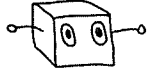


YOU LOOK
LIKE A THING
 AND
I LOVE YOU

How Artificial Intelligence Works
and Why It's Making the World
a Weirder Place

Janelle Shane

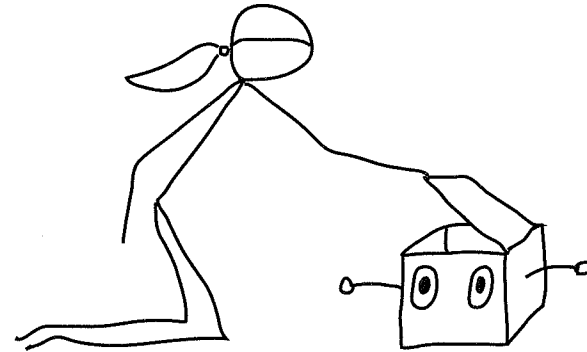


VORACIOUS

Little, Brown and Company
New York ■ Boston ■ London

CHAPTER 3

How does It actually learn?



Remember that in this book I'm using the term *AI* to mean “machine learning programs.” (Refer to the handy chart on page 8 for a list of stuff that I am or am not considering to be AI. Sorry, person in a robot suit.) A machine learning program, as I explained in chapter 1, uses trial and error to solve a problem. But how does that process work? How does a program go from producing a jumble of random letters to writing recognizable knock-knock jokes, all without a human telling it how words work or what a joke even is?

There are lots of different methods of machine learning, many of which have been around for decades, often long before people started calling them AI. Today, these technologies are combined or remixed or made ever more powerful by faster processing and bigger datasets. In this chapter we'll look at a few of the most common types, peeking under the hood to see how they learn.

NEURAL NETWORKS

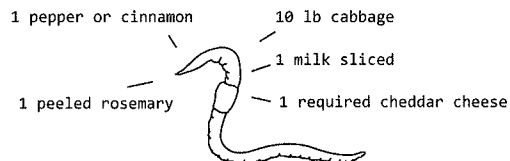
These days, when people talk about AI, or **deep learning**, what they're often referring to are **artificial neural networks (ANNs)**. (ANNs have also been known as **cybernetics**, or **connectionism**.)

There are lots of ways to build artificial neural networks, each meant for a particular application. Some are specialized for image recognition, some for language processing, some for generating music, some for optimizing the productivity of a cockroach farm, some for writing confusing jokes. But they're all loosely modeled after the way the brain works. That's why they're called artificial neural networks—their cousins, **biological neural networks**, are the original, far more complex models. In fact, when programmers made the first artificial neural networks, in the 1950s, the goal was to test theories about how the brain works.

In other words, artificial neural networks are imitation brains.

They're built from a bunch of simple chunks of software, each able to perform very simple math. These chunks are usually called **cells** or **neurons**, an analogy with the neurons that make up our own brains. The power of the neural network lies in how these cells are connected.

Now, compared to actual human brains, artificial neural networks aren't that powerful. The ones I use for a lot of the text generation in this book have as many neurons as... a worm.

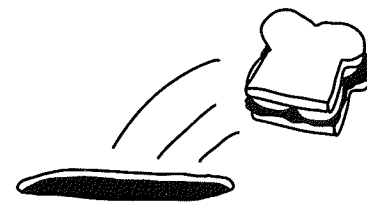


Unlike a human, the neural net is at least able to devote its entire one-worm-power brain to the task at hand (if I don't accidentally distract it

with extraneous data). But how can you solve problems using a bunch of interconnected cells?

The most powerful neural networks, the ones that take months and tens of thousands of dollars' worth of computing time to train, have far more neurons than my laptop's neural net, some even exceeding the neuron count of a single honeybee. Looking at how the size of the world's largest neural networks has increased over time, a leading researcher estimated in 2016 that artificial neural networks might be able to approach the number of neurons in the human brain by around 2050.¹ Will this mean that AI will approach the intelligence of a human then? Probably not even close. Each neuron in the human brain is much more complex than the neurons in an artificial neural network—so complex that each human neuron is more like a complete many-layered neural network all by itself. So rather than being a neural network made of eighty-six billion neurons, the human brain is a neural network made of eighty-six billion neural networks. And there are far more complexities to our brains than there are to ANNs, including many we don't fully understand yet.

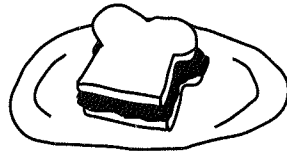
THE MAGIC SANDWICH HOLE



Let's say, hypothetically, that we have discovered a magic hole in the ground that produces a random sandwich every few seconds. (Okay, this is

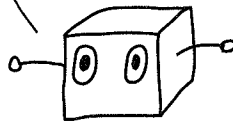
very hypothetical.) The problem is that the sandwiches are very, very random. Ingredients include jam, ice cubes, and old socks. If we want to find the good ones, we'll have to sit in front of the hole all day and sort them.

Alas, ear wax



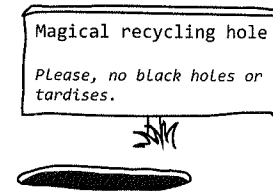
But that's going to get tedious. Good sandwiches are only one in a thousand. However, they *are* very, very good sandwiches. Let's try to automate the job.

I can help!

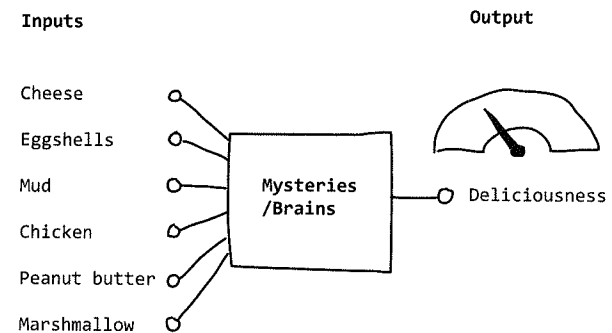


To save ourselves time and effort, we want to build a neural network that can look at each sandwich and decide whether it's good. For now, let's ignore the problem of how to get the neural network to recognize the ingredients the sandwiches are made of—that's a really hard problem. And let's ignore the problem of how the neural network is going to pick up each sandwich. That's also really, really hard—not just recognizing the motion of the sandwich as it flies from the hole but also instructing a robot arm to grab a slim paper-and-motor-oil sandwich or a thick bowling-ball-and-mustard sandwich. Let's assume, then, that the neural net knows what's in each sandwich and that

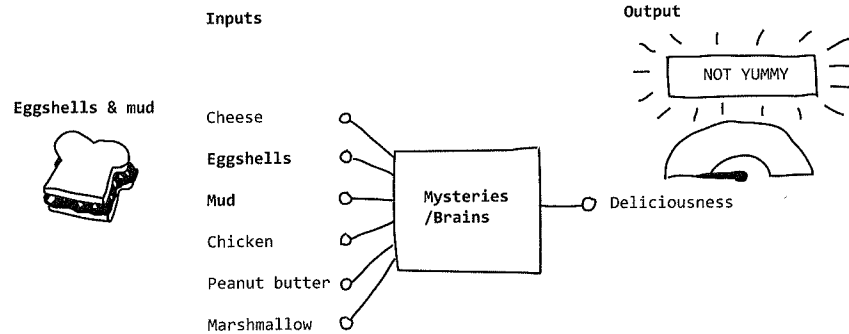
we've solved the problem of physically moving the sandwiches. It just has to decide whether to save this sandwich for human consumption or throw it into the recycling chute. (We're also going to ignore the mechanism of the recycling chute—let's say it's another magic hole.)



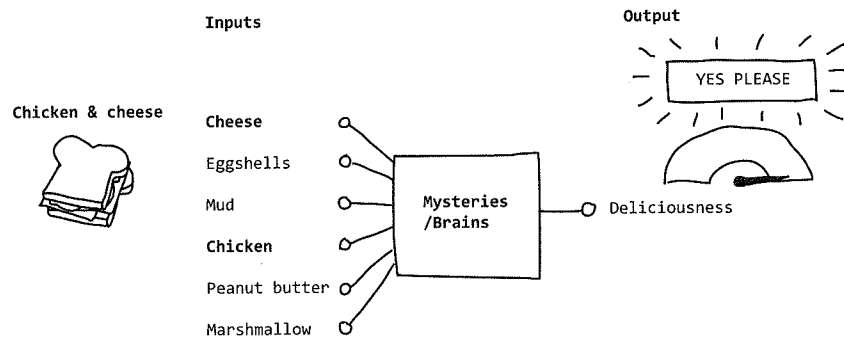
This reduces our task to something simple and narrow—as we discovered in chapter 2, that makes it a good candidate for automation with a machine learning algorithm. We have a bunch of inputs (the names of the ingredients), and we want to build an algorithm that will use them to figure out our single output, a number that indicates whether the sandwich is good. We can draw a simple “black box” picture of our algorithm, and it looks like this:



