Presented at SIGGRAPH 2018 in Vancouver, BC on August 12, 2018. The notes here are just breadcrumbs. You can see the talk itself on YouTube at https://www.youtube.com/watch?v=r0Ogt-q956I . These slides have been slightly updated since the course. Note that the images here were saved at low resolution to keep the file size reasonable.

Photography & Recording Permitted

Almost all of these slides are available in high-res format, for free use in your own talks, classes, and other presentations. Go to GitHub under my userid, blueberrymusic. Look in the repos of the figures for my Deep Learning book. This Keynote file is also at GitHub.

Concepts, terminology, structures, no math, no code. Free open-source libraries do the hard work

This course is based on my books, “Deep Learning from Basics to Practice,” available in digital form at http://amzn.to/2F4nz7k and http://amzn.to/2EQtPR2

Goals

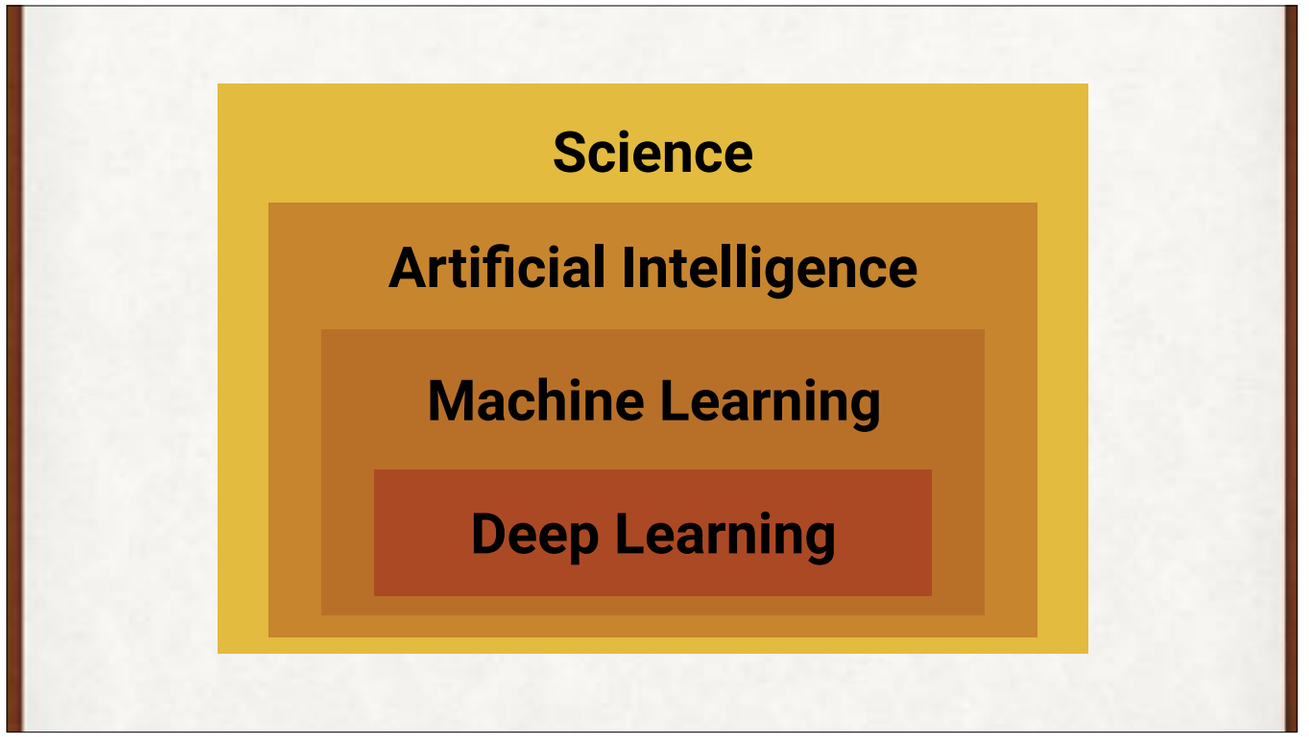
concepts

terminology

algorithms

applications

Our goals for the course.



Where DL sits in the overall scheme of things (homage to Josef Albers). AI : Chess, Go, chatbots: moving goalposts or pulling question into focus? Discuss over dinner.

Intelligence

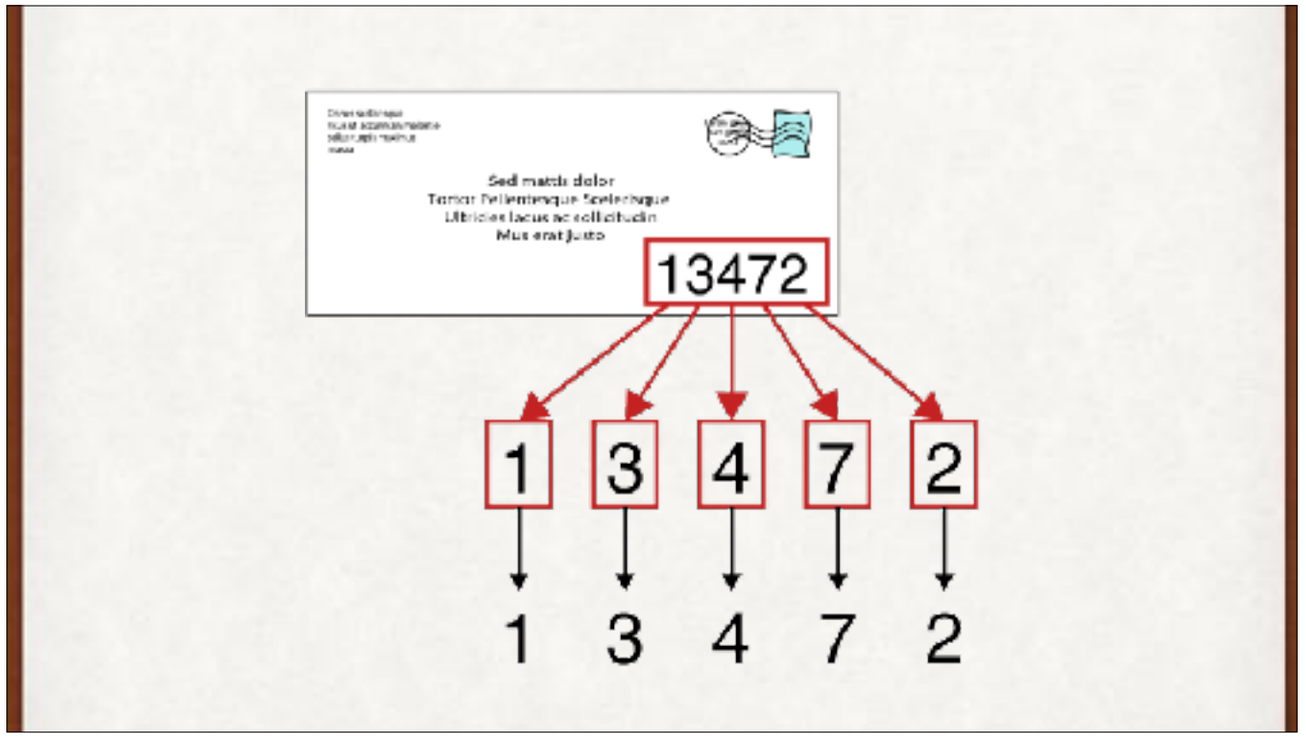
reason

learn

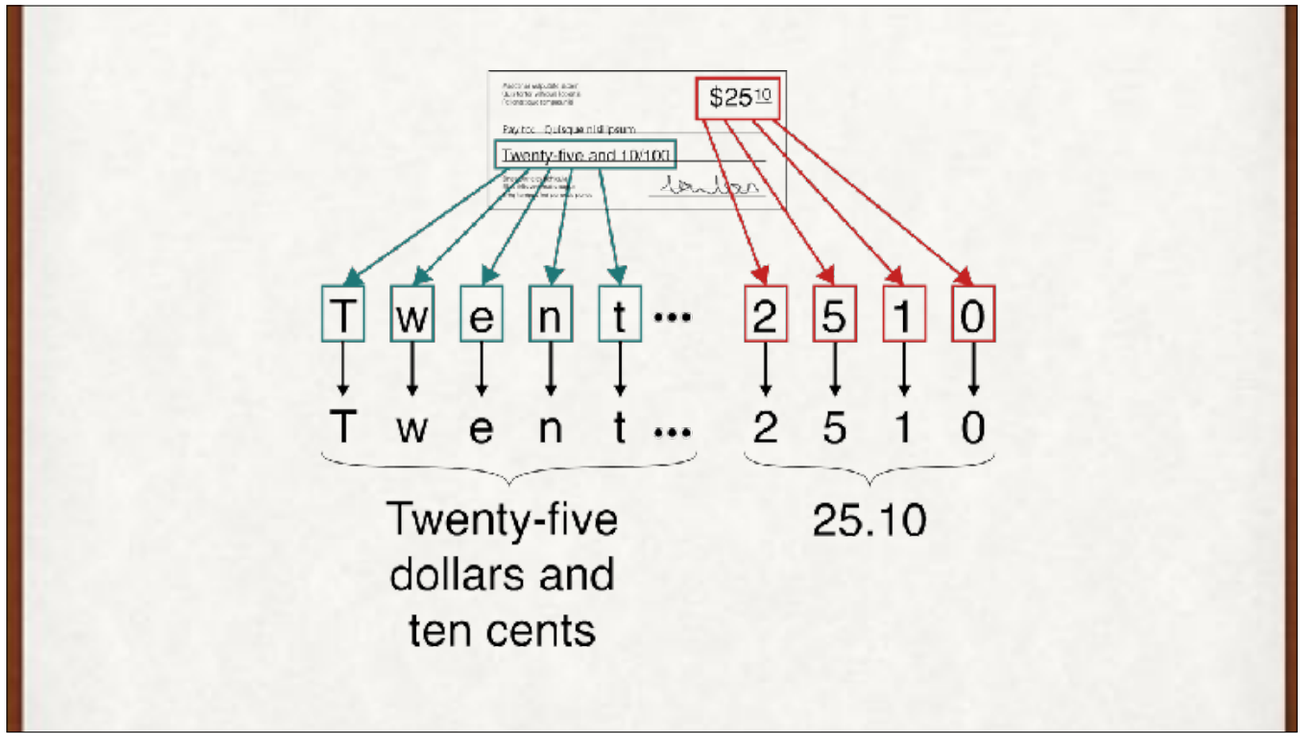
plan

solve problems

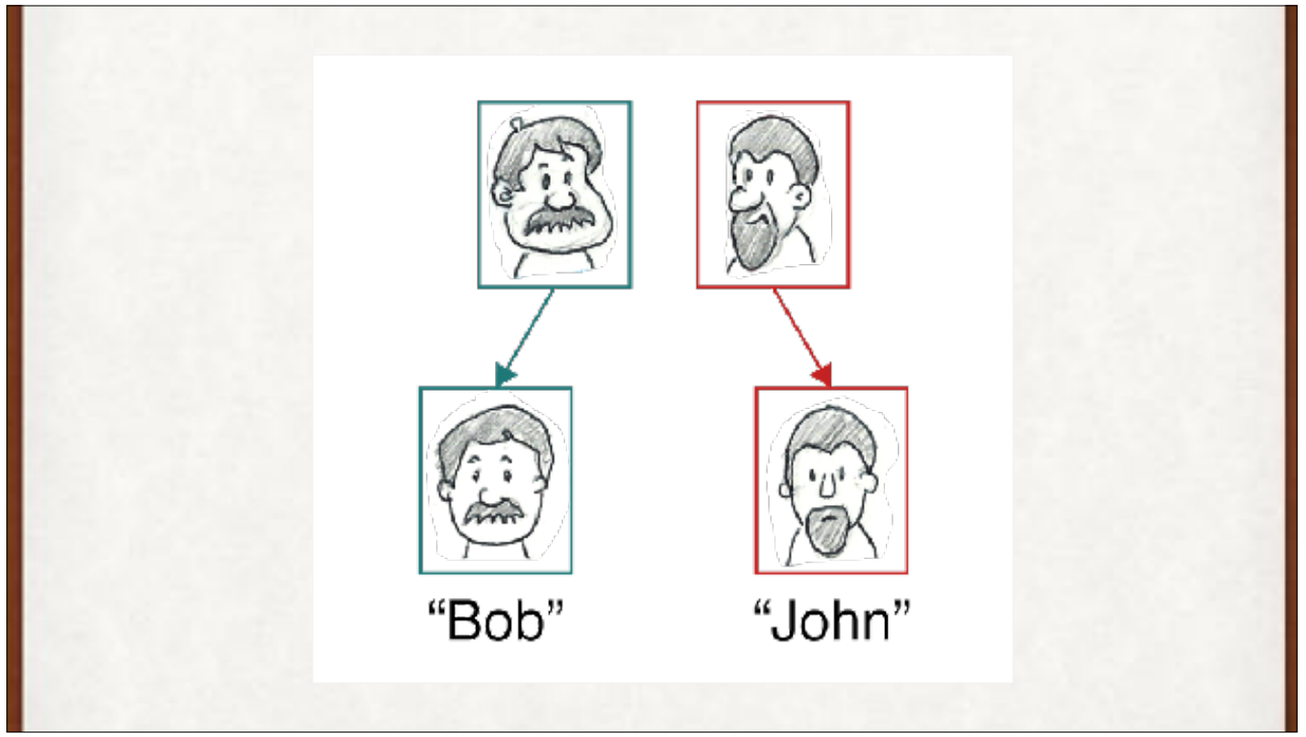
Roughly, some ideas that are characteristics of intelligence



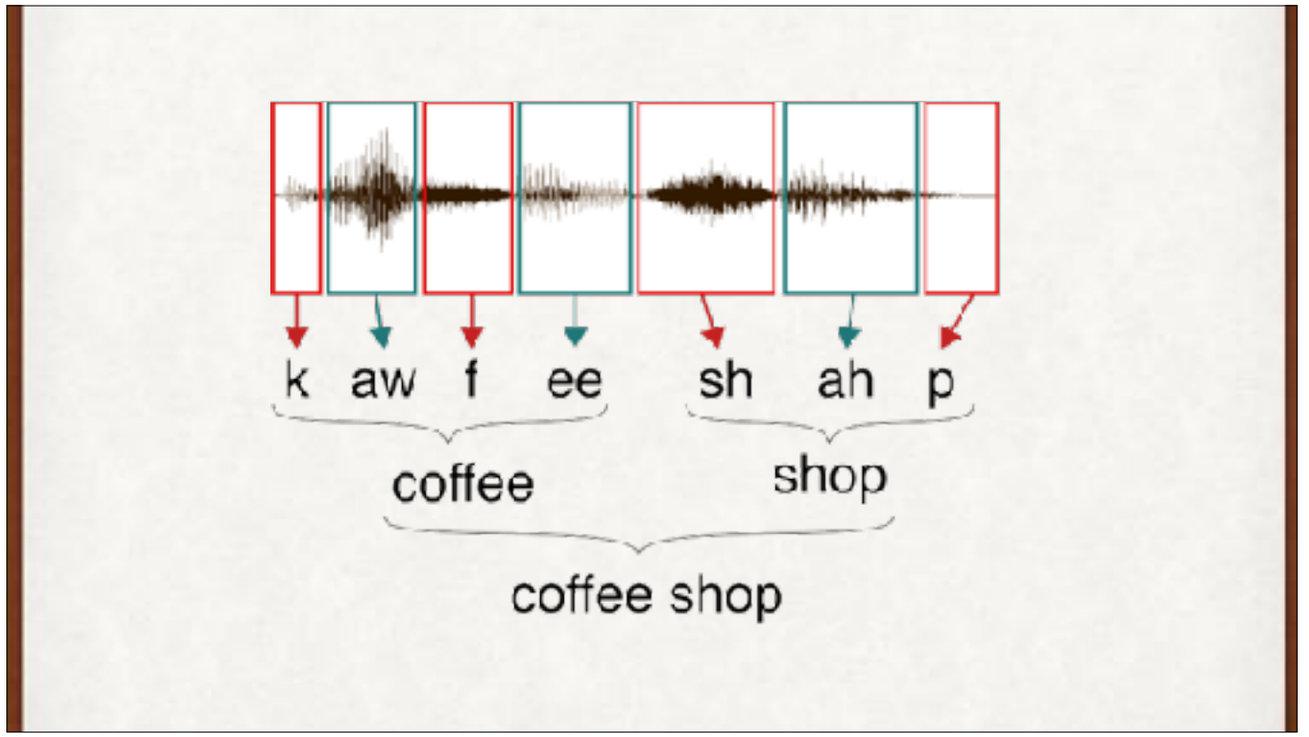
Some ML tasks. Reading digits and letters, recognizing faces.



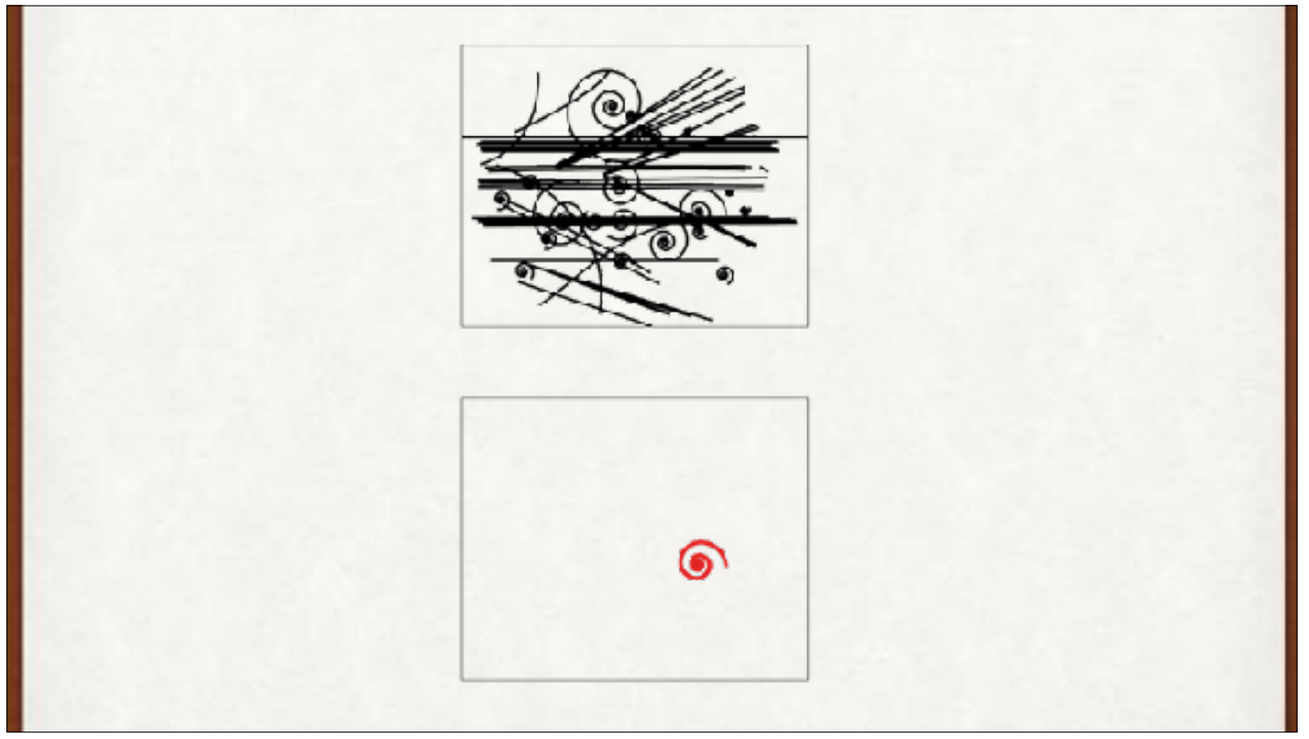
Some ML tasks. Reading digits and letters, recognizing faces.



