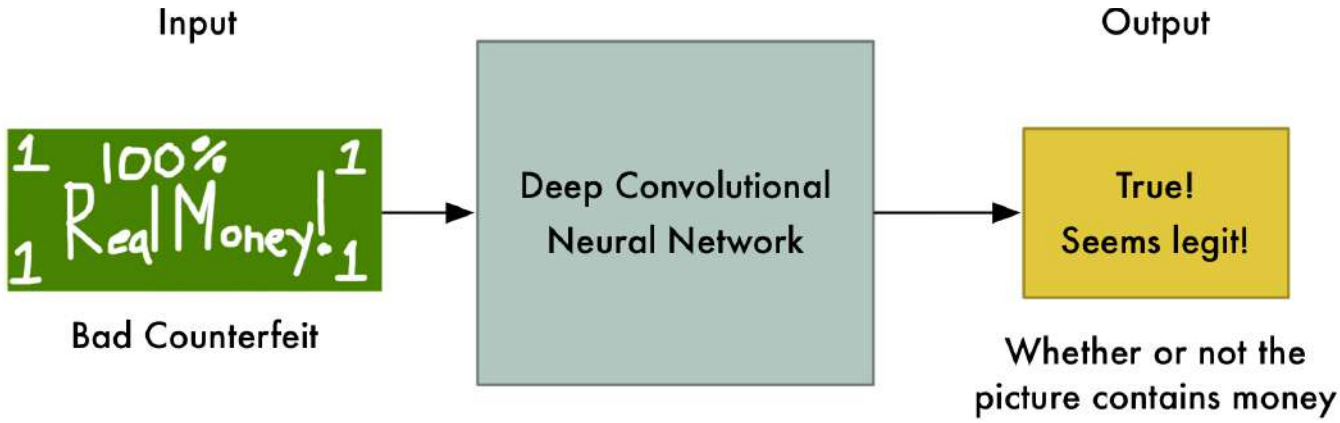
So now we have a police officer (the **Discriminator**) looking for fake money and a counterfeiter (the **Generator**) that's printing fake money. Let's make them battle!

In the first round, the **Generator** will create pathetic forgeries that barely resemble money at all because it knows absolutely nothing about what money is supposed to look like:



The Generator makes the first (terrible) fake dollar

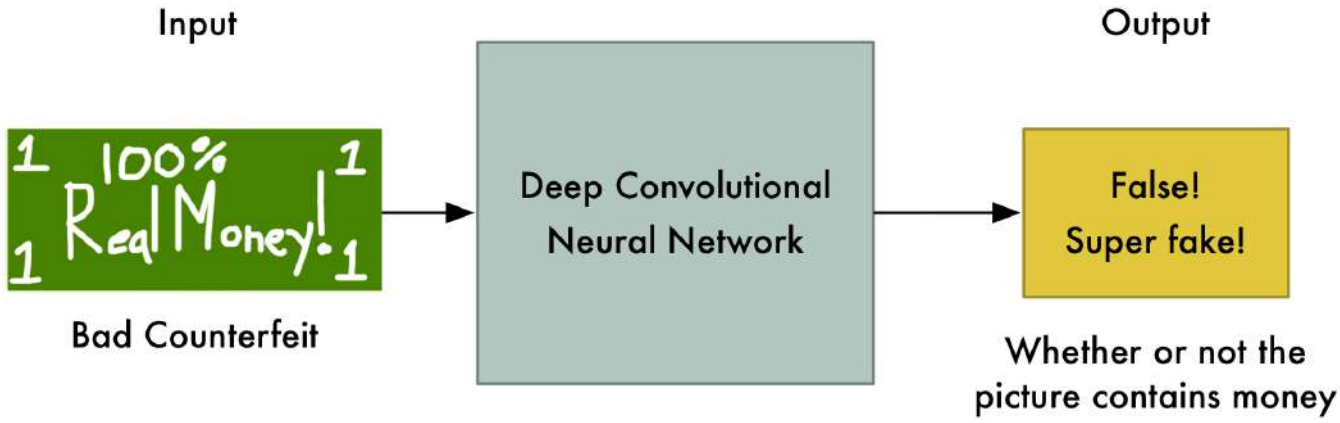
But right now the **Discriminator** is equally terrible at it's job of recognizing money, so it won't know the difference:



The Discriminator thinks the dollar is real!

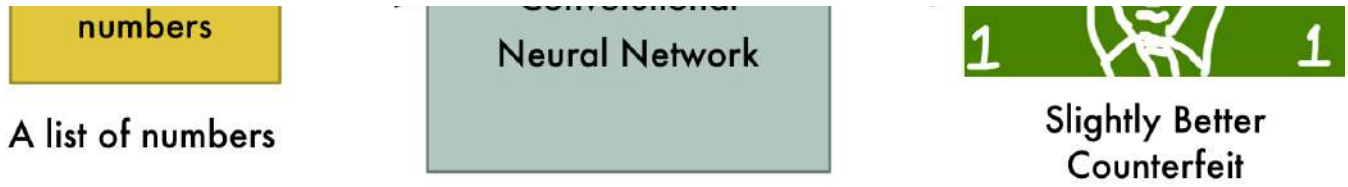
At this point, we step in and tell the Discriminator that this dollar bill is actually fake. Then we show it a real dollar bill and ask it how it looks different from the fake one. The Discriminator looks for a new detail to help it separate the real one from the fake one.

For example, the Discriminator might notice that real money has a picture of a person on it and the fake money doesn't. Using this knowledge, the Discriminator learns how to tell the fake from the real one. It gets a tiny bit better at its job:



The Discriminator levels up! It now can spot very bad fake dollars.

Now we start Round 2. We tell the Generator that it's money images are suddenly getting rejected as fake so it needs to step up it's game. We also tell it that the Discriminator is now looking for faces, so the best way to confuse the Discriminator is to put a face on the bill:



The Generator makes a very slightly better counterfeit dollar

And the fake bills are being accepted as valid again! So now the Discriminator has to look again at the real dollar and find a new way to tell it apart from the fake one.

This back-and-forth game between the Generator and the Discriminator continues thousands of times until both networks are experts. Eventually the Generator is producing near-perfect counterfeits and the Discriminator has turned into a Master Detective looking for the slightest mistakes.

At the point when both networks are sufficiently trained so that humans are impressed by the fake images, we can use the fake images for whatever purpose we want.

Applying this to Video Games

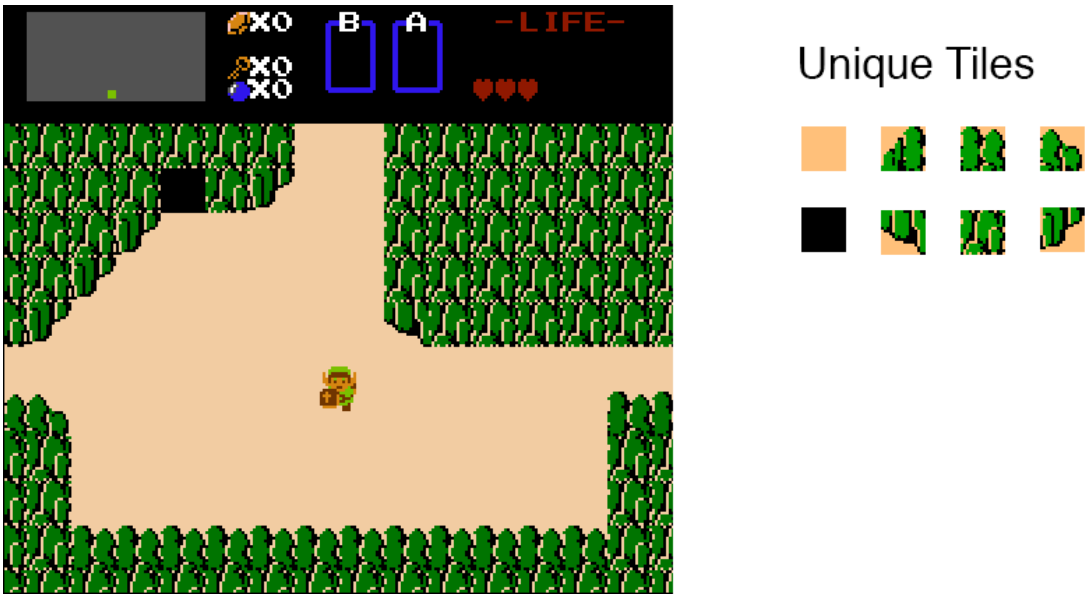
So now that we know how DCGANs work, let's see if we can use one to generate new artwork for 1980s-style video games.

Let's build a DCGAN that tries to produce screenshots of imaginary video games for the Nintendo Entertainment System (or NES) based on screenshots of real games:

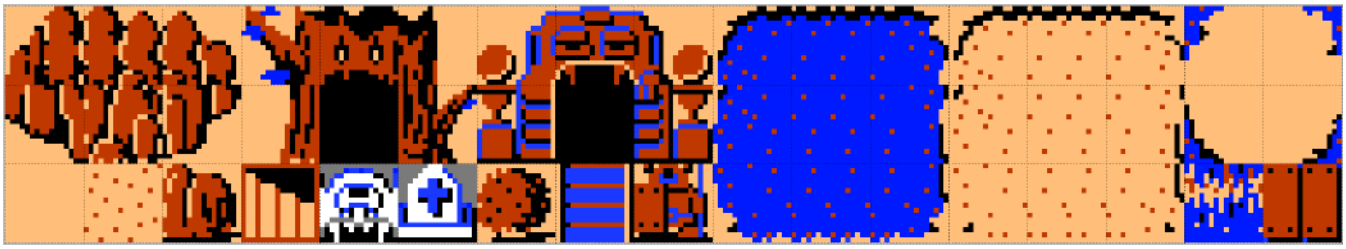
The idea is that if we can generate convincing screenshots of imaginary video games, we could copy and paste bits of art from those screenshots and use it in our own retro-style video game. Since the generated video games never existed, it wouldn't even be stealing (*Maybe... more on this later*).

Video game art in those days was very simple. Since the NES had such a small amount of memory (the games used way less memory than this article takes up!), programmers had to use lots of tricks to fit the game art into memory. To maximize the limited space, games used tile-based graphics where each screen in the game is made up of just a few (usually 16x16 pixel) repeated graphical tiles.

For example, the starting screen of 'The Legend of Zelda' is made up of only 8 unique tiles:



Here are the tiles for entire 'The Legend of Zelda' game map:



Sometimes they swap the colors around to make the different areas look different, but that's it.