We want the “deliciousness” output to change depending on the combination of ingredients in the sandwich. So if a sandwich contains eggshells and mud, our black box should do this:

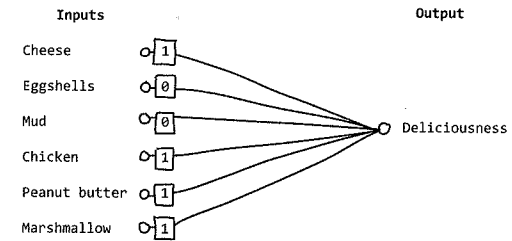


But if the sandwich contains chicken and cheese, it should do this instead:

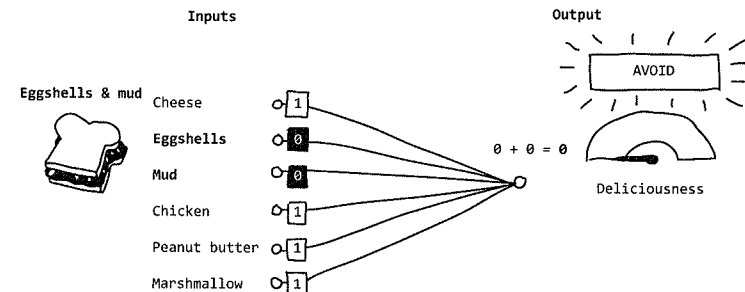


Let's look at how things are hooked up inside the black box.

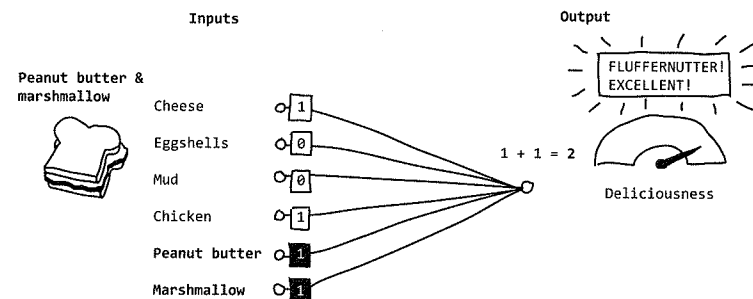
First, let's make it simple. We hook up all the inputs (all the ingredients) to our single output. To get our deliciousness rating, we add each ingredient's contribution. Clearly each ingredient should not contribute equally — the presence of cheese would make the sandwich more delicious, while the presence of mud would make the sandwich less delicious. So each ingredient gets a different weight. The good ones get a weight of 1, while the ones we want to avoid get a weight of 0. Our neural network looks like this:



Let's test it with some sample sandwiches. Suppose the sandwich contains mud and eggshells. Mud and eggshells both contribute a 0, so the deliciousness rating is $0 + 0 = 0$.

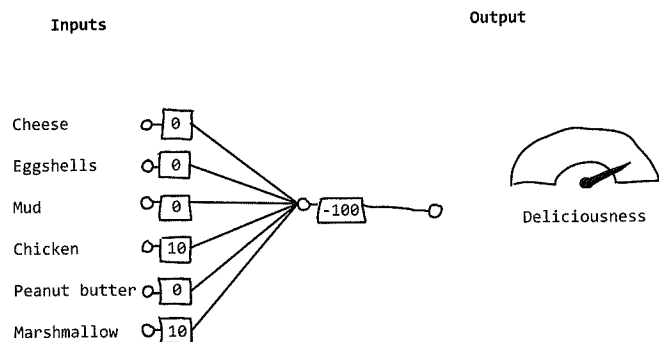


But a peanut-butter-and-marshmallow sandwich will get a rating of $1 + 1 = 2$. (Congratulations! You have been blessed with that New England delicacy, the fluffernutter.)



With this neural network configuration, we successfully avoid all the sandwiches that contain only eggshells, mud, and other inedible things. But this simple one-layer neural network is not sophisticated enough to recognize that some ingredients, while delicious on their own, are not delicious in combination with certain others. It's going to rate a chicken-and-marshmallow sandwich as delicious, the equal of the fluffernutter. It's also susceptible to something we'll call the **big sandwich bug**: a sandwich that contains mulch might still be rated as tasty if it contains enough good ingredients to cancel out the mulch.

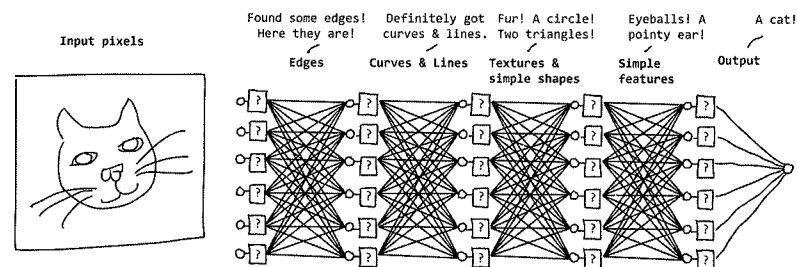
To get a better neural network, we're going to need another layer of cells.



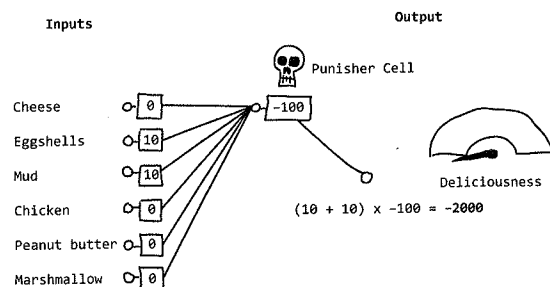
Here's our neural network now. Each ingredient is connected to our new layer of cells, and each cell is connected to the output. This new layer is called a **hidden layer**, because the user only sees the inputs and the outputs. Just as before, each connection has its own weight, so it affects our final deliciousness output in different ways. This isn't deep learning yet (that would require even more layers), but we're getting there.

DEEP LEARNING

Adding hidden layers to our neural network gets us a more sophisticated algorithm, one that's able to judge sandwiches as more than the sum of their ingredients. In this chapter, we've only added one hidden layer, but real-world neural networks often have several. Each new layer means a new way to combine the insights from the previous layer—at higher and higher levels of complexity, we hope. This approach—lots of hidden layers for lots of complexity—is known as **deep learning**.

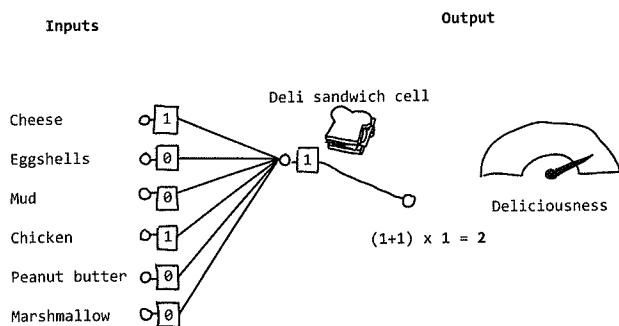


With this neural network, we can finally avoid bad ingredients by connecting them to a cell that we'll call the punisher. We'll give that cell a huge negative weight (let's say -100) and connect everything bad to it with a weight of 10. Let's make the first cell the punisher and connect the mud and eggshells to it. Here's what that looks like:



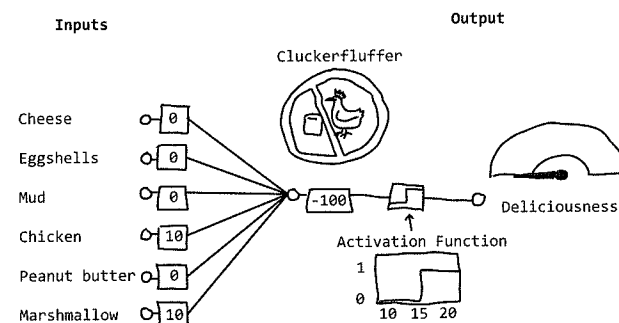
Now, no matter what happens in the other cells, a sandwich is likely to fail if it contains eggshells or mud. Using the punisher cell, we can beat the big sandwich bug.

We can do other things with the rest of the cells—like finally make a neural network that knows which ingredient combos work. Let's use the second cell to recognize chicken-and-cheese-type sandwiches. We'll refer to it as the deli sandwich cell. We connect chicken and cheese to it with weights of 1 (we'll also do this with ham and turkey and mayo) and connect everything else to it with weights of 0. And this cell gets connected to the output with a modest weight of 1. The deli sandwich cell is a good thing, but if we get too excited about it and assign it a very high weight, we'll be in danger of making the punisher cell less powerful. Let's look at what this cell does.



A chicken-and-cheese sandwich will cause this cell to contribute a cheerful $1 + 1 = 2$ to the final output. But adding marshmallow to the chicken-and-cheese sandwich doesn't hurt it at all, even though it makes a pretty objectively less delicious sandwich. To fix that, we'll need other cells that specifically look for and punish incompatibilities.