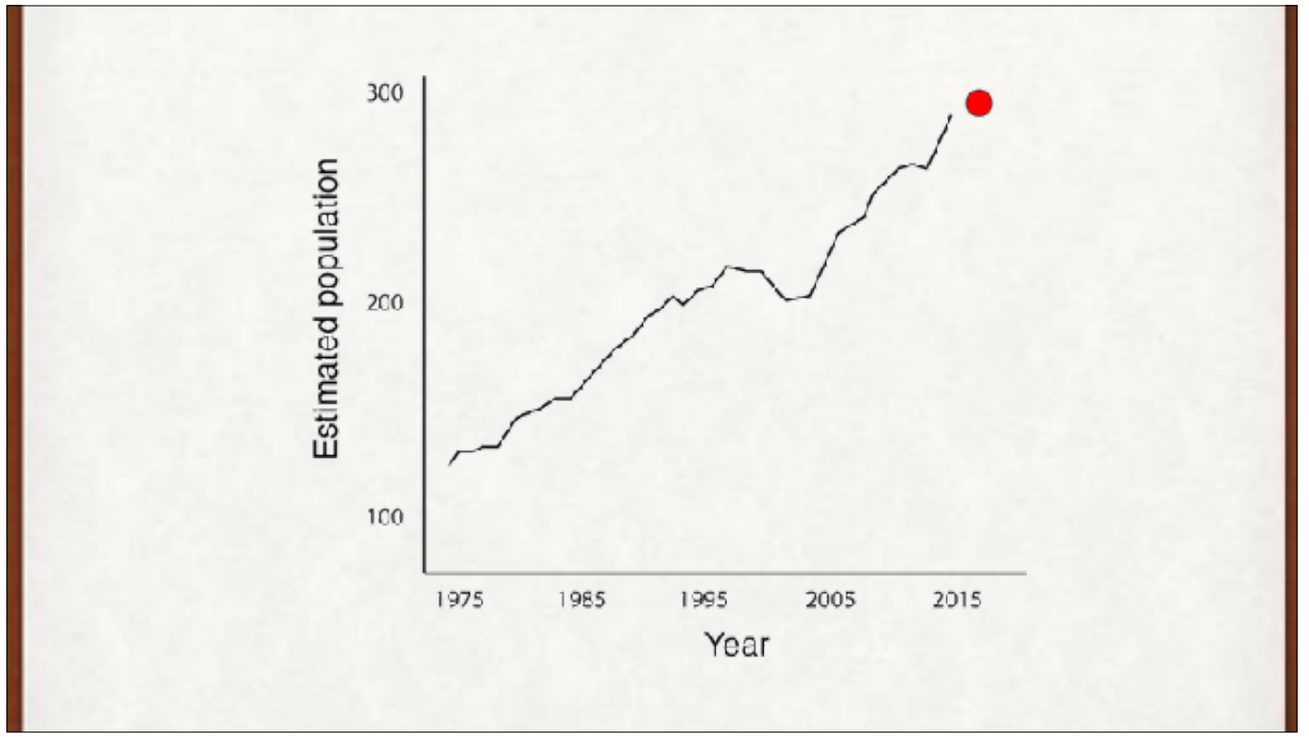
Some ML tasks. Reading digits and letters, recognizing faces.



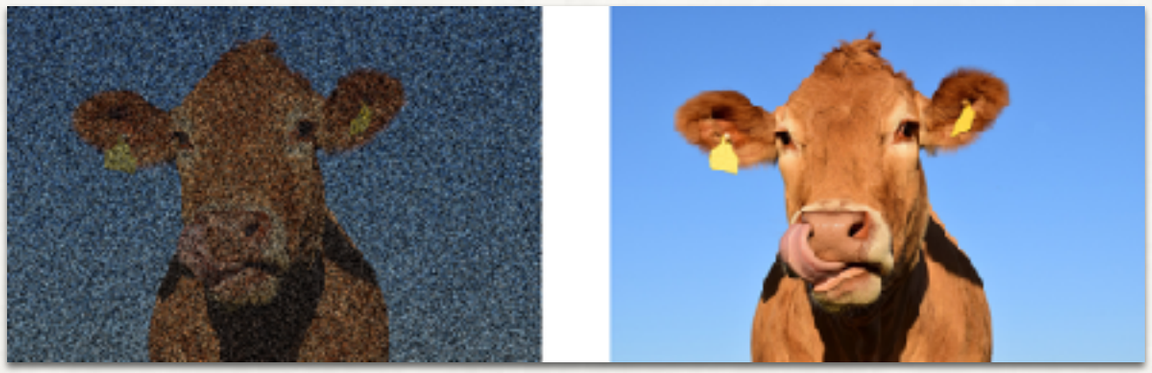
More ML tasks. Speech to words, finding a needle in a haystack, predicting missing data.



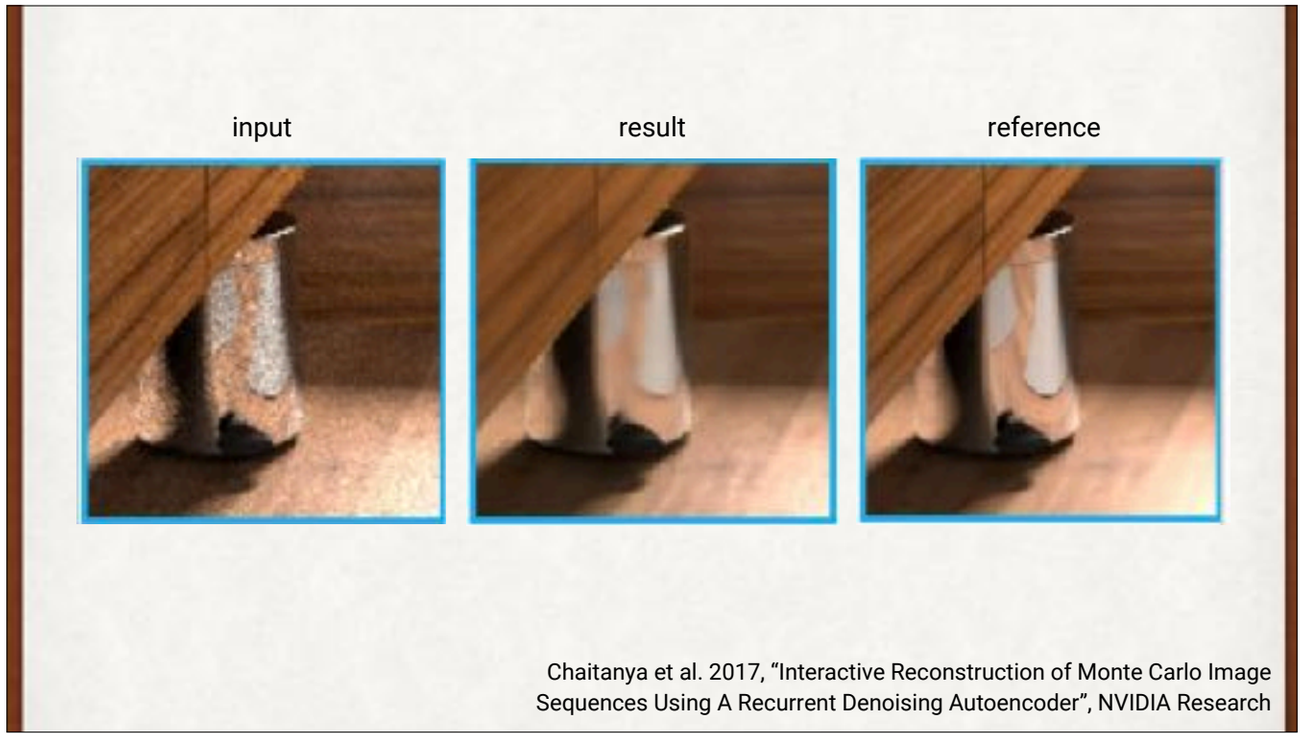
More ML tasks. Speech to words, finding a needle in a haystack, predicting missing data.



regression to mediocrity -> regression to the mean



Applications: Denoising, from the left to the right



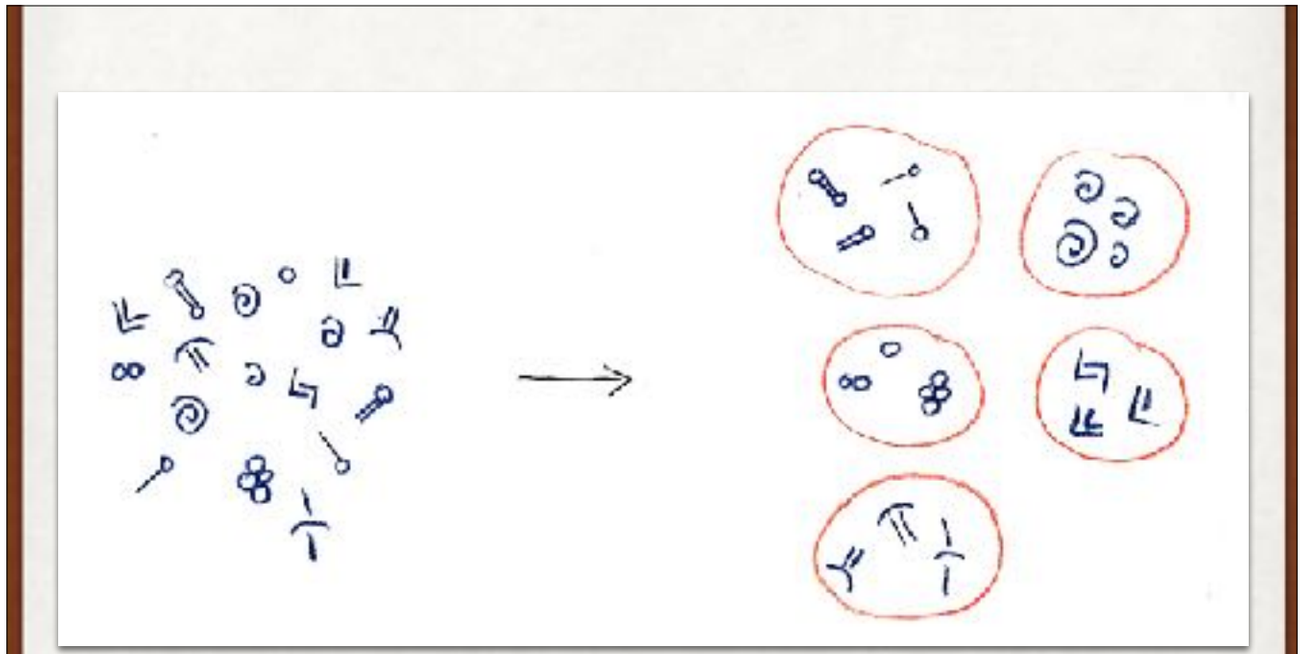
Chaitanya et al. 2017, “Interactive Reconstruction of Monte Carlo Image Sequences Using A Recurrent Denoising Autoencoder”, NVIDIA Research

Close-up reconstructions from the autoencoder.

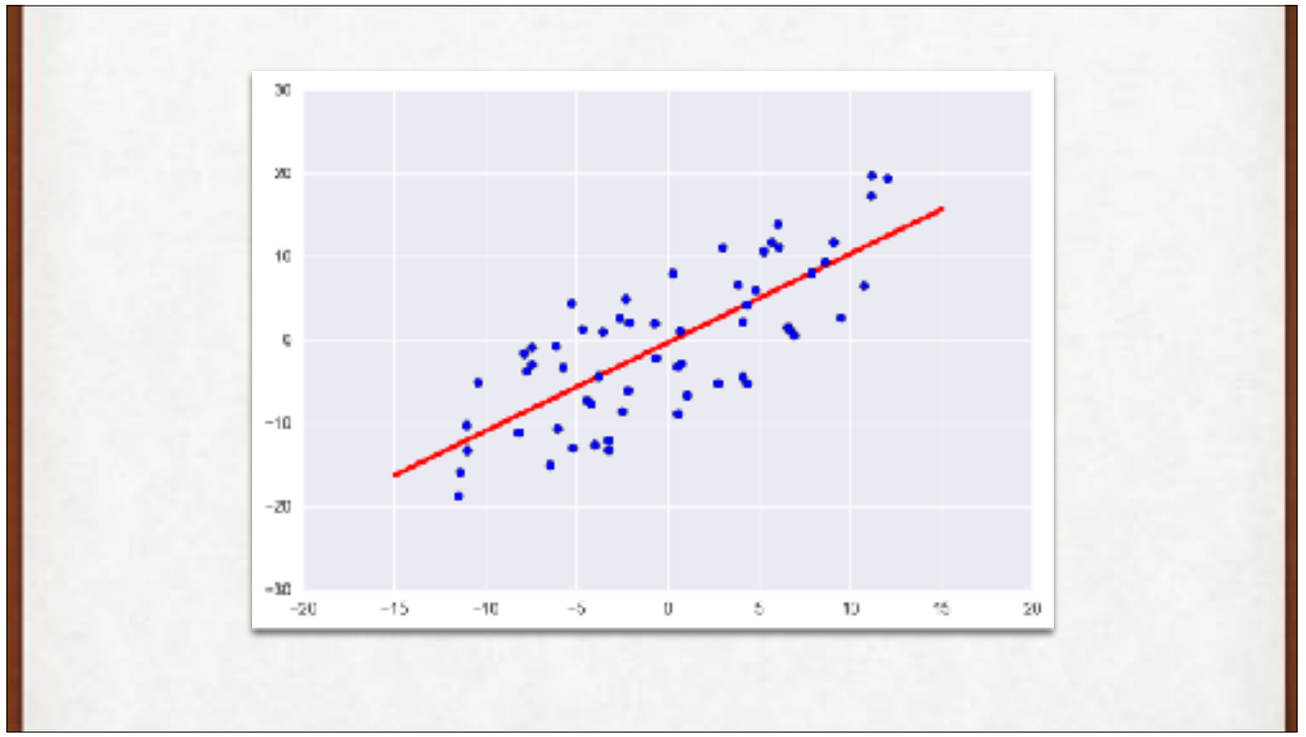
UNSUPERVISED LEARNING

Unsupervised learning is when we have a bunch of data without labels. Usually we want to group or cluster them somehow or figure out something about their relationships.

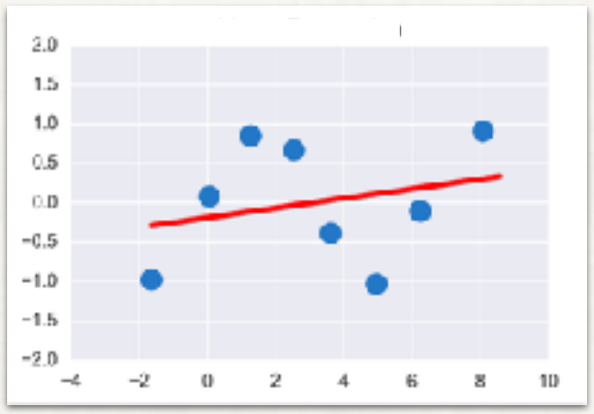
Discovered pottery markings. Cluster them into similar groups, without any help from us.



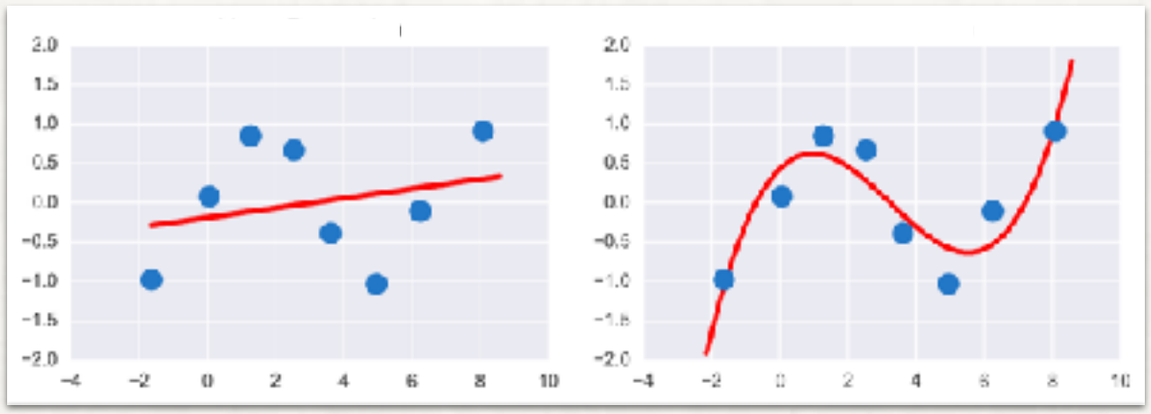
Discovered pottery markings. Cluster them into similar groups, without any help from us.



Linear regression: find the best line that approximates the data.



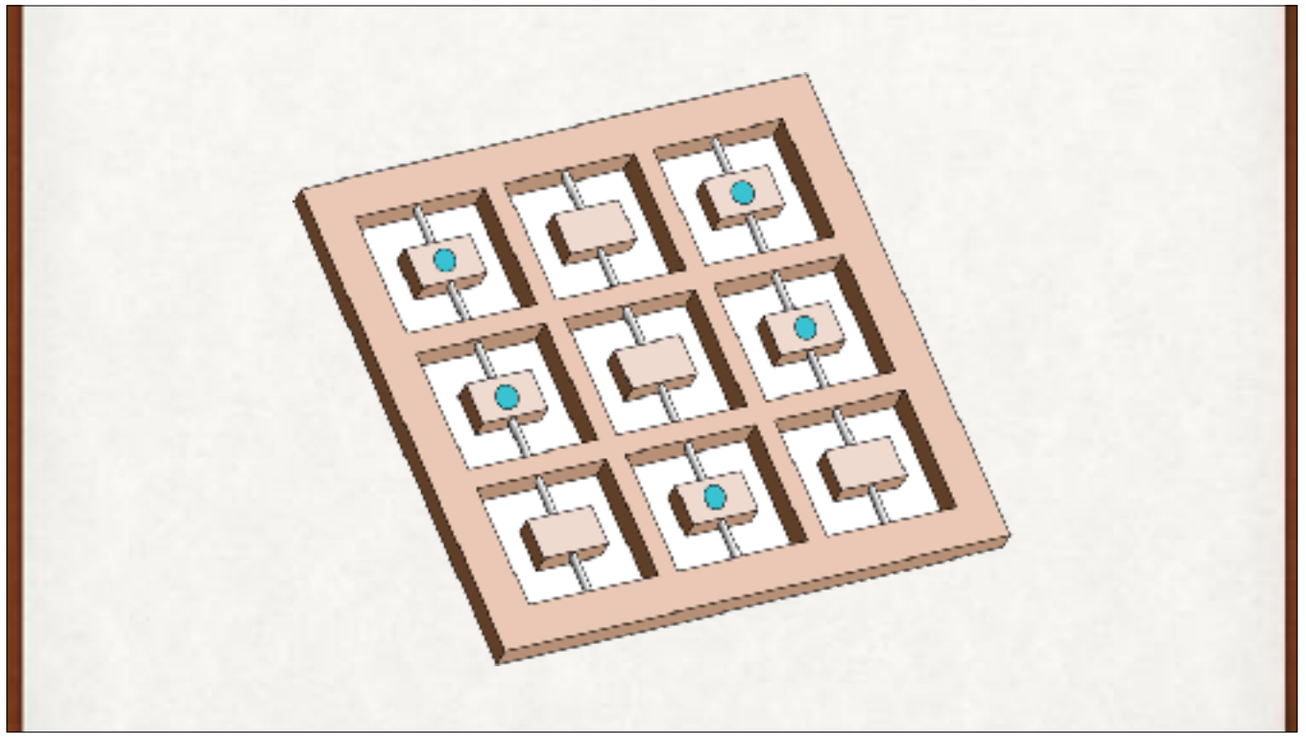
Sometimes a curve fits the data better than a line.



Sometimes a curve fits the data better than a line.

REINFORCEMENT LEARNING

Like training a dog. Operant conditioning. Rewards for good behavior.