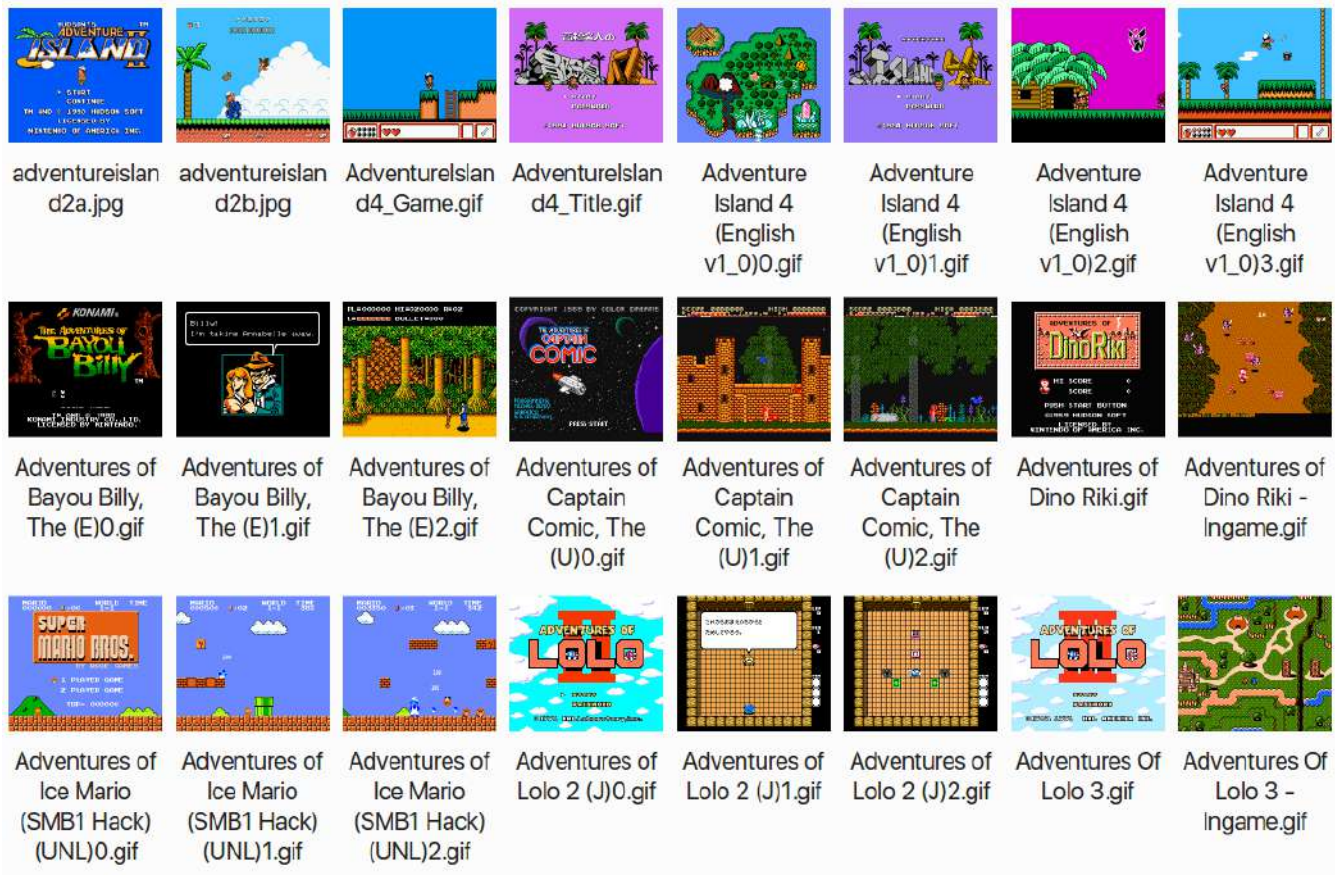
Our goal is to create a similar tile sheet for our game. Because of that, we don't really care if the game screenshots we generate look completely realistic. Instead, we're just looking for the shapes and patterns that we can use as 16 x 16 tiles in our game —

things like stones, water, bridges, etc. Then we can use those tiles to build our own 8-bit-style video game levels.

Getting Data

To train our system, we need lots of data. Luckily there are over 700 games for the NES that we can pull from.

I used wget to download all the NES game screenshots on The Video Game Museum website (sorry for scraping your site!). After a few minutes of downloading, I had a little over 10,000 screenshots of hundreds of NES games:



Just a few of the 10,000 screenshots that make up the data set

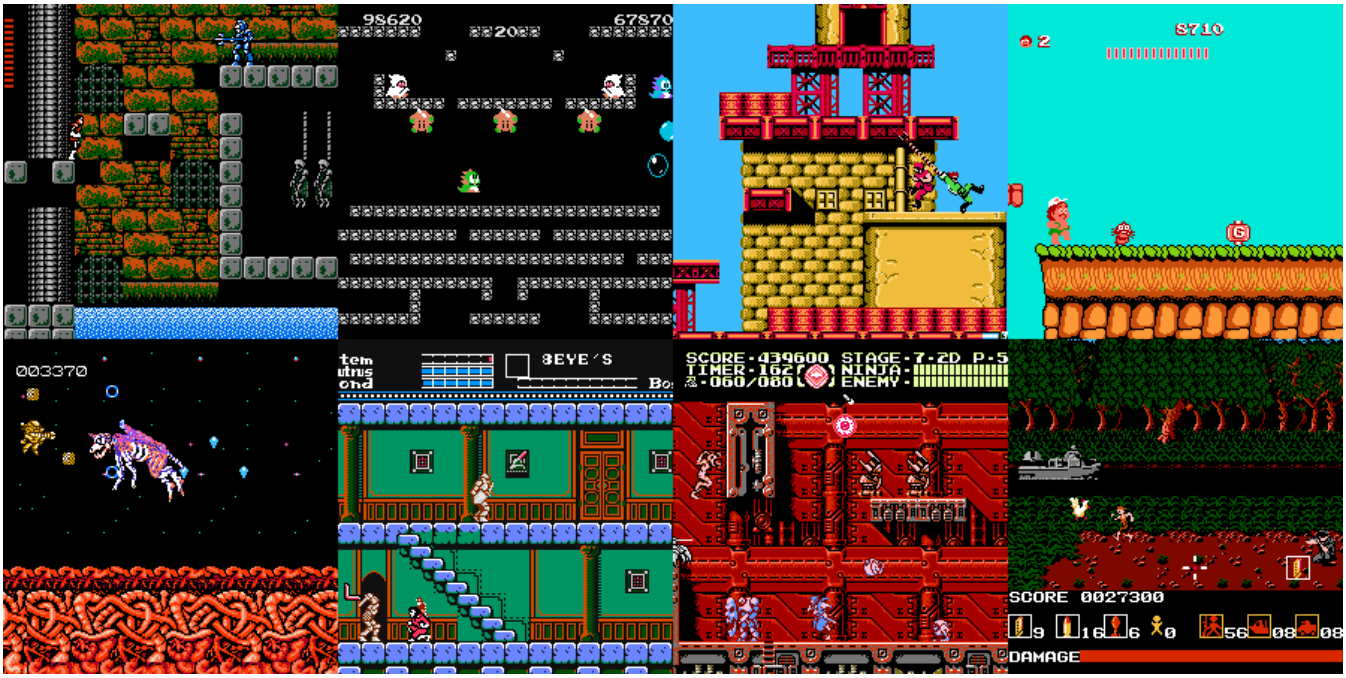
Right now, DCGANs only work on pretty small images — 256 pixels square or so. But the entire screen resolution of the NES was only 256 pixels by 224 pixels, so that's not a problem. To make things simple, I cropped each NES screenshot to 224 pixels square.

Setting up the DCGAN

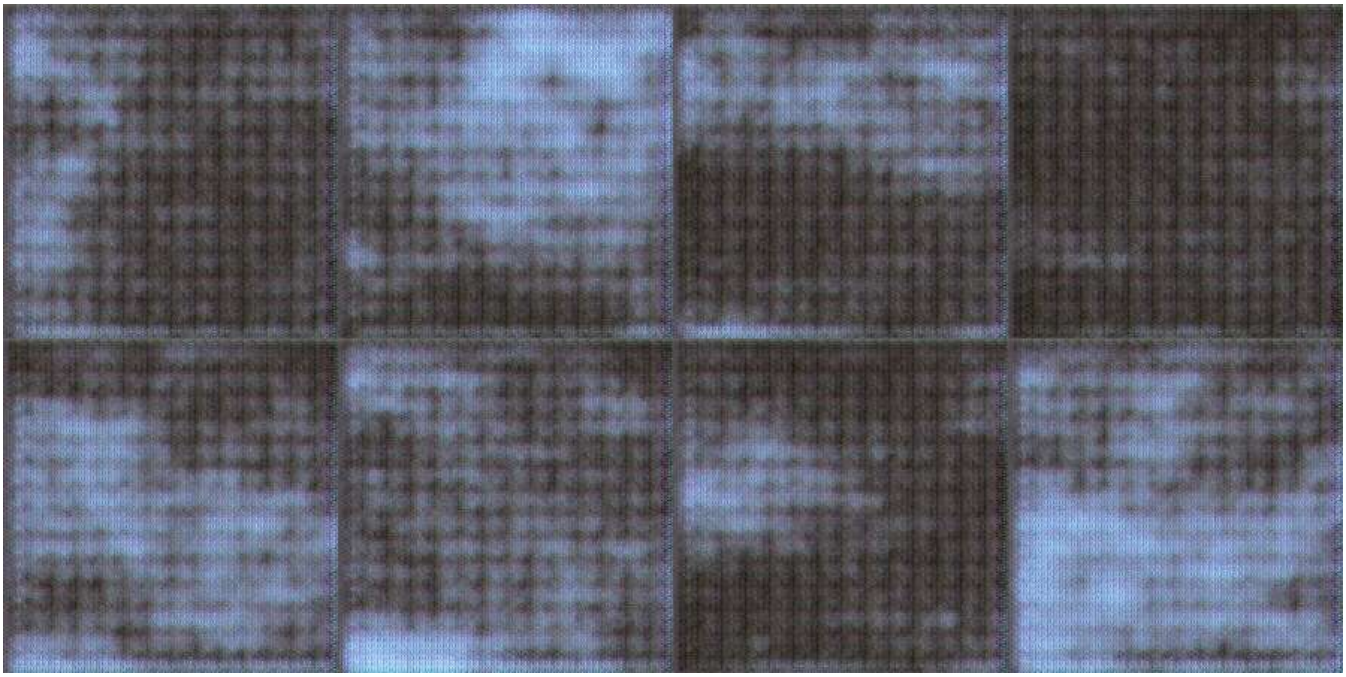
There are several open-source implementations of DCGANs on github that you can try out. I used Taehoon Kim's Tensorflow implementation. Since DCGANs are

unsupervised, all you have to do is put the data in a folder, tweak the basic parameters, start it training and then wait to see what results you get.