Cell 3, for example, might look for the chicken-marshmallow combination (let's call it the cluckerfluffer) and severely punish any sandwich that contains it. It would be hooked up like this:



Cell 3 returns a devastating $(10 + 10) \times -100 = -2000$ to any sandwich that dares to combine chicken and marshmallow. It's acting like a very specialized punisher cell, designed specifically to punish chicken and marshmallow. Notice that I've shown an extra part of the cluckerfluffer cell here, called the **activation function**, because without it, the cell will punish *any* sandwich that contains chicken *or* marshmallow. With a threshold of 15, the activation function stops the cell from turning on when just chicken (10 points) or marshmallow (10 points) is present—it will return a neutral 0. But if *both* are present ($10 + 10 = 20$ points), the threshold of 15 is exceeded, and the cell turns on. *Boom!* The activated cell punishes any combination of ingredients that exceeds its threshold.

Cluckerfluffer



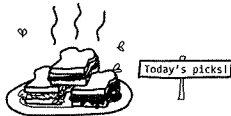
With all the cells connected in similarly sophisticated configurations, we have a neural net that can sort out the best sandwiches the magic hole has to offer.



THE TRAINING PROCESS

So now we know what a well-configured sandwich-picking neural network might look like. But the point of using machine learning is that we don't have to set up the neural network by hand. Instead, it should be able to configure *itself* into something that does a great sandwich-picking job. How does this training process work?

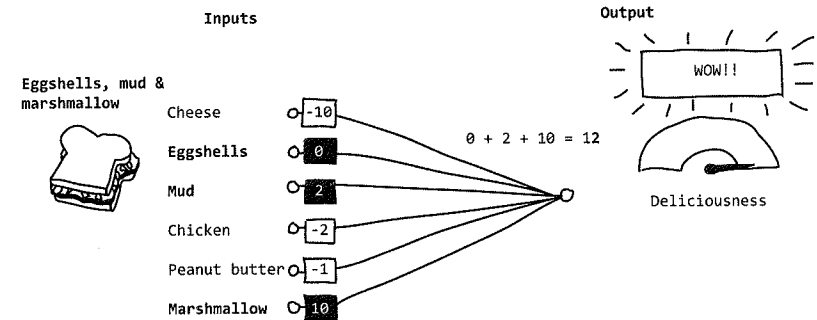
Let's go back to a simple two-layer neural network. At the beginning of the training process, it's starting completely from scratch, with random weights for each ingredient. Chances are it's very, very bad at rating sandwiches.



We'll need to train it with some real-world data — some examples of the correct way to rate a sandwich, as demonstrated by real humans. As the

neural net rates each sandwich, it needs to compare its ratings against those of a panel of cooperative sandwich judges. Note: never volunteer to test the early stages of a machine learning algorithm.

For this example, we'll go back to the very simple neural network. Remember, since we're trying to train it from scratch, we're ignoring all our prior knowledge about what the weights should be, and starting from random ones. Here they are:



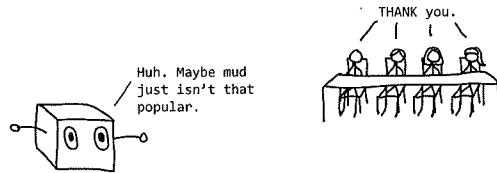
It *hates* cheese. It *loves* marshmallow. It's rather fond of mud. And it can take or leave eggshells.

The neural net looks at the first sandwich that pops out of the magic sandwich hole and using its (terrible) judgment, gives it a score. It's a marshmallow, eggshell, and mud sandwich, so it gets a score of $10 + 0 + 2 = 12$. Wow! That's a really, really great score!

It presents the sandwich to the panel of human judges. Harsh reality: it's not a popular sandwich.

Now comes the part where the neural net has a chance to improve: it looks at what would have happened if its weights were slightly different. From this one sandwich, it doesn't know what the problem is. Was it too excited about the marshmallow? Are eggshells not neutral but maybe even a teensy bit bad? It can't tell. But if it looks at a batch of ten sandwiches, the scores it gave them, and the scores the human judges gave them, it can

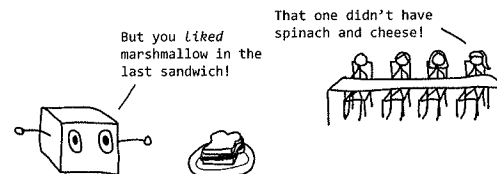
discover that if it had in general given mud a lower weight, lowering the score of any sandwich that contains mud, its scores would match those of the human judges a bit better.



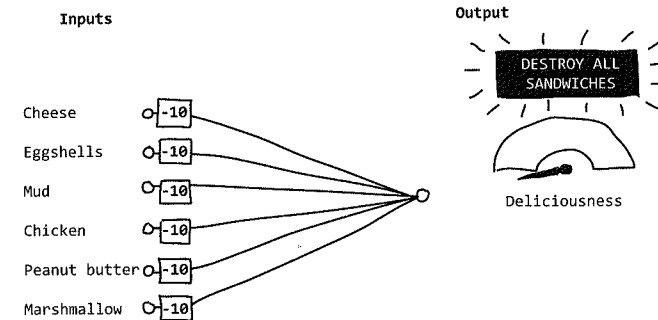
With its newly adjusted weights, it's time for another iteration. The neural net rates another bunch of sandwiches, compares its scores against those of the human judges, and adjusts its weights again. After thousands more iterations and tens of thousands of sandwiches, the human judges are very, very sick of this, but the neural network is doing a lot better.



There are plenty of pitfalls in the way of progress, though. As I mentioned above, this simple neural network only knows if particular ingredients are generally good or generally bad, and it isn't able to come up with a nuanced idea of which combinations work. For that, it needs a more sophisticated structure, one with hidden layers of cells. It needs to evolve punishers and deli sandwich cells.



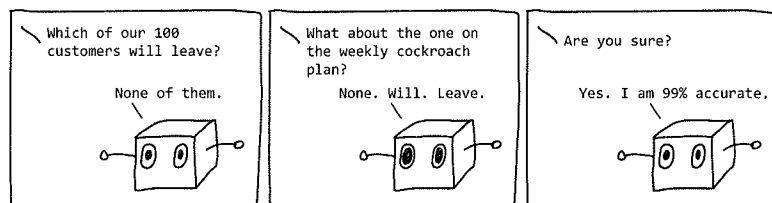
Another pitfall that we'll have to be careful of is the issue of **class imbalance**. Remember that only a handful of every thousand sandwiches from the sandwich hole are delicious. Rather than go through all the trouble of figuring out how to weight each ingredient, or how to use them in combination, the neural net may realize it can achieve 99.9 percent accuracy by rating each sandwich as terrible, no matter what.



To combat class imbalance, we'll need to prefilter our training sandwiches so that there are approximately equal proportions of sandwiches that are delicious and awful. Even then, the neural net might not learn about ingredients that are usually to be avoided but delicious in very specific circumstances. Marshmallow might be an example of such an ingredient—awful with most of the usual sandwich ingredients but delicious in a fluffernutter (and maybe with chocolate and bananas). If the neural net doesn't see fluffernutters in training, or sees them very rarely, it may decide that it can achieve pretty good accuracy by rejecting anything that contains marshmallow.

Class imbalance-related problems show up all the time in practical applications, usually when we ask AI to detect a rare event. When people try to predict when customers will leave a company, they have a lot

more examples of customers who stay than customers who leave, so there's a danger the AI will take the shortcut of deciding that all customers will stay forever. Detecting fraudulent logins and hacking attacks has a similar problem, since actual attacks are rare. People also report class imbalance problems in medical imaging, where they may be looking for just one abnormal cell among hundreds—the temptation is for the AI to shortcut its way to high accuracy just by predicting that all cells are healthy. Astronomers also run into class imbalance problems when they use AI, since many interesting celestial events are rare—there was a solar-flare-detecting program that discovered it could achieve near 100 percent accuracy by predicting zero solar flares, since these were very rare in the training data.²



WHEN CELLS WORK TOGETHER

In the sandwich-sorting example above, we saw how a layer of cells can increase the complexity of the tasks a neural network can perform. We built a deli sandwich cell that responded to combinations of deli meats and cheeses, and we built a cluckerfluffer cell that punished any sandwich that tried to use chicken and marshmallow in combination. But in a neural network that trains itself, using trial and error to adjust the connections between cells, it's usually a lot harder to identify each cell's job. Tasks tend

to be spread among several cells — and in the case of some cells, it's difficult or impossible to tell what tasks they accomplish.

To explore this phenomenon, let's look at some of the cells of a fully trained neural net. Built and trained by researchers at OpenAI,³ this particular neural net looked at more than eighty-two million Amazon product reviews letter by letter and tried to predict which letter would come next. This is another recurrent neural network, the same general sort as the one that generated the knock-knock jokes, ice cream flavors, and recipes listed in chapters 1 and 2. This one's larger — it has approximately as many neurons as a jelly fish. Here are a few examples of reviews it generated:

This is a great book that I would recommend to anyone who loves the great story of the characters and the series of books.