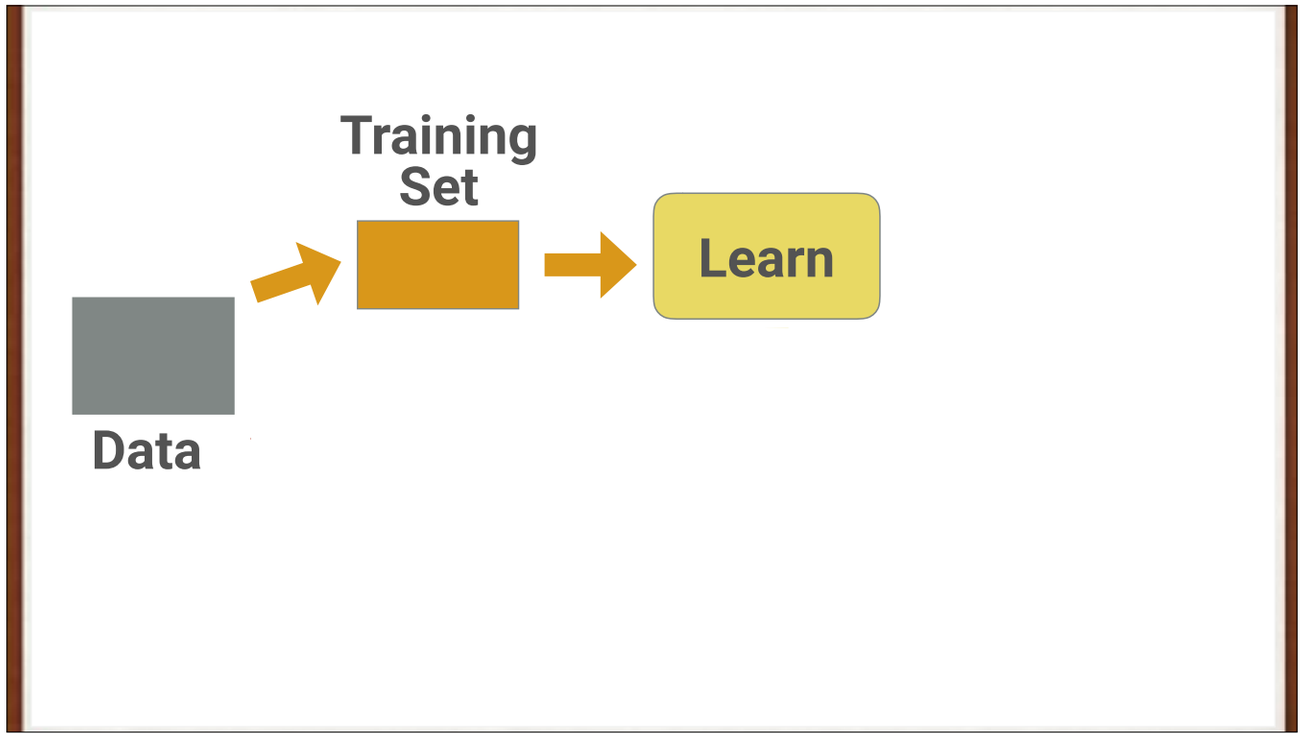
We’ll use RL to learn how to play Flippers later on. It’s a solitaire tic-tac-toe game.

SUPERVISED LEARNING

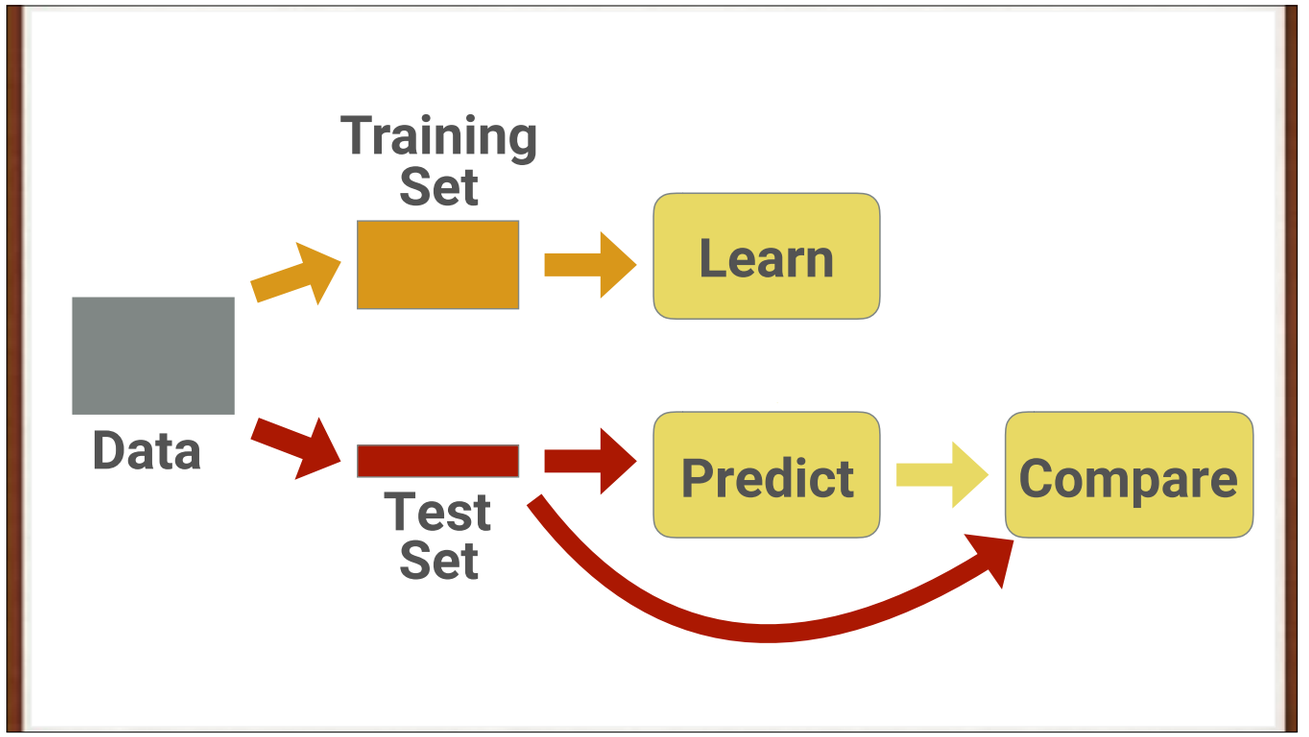
We have a label for each piece of data, and we want to learn how to predict that label.

Hummingbird Spoon Corkscrew Headphones

Some labels we’ve assigned to objects, so the system can learn which labels go with what objects.



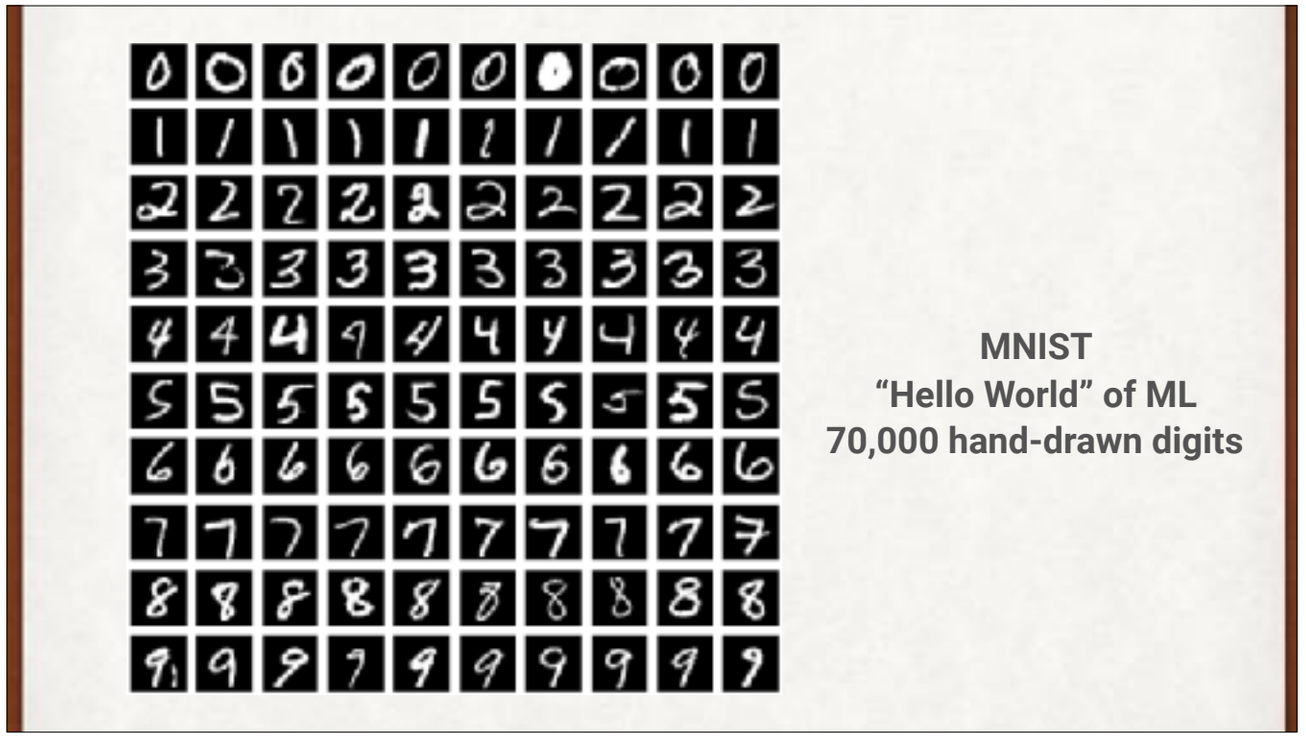
The basic supervised learning flow, using clunky Keynote graphics.



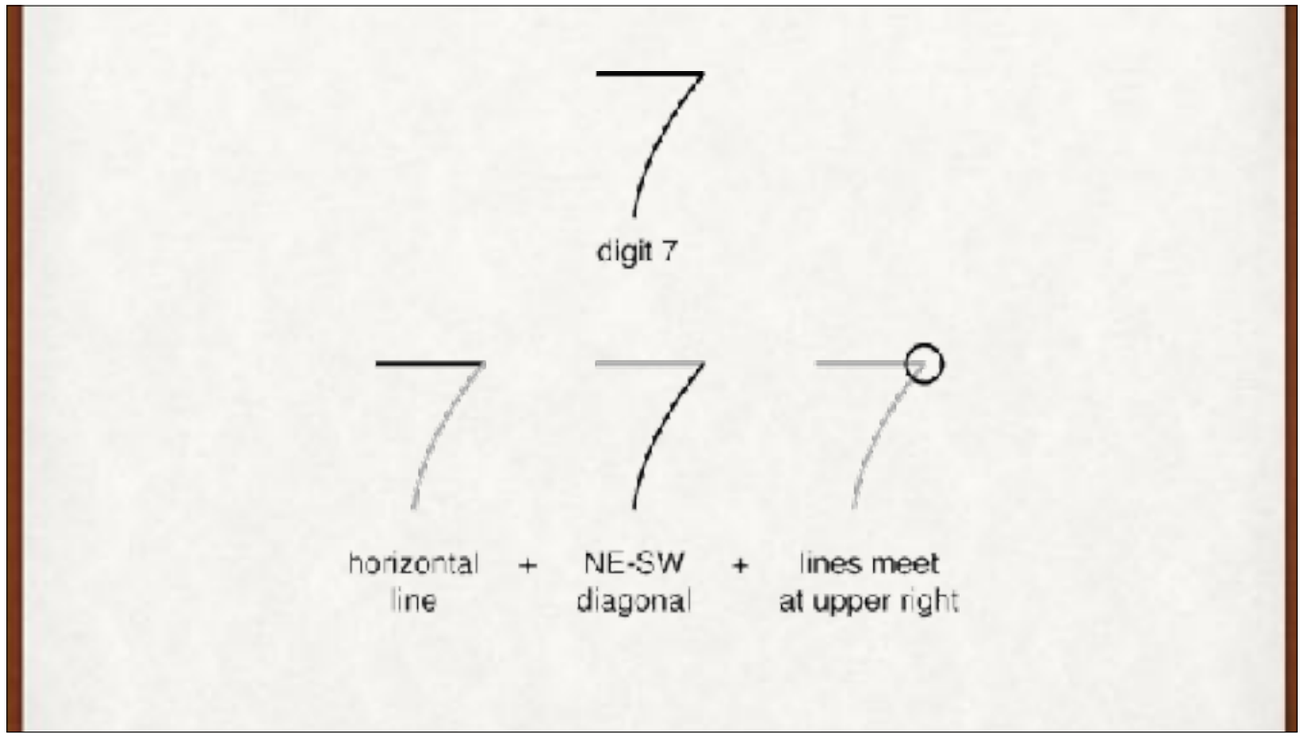
The basic supervised learning flow, using clunky Keynote graphics. The test set is used only once, at the very end. Think of that data as final exam questions. Once seen, they cannot be used again. If performance is not good enough, we need to start over with an untrained network.

LET’S IDENTIFY DIGITS!

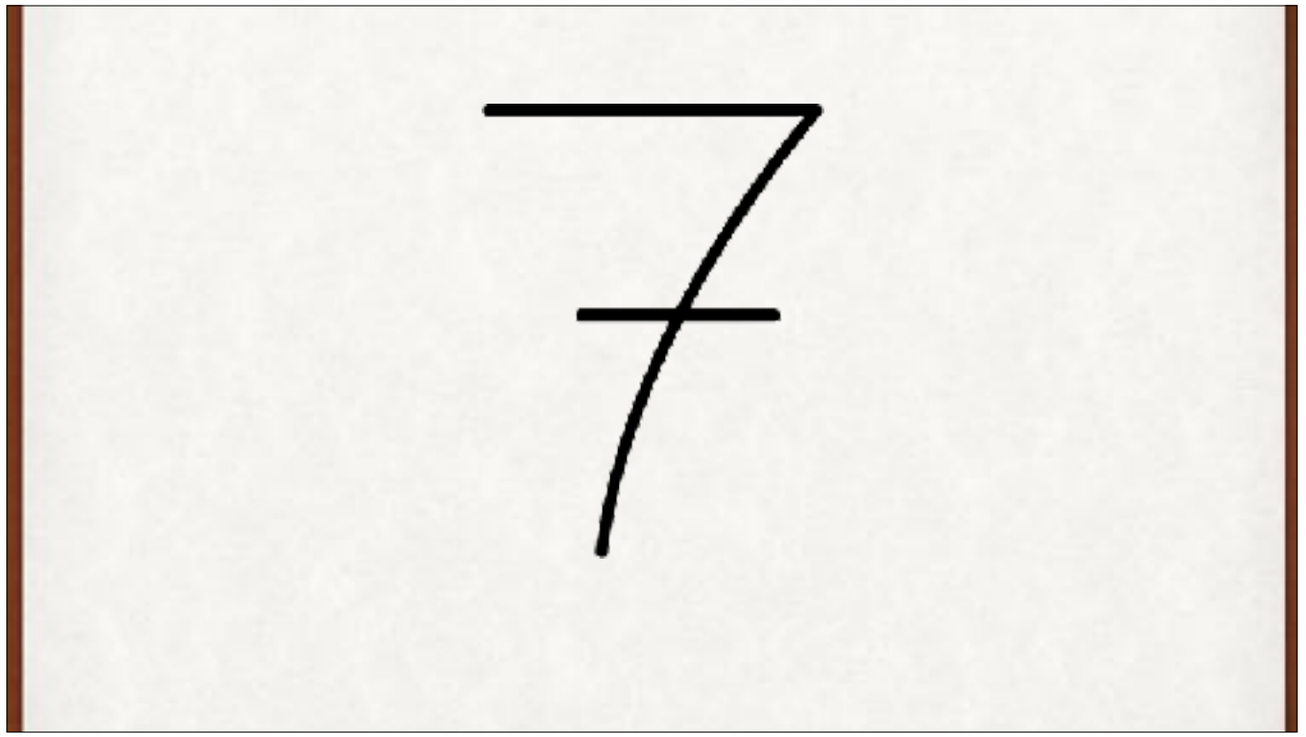
Let’s use ML to identify hand-drawn digits.



Everyone’s favorite classification problem! Called MNIST, it’s simple, clean, and just big and complicated enough to be interesting. 60k train 10k test. US National Institute of Standards and Technology. 1/2 census bureau, 1/2 high schoolers.

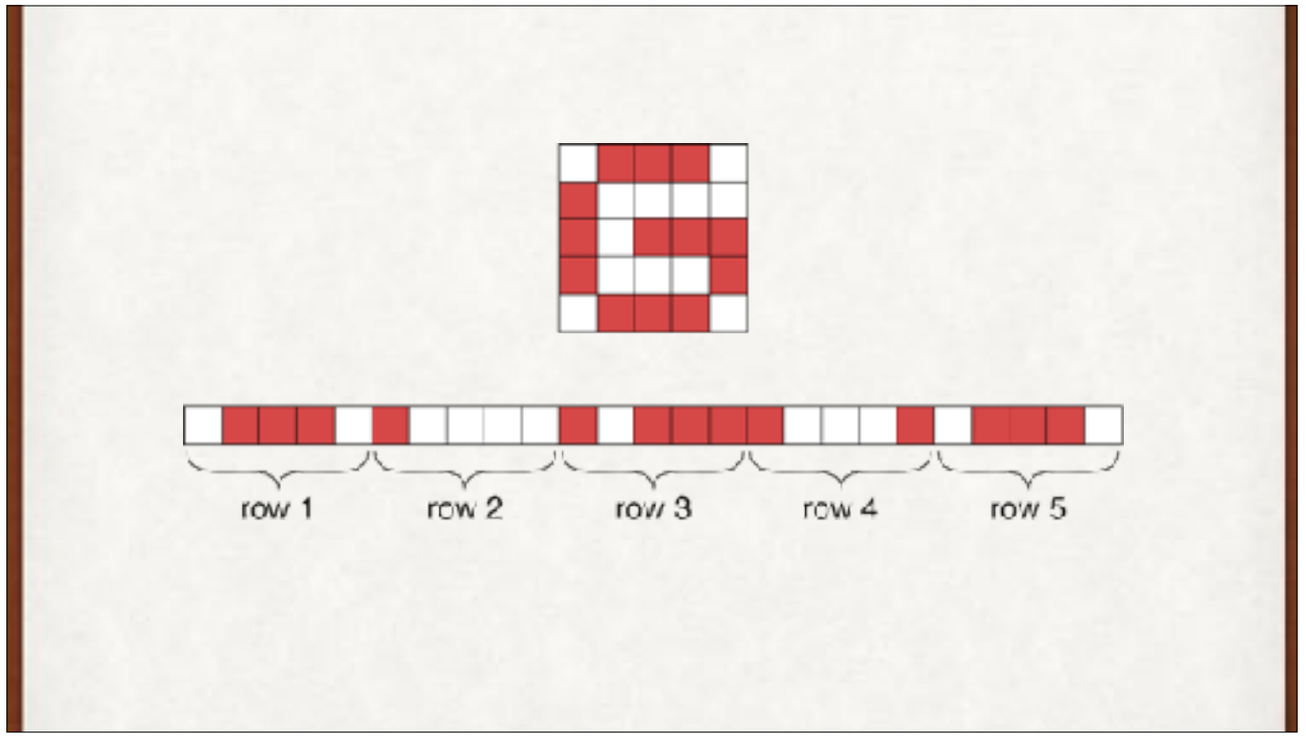


Expert system: craft rules that tell us which digit this is.

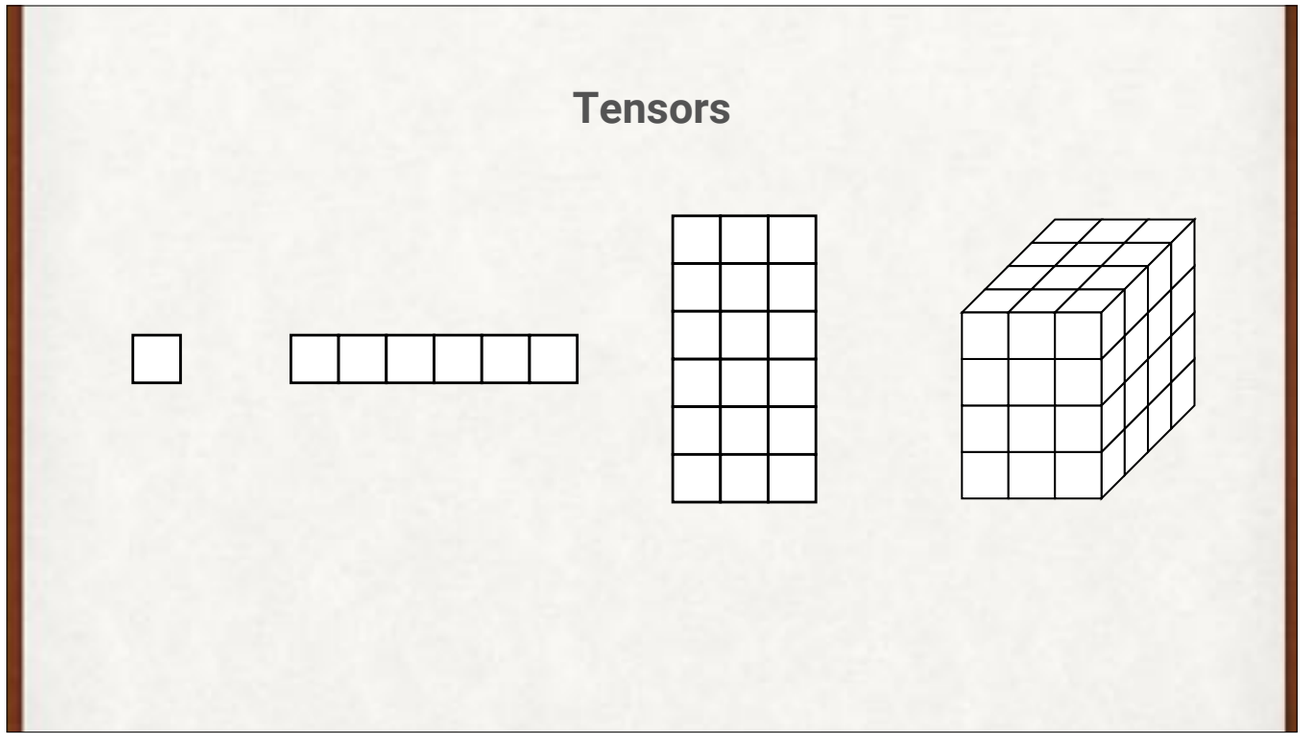


Uh oh, we forgot the line some people use. This is not going to scale up well for digits, much less reading X-rays or piloting an airplane. Instead of hand-crafting rules, let’s try DL.