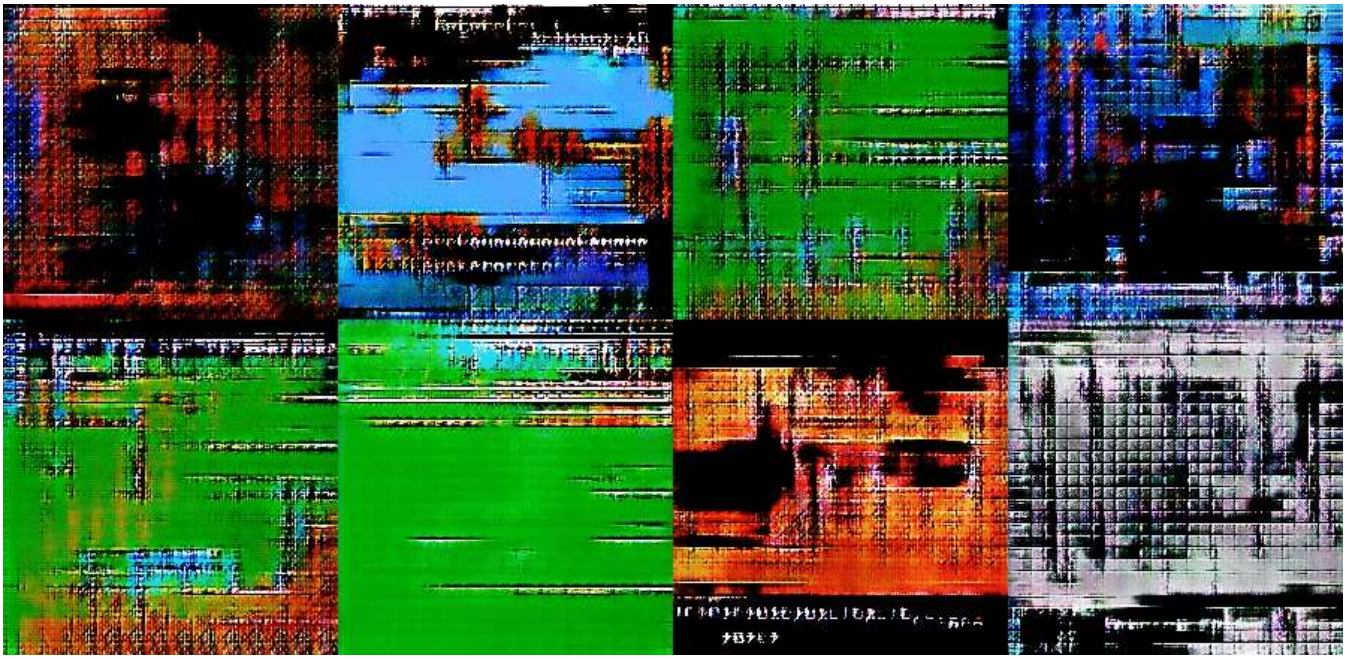
Here's what a sample of the original training data looks like:



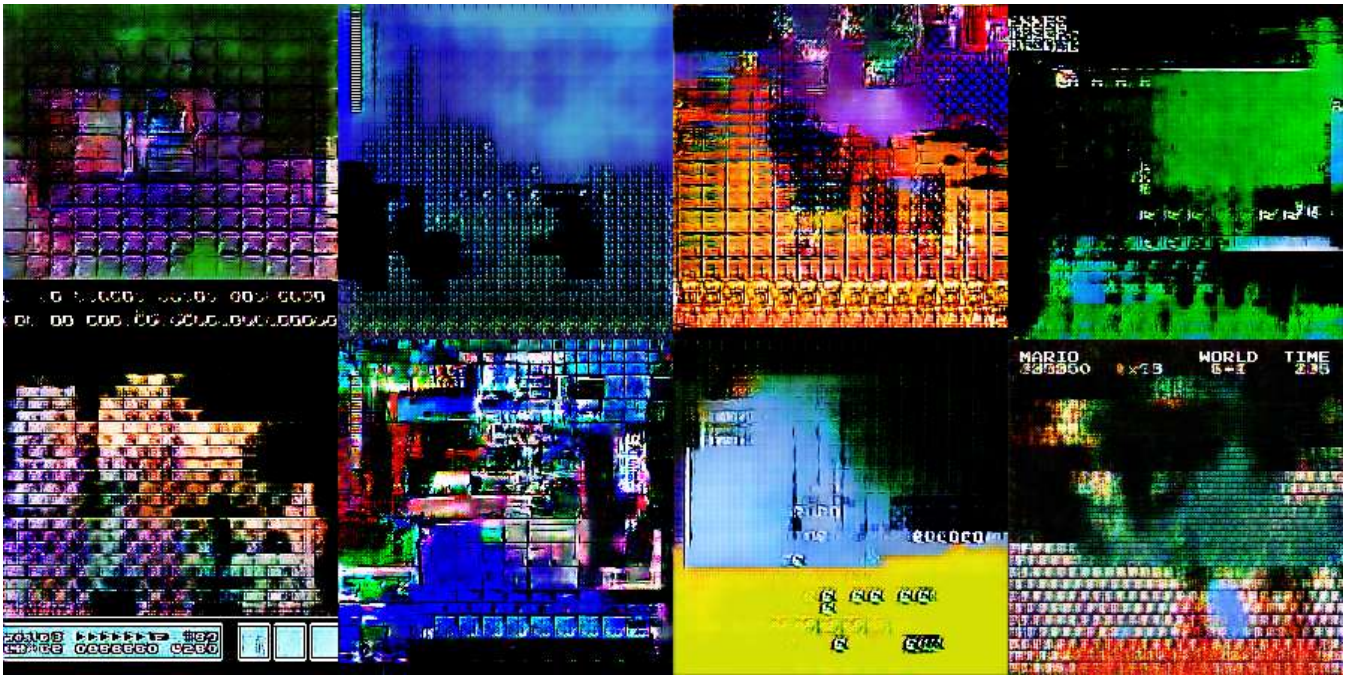
Now training begins. At first, the output from the Generator is pure noise. But it slowly start to take shape as the Generator learns to do a better job:



After several more training rounds, the images start to resemble nightmare-ish versions of classic Nintendo games:



As training continues further, we start to see the bricks and blocks we are hoping to find. You can also see screen elements like life bars and even some text:



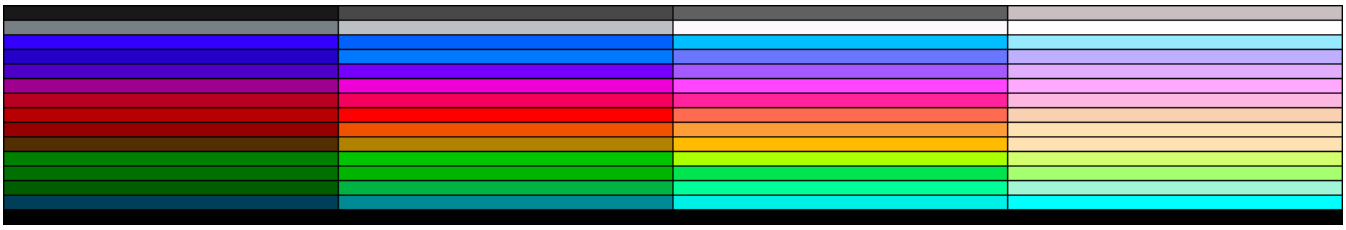
This is where things get complicated. How do we know the computer is creating brand new art and not just regurgitating art directly from the training images? In two of these images, you can clearly see the menu bar from Super Mario Bros. 3 and the header bar and bricks from the original Super Mario Bros.

Regurgitating training data is definitely something that can happen. By using a large training data set and not training too long, we can try to reduce the chance that this happens. But it's a thorny issue and research on it continues.

Since I'm just going for aesthetics, I tweaked the model until it produced art that looked original to me. But I can't *prove* that the new art is totally original except by searching the training data for similar art and verifying that there isn't any.

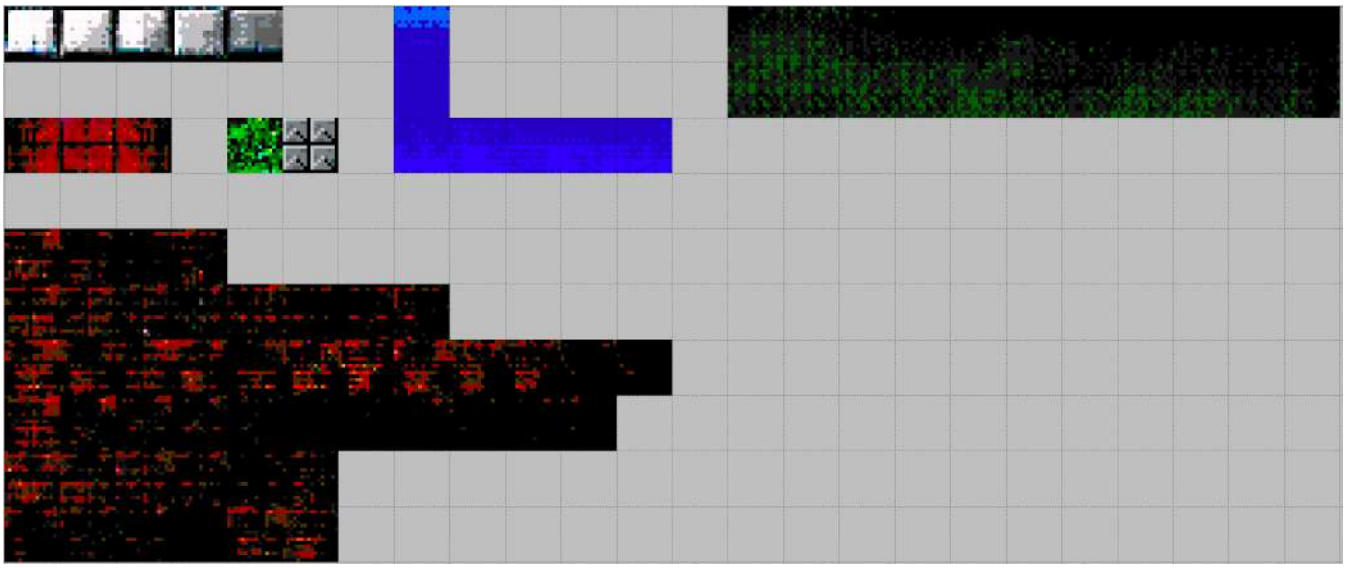
With a few hours of training, the generated images contained 16 x 16 tiles that looked nice to me. I was looking for some variations on a basic stone block, brick patterns, water patterns, bushes, and some general "spooky-looking" background atmosphere tiles.

Next I need to pre-process the generated images to the make sure they only used the 64 colors that are available on the NES:



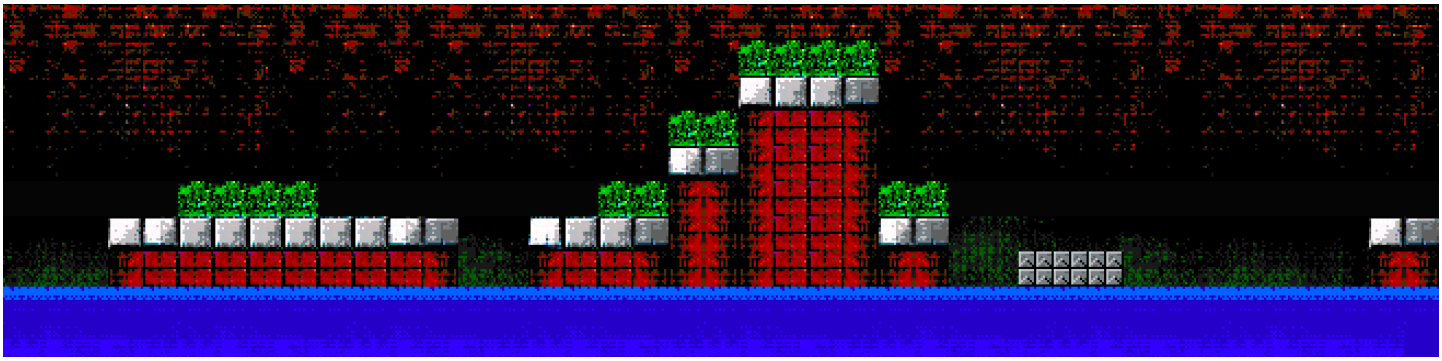
The original Nintendo could only display these 64 colors. Technically there's only 54 unique colors because some of them are duplicates.