I love this song. I listen to it over and over again and never get tired of it. It is so addicting. I love it!!

This is the best product I have ever used to clean my shower stall. It is not greasy and does not strip the water of the water and stain the white carpet. I have been using it for a few years and it works well for me.

These workout DVDs are very useful. You can cover your whole butt with them.

I bought this thinking it would be good for the garage. Who has a lot of lake water? I was totally wrong. It was simple and fast. The night grizzly has not harmed it and we have had this for over 3 months. The guests are inspired and they really enjoy it. My dad loves it!

This particular neural net has an input for each letter or punctuation

mark it could encounter (similar to the sandwich sorter, which had one input for each sandwich ingredient) and can look back at the past few letters and punctuation marks. (It is as if the sandwich rater's scoring depended a bit on the last few sandwiches it had seen — maybe it can keep track of whether we might be sick of cheese sandwiches and adjust the next cheese sandwich's rating accordingly.) And rather than having a single output, as the sandwich sorter does, the review-writing neural net has a lot of them, one output for each letter or punctuation mark that it could choose as most likely to come next in the review. If it sees the sequence "I own twenty eggbeaters and this is my very favorit," then the letter *e* will be the most likely next choice.



Based on the outputs, we can take a look at each cell and see when it's "active," letting us make an educated guess about what its function is. In our sandwich-sorter example above, the deli sandwich cell would be active when it sees lots of meat and cheese and inactive when it sees socks or marbles or peanut butter. However, most of the neurons in the Amazon product-review neural net are going to be nowhere near as interpretable as deli cells and punisher neurons. Instead, most of the rules the neural net comes up with are going to be unintelligible to us. Sometimes we can guess what a cell's function will be, but far more frequently, we have no idea what it's doing.

Here's the activity of one of the product-review algorithm's cells (the 2,387th) as it generates a review (white = active, dark = inactive):

For me, this is one of the few albums of theirs I own
that actually made me an instant classic pop fan. I also

had a major problem with the audio with 10 new songs;
the execution of the vocals and editing was awful. The
next day, I was in a recording studio and I can't tell
you how many times I had to hit the play button to see
where the song was going.

This cell is contributing to the neural net's prediction of which letters come next, but its function is mysterious. It's reacting to certain letters, or certain combinations of letters, but not in a way that makes sense to us. Why was it really excited about the letters *um* in *album* but not the letters *al*? What is it actually doing? It's just one small piece of the puzzle working with a lot of other cells. Almost all the cells in a neural net are as mysterious as this one.

However, every once in a while, there will be a cell whose job is recognizable — a cell that activates whenever we're between a pair of parentheses or that activates increasingly strongly the longer a sentence gets.⁴ The people who trained the product-review neural net noticed that it had one cell that was doing something they could recognize: it was responding to whether the review was positive or negative. As part of its task of predicting the next letter in a review, the neural net seems to have decided it was important to determine whether to praise the product or trash it. Here's the activation of the "sentiment neuron" on that same review. Note that a light color indicates high activation, which means it thinks the review is positive:

For me, this is one of the few albums of theirs I own
that actually made me an instant classic pop fan. I also
had a major problem with the audio with 10 new songs;
the execution of the vocals and editing was awful. The
next day, I was in a recording studio and I can't tell
you how many times I had to hit the "play" button to see
where the song was going.

The review starts out very positive, and the sentiment neuron is highly activated. Midway through, however, it switches tone, and the cell's activation level goes way down.

Here's another example of the sentiment neuron at work. It has low activity when the review is neutral or critical but quickly swings into high gear whenever it detects a change in sentiment:

The Harry Potter File, from which the previous one was based (which means it has a standard size liner) weighs a ton and this one is huge! I will definitely put it on every toaster I have in the kitchen since, it is that good. This is one of the best comedy movies ever made. It is definitely my favorite movie of all time. I would recommend this to ANYONE!

But it's less good at detecting sentiment in other kinds of text. Most people would not classify this passage from Edgar Allan Poe's "The Fall of the House of Usher" as positive in sentiment, but this particular neural net thinks it's mostly positive:

Overpowered by an intense sentiment of horror, unaccountable yet unendurable, I threw on my clothes with haste (for I felt that I should sleep no more during the night,) and endeavoured to arouse myself from the pitiable condition into which I had fallen, by pacing rapidly to and fro through the apartment.

I guess a movie could overpower you by an intense sentiment of horror and be a good movie if that's what it was supposed to do.

Again, it's unusual to find a cell in a text-generating or text-analyzing algorithm that behaves as transparently as the sentiment neuron. The

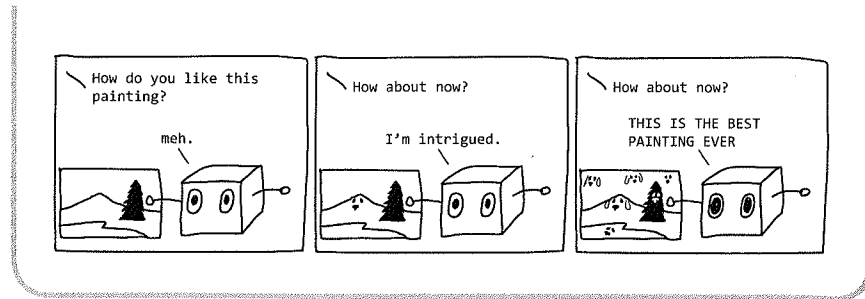
same goes for other types of neural networks — and that's too bad, since we'd love to be able to tell when they're making unfortunate mistakes and to learn from their strategies.

In image-recognizing algorithms, though, it's a bit easier to find cells whose jobs you can identify. There the inputs are the individual pixels of a particular image, and the outputs are the various possible ways to classify the image (dog, cat, giraffe, cockroach, and so on). Most image recognition algorithms have lots and lots of layers of cells in between — the hidden layers. And in most image recognition algorithms, there are cells or groups of cells whose functions we can identify if we analyze the neural net in the right way. We can look at the collections of cells that activate when they see particular things, or we can tweak the input image and see which changes make the cells activate most strongly.

DEEP DREAMING

Tweaking an image to make the neurons more excited about it is the technique used to make the famous Google DeepDream images where an image-identifying neural network turned ordinary images into landscapes full of trippy dog faces and fantastic conglomerations of arches and windows.

To make a DeepDream image, you start with a neural network that has been trained to recognize something — dogs, for example. Then you choose one of its cells and gradually change the image to make that cell increasingly more excited about it. If the cell is trained to recognize dog faces, then it will get more excited the more it sees areas in the image that look like dog faces. By the time you've changed the image to the cell's liking, it will be highly distorted and covered in dogs.



The smallest groups of cells seem to look for edges, colors, and very simple textures. They might report vertical lines, curves, or green grassy textures. In subsequent layers, larger groups of cells look for collections of edges, colors, and textures or for simple features. Some researchers at Google, for example, analyzed their GoogLeNET image recognition algorithm and found that it had several collections of cells that were looking specifically for floppy versus pointy ears on animals, which helped it distinguish dogs from cats.⁵ Other cells got excited about fur or eyeballs.

Image-generating neural networks also have some cells that do identifiable jobs. We can do “brain surgery” on image-generating neural networks, removing certain cells to see how the generated image changes.⁶ A group at MIT found that it could deactivate cells to remove elements from generated images. Interestingly, elements that the neural net deemed “essential” were more difficult to remove than others — for example, it was easier to remove curtains from an image of a conference room than to remove the tables and chairs.

Now let's look at another kind of algorithm, one you've probably interacted with directly if you've used the predictive-text feature of a smartphone.

MARKOV CHAINS

A **Markov chain** is an algorithm that can tackle many of the same problems as the recurrent neural network (RNN) that generated the recipes, ice cream flavors, Amazon reviews, and metal bands in this book. Like the RNN, it looks at what happened in the past (words previously used in a sentence or last week's weather, for example) and predicts what's most likely to happen next.

Markov chains are more lightweight than most neural networks and quicker to train. That's why the predictive-text function of smartphones is usually a Markov chain rather than an RNN.

However, a Markov chain gets exponentially more unwieldy as its memory increases. Most predictive-text Markov chains, for example, have memories that are only three to five words long. RNNs, by contrast, can have memories that are hundreds of words long — or even longer with the use of **LSTM** (long short-term memory) and convolution tricks. In chapter 2 we saw how important memory length is when short memory made an RNN lose track of important information. The same is true for Markov chains.

I trained a Markov chain with a dataset of Disney songs using a trainable predictive-text keyboard.⁷ Training took only a few seconds as opposed to a few minutes for an RNN. But this Markov chain has a three-word memory. That is, the words it suggests are the ones it thinks are the most likely based on the previous three words in the song. When I had it generate a song, choosing only its top suggestion at every step, here is what it produced:

The sea)
 under the sea)
 under the sea)
 under the sea)

under the sea)
 under the sea)
 under the sea)

It doesn't know how many times to sing "under the sea" because it doesn't know how many times it has already sung it.

If I start it out with the beginning of the song "Beauty and the Beast" ("Tale as Old as Time"), it quickly gets stuck again.