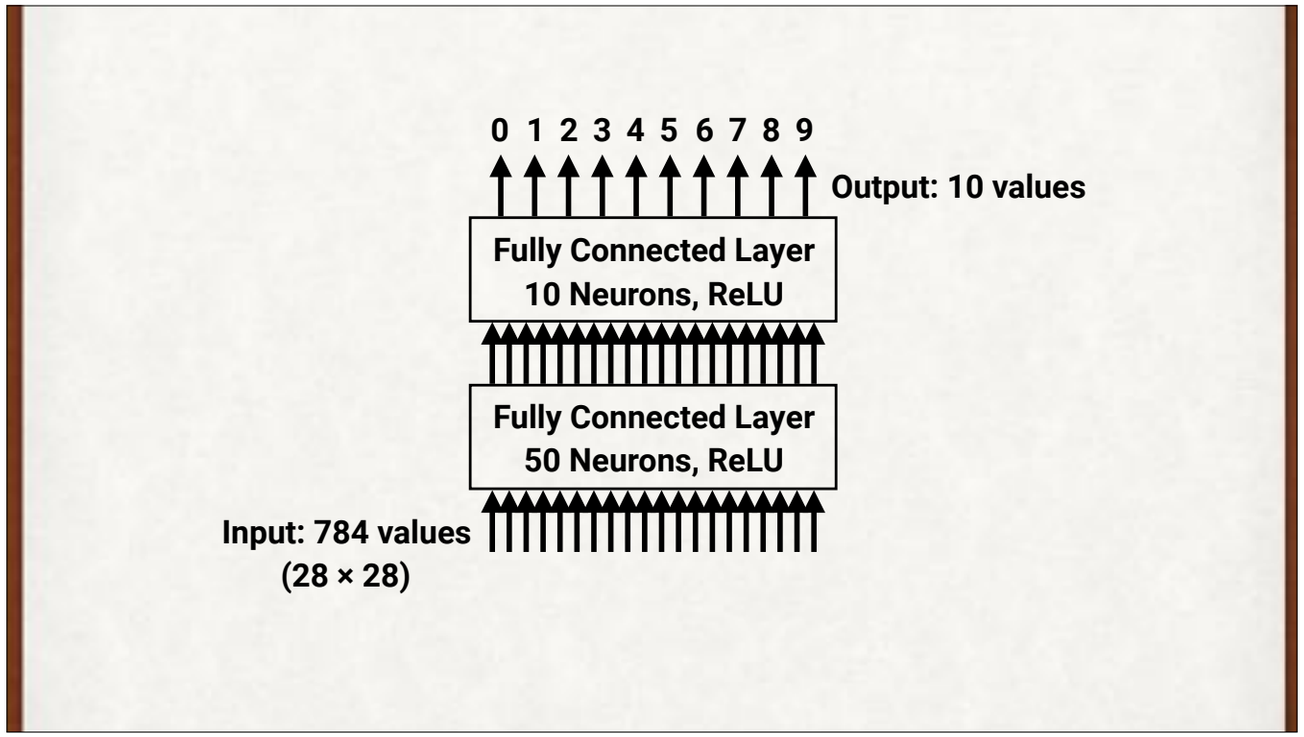
Everyone says ML is like Lego. Just snap the layers together.



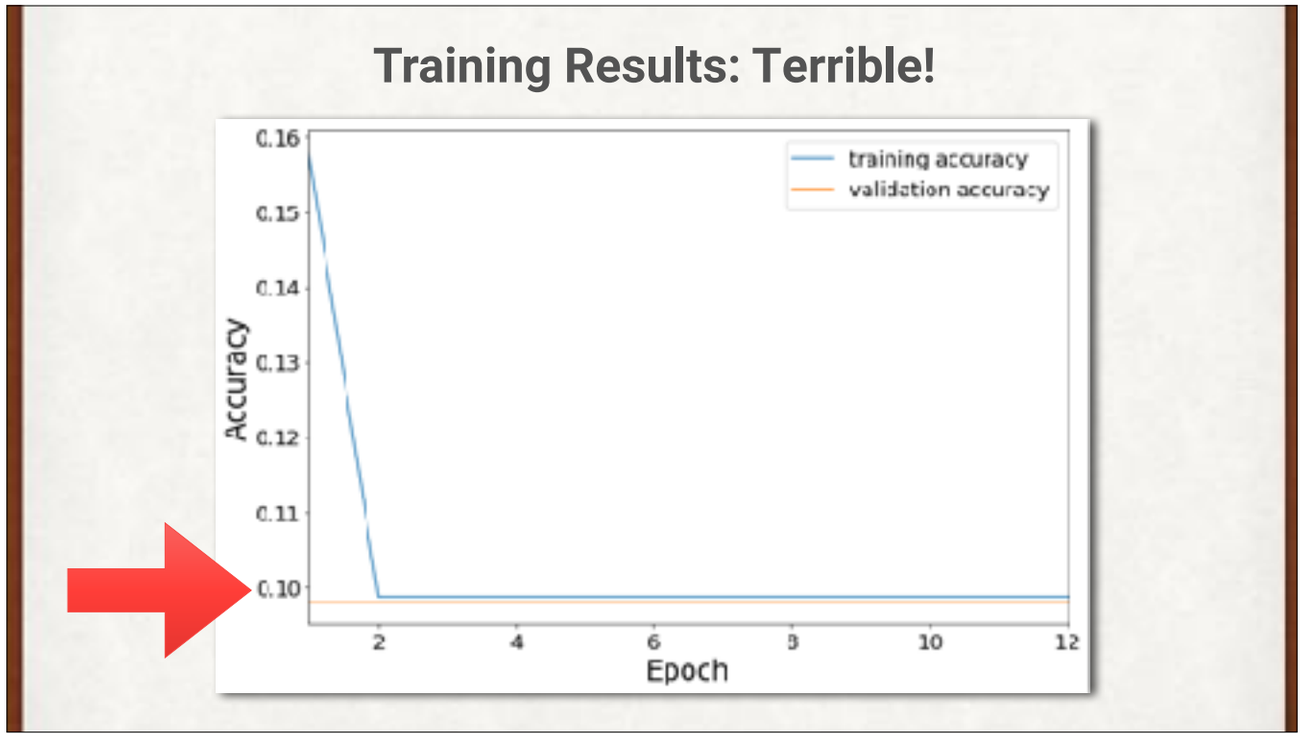
Input: 2D images as 1D arrays. Data shaping is important in DL.



Input: 2D images as 1D arrays. Data shaping is important in DL. Tensors are just multidimensional arrays. No holes, no extra bits sticking out.

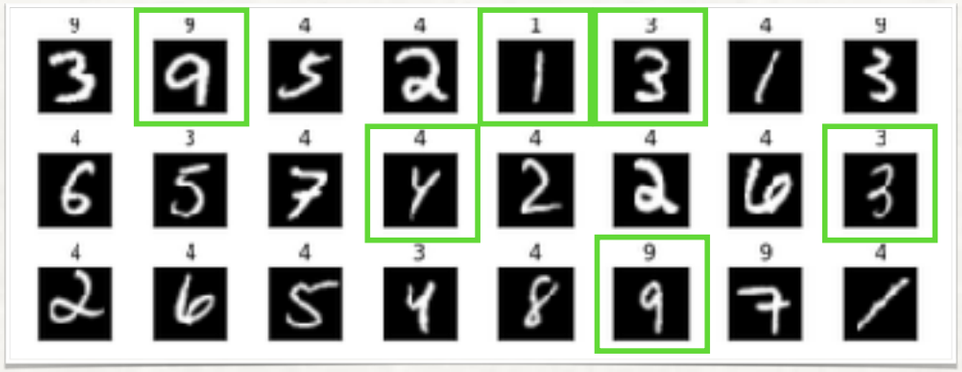


Our first DL. Just snap a few existing pieces, let’s not worry about the details.



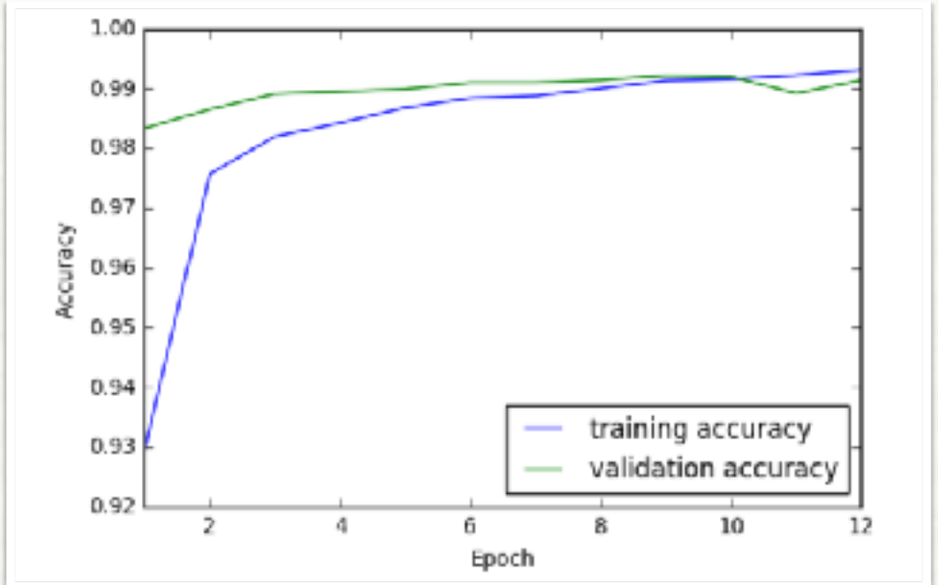
Oh no! That’s not very good at all. The red line is 10%, or a 1 in 10 chance of getting the right answer. That’s what we get from guessing randomly. We’re not doing any better than chance.

Network Predictions: Terrible!



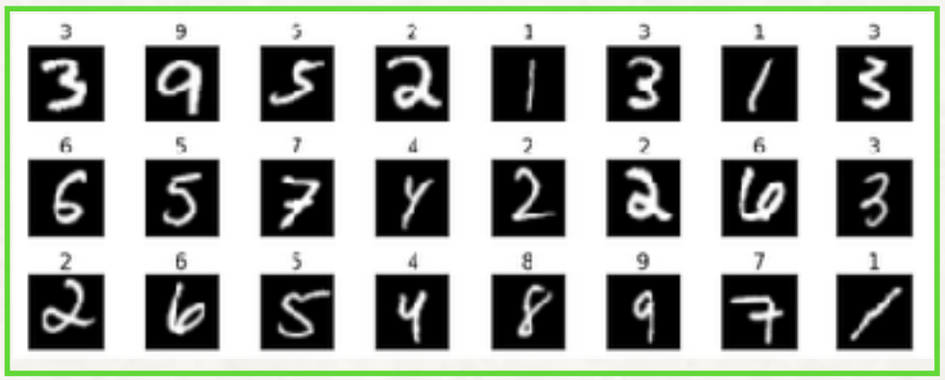
Only the green predictions are correct. Blech.

Training Results We Want



This is the kind of thing we’re hoping for: better than 99% accuracy.

Network Predictions We Want



Yeah, this is what we want: correct labels every time.

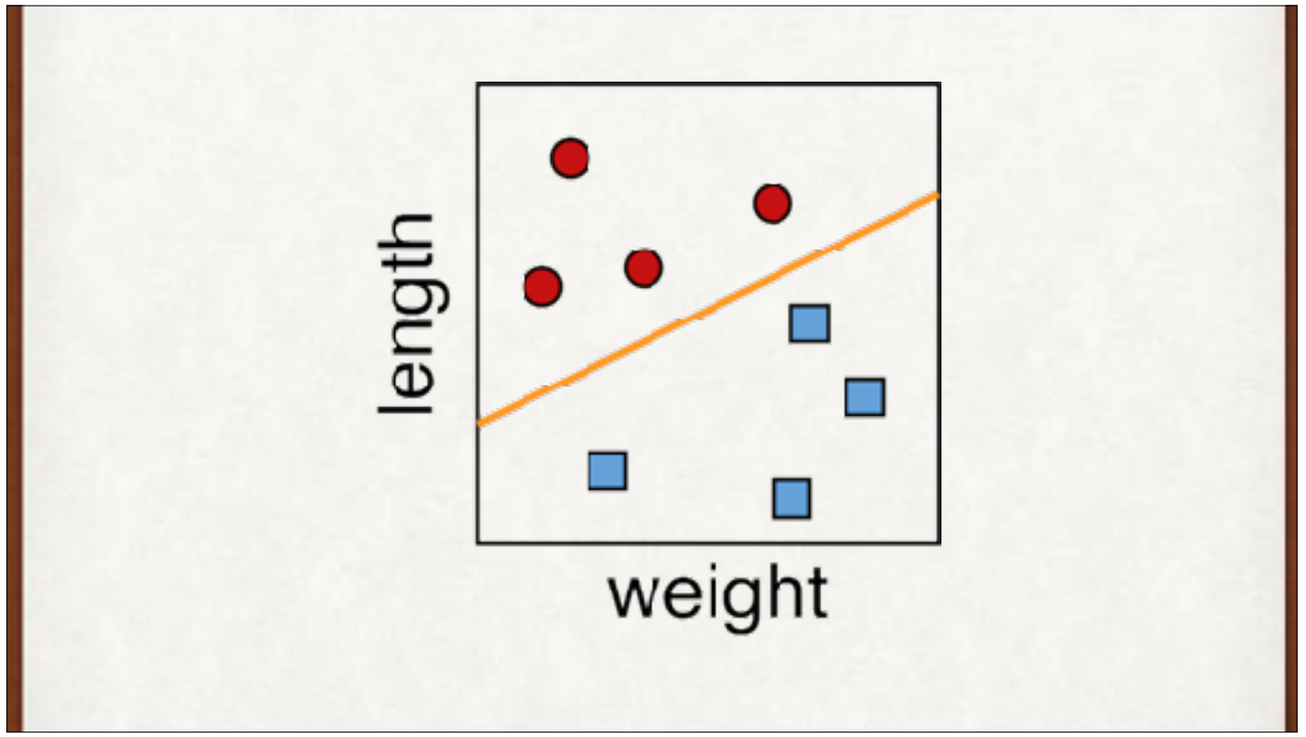
How to
get there

To get good results, we have to actually know something about what we’re doing! So let’s back up a bit.

A quick break to catch our breath and change topic.

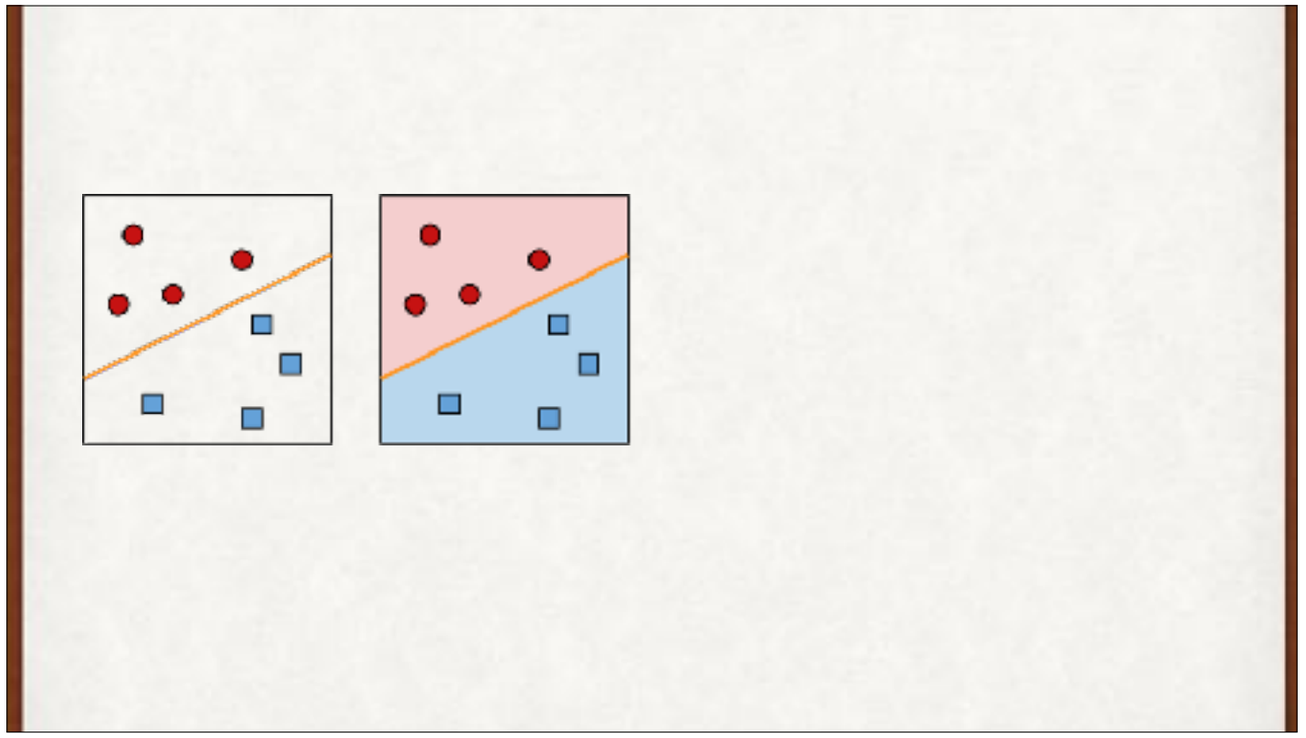
Classification

Digit labeling is a problem in “classification,” or assigning a category to each input.