Then I'll open up the 64-color images in the Tiled Map Editor. From there, I can easily grab the 16 x 16 tiles that match the aesthetic I want:



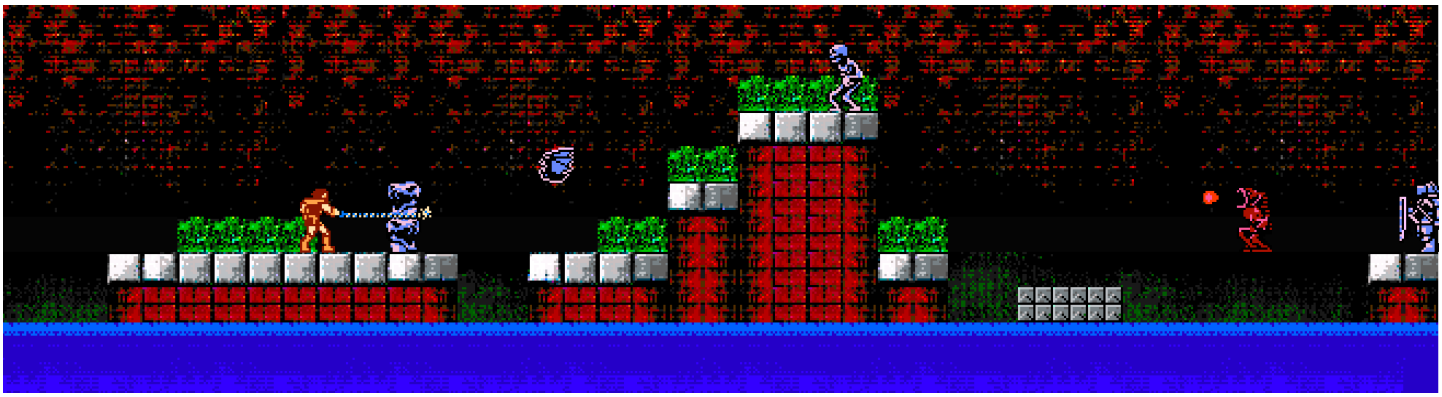
The tiles I grabbed out of the generated screenshots

Then inside of Tiled Map Editor, I'll arrange those 16 x 16 tiles into a simple level layout reminiscent of the NES game 'Castlevania':

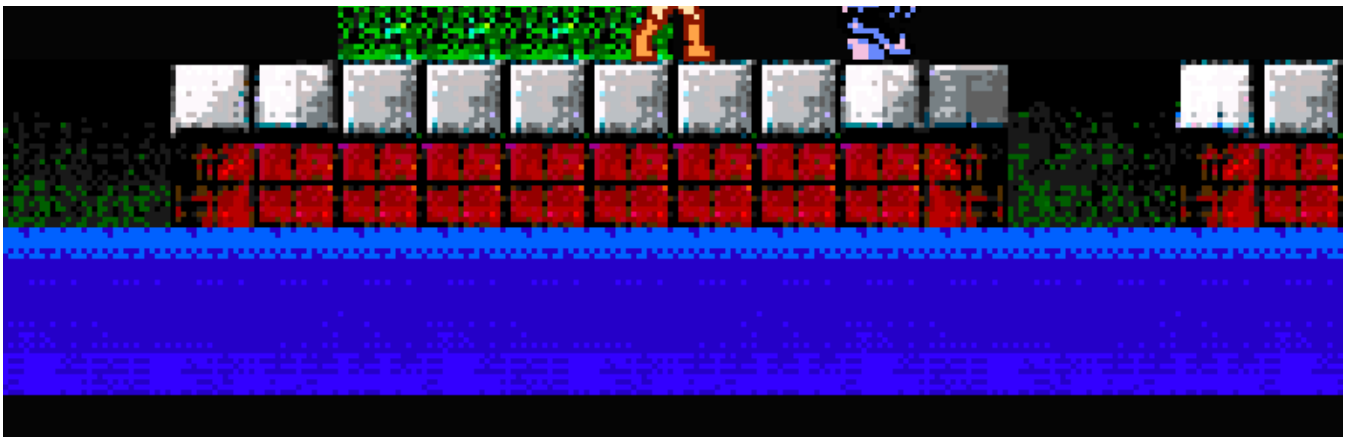


I think that looks pretty good! Keep in mind I didn't touch a single pixel with an image editor. Every tile came straight out of the DCGAN model.

Next, let's throw in the main character and some enemies from 'Castlevania' so we can see what this level would look like in action:

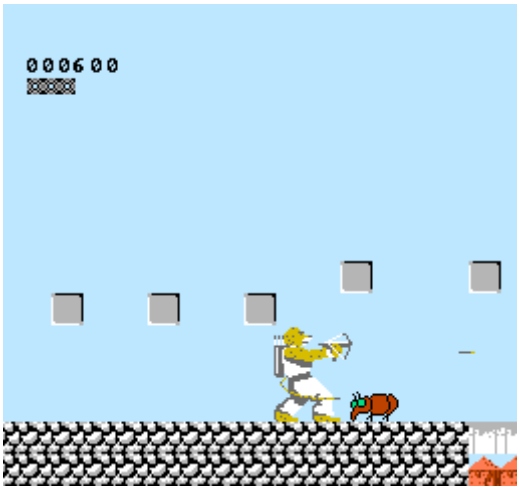


To get the full effect, let's see what the level would look like inside the game with the menu elements added:



So spooooky

I think that looks like the NES games that I remember! I'm not claiming it's the best NES art ever created, but it's certainly not the worst:



The Cheetahmen is not a good game.

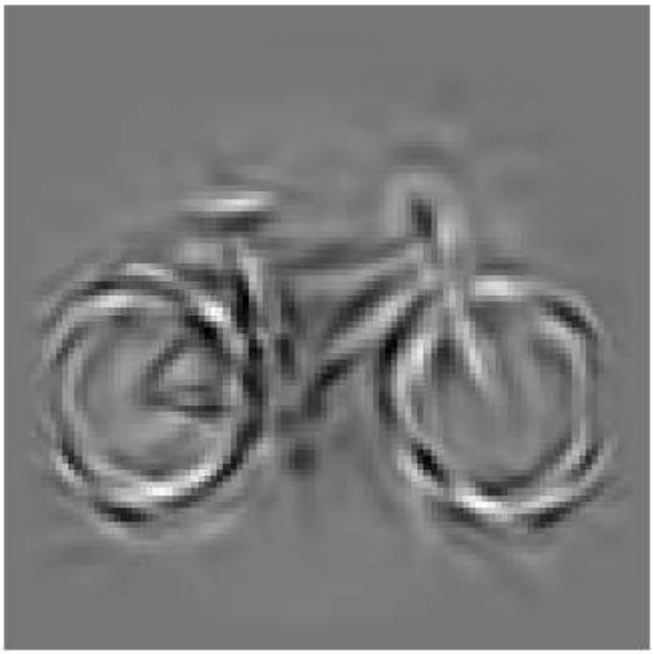
Is that it?

I get really excited about generative models like this. The idea of one day cranking out endless artwork with computers is fascinating to me. But when I talk to other people about this stuff, sometimes the response is "Is that it? That's so basic."

There's certainly a lot of hype around generative models right now. GANs are already being called the future of AI despite being notoriously hard to train and limited to generating tiny images. In fact, the very best models can currently only generate postage-stamp-sized pictures of mutant dogs:

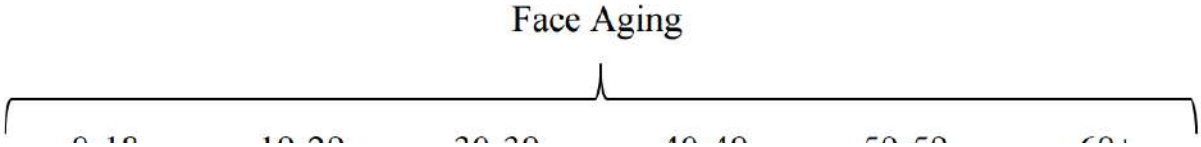
A nightmare animal! Photo from Ian Goodfellow's GAN Tutorial paper

But a couple of years ago, we couldn't do anything close to that. We were pretty excited by generated pictures that looked like this:



It's a bicycle! I swear!

And the technology is improving every single day. Here's a random paper that came out *this week* that uses GANs to age the faces of people:



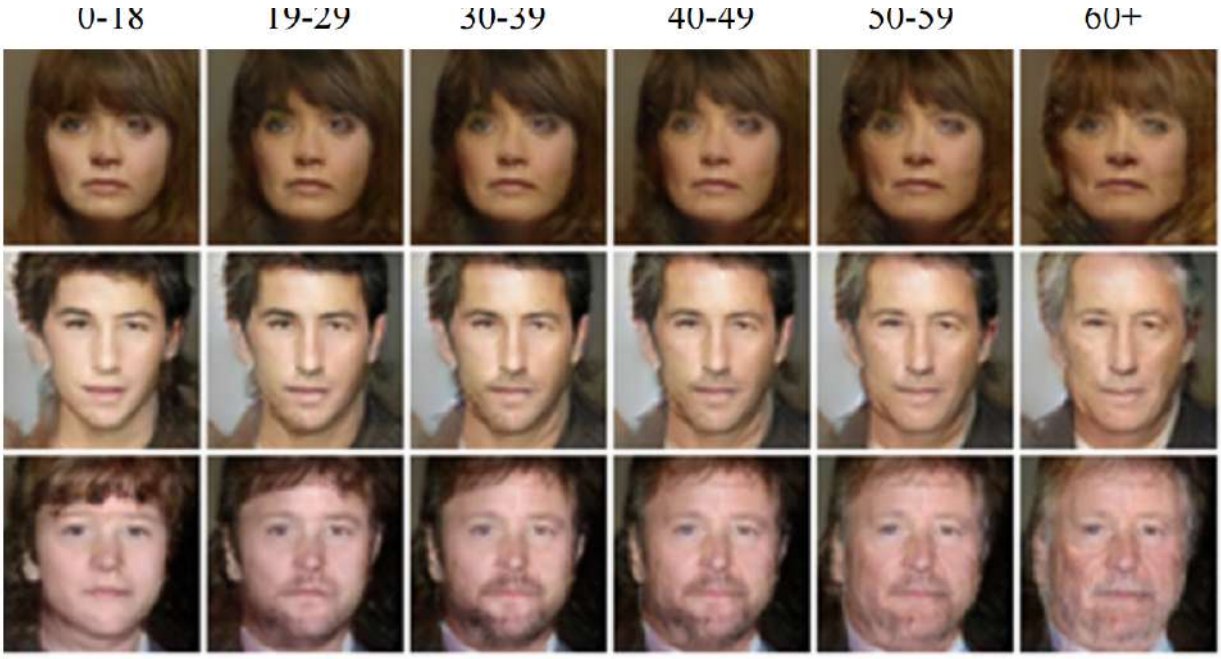


Image from "Face Aging With Conditional Generative Adversarial Networks"

If things keep improving at this pace, it won't be too long before generative models are a mainstream tool helping us create. It's a great time to start experimenting!