Tale as old as time
 song as old as time
 song as old as time
 song as old as time

In several verses of "Beauty and the Beast," the words "tale as old as time" are immediately followed by the words "song as old as rhyme." But when this Markov chain is looking at the phrase "as old as," it doesn't know which of those two verses it's in the middle of writing.

I can get it out of its trap by choosing the *second* most probable word at every step. Then it writes this:

A whole world
 bright young master
 you're with all
 ya think you're by wonder
 by the powers
 and i got downhearted
 alone hellfire dark side

And choosing the third most probable word each time:

You think i can open up
 where we'll see how you feel
 it all my dreams will be mine
 is something there before
 she will be better time

These are a lot more interesting, but they don't make much sense. And songs—and poetry—are pretty forgiving when it comes to grammar, structure, and coherence. If I give the Markov chain a different dataset to learn, then its shortcomings become even more obvious.

Here's a Markov chain trained on a list of April Fool's Day pranks as it chooses the most probable next word at each step. (It never suggested punctuation, so the line breaks are my additions.)

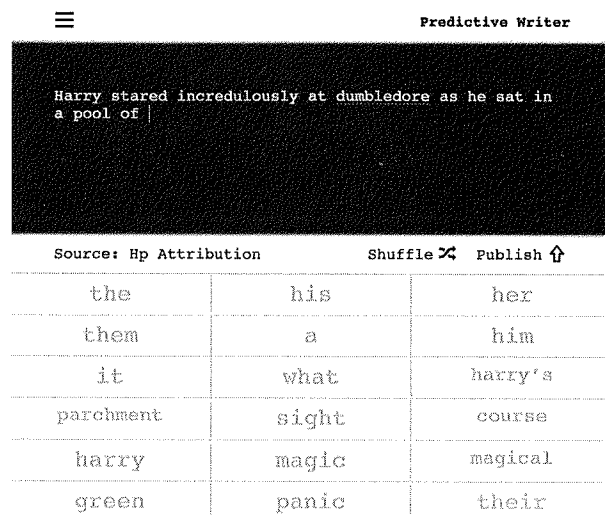
The door knob off a door and put it back on backwards
 softly
 Do nothing all day to a co of someone's ad in the paper
 for a garage sale at someone of an impending prank
 Then do nothing all day to a co of someone's ad in the
 paper for a garage sale at . . .

A predictive-text Markov chain isn't likely to hold a conversation with a customer or write a story that can be used as a new video-game quest (both of which are things that people are trying to get RNNs to do one day). But one thing it can do is suggest likely words that might come next in a particular training set.

The people at Botnik, for example, use Markov chains trained on various datasets (Harry Potter books, *Star Trek* episodes, Yelp reviews, and more) to suggest words to human writers. The unexpected Markov chain suggestions often help the writers take their texts in weirdly surreal directions.

Rather than allowing the Markov chain and its short memory to try to choose the next word, I can let it come up with a bunch of options and present them to me—just as predictive text does when I’m composing a text message to someone.

Here’s an example of what it looks like to interact with one of Botnik’s trained Markov chains, this one trained on Harry Potter books:



And here are some new April Fool’s Day pranks I wrote with the help of the predictive text of a trained Markov chain:

Put plastic wrap pellets on your lips.

Arrange the kitchen sink into a chicken head.

Put a glow stick in your hand and pretend to sneeze on the roof.

Make a toilet seat into pants and then ask your car to pee.

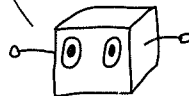
For the sake of comparison, I also used a more complex, data-intensive RNN to generate April Fool’s Day pranks. In this case, the RNN generated the entire prank, punctuation and all. However, there was still an element of human creativity involved—I had to sort through all the RNN-generated pranks looking for the funniest ones.

Make a food in the office computer of someone.
 Hide all of the entrance to your office building
 if it only has one entrance.
 Putting googly eyes on someone’s computer mouse
 so that it won’t work.
 Set out a bowl filled with a mix of M&M’s,
 Skittles, and Reese’s Pieces.
 Place a pair of pants and shoes in your ice
 dispenser.

You can conduct similar experiments with the predictive text included in most phone messaging apps. If you start with “I was born...” or “Once upon a time...” and keep clicking the phone’s suggested words, you’ll get a strange piece of writing straight from the innards of a machine learning algorithm. And because training a new Markov chain is relatively quick and easy, the text you get is specific to you. Your phone’s predictive text and autocorrect Markov chains update themselves as you type, training themselves on what you write. That’s why if you make a typo, it may haunt you for quite some time.

Google Docs may have fallen victim to a similar effect when users reported its autocorrect would change “a lot” to “alot” and suggested “gonna” instead of “going.” Google was using a context-aware autocorrect

Did you mean:
spgheitt sauce?



that scanned the internet to decide which suggestions to make.⁸ On the plus side, a context-aware autocorrect is able to spot typos that form real words (like “gong” typed instead of “going”), and add new words as soon as they become common. However, as any user of the internet knows, common usage rarely dovetails with the grammatically “correct” formal usage you’d want in a word processor’s autocorrect feature. Although Google hasn’t talked specifically about these autocorrect bugs, the bugs do tend to disappear after users report them.

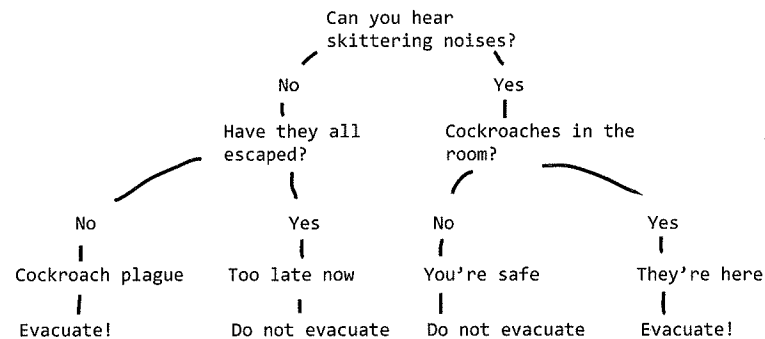
RANDOM FORESTS

A **random forest algorithm** is a type of machine learning algorithm frequently used for prediction and classification—predicting customer behavior, for example, or making book recommendations or judging the quality of a wine—based on a bunch of input data.

To understand the forest, let’s start with the trees. A random forest algorithm is made of individual units called decision trees. A **decision tree** is basically a flowchart that leads to an outcome based on the information we have. And, pleasingly, decision trees do kind of look like upside-down trees.

On the next page is a sample decision tree for, hypothetically, whether to evacuate a giant cockroach farm.

The decision tree keeps track of how we use information (ominous noises, the presence of cockroaches) to make decisions about how to handle the situation. Just as our sandwich decisions become more sophisti-



cated as the number of cells in our neural network increases, we can handle the cockroach situation with more nuance if we have a larger decision tree.

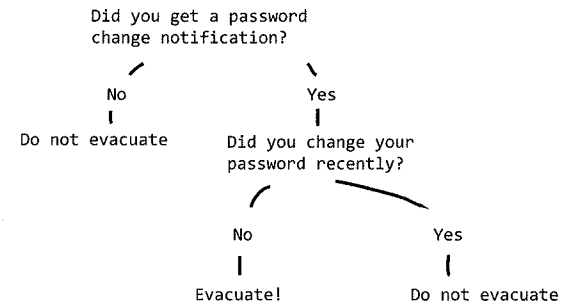
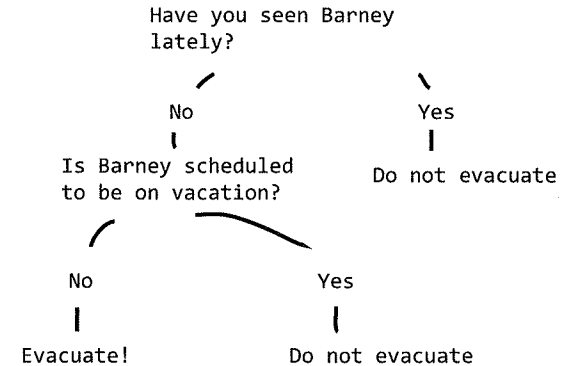
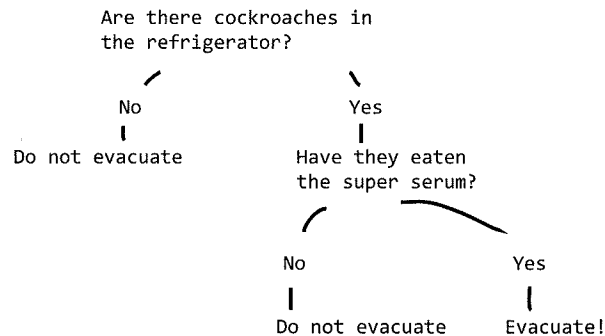
If the cockroach farm is strangely quiet, yet the roaches have not escaped, then there may be other explanations (perhaps even more unsettling) besides “they’re all dead.” With a larger tree we could ask whether there are dead cockroaches around, how smart the cockroaches are known to be, and whether the cockroach-crushing machines have been mysteriously sabotaged.