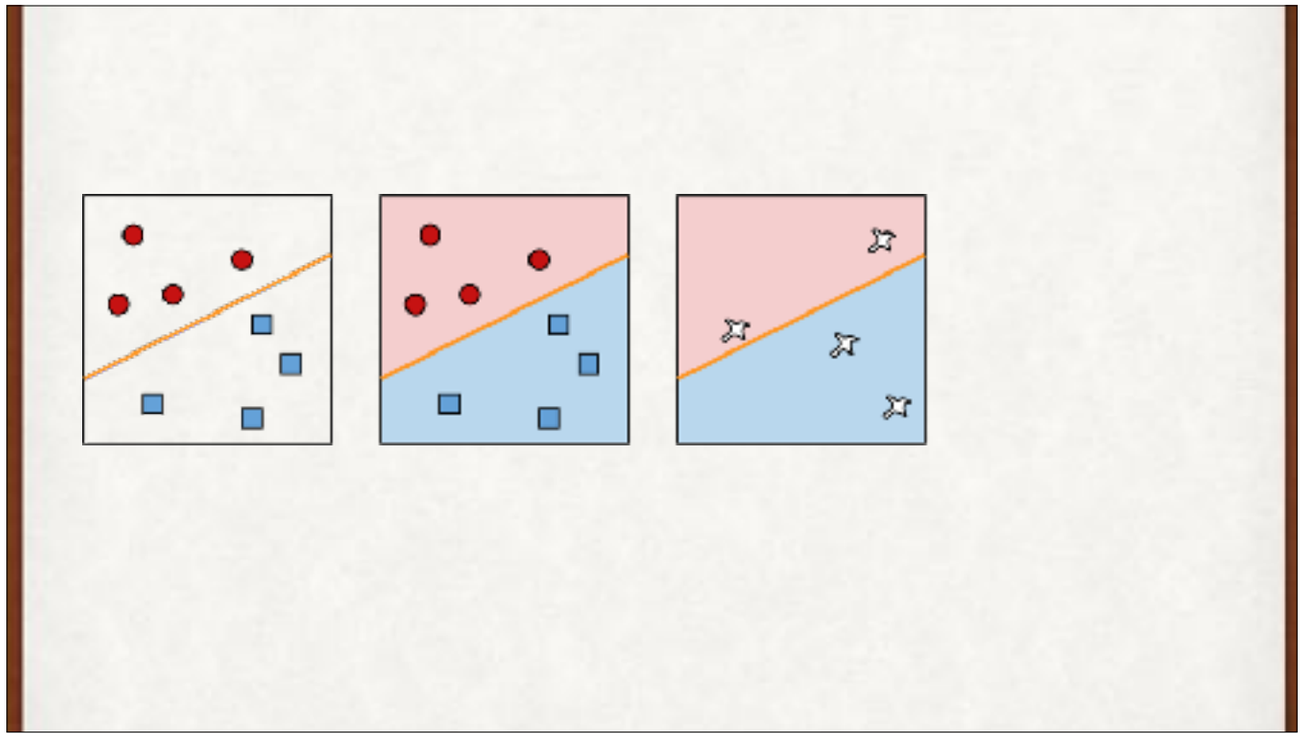


Suppose we’re looking at two species of bananas. One species (red circles) tends to be long but not too heavy, while the other species (blue squares) tends to be heavy but not too long. The orange line splits the groups.

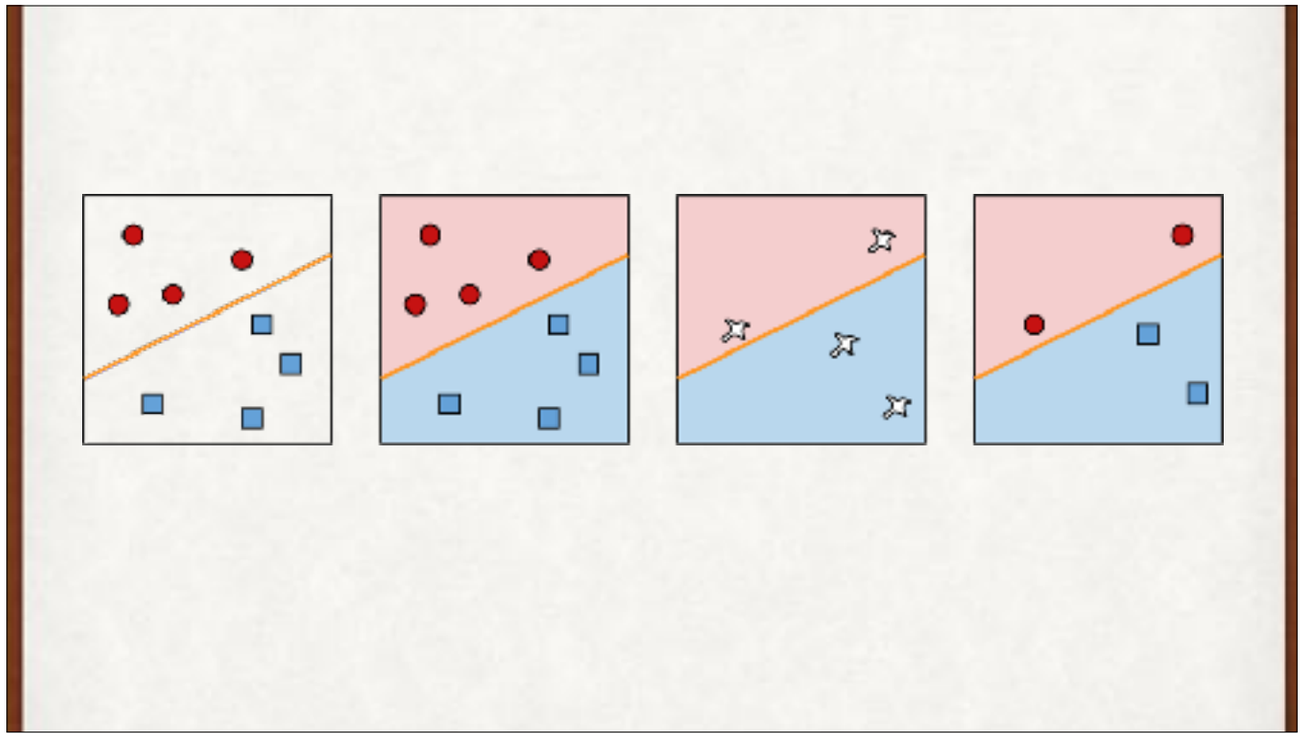
Now when new bananas arrive (white splats), we can tell which species they are by seeing which side of the line they fall on.



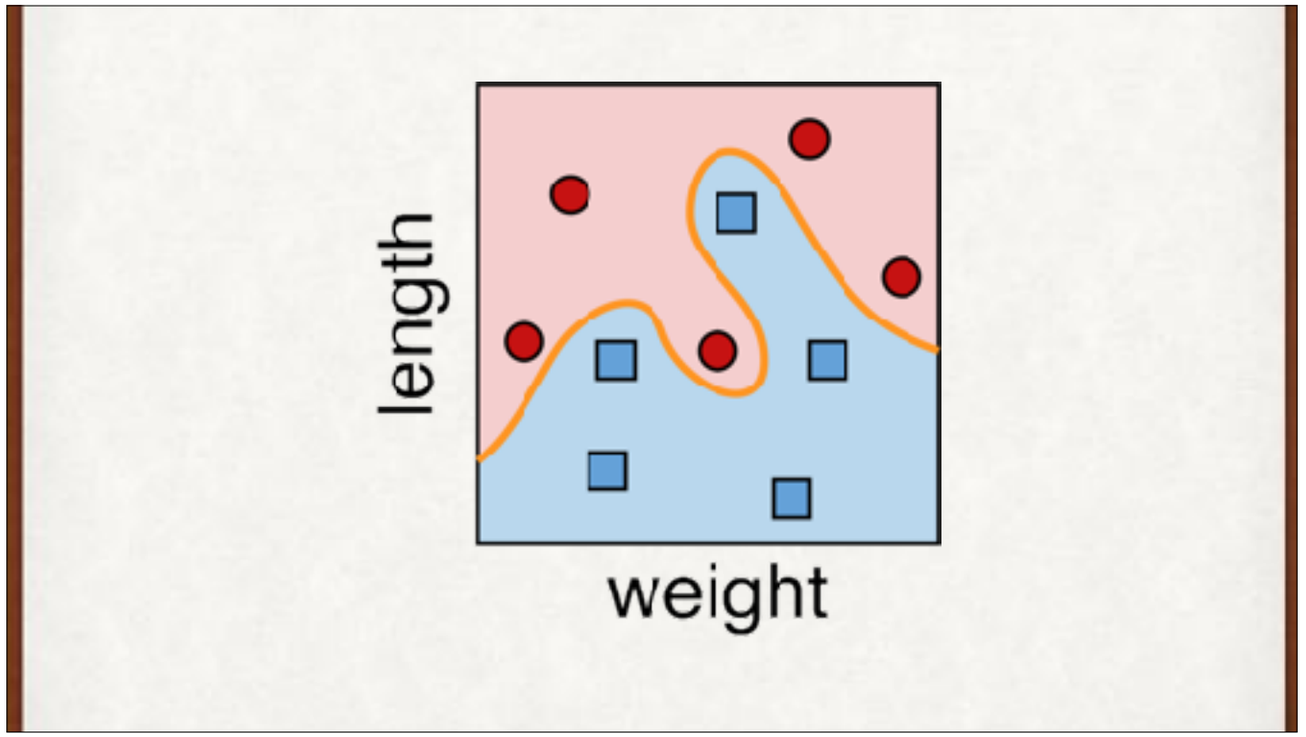
Now when new bananas arrive (white splats), we can tell which species they are by seeing which side of the line they fall on.



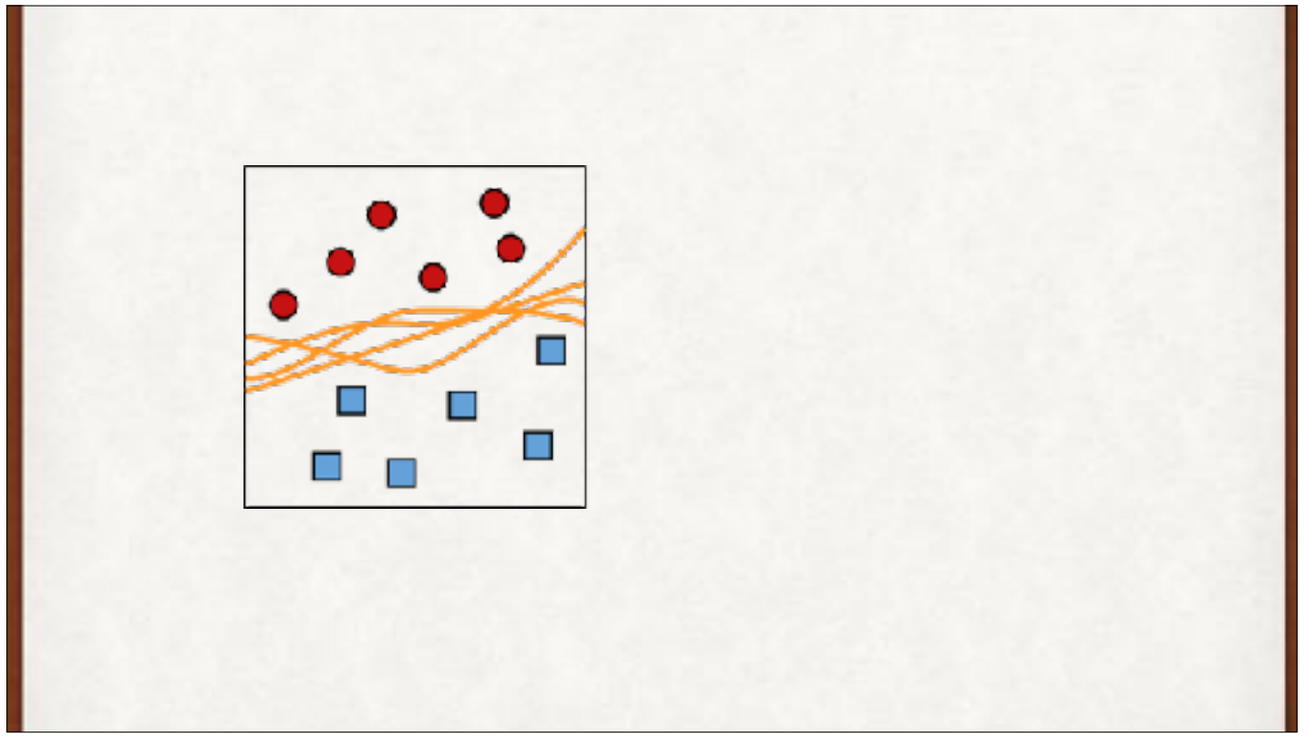
Now when new bananas arrive (white splats), we can tell which species they are by seeing which side of the line they fall on.



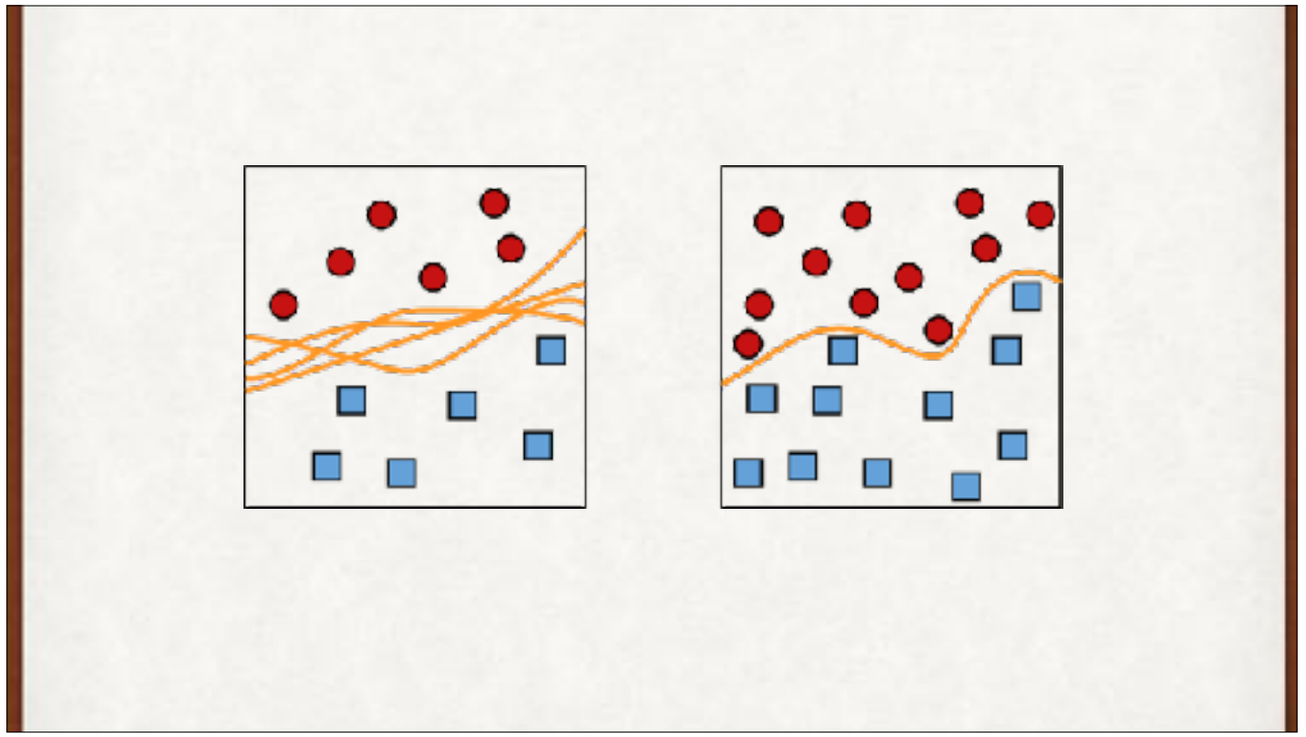
Now when new bananas arrive (white splats), we can tell which species they are by seeing which side of the line they fall on. This data is **linearly separable**.



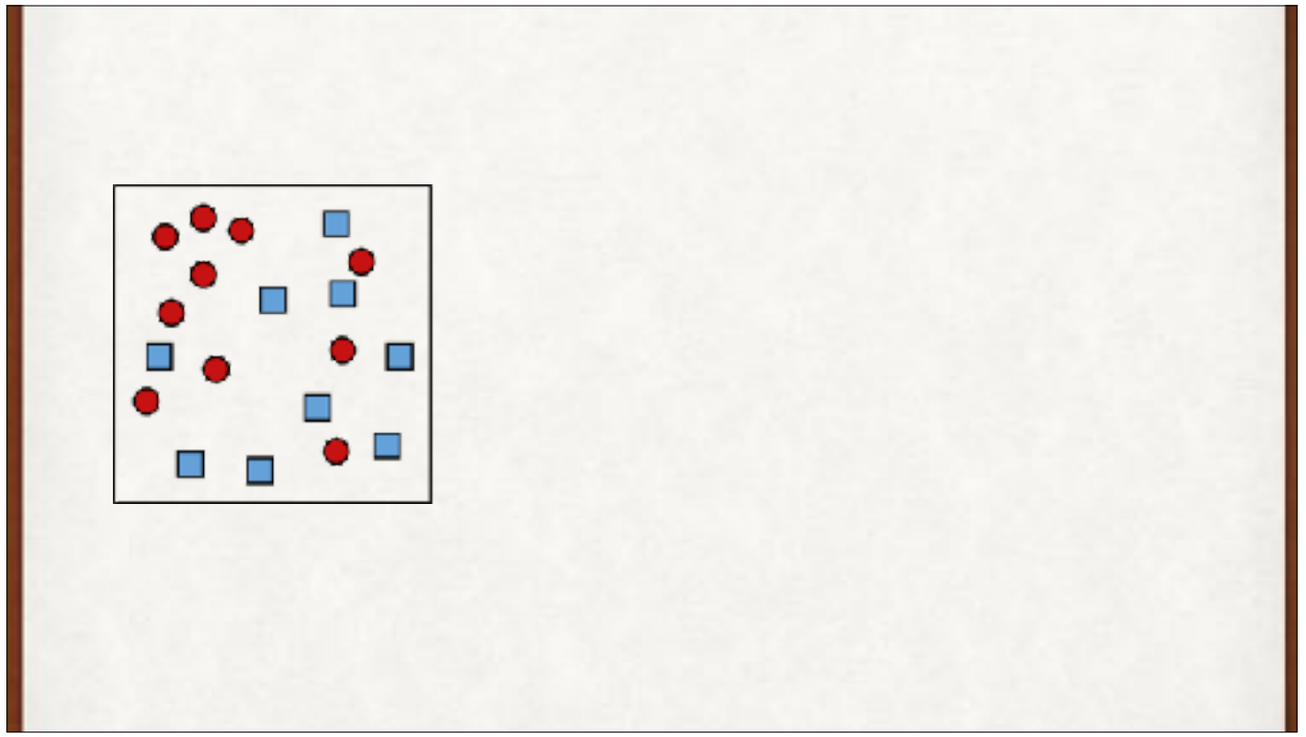
Here’s a couple of different species of banana. Now a straight line won’t do, but a curve can separate the two groups so we can predict new bananas.



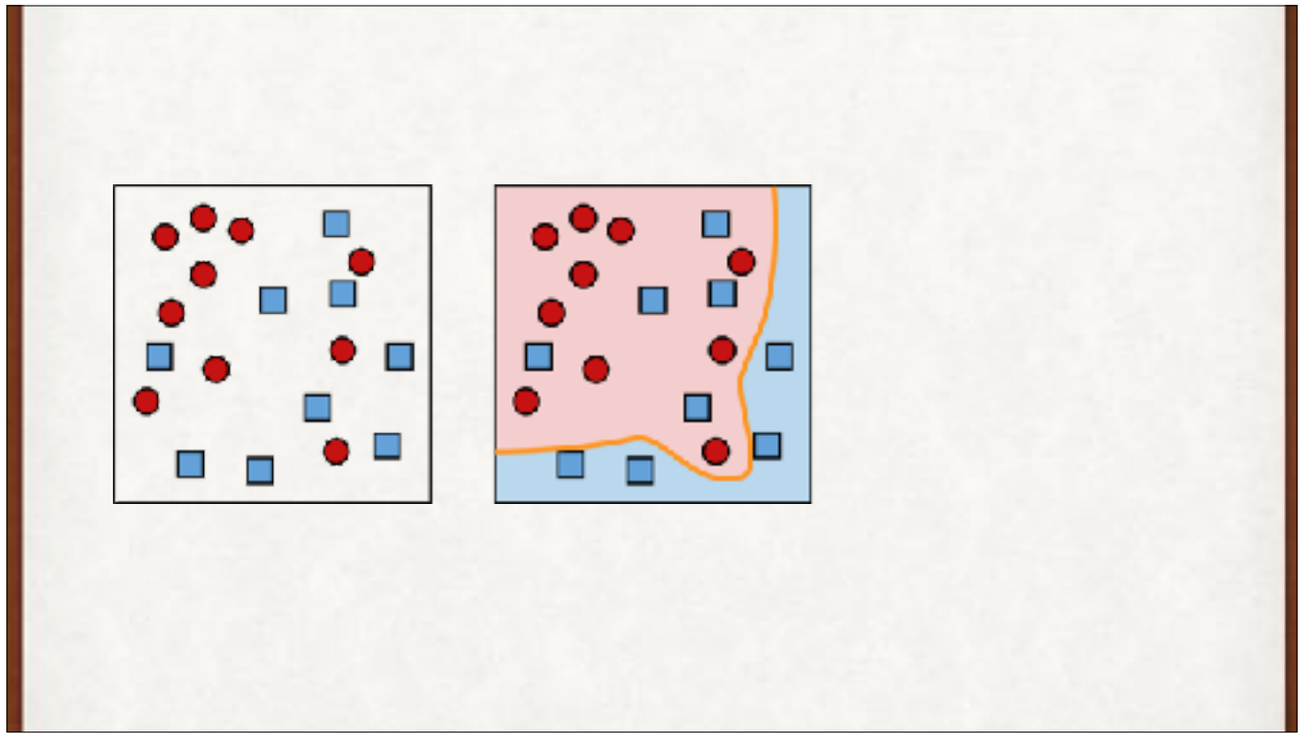
It’s not always clear which line, or curve, is best. On the left, they all split the two groups. On the right, after getting more data, and we can nail down the shape of the curve a little more.