Keep Learning

If you want to learn more in depth about generative models and DCGANs, here are some recommended resources:

- Conditional generative adversarial networks for face generation by Jon Gauthier
- Generative Models overview from OpenAI
- Image Completion with Deep Learning in TensorFlow by Brandon Amos
- See how Tom White uses generative models to make art in his Neural Facegrid project
- Ian Goodfellow's original paper on GANs and his recent tutorial on them

. . .

This article is part of my **Machine Learning is Fun** series. You can check out the earlier parts here: *Part 1, Part 2, Part 3, Part 4, Part 5 and Part 6*

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.

Artificial Intelligence Machine Learning Gaming

Machine Learning is Fun Part 8: How to Intentionally Trick Neural Networks

A Look into the Future of Hacking

Adam Geitgey
Aug 17, 2017 · 11 min read

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 Русский, Tiế´ng Việt or 한국어.

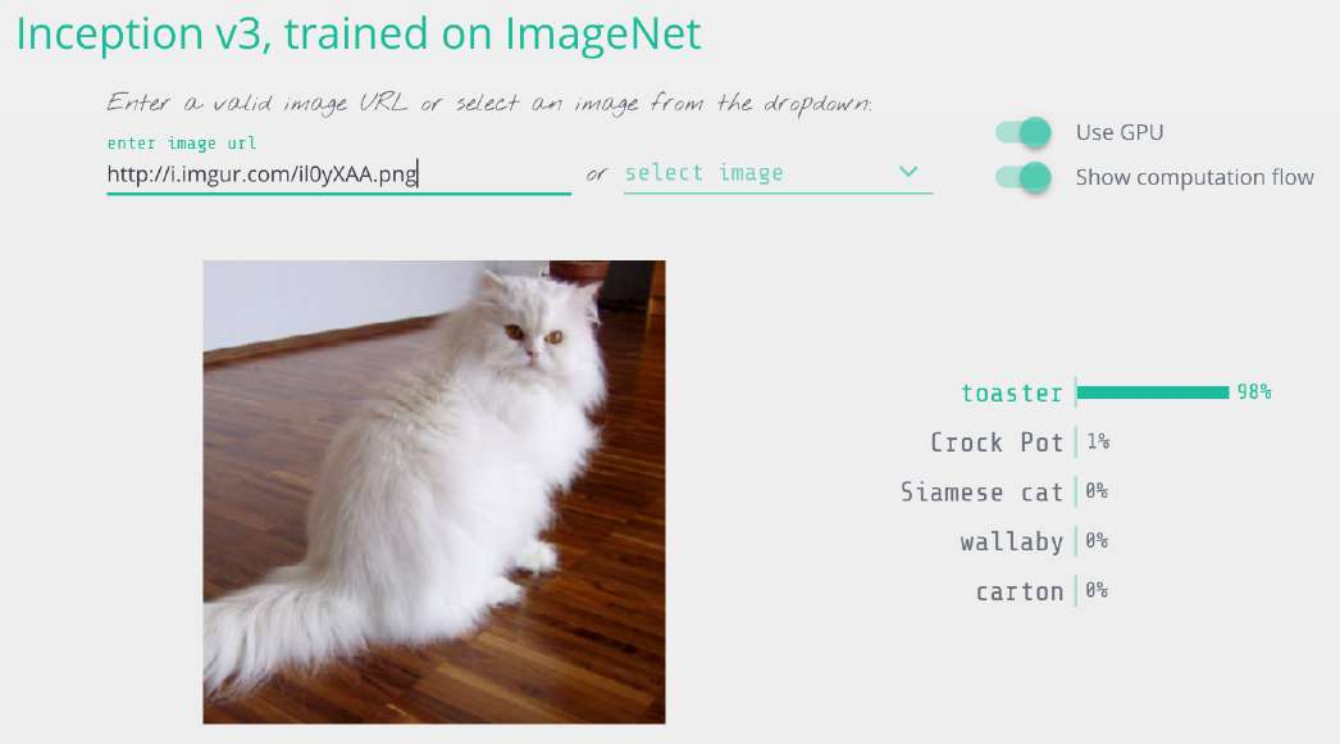
***Giant update:** I've written a new book based on these articles! It not only expands and updates all my articles, but it has tons of brand new content and lots of hands-on coding projects. Check it out now!*

Almost as long as programmers have been writing computer programs, computer hackers have been figuring out ways to exploit those programs. Malicious hackers take advantage of the tiniest bugs in programs to break into systems, steal data and generally wreak havoc.

100% Real Hackers™

But systems powered by deep learning algorithms should be safe from human interference, right? How is a hacker going to get past a neural network trained on terabytes of data?

It turns out that even the most advanced deep neural networks can be easily fooled. With a few tricks, you can force them into predicting whatever result you want:



I modified this cat picture so it would be recognized as a toaster.

So before you launch a new system powered by deep neural networks, let's learn exactly how to break them and what you can do to protect yourself from attackers.

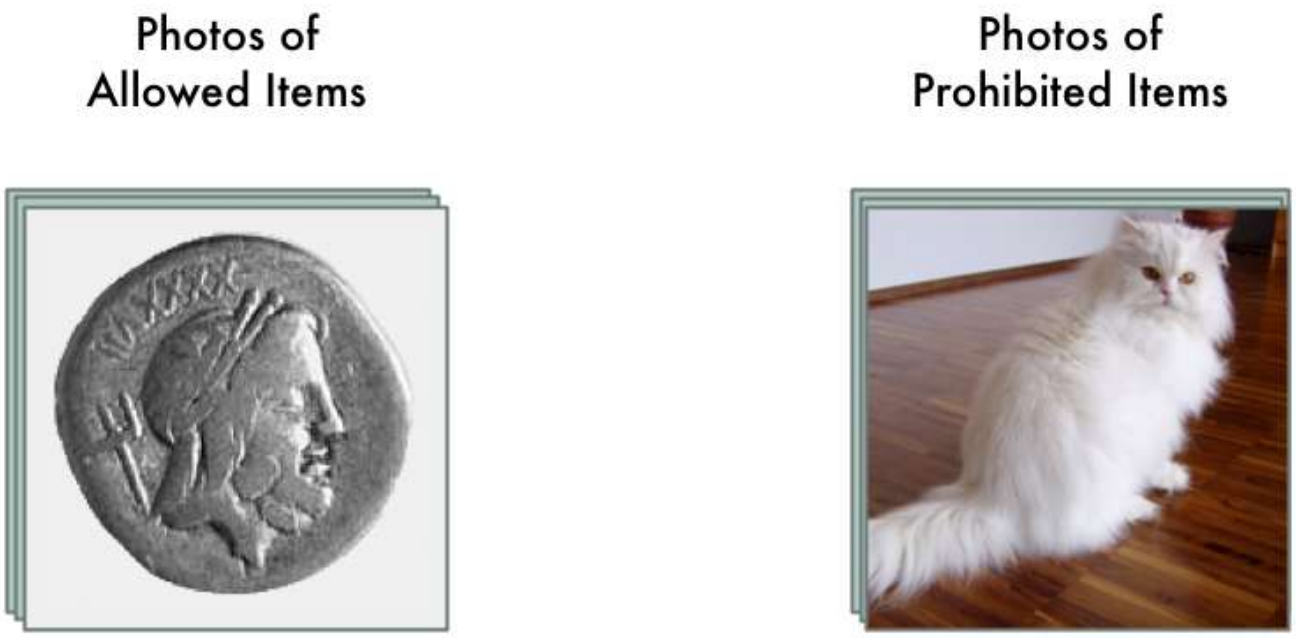
Neural Nets as Security Guards

Let's imagine that we run an auction website like Ebay. On our website, we want to prevent people from selling prohibited items — things like live animals.

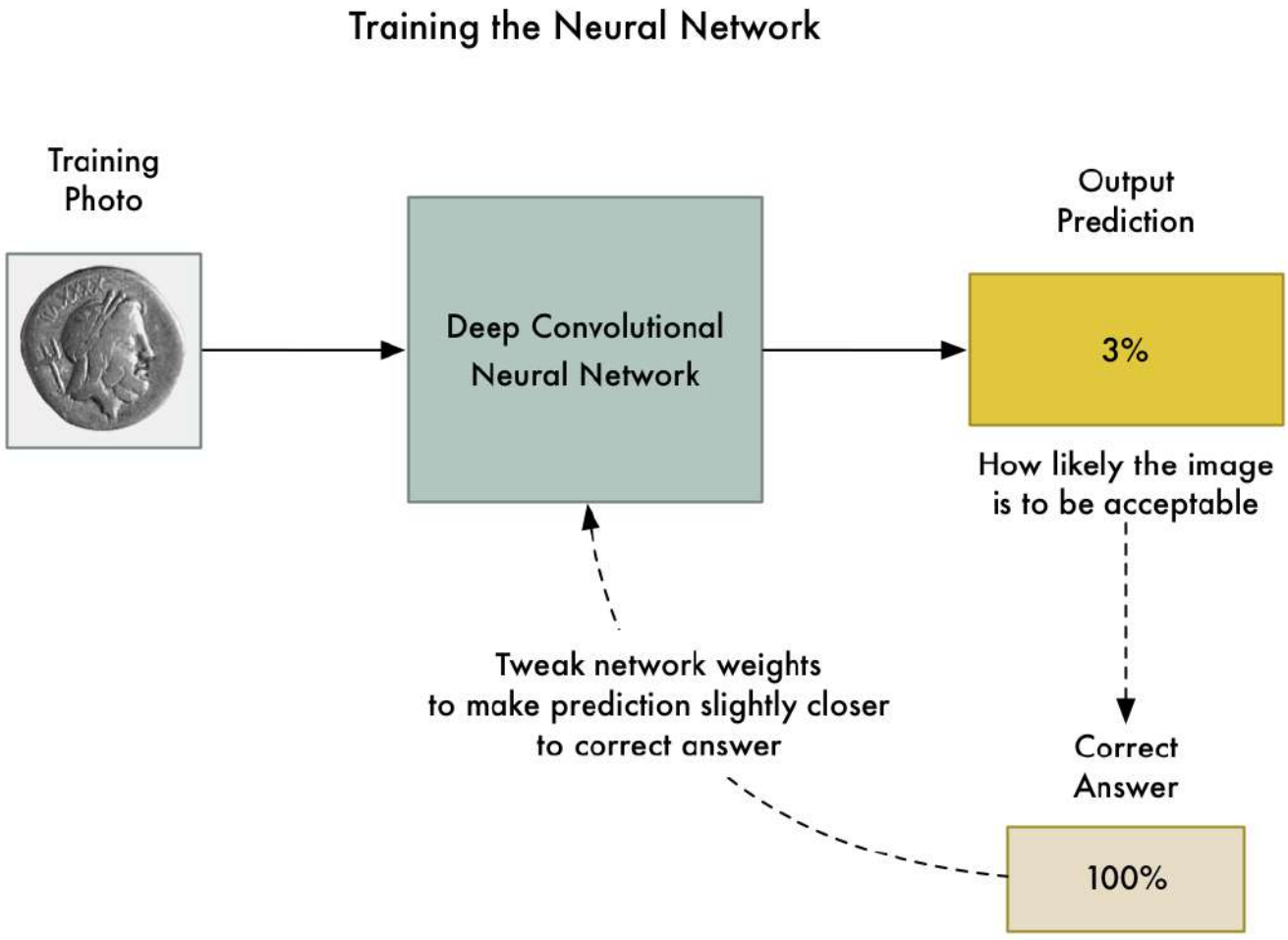
Enforcing these kinds of rules are hard if you have millions of users. We could hire hundreds of people to review every auction listing by hand, but that would be expensive. Instead, we can use deep learning to automatically check auction photos for prohibited items and flag the ones that violate the rules.