With lots and lots of inputs and choices, the decision tree can become hugely complex (or, to use the programming parlance of deep learning, very deep). It could become so deep that it encompasses every possible input, decision, and outcome in the training set, but then the chart would only work for the specific situations from the training set. That is, it would overfit the training data. A human expert could cleverly construct a huge decision tree that avoids overfitting and can handle most decisions without fixating on specific, probably irrelevant data. For example, if it was cloudy and cool the last time the cockroaches got out, a human is smart enough to know that having the same weather doesn’t necessarily have anything to do with whether the cockroaches will escape again.

But an alternative approach to having a human carefully build a huge

decision tree is to use the random forest method of machine learning. In much the same way as a neural network uses trial and error to configure the connections between its cells, a random forest algorithm uses trial and error to configure itself. A random forest is made of a bunch of tiny (that is, shallow) trees that each consider a tiny bit of information to make a couple of small decisions. During the training process, each shallow tree learns which information to pay attention to and what the outcome should be. Each tiny tree's decision probably won't be very good, because it's based on very limited information. But if all the tiny trees in the forest pool their decisions and vote on the final outcome, they will be much more accurate than any individual tree. (The same phenomenon holds true for human voters: if people try to guess how many marbles are in a jar, individually their guesses may be way off, but on average their guesses will likely be very close to the real answer.) The trees in a random forest can pool their decisions on all sorts of topics, coming up with an accurate picture of staggeringly complex scenarios. One recent application, for example, was sorting through hundreds of thousands of genomic patterns to determine which species of livestock was responsible for a dangerous *E. coli* outbreak.⁹

If we used a random forest to handle the cockroach situation, here's what a few of its trees might look like:

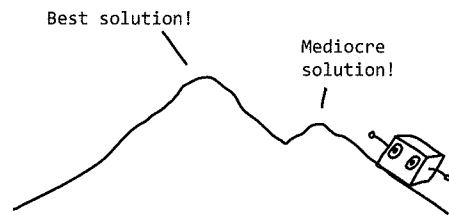


Now, each individual tree is only seeing a very small bit of the situation. There may be a perfectly reasonable explanation for why Barney isn't around — perhaps Barney has merely called in sick. And if the cockroaches have not actually eaten the super serum, that doesn't necessarily mean we're safe. Maybe the cockroaches have taken samples of the super serum and are even now brewing up a huge batch, enough for the 1.7 billion cockroaches in the facility.

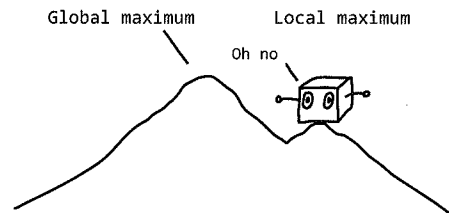
But the trees are combining their individual hunches, and with Barney mysteriously missing, the serum gone, and your password mysteriously changed, the decision to evacuate may be a prudent one.

EVOLUTIONARY ALGORITHMS

AI refines its understanding by making a guess about a good solution, then testing it. All three machine learning algorithms above use trial and error to refine their own structures, producing the configuration of neurons, chains, and trees that lets them best solve the problem. The simplest methods of trial and error are those in which you always travel in the direction of improvement — often called **hill climbing** if you're trying to maximize a number (say, the number of points collected during a game of Super Mario Bros.) or **gradient descent** if you're trying to minimize a number (like the number of escaped cockroaches). But this simple process of getting closer to your goal doesn't always yield the best results. To visualize the pitfalls of simple hill climbing, imagine you're somewhere on a mountain (in deep fog) and trying to find its highest point.

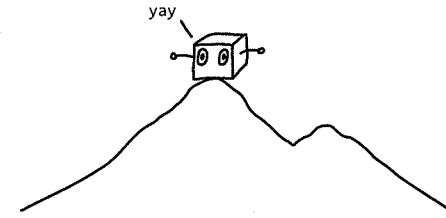


If you use a simple hill-climbing algorithm, you'll head uphill no matter what. But depending on where you start, you might end up stopping at the



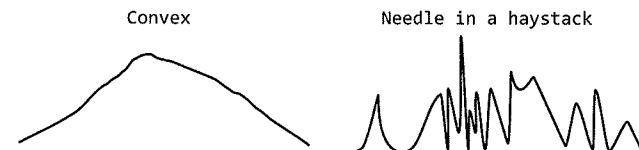
lowest peak — a **local maximum** — rather than the highest peak, the **global maximum**.

So there are more complex methods of trial and error designed to force you to try out more parts of the mountain, maybe doing a few test hikes in a few different directions before deciding where the most promising areas are. With those strategies, you might end up exploring the mountain more efficiently.



In machine learning terms, the mountain is called your **search space** — somewhere in that space is your goal (that is, somewhere on the mountain is the peak), and you're trying to find it. Some search spaces are **convex**, meaning that a basic hill-climbing algorithm will find you the peak each time. Other search spaces are much more annoying. The worst are the so-called **needle-in-the-haystack problems**, in which you might have very little clue how close you are to the best solution until the moment you stumble upon it. Searching for prime numbers is an example of a needle-in-the-haystack problem.

The search space of a machine learning algorithm could be anything. For example, the search space could be the shapes of parts that make up a



walking robot. Or it could be the set of possible weights of a neural network, and the “peak” is the weights that help you identify fingerprints or faces. Or the search space could be the set of possible configurations of a random forest algorithm, and your goal is to find a configuration that’s good at predicting a customer’s favorite books — or whether the cockroach factory should be evacuated.

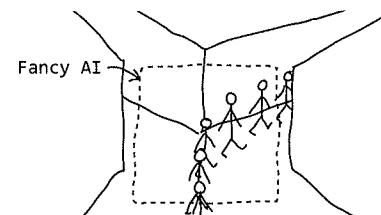
As we learned above, a basic search algorithm like hill climbing or gradient descent might not get you very far if the search space of possible neural net configurations is not very convex. So machine learning researchers sometimes turn to other, more complex trial-and-error methods.

One of these strategies takes its inspiration from the process of evolution. It makes a lot of sense to imitate evolution — after all, what is evolution if not a generational process of “guess and check”? If a creature is different from its neighbors in some way that makes it more likely to survive and therefore reproduce, then it will be able to pass its useful traits on to the next generation. A fish that can swim a tiny bit faster than other individuals of its species may be more likely to escape predators, and after a few generations of this, its fast-swimming offspring may be a bit more common than the descendants of slower-swimming fish. And evolution is a powerful, powerful process — one that has solved countless locomotion and information-processing problems, figured out how to extract food from sunlight and from hydrothermal vents, and figured out how to glow, fly, and hide from predators by looking like bird dung.

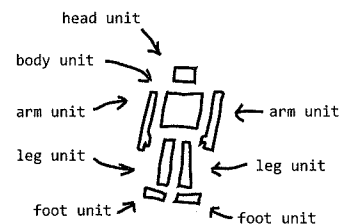
In **evolutionary algorithms**, each potential solution is like an organism. In each generation, the most successful solutions survive to reproduce, mutating or mating with other solutions to produce different — and, one hopes, better — children.

If you’ve ever struggled to solve a complex problem, it might be mind-boggling to think of each potential solution as a living being — eating, mating, whatever. But let’s think about it in concrete terms. Let’s say we’re trying to solve a crowd-control problem: we have a hallway that splits into a

fork, and we want to design a robot that can direct people to take one hallway or the other.



The first thing we do is come up with the bits that the evolutionary algorithm can vary, deciding what about our robot we want to be constant and what the algorithm is free to play with. We could make these variable elements very limited, with a fixed body design, and just allow the program to change the way the robot moves around. Or we could allow the algorithm to build a body design completely from scratch, starting from random blobs. Let’s say that the owners of this building are insisting on a humanlike robot design for sci-fi-aesthetic reasons. No messy jumble of crawling blocks (which is what an evolutionary algorithm’s creatures tend to look like, given absolute freedom). Within a basic humanlike form, there’s still a lot we could vary, but let’s keep it simple and say that the algorithm will be allowed to vary the size and shape of a few basic body parts, with each one having a simple range of motion. In evolutionary terms, this is the robot’s **genome**.



Robot Genome

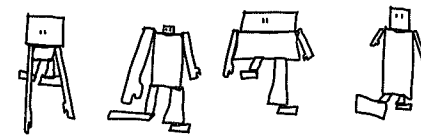
Body part dimensions:
 Head unit: length, width, height
 Body unit: length, width, height
 ...
Behaviors:
 Default behavior
 When human present
 When human moves left
 When human moves right
 ...

The next thing we need to do is define the problem we're trying to solve in such a way that there's a single number we can optimize. In evolutionary terms, this number is the **fitness function**: a single number that will describe how fit an individual robot is for our task. Since we're trying to build a robot that can direct humans down one hallway or the other, let's say that we're trying to minimize the number of humans that take the left-hand fork. The closer that number is to zero, the higher the fitness.