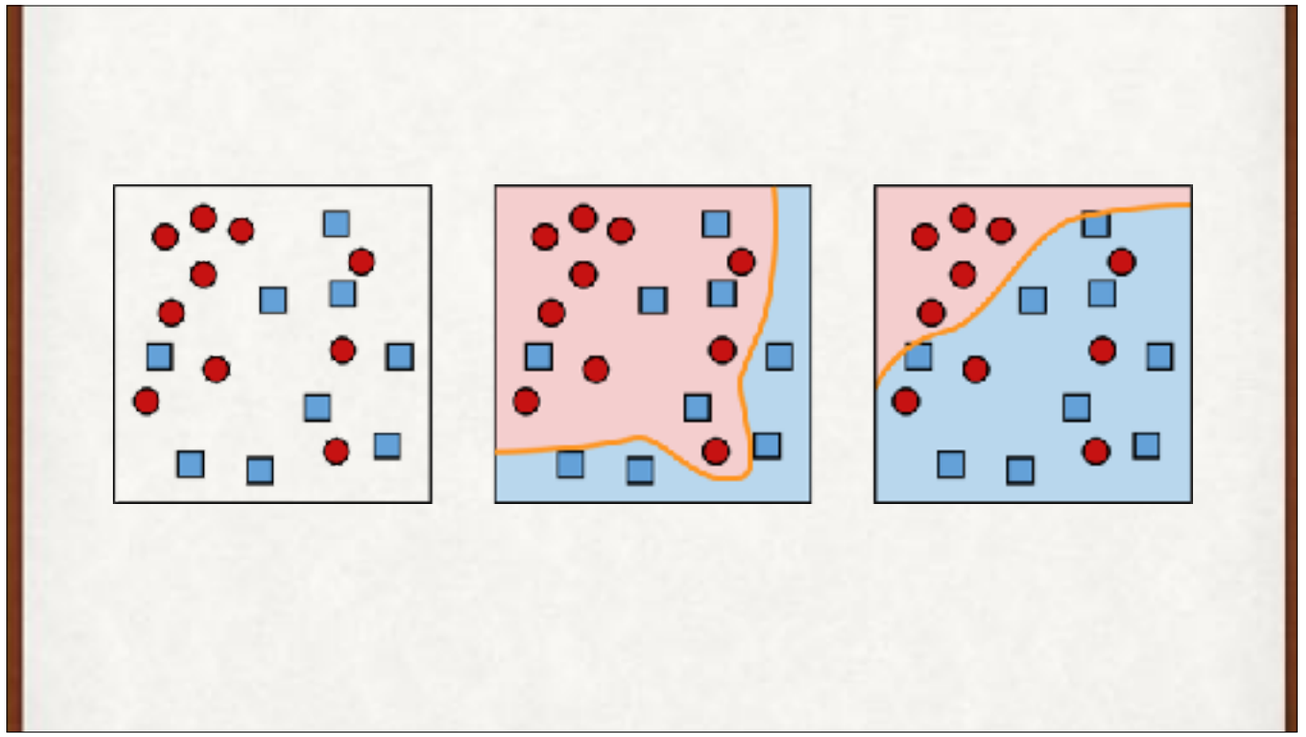
It’s not always clear which line, or curve, is best. On the left, they all split the two groups. On the right, we get more data, and we can nail down the shape of the curve a little more.



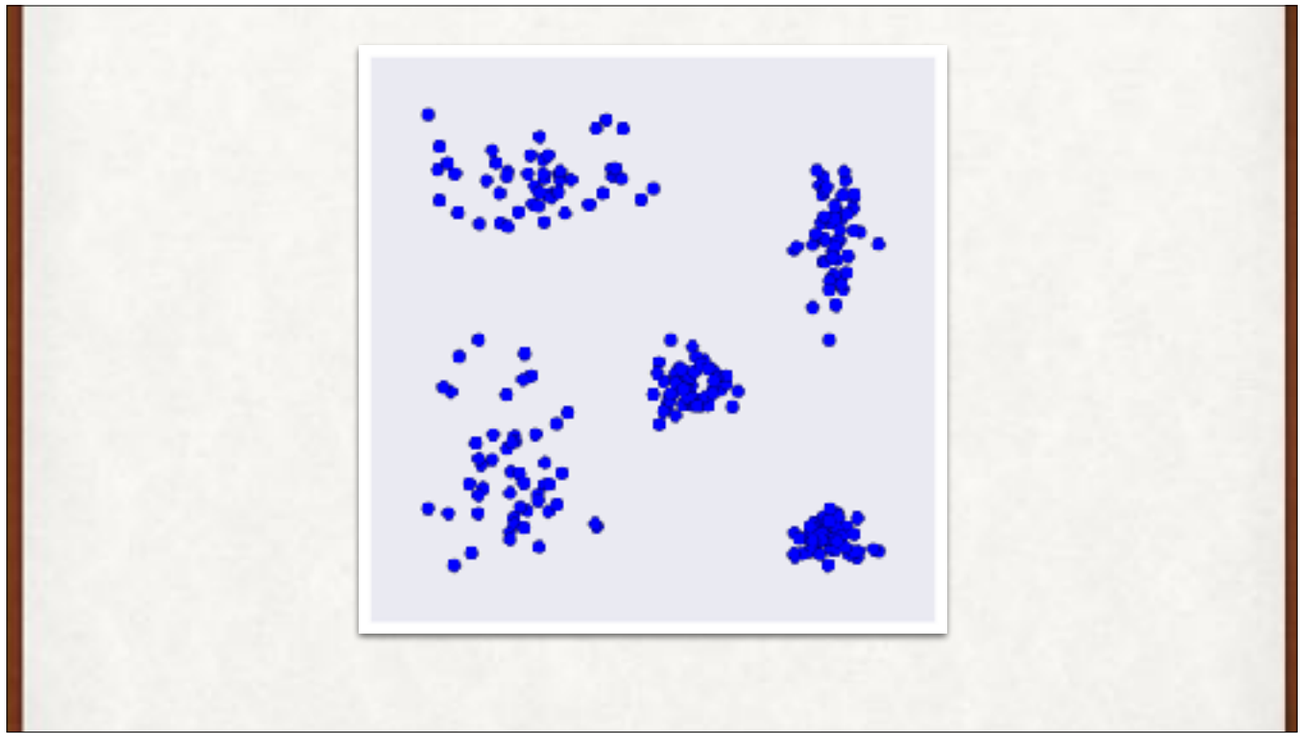
What’s better, a very wiggly but precise curve (middle), or a smooth and imprecise one (right)? Which is more likely to do better on future data? It depends on the data and what we want from our system. It’s a policy decision.



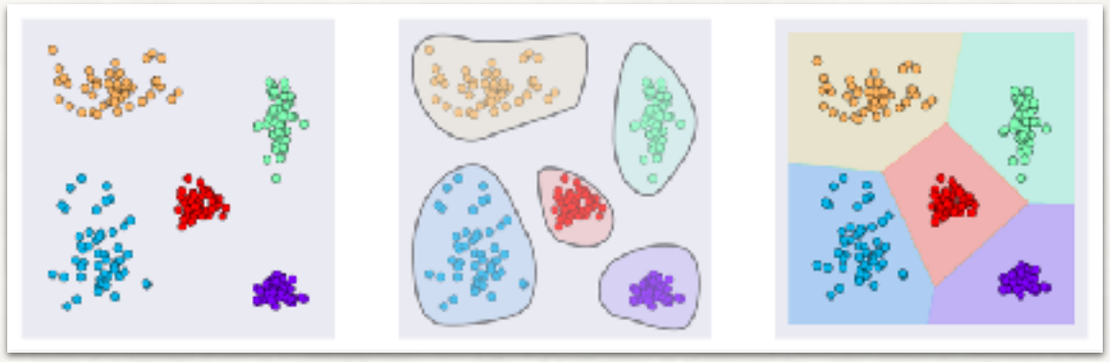
What’s better, a very wiggly but precise curve (middle), or a smooth and imprecise one (right)? Which is more likely to do better on future data? It depends on the data and what we want from our system.



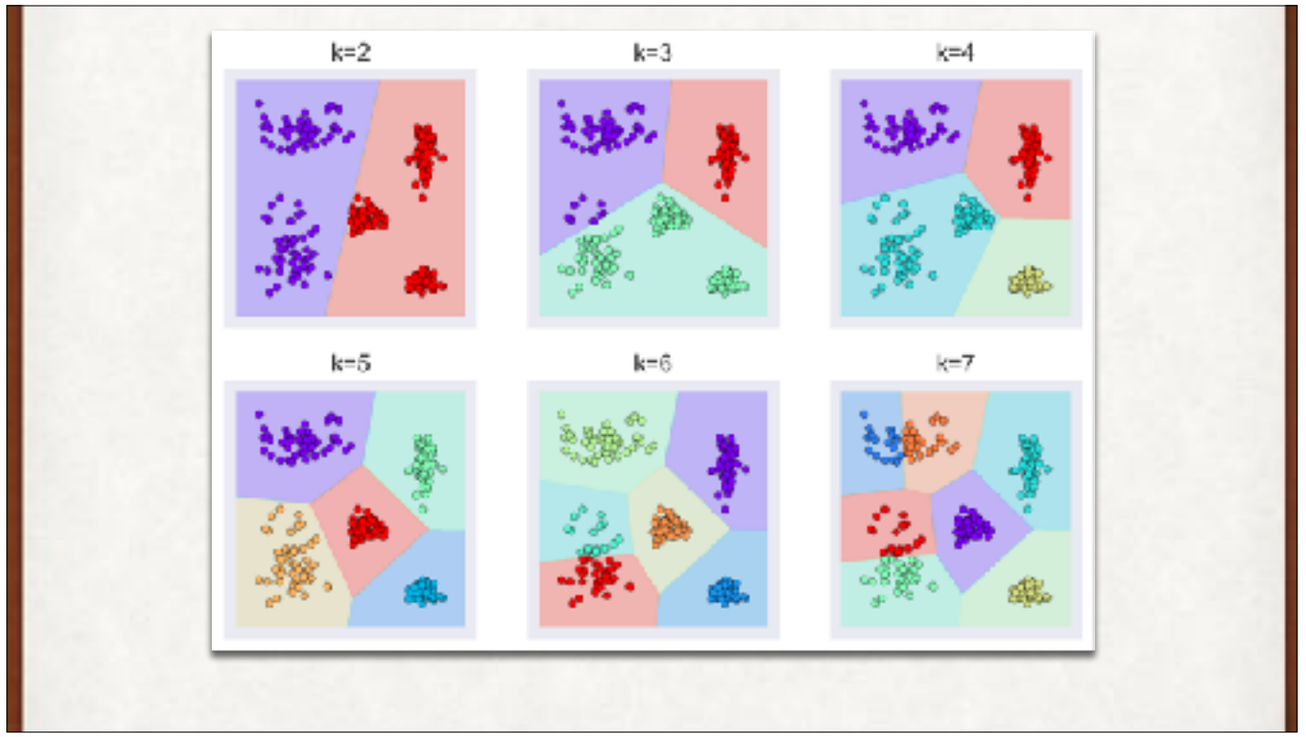
What’s better, a very wiggly but precise curve (middle), or a smooth and imprecise one (right)? Which is more likely to do better on future data? It depends on the data and what we want from our system.



A different approach. Maybe we can just cluster things by finding natural boundaries.



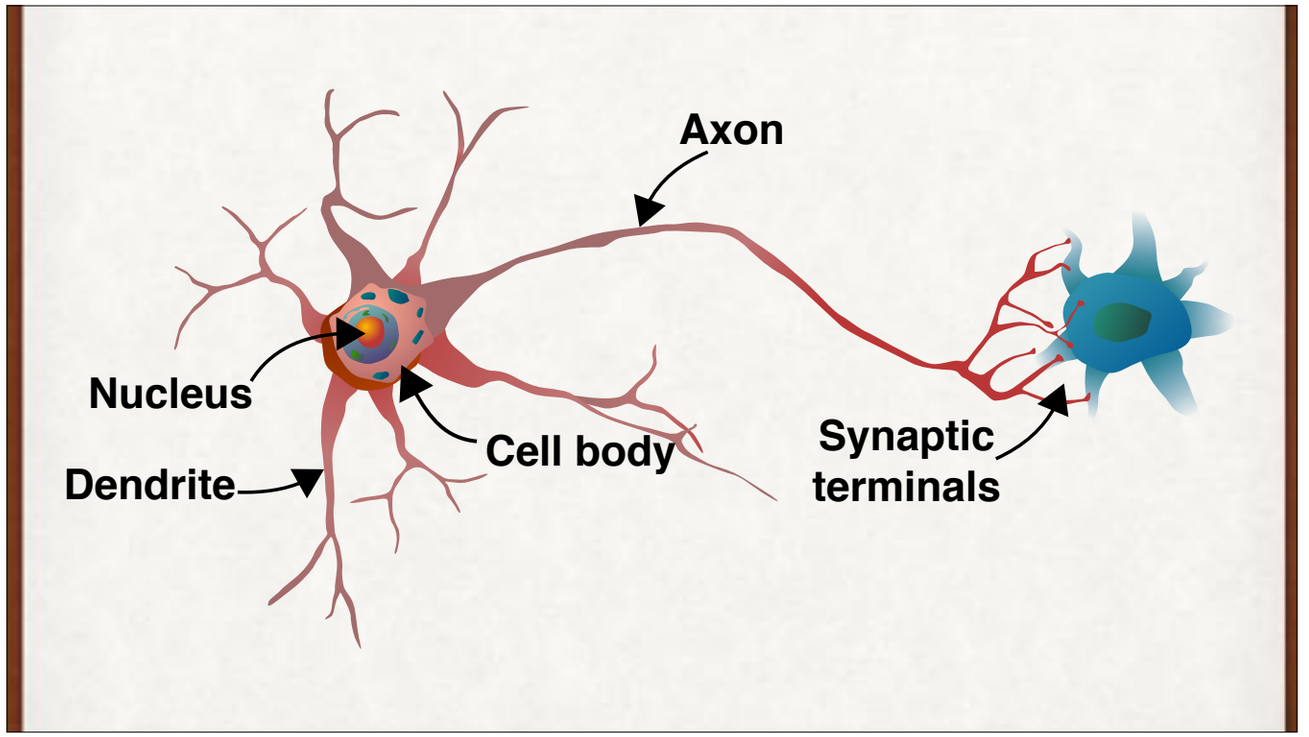
This algorithm, called k-means clustering, seems to be doing a good job. But we have to tell it how many clusters to use.



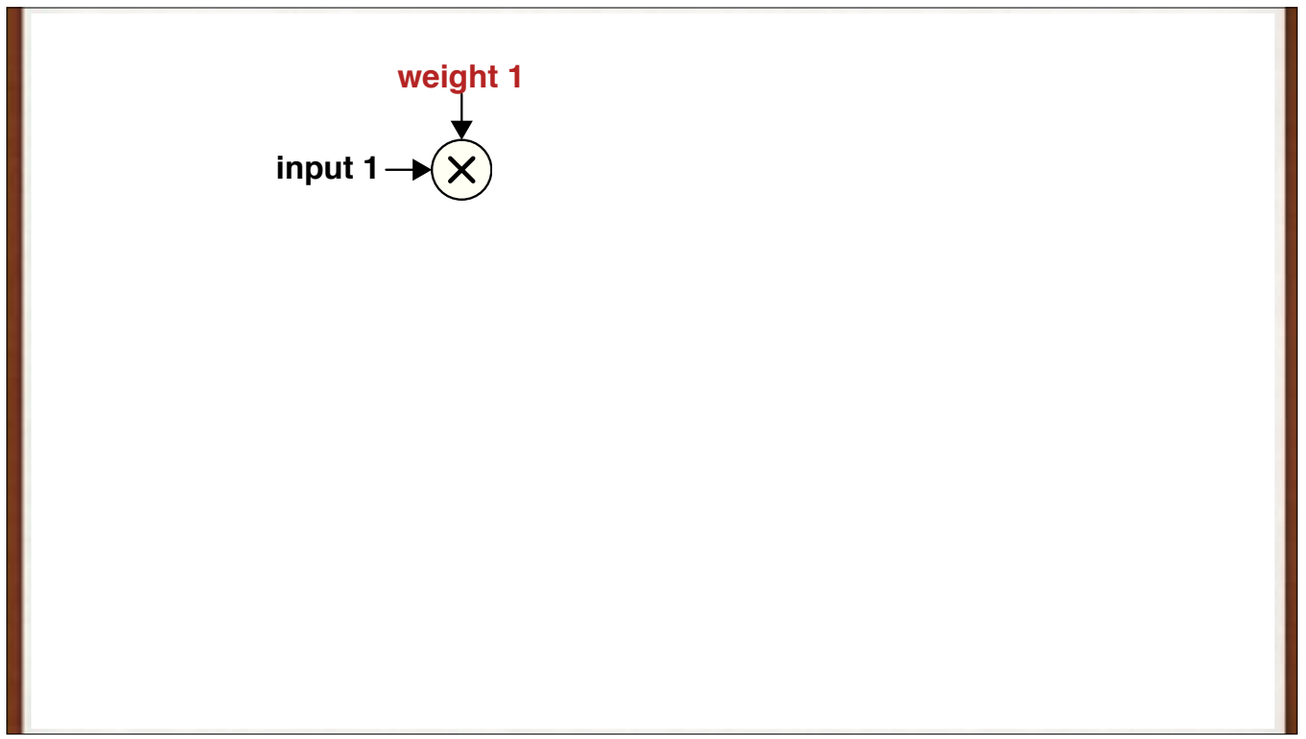
If the “k” (number of clusters) we pick is too small or too big, the results aren’t so great.

Neurons

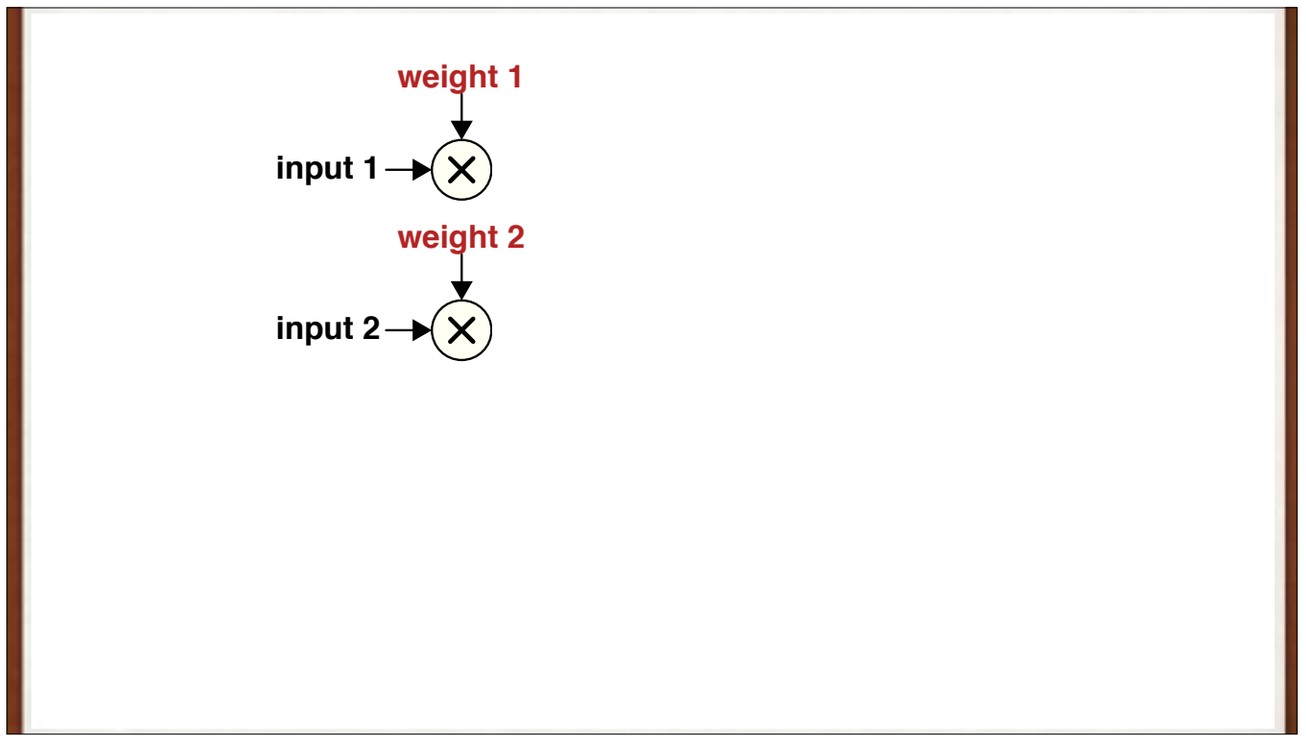
To find a good automatic classifier, we'll start over. We'll begin with artificial neurons.