This is a typical image classification problem. To build this, we'll train a deep convolutional neural network to tell prohibited items apart from allowed items and then we'll run all the photos on our site through it.

First, we need a data set of thousands of images from past auction listings. We need images of both allowed and prohibited items so that we can train the neural network to tell them apart:

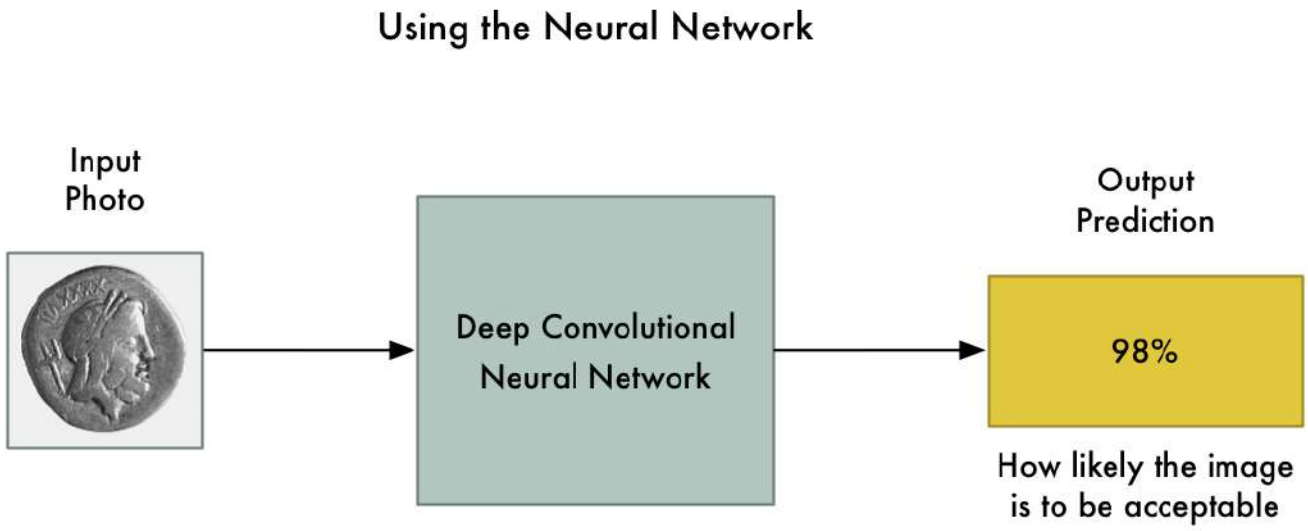


To train then neural network, we use the standard *back-propagation* algorithm. This is an algorithm were we pass in a training picture, pass in the expected result for that picture, and then walk back through each layer in the neural network adjusting their weights slightly to make them a little better at producing the correct output for that picture:



We repeat this thousands of times with thousands of photos until the model reliably produces the correct results with an acceptable accuracy.

The end result is a neural network that can reliably classify images:

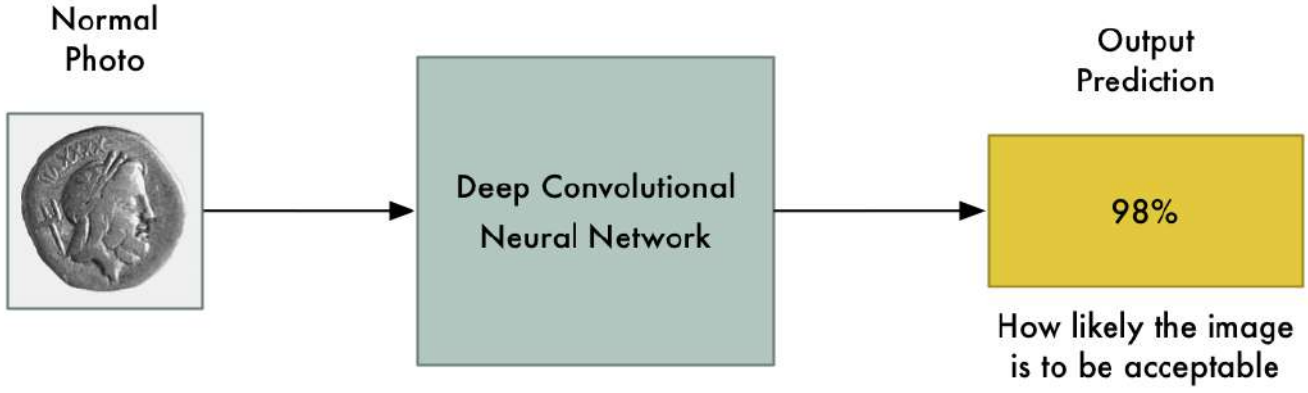


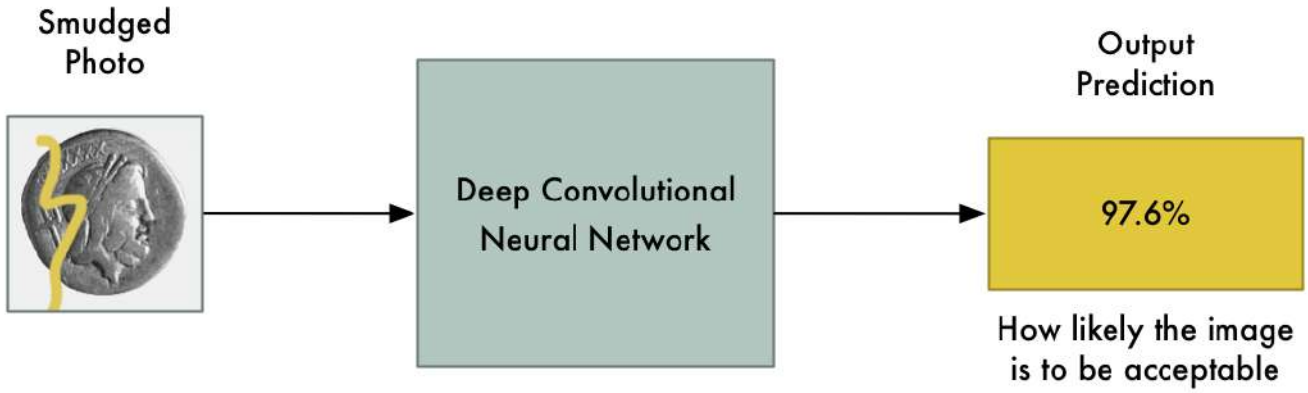
Note: If you want more detail on how convolution neural networks recognize objects in images, check out Part 3.

But things are not as reliable as they seem...

Convolutional neural networks are powerful models that consider the entire image when classifying it. They can recognize complex shapes and patterns no matter where they appear in the image. In many image recognition tasks, they can equal or even beat human performance.

With a fancy model like that, changing a few pixels in the image to be darker or lighter shouldn't have a big effect on the final prediction, right? Sure, it might change the final likelihood slightly, but it shouldn't flip an image from "prohibited" to "allowed".