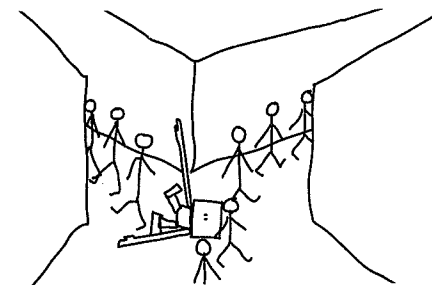
We'll also need a simulation, because there's no way we're building thousands of robots to order or hiring people to walk down a hall thousands of times. (Not using real humans is also a safety consideration — for reasons that will be clear later.) So let's say it's a simulated hall in a world with simulated gravity and friction and other simulated physics. And of course we need simulated people with simulated behaviors, including walking, lines of sight, crowding, and various phobias, motivations, and levels of cooperativeness. The simulation itself is a really hard problem, so let's just say we've solved it already. (Note: in actual machine learning, it's never this easy.)

One handy way of getting a ready-made simulation that can train an AI is to use video games. That's partly why there are so many researchers training AIs to play Super Mario Bros. or old Atari games—these old video games are small, quick-to-run programs that can test various problem-solving skills. Just like human video-game players, though, AIs tend to find and exploit bugs in the games. More about this in chapter 5.

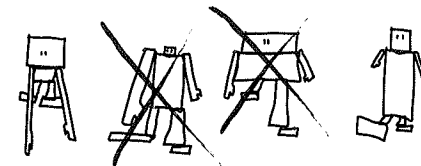
We let the algorithm randomly create our first generation of robots. They're... very random. A typical generation produces hundreds of robots, each with a different body design.



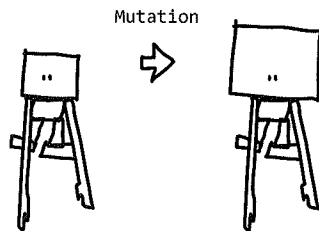
Then we test each robot individually in our simulated hallway. They don't do well. People walk right past them as they flop on the ground and flail ineffectually. Maybe one of them falls a bit more to the left than the others and blocks that hallway slightly, and a few of the more timid humans decide to take the right hallway instead. It scores slightly better than the other robots.



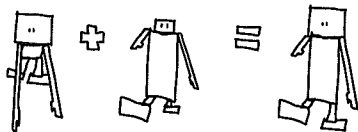
Now it's time to build the next generation of robots. First, we'll choose which robots are going to survive to reproduce. We could save just the very best robot, but that would make the population pretty uniform, and we wouldn't get to try out some other robot designs that might end up being better if evolution gets a chance to tweak them. So we'll save some of the best robots and throw out the rest.



Next, we have lots of choices about how the surviving robots are going to reproduce. They can't simply make identical copies of themselves, because we want them to be evolving toward something better. One option we have is **mutation**: pick a random robot and randomly vary something about it.



Another option we might decide to use is **crossover**: two robots produce offspring that are random combinations of the two parents.

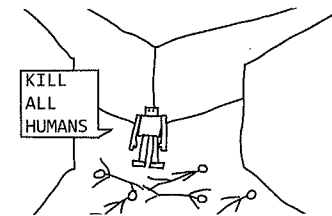


We also have to decide how many offspring each robot can have (should the most successful robots have the most offspring?), which robots can cross with which other robots (or if we use crossover at all), and whether we're going to replace all the dead robots with offspring or with a few randomly generated robots. Tweaking all these options is a big part of building an evolutionary algorithm, and sometimes it's hard to guess which options — which **hyperparameters** — are going to work best.

Once we've built the new generation of robots, the cycle begins again as we test their crowd-controlling abilities in the simulation. More of them are now flopping over to the left because they're descended from that first marginally successful robot.

After many more generations of robots, some distinct crowd-control strategies start to emerge. Once the robots learn to stand up, the original “fall to the left and be kinda in the way” strategy has evolved into a “stand in the left hallway and be even more annoying” strategy. Another strategy also emerges — the “point vigorously to the right” strategy. But none of the strategies is perfectly solving our problem yet: each robot is still letting plenty of people leak into the left hallway.

After many more generations, a robot emerges that is very good at preventing people from entering the left hallway. Unfortunately, by a stroke of bad luck, it just so happens that the solution it found was “murder everyone.” Technically that solution works because all we told it to do was minimize the number of people entering the left hallway.



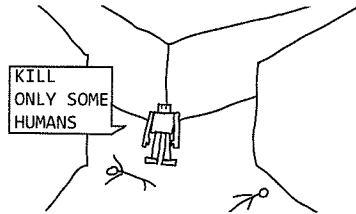
Because of a problem with our fitness function, evolution directed the algorithm toward a solution that we hadn't anticipated. Unfortunate shortcuts happen in machine learning all the time, although not usually this dramatically. (Fortunately for us, in real life, “kill all humans” is usually very impractical. Don't give autonomous algorithms deadly weapons is the message here.) Still, *this* is why we used simulated humans rather than real humans in our thought experiment.

We'll have to start over again, this time with a fitness function that, rather than minimizing the number of humans in the left-hand hallway, maximizes the number of humans who take the right-hand hallway.

Actually, we can take a (somewhat gory) shortcut and just change the fitness function rather than completely starting over. After all, our robots

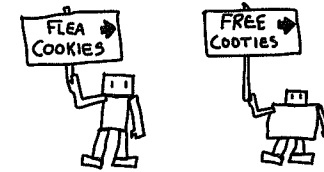
have learned many useful skills besides murdering people. They've learned to stand, detect people, and move their arms in a scary manner. Once our fitness function changes to maximizing the number of survivors who enter the right-hand hallway, the robots *should* quickly learn to forsake their murdering ways. (Recall that this strategy of reusing a solution from a different but related problem is called transfer learning.)

So we start with the group of murdering robots and sneakily swap the fitness function on them. Suddenly, murdering isn't working very well at all, and they don't know why. In fact, the robot that was the worst at murdering is now at the top of the heap, because some of its screaming victims managed to escape down the right-hand hallway. Over the next few generations, the robots quickly become ever worse at murdering.

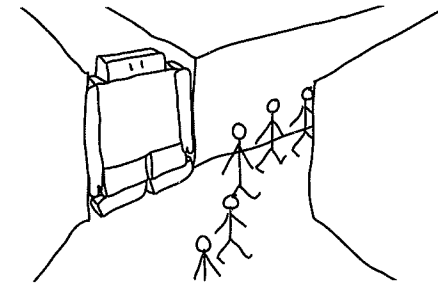


Eventually, maybe they only look like they might *want* to murder you, which would scare most humans into entering the right-hand hallway. By starting with a population of murderbots, we do restrict the path that evolution is likely to take. Had we started over instead, we might have evolved robots that stood at the end of the right-hand hallway and beckoned people or even robots whose hands evolved into signs that said *FREE COOKIES*. (The “free cookies” robot would be hard to evolve, though, because getting the sign merely partially right wouldn't work at all, and it would be hard to reward a solution that was only getting close. In other words, it's a needle-in-the-haystack solution.)

All murderbots aside, the most likely path that evolution would have taken is the “fall down and be in the way” robot getting ever more annoy-



ingly in the way. (Falling down is pretty easy to do, so if an evolved robot can solve a problem by falling down, it will tend to do that.) Through that path we may arrive at a robot that solves the problem perfectly by causing 100 percent of humans to enter the right-hand hallway (murdering none of them in the process). The robot looks like this:



Yes, we have evolved: a door.

That's the other thing about AI. It can sometimes be a needlessly complicated substitute for a commonsense understanding of the problem.

Evolutionary algorithms are used to evolve all kinds of designs, not just robots. Car bumpers that dissipate force when they crumple, proteins that bind to other medically useful proteins, flywheels that spin just so — these are all problems that people have used evolutionary algorithms to solve. The algorithm doesn't have to stick to a genome that describes a physical object, either. We could have a car or bicycle with a fixed design and a control program that evolves. I mentioned earlier that the genome can even be the weights of a neural network or the arrangement of a decision tree.

Different kinds of machine learning algorithms are often combined like this, each playing to its strength.

When we consider the huge array of life that has arisen on our planet via evolution, we get an idea of the magnitude of possibility that's available to us by using virtual evolution at a massively accelerated speed. Just as real-life evolution has managed to produce wonderfully complex creatures and allow them to take advantage of the weirdest, most specific food sources, evolutionary algorithms continue to surprise and delight us with their ingenuity. Of course, sometimes evolutionary algorithms can be a little *too* creative — as we'll see in chapter 5.