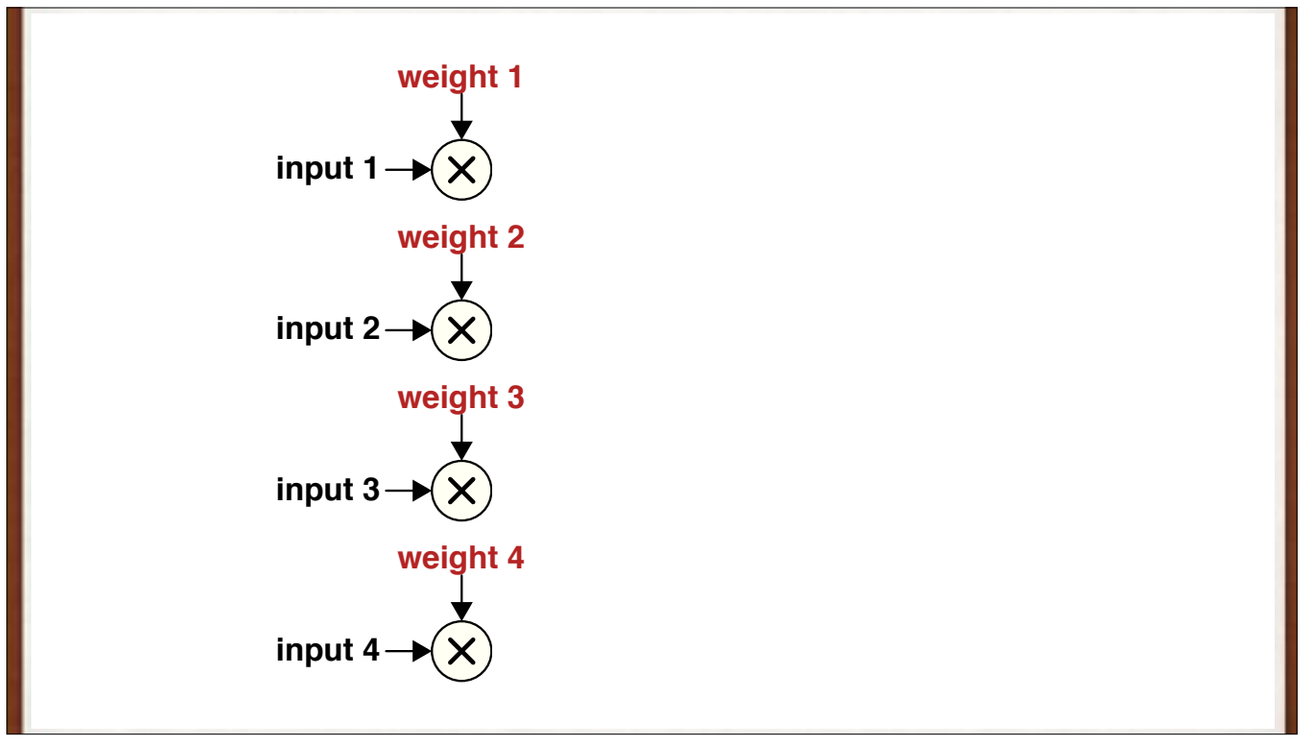
A real neuron is crazy complicated. Biology, chemistry, physics, timing, electricity, thresholds, emissions, absorptions - it’s super complex and we still don’t fully understand even a single neuron.



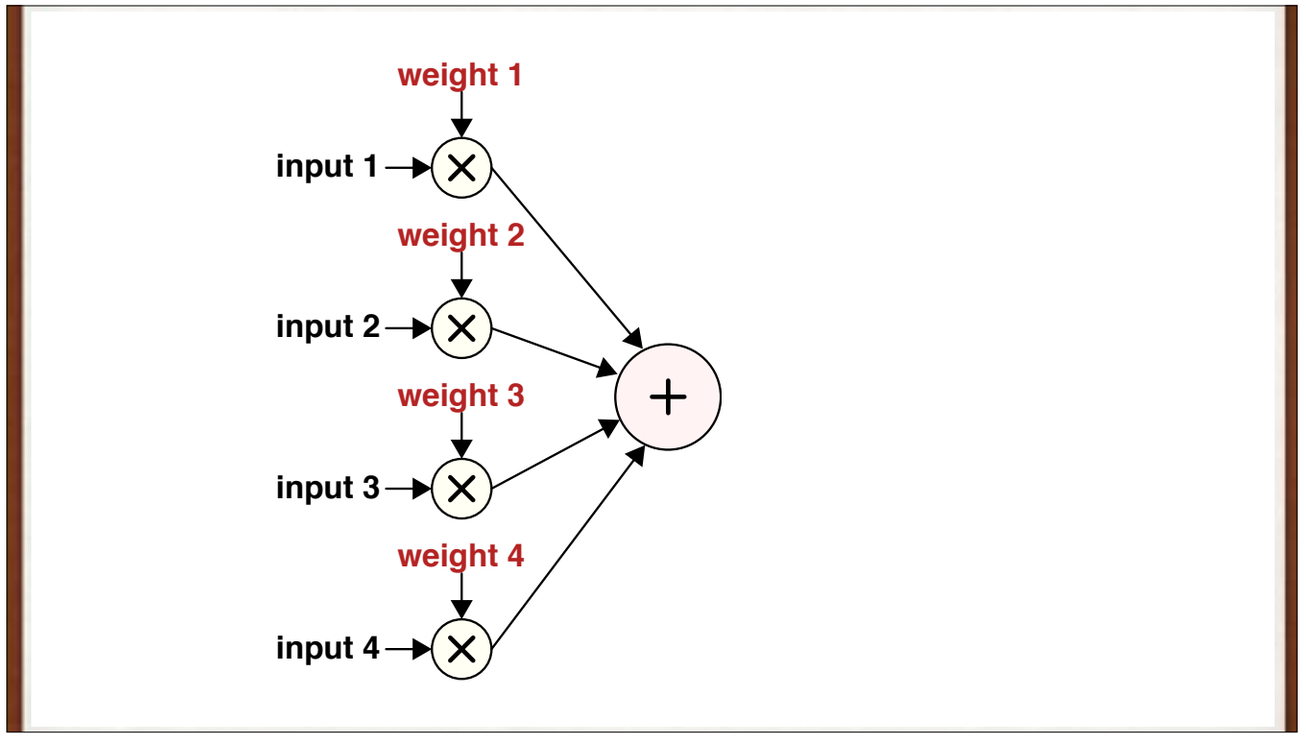
Our “artificial neuron”, or **perceptron**: inputs (numbers) come in. Each is “weighted” by another number. We add those all up, and threshold the result. The output is then -1 or 1 (some people use 0 and 1).



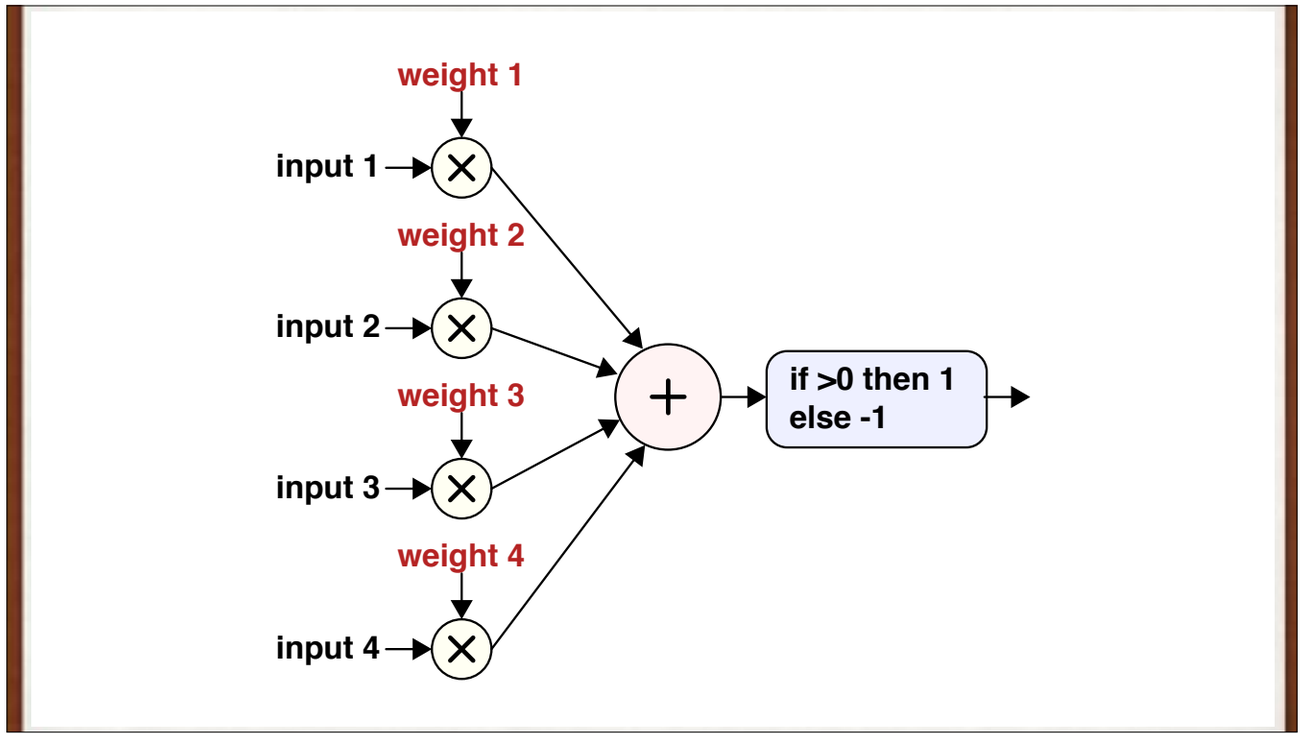
Our “artificial neuron”, or perceptron: inputs (numbers) come in. Each is “weighted” by another number. We add those all up, and threshold the result. The output is then -1 or 1 (some people use 0 and 1).



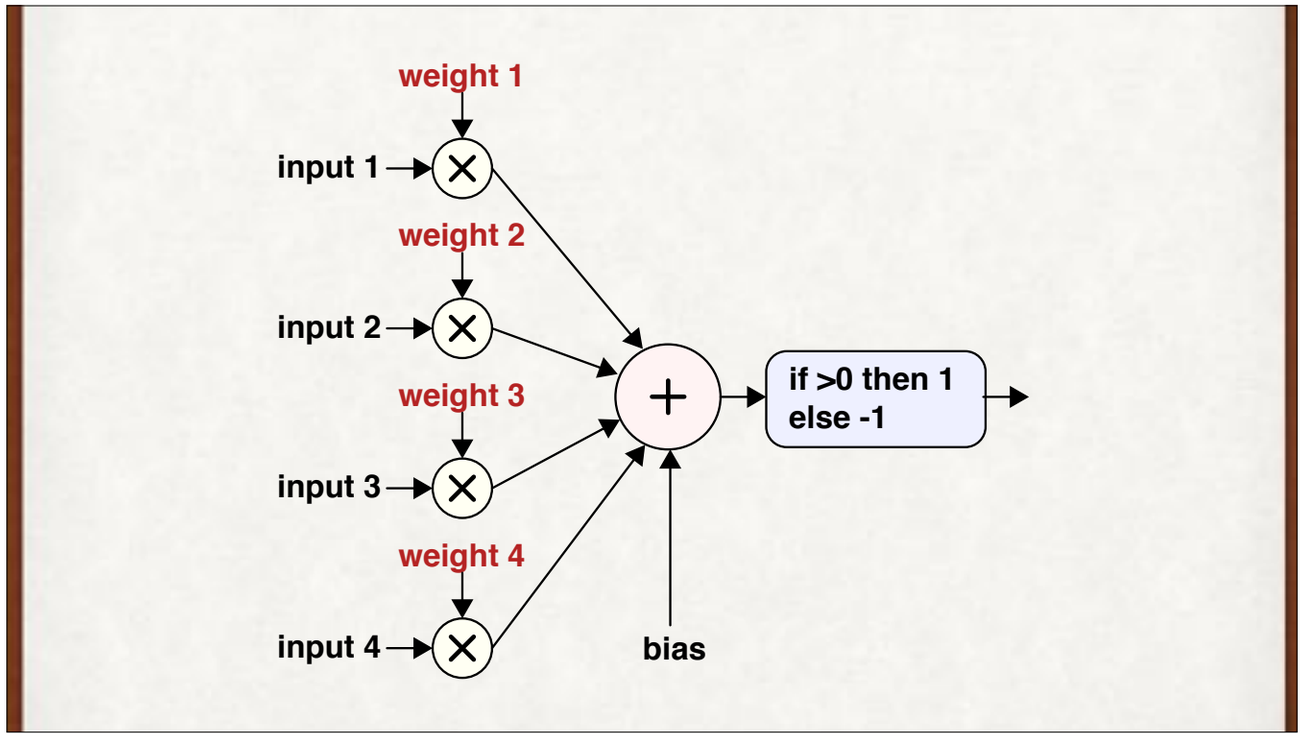
Our “artificial neuron”, or perceptron: inputs (numbers) come in. Each is “weighted” by another number. We add those all up, and threshold the result. The output is then -1 or 1 (some people use 0 and 1).



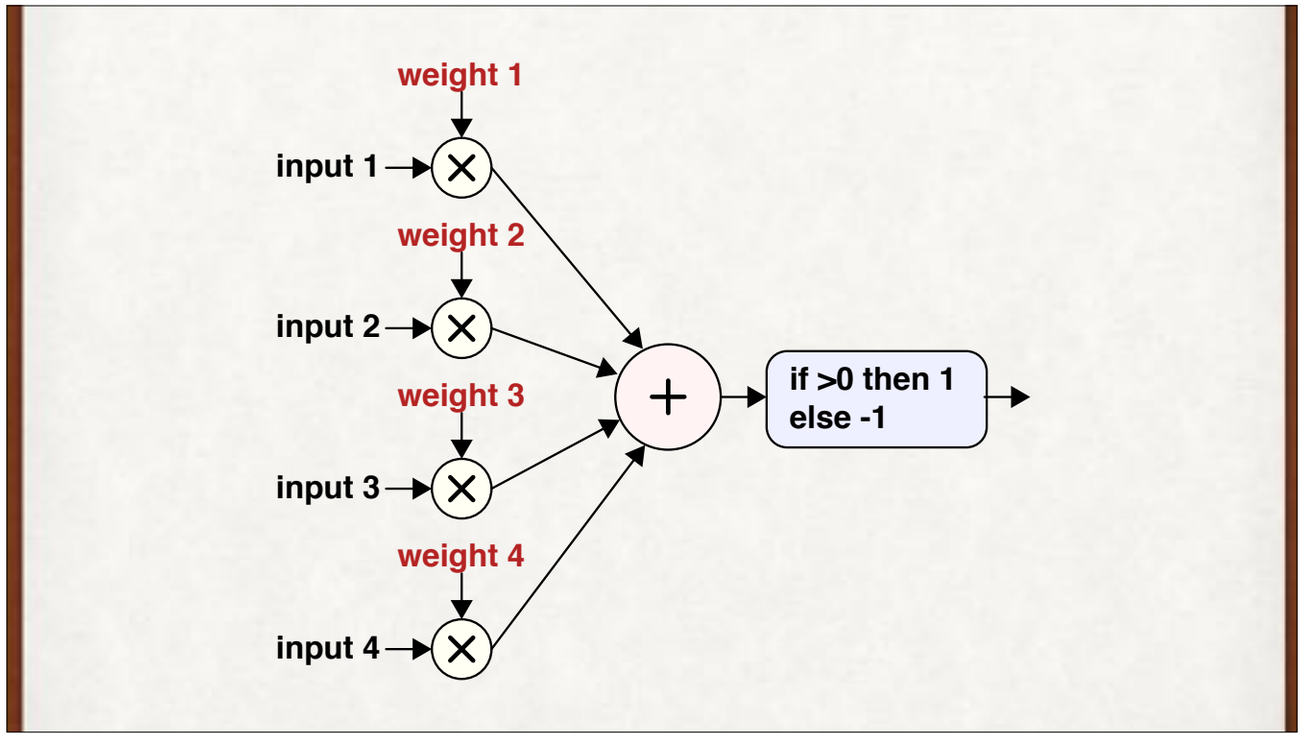
Our “artificial neuron”, or perceptron: inputs (numbers) come in. Each is “weighted” by another number. We add those all up, and threshold the result. The output is then -1 or 1 (some people use 0 and 1).



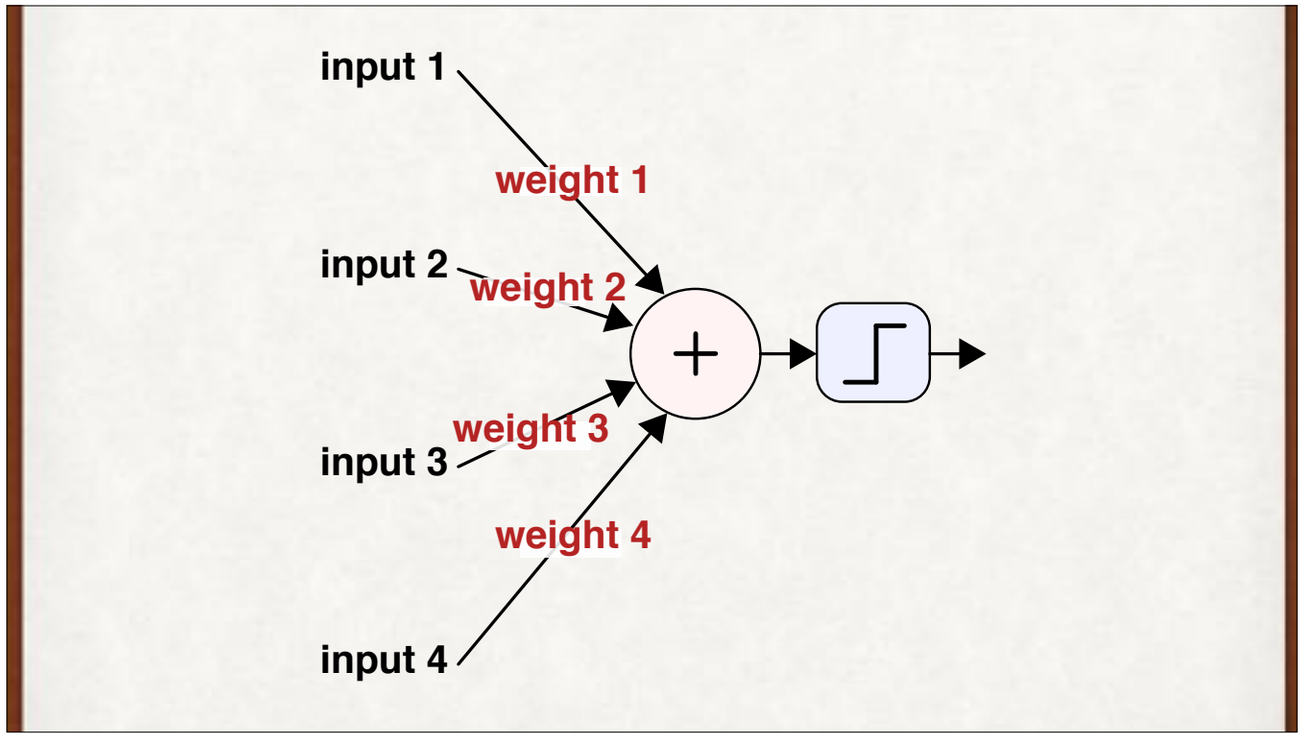
Our “artificial neuron”, or perceptron: inputs (numbers) come in. Each is “weighted” by another number. We add those all up, and threshold the result. The output is then -1 or 1 (some people use 0 and 1).