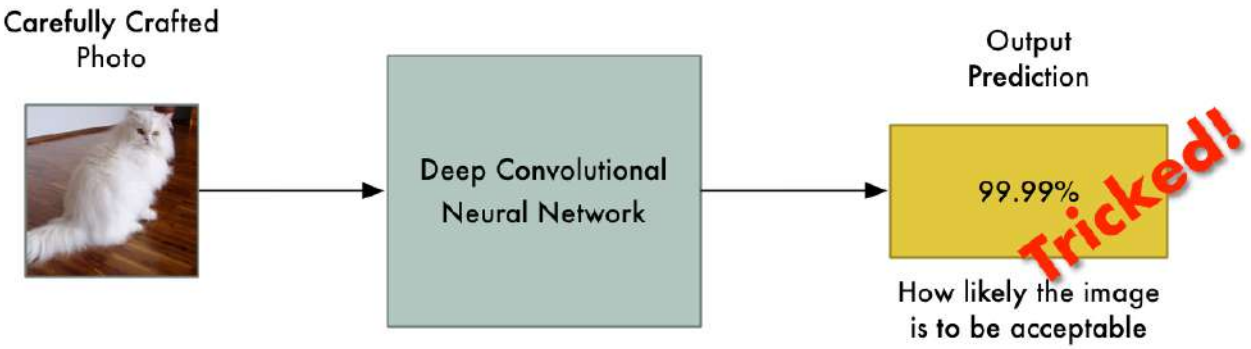




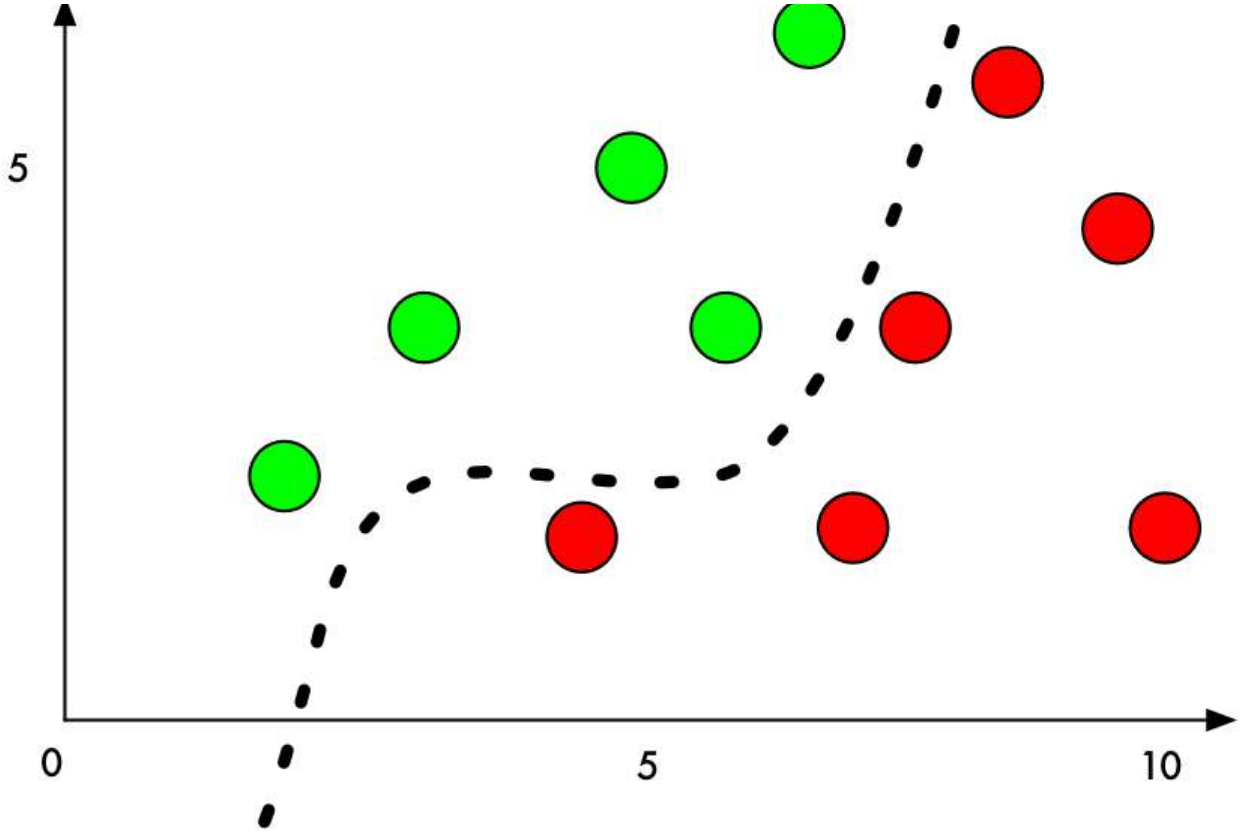
What we expect: Small changes to the input photo should only cause small changes to the final prediction.

But in a famous paper in 2013 called Intriguing properties of neural networks, it was discovered that this isn't always true. If you know *exactly which pixels to change* and *exactly how much to change them*, you can intentionally force the neural network to predict the wrong output for a given picture without changing the appearance of the picture very much.

That means we can intentionally craft a picture that is clearly a prohibited item but which completely fools our neural network:



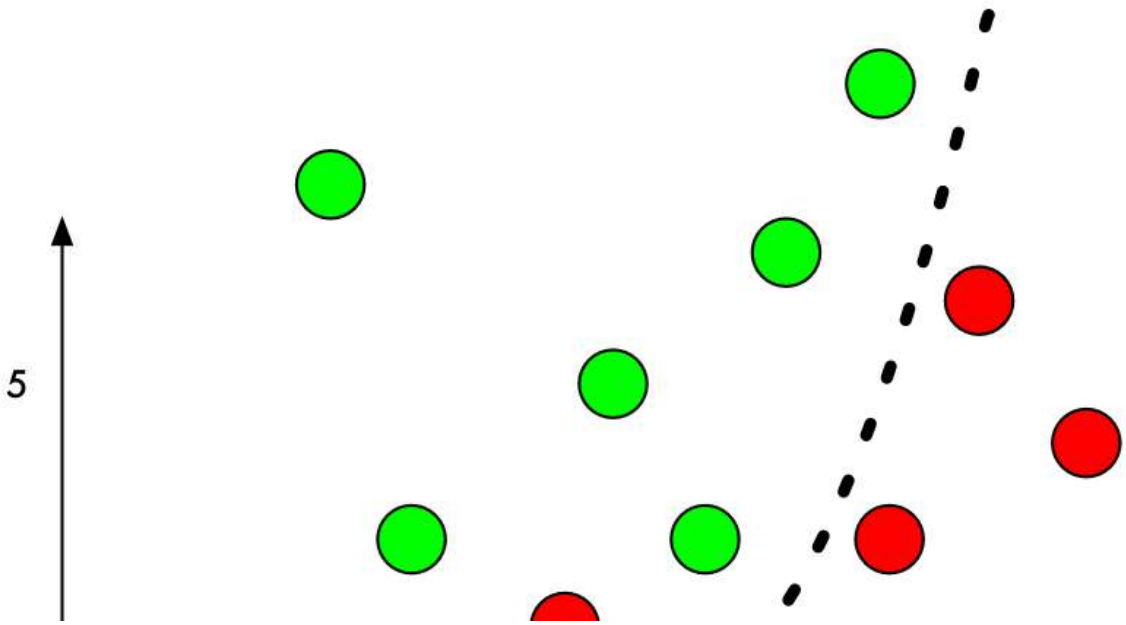
Why is this? A machine learning classifier works by finding a dividing line between the things it's trying to tell apart. Here's how that looks on a graph for a simple two-dimensional classifier that's learned to separate green points (acceptable) from red points (prohibited):

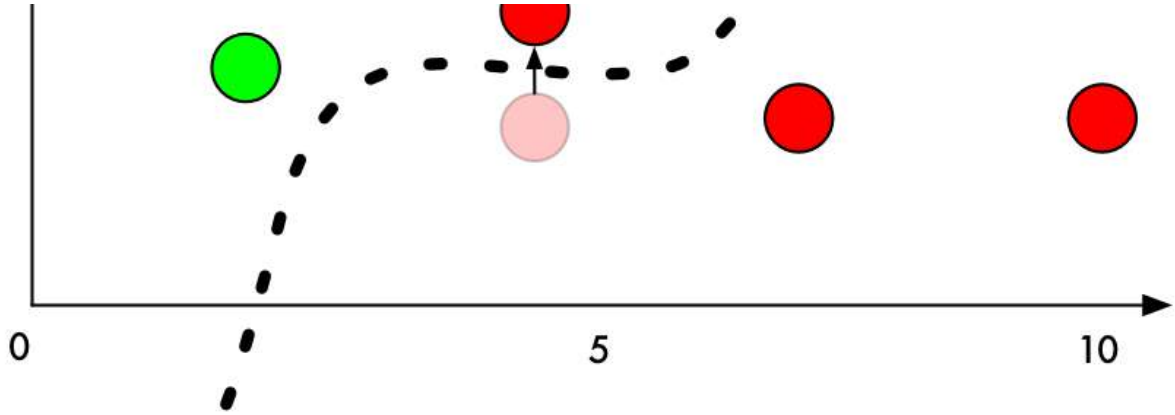


Right now, the classifier works with 100% accuracy. It's found a line that perfectly separates all the green points from the red points.

But what if we want to trick it into mis-classifying one of the red points as a green point? What's the minimum amount we could move a red point to push it into green territory?

If we add a small amount to the Y value of a red point right beside the boundary, we can just barely push it over into green territory:





So to trick a classifier, we just need to know which direction to nudge the point to get it over the line. And if we don't want to be too obvious about being nefarious, ideally we'll move the point as little as possible so it just looks like an honest mistake.

In image classification with deep neural networks, each "point" we are classifying is an entire image made up of thousands of pixels. That gives us *thousands* of possible values that we can tweak to push the point over the decision line. And if we make sure that we tweak the pixels in the image in a way that isn't too obvious to a human, we can fool the classifier without making the image look manipulated.

In other words, we can take a real picture of one object and change the pixels very slightly so that the image completely tricks the neural network into thinking that the picture is something else — and we can control exactly what object it detects instead:

Turning a cat into a toaster. Image detection results from the Keras.js web-based demo

How to Trick a Neural Network

We've already talked about the basic process of training a neural network to classify photos:

1. Feed in a training photo.

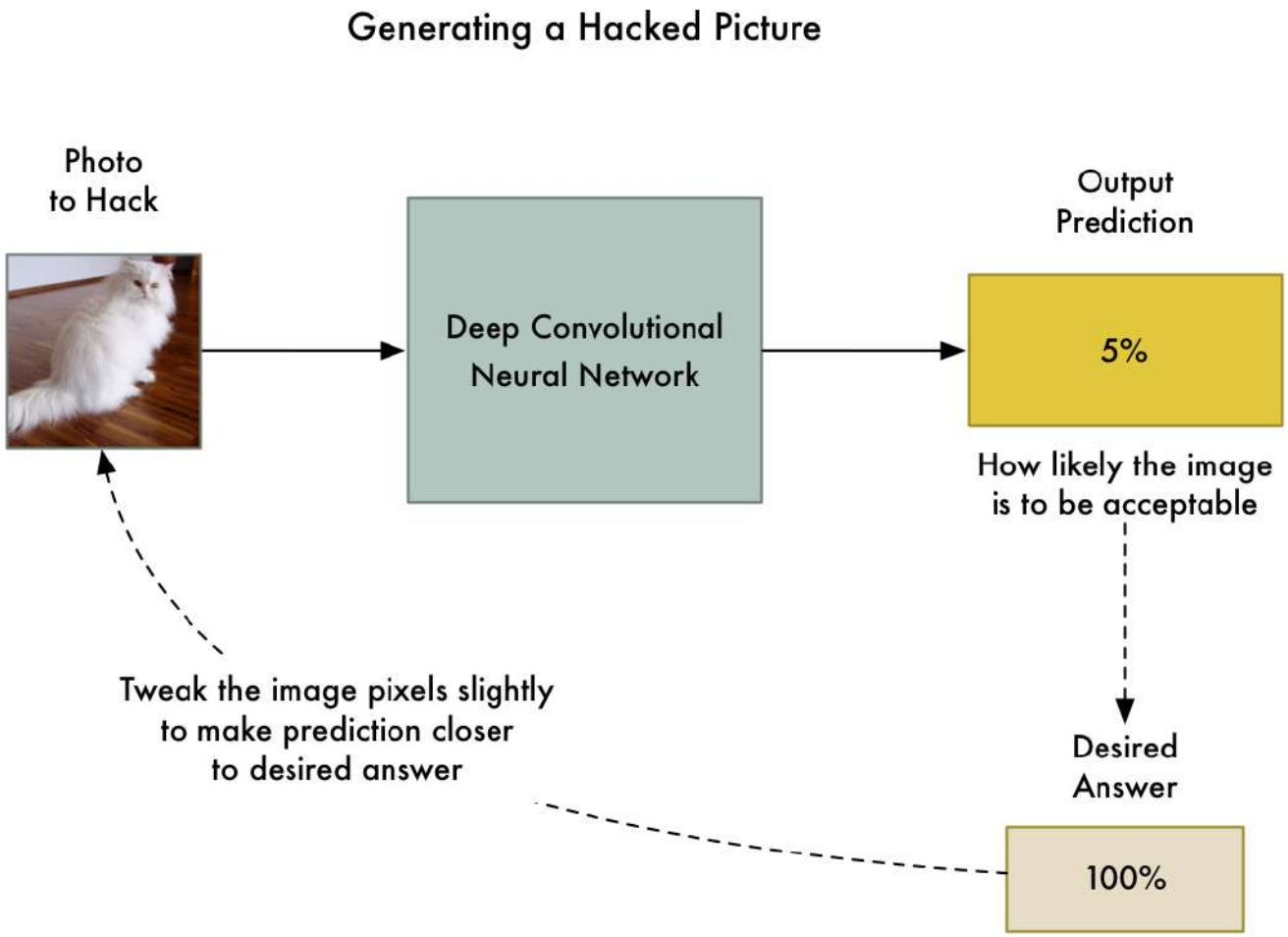
2. Check the neural network's prediction and see how far off the is from the correct answer.