GENERATIVE ADVERSARIAL NETWORKS (GANs)

AI can do amazing things with images, turning a summer scene into a winter one, generating faces of imaginary people, or changing a photo of someone's cat into a cubist painting. These showy image-generating, image-remixing, and image-filtering tools are usually the work of **GANs (generative adversarial networks)**. They're a subvariety of neural networks, but they deserve their own mention. Unlike the other kinds of machine learning in this chapter, GANs haven't been around very long — they were only introduced by Ian Goodfellow and other Université de Montréal researchers in 2014.¹⁰

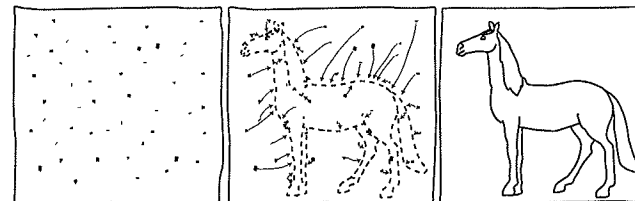
The key thing about GANs is they're really two algorithms in one — two adversaries that learn by testing each other. One, the **generator**, tries to imitate the input dataset. The other, the **discriminator**, tries to tell the difference between the generator's imitation and the real thing.

To see why this is a helpful way of training an image generator, let's go through a hypothetical example. Suppose we want to train a GAN to generate images of horses.

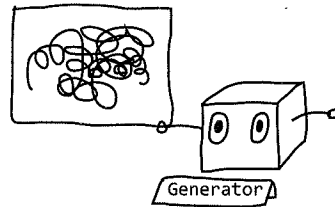
The first thing we'll need is lots of example pictures of horses. If they all

show the same horse in the same pose (maybe we're obsessed with that particular horse), the GAN will learn more quickly than if we give it a huge variety of colors and angles and lighting conditions. We can also simplify things by using a plain, consistent background. Otherwise the GAN will spend a long time trying to learn when and how to draw fences, grass, and parades. Most of the GANs that can generate photorealistic faces, flowers, and foods were given very limited, consistent datasets — pictures of just cat faces, for example, or bowls of ramen photographed only from the top. A GAN trained just on photos of tulip heads may produce very convincing tulips but will have no idea about other kinds of flowers or even any concept that tulips have leaves or bulbs. A GAN that can generate photorealistic human head shots won't know what's below the neck, what's on the back of the head, or even that human eyes can close. So this is all to say that if we're going to make a horse-generating GAN, we'll have better success if we make its world a very simple one and only give it pictures of horses photographed from the side against a plain white background. (Conveniently, this is also about the extent of my drawing ability.)

Now that we have our dataset (or, in our case, now that we've imagined one), we're ready to start training the two parts of the GAN, the generator and the discriminator. We want the generator to look at our set of horse pictures and figure out some rules that will let it make pictures similar to them. Technically what we are asking the generator to do is warp random noise into pictures of horses — that way, we can get it to generate not just one single horse picture but also a different horse for every random noise pattern.

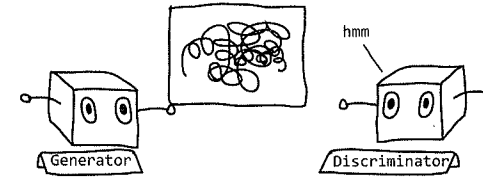


At the beginning of the training, though, the generator hasn't learned any rules about drawing horses. It starts with our random noise and does something random to it. As far as it knows, that is how you draw a horse.

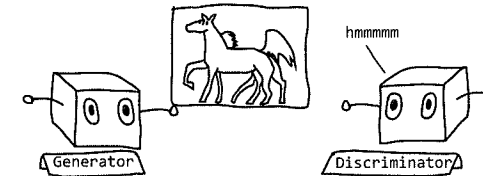
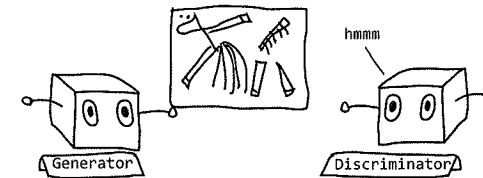


How can we give the generator useful feedback on its terrible drawings? Since this is an algorithm, it needs feedback in the form of a number, some kind of quantitative rating that the generator can work on improving. One useful metric would be the percentage of instances in which it makes a drawing that's so good that it looks just like a real horse. A human could easily judge this—we're pretty good at telling the difference between a smear of fur and a horse. But the training process is going to require many thousands of drawings, so it's impractical to have a human judge rate them all. And a human judge would be too harsh at this stage—they would look at two of the generator's scribbles and rate them both as "not horse," even if one of them is actually ever so imperceptibly more horselike than the other. If we give the generator feedback on how often it manages to fool a human into thinking one of its drawings is real, then it will never know if it's making progress because it will never fool the human.

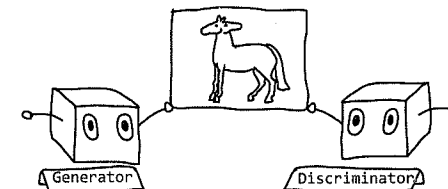
This is where the discriminator comes in. The discriminator's job is to look at the drawings and decide if they're real horses from the training set. At the beginning of training, the discriminator is just about as awful at its job as the generator is: it can barely tell the difference between the generator's scribbles and the real thing. The generator's almost imperceptibly horselike scribbles might actually succeed in fooling the discriminator.



Through trial and error, both the generator and the discriminator get better.



The GAN is, in a way, using its generator and discriminator to perform a Turing test in which it is both judge and contestant. The hope is that by the time training is over, it's generating horses that would fool a human judge as well.



Sometimes people will design GANs that don't try to match the input dataset exactly but instead try to make something "similar but different." For example, some researchers designed a GAN to produce abstract art, but they wanted art that wasn't a boring knockoff of the art in the training data. They set up the discriminator to judge whether the art was like the training data yet not identifiable as belonging to any particular category. With these two somewhat contradictory goals, the GAN managed to straddle the line between conformity and innovation.¹¹ And consequently, its images were popular — human judges even rated the GAN's images more highly than human-painted images.

MIXING, MATCHING, AND WORKING TOGETHER

We learned that GANs work by combining two algorithms — one that generates images and one that classifies images — to reach a goal.

In fact, a lot of AIs are made of combinations of other, more specialized machine learning algorithms.

Microsoft's Seeing AI app, for example, is designed for people with vision impairments. Depending on which "channel" a user selects, the app can do things like

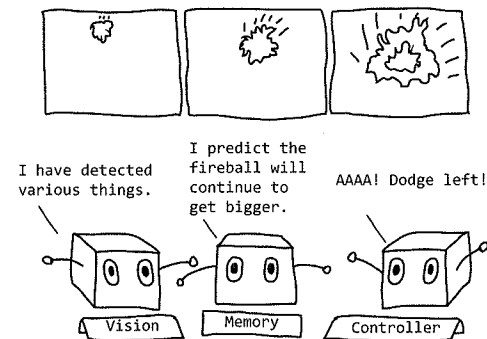
- recognize what's in a scene and describe it aloud,
- read text held up to a smartphone's camera,
- read denominations of currency,
- identify people and their emotions, and
- locate and scan bar codes.

Each one of these functions—including its crucial text-to-speech function — is likely powered by an individually trained AI.

Artist Gregory Chatonsky used three machine learning algorithms to

generate paintings for a project called *It's Not Really You*.¹² One algorithm was trained to generate abstract art, and another algorithm's job was to transform the first algorithm's artwork into various painterly styles. Finally, the artist used an image recognition algorithm to give the images titles such as *Colorful Salad*, *Train Cake*, and *Pizza Sitting on a Rock*. The final artwork was a multialgorithm collaboration planned and orchestrated by the artist.

Sometimes the algorithms are even more tightly integrated, using multiple functions at once without human intervention. For example, researchers David Ha and Jürgen Schmidhuber used evolution to train an algorithm inspired by the human brain to play one level of the computer game Doom.¹³ The algorithm consisted of three algorithms working together. A vision model was in charge of perceiving what was going on in the game — were there fireballs in view? Were there walls nearby? It transformed the 2-D image of pixels into the features it had decided were important to keep track of. The second model, a memory model, was in charge of trying to predict what would happen next. Just as the text-generating RNNs in this book look at past history to predict what letter or word is likely to come next, the memory model was an RNN that looked at previous moments in the game and tried to predict what would happen next. If there had been a fireball moving to the left a few moments earlier, it's probably going to still be there in the next image, just a bit farther to the left. If the fireball had



been getting bigger, it's probably going to continue to get bigger (or it may hit the player and cause a huge explosion). Finally, the third algorithm was the controller, whose job was to decide what actions to take. Should it dodge to the left to avoid being hit by the fireball? Maybe that would be a good idea.

So the three parts worked together to see fireballs, realize they were approaching, and dodge out of the way. The researchers chose each subalgorithm's form so that it would be optimized for its specific task. This makes sense, since we learned in chapter 2 that machine learning algorithms do best when they have a very narrow task to work on. Choosing the correct form for a machine learning algorithm, or breaking a problem into tasks for subalgorithms, is a key way programmers can design for success.

In the next chapter, we'll look at more ways that AIs can be designed for success — or not.