3. Tweak the weighs of each layer in the neural network using back-propagation to make the final prediction slightly closer to the correct answer.

4. Repeat steps 1–3 a few thousand times with a few thousand different training photos.

But what if instead of tweaking the weights of the layers of the neural network, we instead tweaked the input image itself until we get the answer we want?

So let's take the already-trained neural network and "train" it again. But let's use back-propagation to adjust the ***input image*** instead of the neural network layers:



So here's the new algorithm:

1. Feed in the photo that we want to hack.

2. Check the neural network's prediction and see how far off the is from the ***answer we want to get*** for this photo.

3. Tweak our photo using back-propagation to make the final prediction slightly closer to the answer we want to get.

4. Repeat steps 1–3 a few thousand times with ***the same photo*** until the network gives us the answer we want.

At end of this, we'll have an image that fools the neural network without changing anything inside the neural network itself.

The only problem is that by allowing any single pixel to be adjusted without any limitations, the changes to the image can be drastic enough that you'll see them. They'll show up as discolored spots or wavy areas:

A hacked image with no constraints on how much a pixel can be tweaked. You can see green discolored spots around the cat and wavy patterns in the white wall.

To prevent these obvious distortions, we can add a simple constraint to our algorithm. We'll say that no single pixel in the hacked image can ever be changed by more than a tiny amount from the original image — let's say something like 0.01%. That forces our algorithm to tweak the image in a way that still fools the neural network without it looking too different from the original image.

Here's what the generated image looks like when we add that constraint:

A hacked image generated with a limit placed on how much any single pixel can be changed.

Even though that image looks the same to us, it still fools the neural network!

Let's Code It

To code this, first we need a pre-trained neural network to fool. Instead of training one from scratch, let's use one created by Google.

Keras, the popular deep learning framework, comes with several pre-trained neural networks. We'll use its copy of Google's Inception v3 deep neural network that was pre-trained to detect 1000 different kinds of objects.

Here's the basic code in Keras to recognize what's in a picture using this neural network. Just make sure you have Python 3 and Keras installed before you run it:

```
import numpy as np
from keras.preprocessing import image
from keras.applications import inception_v3

# Load pre-trained image recognition model
model = inception_v3.InceptionV3()

# Load the image file and convert it to a numpy array
img = image.load_img("cat.png", target_size=(299, 299))
input_image = image.img_to_array(img)

# Scale the image so all pixel intensities are between [-1, 1] as the model expects
input_image /= 255.
input_image -= 0.5
input_image *= 2.

```

```
# Add a 4th dimension for batch size (as Keras expects)
input_image = np.expand_dims(input_image, axis=0)

# Run the image through the neural network
predictions = model.predict(input_image)

# Convert the predictions into text and print them
predicted_classes = inception_v3.decode_predictions(predictions, top=1)
imagenet_id, name, confidence = predicted_classes[0][0]
print("This is a {} with {:.4}% confidence!".format(name, confidence * 100))
```

When we run it, it properly detects our image as a Persian cat:

```
$ python3 predict.py

This is a Persian_cat with 85.7% confidence!
```

Now let's trick it into thinking that this cat is a toaster by tweaking the image until it fools the neural network.

Keras doesn't have a built-in way to train against the input image instead of training the neural network layers, so I had to get a little tricky and code the training step manually.

Here's the code:

```
import numpy as np
from keras.preprocessing import image
from keras.applications import inception_v3
from keras import backend as K
from PIL import Image

# Load pre-trained image recognition model
model = inception_v3.InceptionV3()

# Grab a reference to the first and last layer of the neural net
model_input_layer = model.layers[0].input
model_output_layer = model.layers[-1].output

# Choose an ImageNet object to fake
# The list of classes is available here: https://gist.github.com/ageitgey/4e1342c10a71981d0b491
# Class #859 is "toaster"
object_type_to_fake = 859

```

```
# Load the image to hack
img = image.load_img("cat.png", target_size=(299, 299))
original_image = image.img_to_array(img)

# Scale the image so all pixel intensities are between [-1, 1] as the model expects
original_image /= 255.
original_image -= 0.5
original_image *= 2.

# Add a 4th dimension for batch size (as Keras expects)
original_image = np.expand_dims(original_image, axis=0)

# Pre-calculate the maximum change we will allow to the image
# We'll make sure our hacked image never goes past this so it doesn't look funny.
# A larger number produces an image faster but risks more distortion.
max_change_above = original_image + 0.01
max_change_below = original_image - 0.01

# Create a copy of the input image to hack on
hacked_image = np.copy(original_image)

# How much to update the hacked image in each iteration
learning_rate = 0.1

# Define the cost function.
# Our 'cost' will be the likelihood out image is the target class according to the pre-trained
cost_function = model_output_layer[0, object_type_to_fake]

# We'll ask Keras to calculate the gradient based on the input image and the currently predicte
# In this case, referring to "model_input_layer" will give us back image we are hacking.
gradient_function = K.gradients(cost_function, model_input_layer)[0]

# Create a Keras function that we can call to calculate the current cost and gradient
grab_cost_and_gradients_from_model = K.function([model_input_layer, K.learning_phase()], [cost_

cost = 0.0

# In a loop, keep adjusting the hacked image slightly so that it tricks the model more and more
# until it gets to at least 80% confidence
while cost < 0.80:
    # Check how close the image is to our target class and grab the gradients we
    # can use to push it one more step in that direction.
    # Note: It's really important to pass in '0' for the Keras learning mode here!
    # Keras layers behave differently in prediction vs. train modes!
    cost, gradients = grab_cost_and_gradients_from_model([hacked_image, 0])

    # Move the hacked image one step further towards fooling the model
    hacked_image += gradients * learning_rate
```

```
        hacked_image += gradients * learning_rate

        # Ensure that the image doesn't ever change too much to either look funny or to become an i
        hacked_image = np.clip(hacked_image, max_change_below, max_change_above)
        hacked_image = np.clip(hacked_image, -1.0, 1.0)

        print("Model's predicted likelihood that the image is a toaster: {:.8}%".format(cost * 100)

    # De-scale the image's pixels from [-1, 1] back to the [0, 255] range
    img = hacked_image[0]
    img /= 2.
    img += 0.5
    img *= 255.

    # Save the hacked image!
    im = Image.fromarray(img.astype(np.uint8))
    im.save("hacked-image.png")
```

```
$ python3 generated_hacked_image.py

Model's predicted likelihood that the image is a toaster: 0.00072%

[ .... a few thousand lines of training .... ]

Model's predicted likelihood that the image is a toaster: 99.4212%
```

Note: If you don't have a GPU, this might take a few hours to run. If you do have a GPU properly configured with Keras and CUDA, it shouldn't take more than a couple of minutes to run.

Now let's test the hacked image that we just made by running it through the original model again:

```
$ python3 predict.py

This is a toaster with 98.09% confidence!
```

We did it! We tricked the neural network into thinking that a cat is a toaster!

What can we do with a Hacked Image?

Created a hacked image like this is called “generating an adversarial example”. We’re intentionally crafting a piece of data so that a machine-learning model will misclassify it. It’s a neat trick, but why does this matter in the real world?

Research has show that these hacked images have some surprising properties:

1. Hacked images can still fool neural networks even when they are printed out on paper! So you can use these hacked images to fool physical cameras or scanners, not just systems where upload an image file directly.
2. Images that fool one neural network tend to fool other neural networks with entirely different designs if they were trained on similar data.

So we can potentially do a lot with these hacked images!

But there is still a big limitation with how we create these images — our attack requires direct access to the neural network itself. Because we are actually “training” against the neural network to fool it, we need a copy of it. In the real world, no company is going to let you download their trained neural network’s code, so that means we can’t attack them… Right?

Nope! Researchers have recently shown that you can train your own substitute neural network to mirror another neural network by probing it to see how it behaves. Then you can use your substitute neural network to generate hacked images that still often fool the original network! This is called a *black-box attack*.

The applications of black-box attacks are limitless. Here are some plausible examples:

- Trick self-driving cars into seeing a stop sign as a green light — this could cause car crashes!
- Trick content filtering systems into letting offensive/illegal content through.
- Trick ATM check scanners into thinking the handwriting on a check says the check is for a greater amount then it actually is (with plausible deniability if you get caught!)

And these attack methodology isn’t limited to just images. You can use the same kind of approach to fool classifiers that work on other types of data. For example, you could

trick virus scanners into recognizing your virus as safe code!

How can we protect ourselves against these attacks?

So now that we know it's possible to trick neural networks *(and all other machine learning models too)*, how do we defend against this?

The short answer is that no one is *entirely* sure yet. Preventing these kinds of attacks is still an on-going area of research. The best way to keep up with the latest developments is by reading the cleverhans blog maintained by Ian Goodfellow and Nicolas Papernot, two of the most influential researchers in this area.

But there are some things we do know so far:

- If you simply create lots of hacked images and include them in your training data set going forward, that seems to make your neural network more resistant to these attacks. This is called Adversarial Training and is probably the most reasonable defense to consider adopting right now.
- There is another somewhat effective approach called Defensive Distillation where you train a second model to mimic your original model. But this approach is new and rather complicated, so I wouldn't invest in this yet unless you have specialized needs.
- Pretty much every other idea researchers have tried so far has failed to be helpful in preventing these attacks.

Since we don't have any final answers yet, its worth thinking about the scenarios where you are using neural networks so that you can at least lessen the risk that this kind of attack would cause damage your business.

For example, if you have a single machine learning model as the only line of defense to grant access to a restricted resource and assume it can't be fooled, that's probably a bad idea. But if you use machine learning as a step in a process where there is still human verification, that's probably fine.

In other words, treat machine learning models in your architecture like any other component that can potentially be bypassed. Think through the implications of what would happen if a user intentionally sets out to fool them and think of ways to mitigate those scenarios.

Learning More

Want to learn more about Adversarial Examples and protecting against them?

- This area of research is only a few years old, so it's easy to get caught up by reading a few key papers: Intriguing properties of neural networks, Explaining and Harnessing Adversarial Examples, Practical Black-Box Attacks against Machine Learning and Adversarial examples in the physical world.
- Follow Ian Goodfellow and Nicolas Papernot's cleverhans blog to keep up with the latest research.
- Check out the on-going Kaggle competition that's challenging researchers to come up with new ways to defend against these attacks.

. . .

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.

Machine Learning Hacking Neural Networks