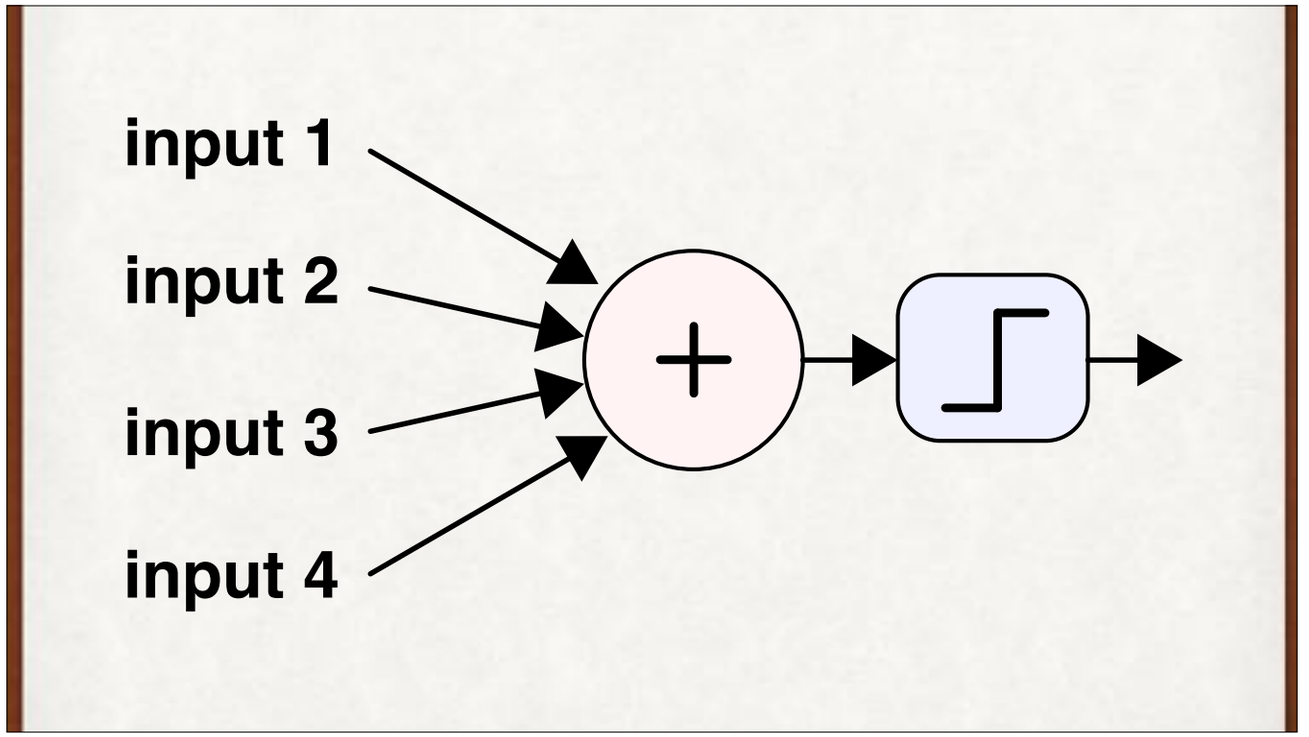
There’s a bias term. We can think of this as an input with a value of 1, and a weight that is the bias. So we can treat this term as just another input. This “bias trick” makes the code and math easier, so we’ll use it from now on. So the bias is always there, but we won’t draw it.



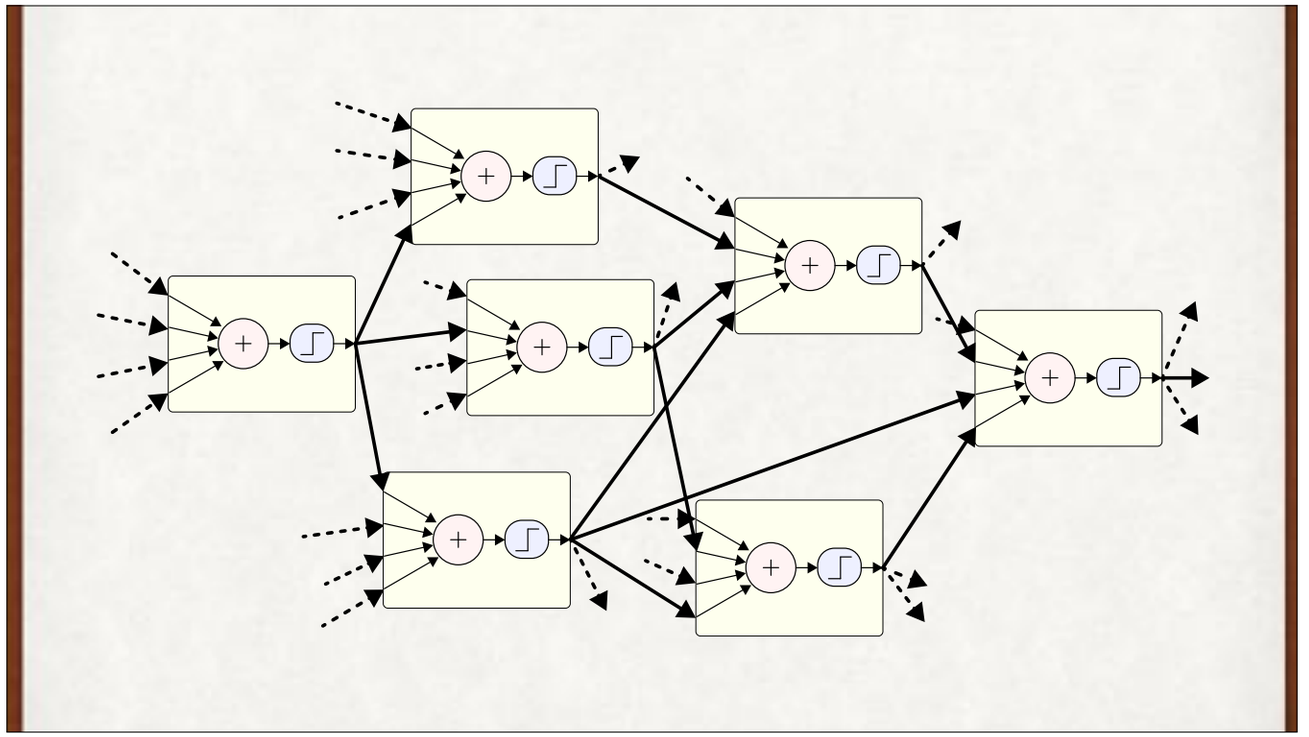
Our “artificial neuron”, or perceptron: inputs (numbers) come in. Each is “weighted” by another number. We add those all up, and threshold the result. The output is then -1 or 1 (some people use 0 and 1).



A simpler way to draw the last figure. The weights implicitly multiply the values on each “wire.”



An even simpler drawing. This is how a perceptron is normally drawn. Here, the weights are only implied. They are still there. It’s our job to imagine them. This is very important! Remember that the weights are there, on every input.



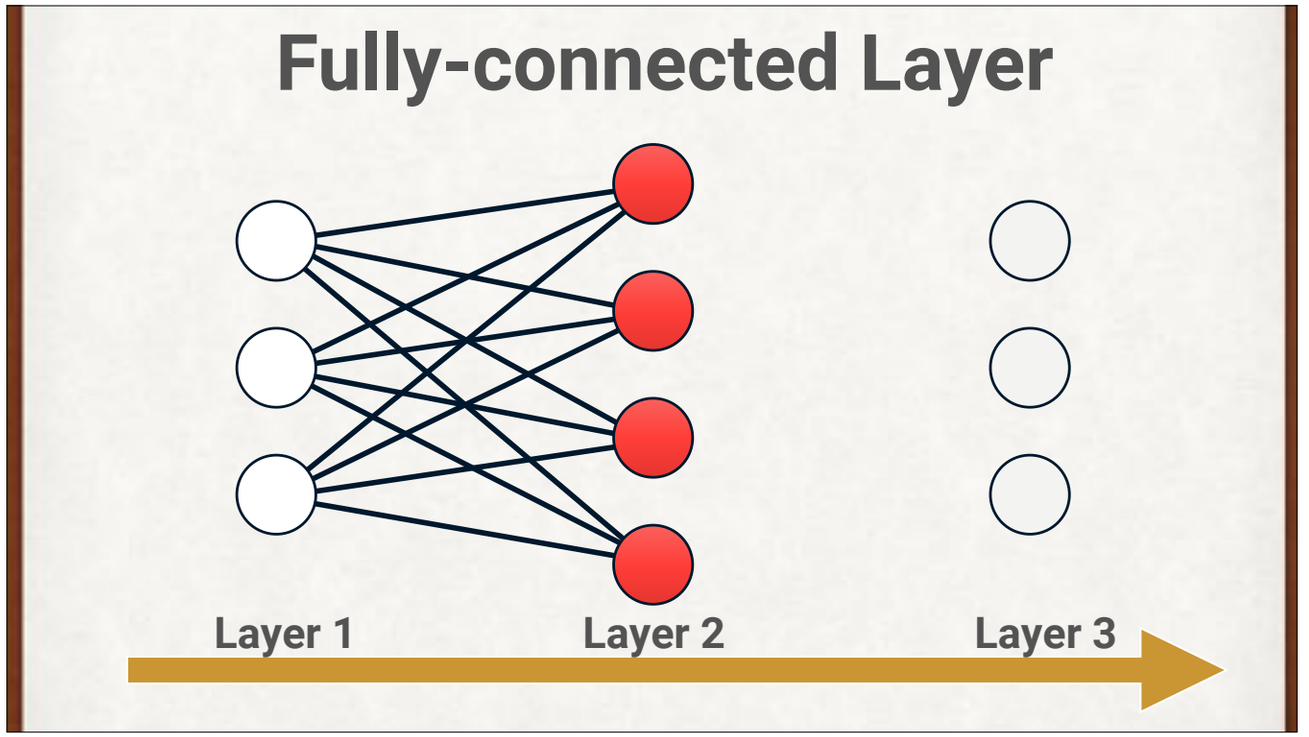
We can hook the output of each neuron into other neurons. A neural network!

Layers
make it deep

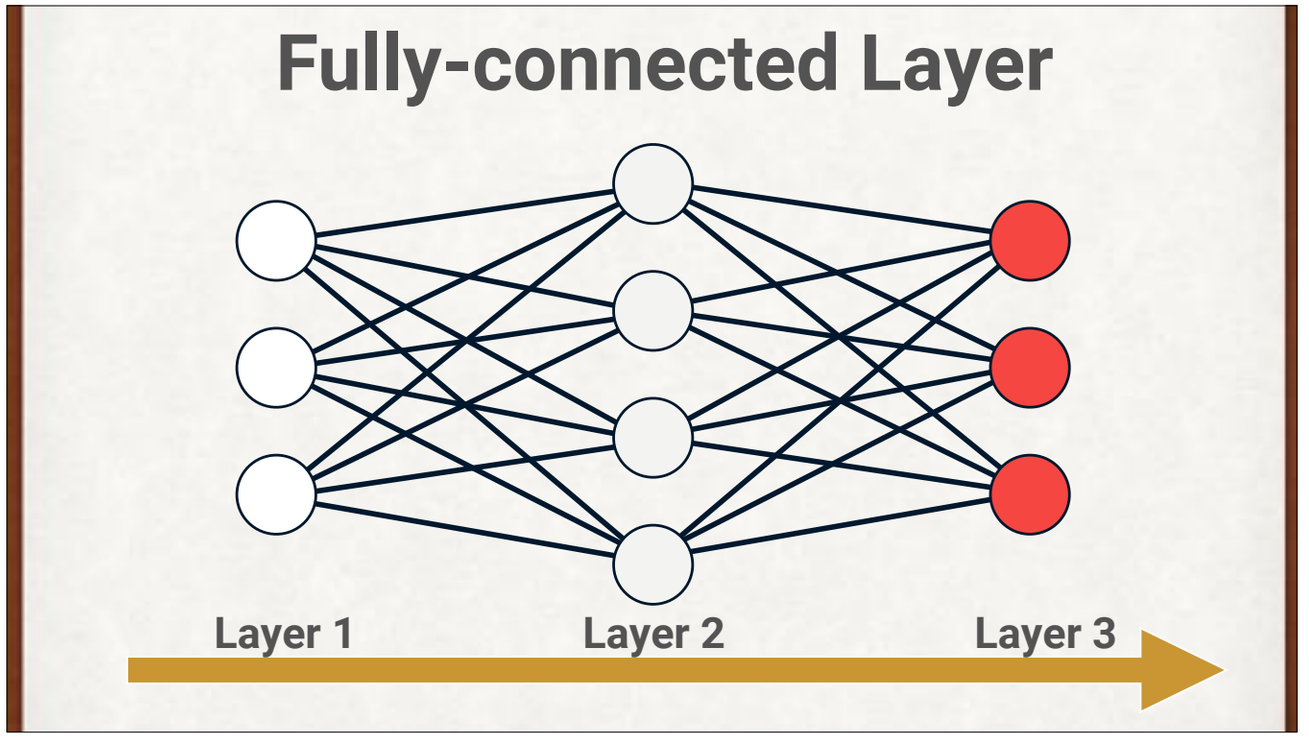
We’ll organize our neurons into layers. Many layers = “deep” network.

Fully-connected Layer

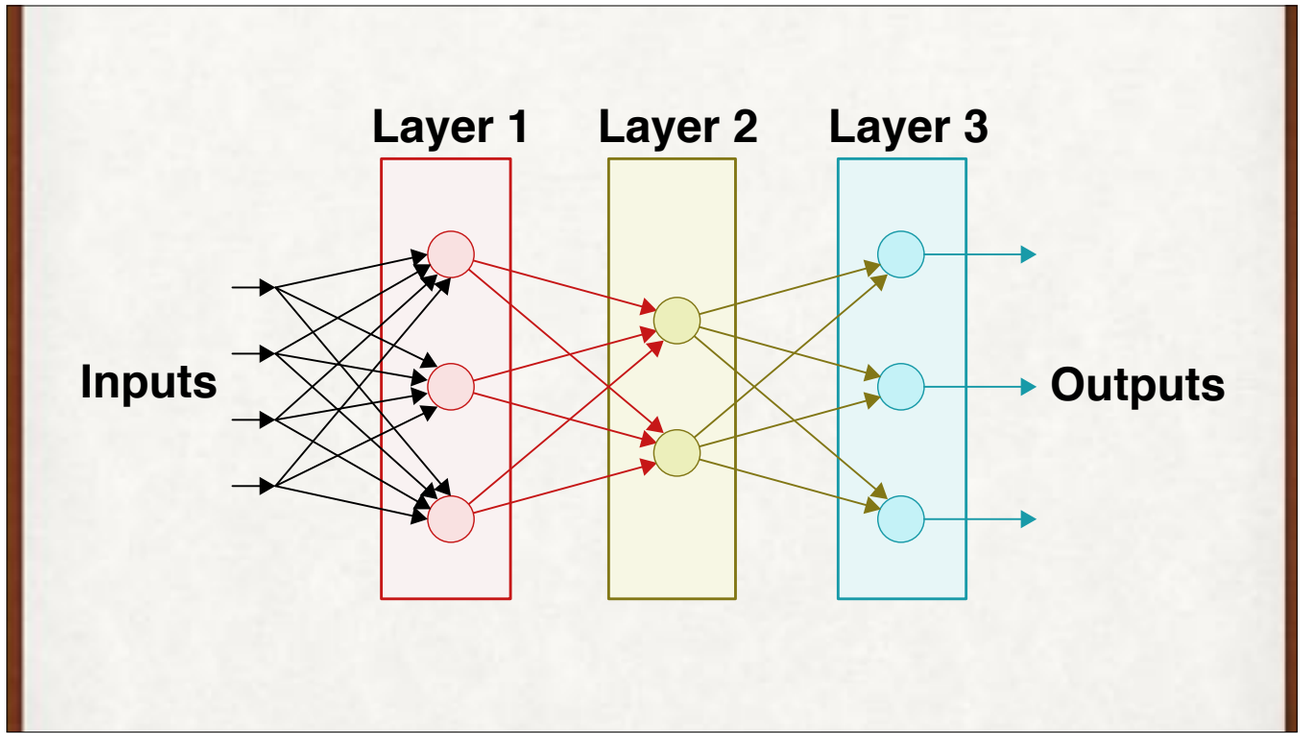
To learn from mistakes, we first have to know when we’ve made one.



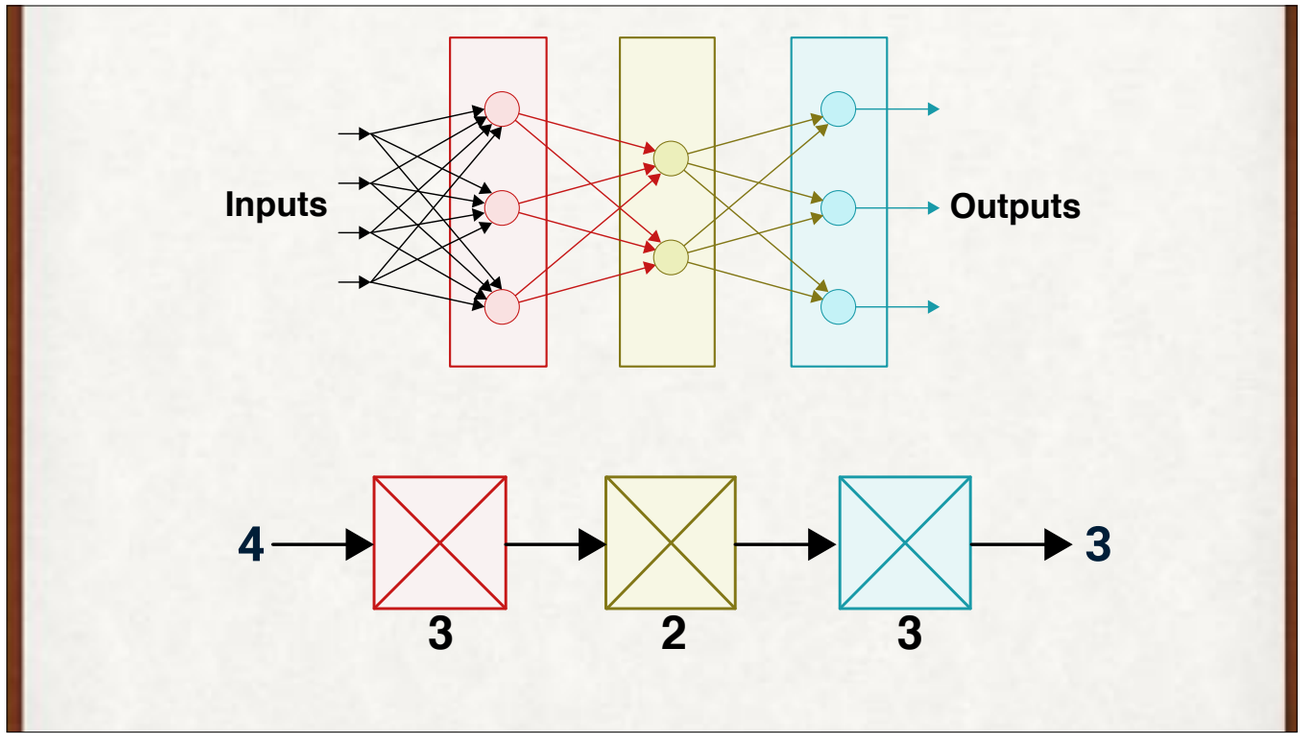
A fully-connected layer (in red) means that each neuron receives an input from every neuron on the previous layer.



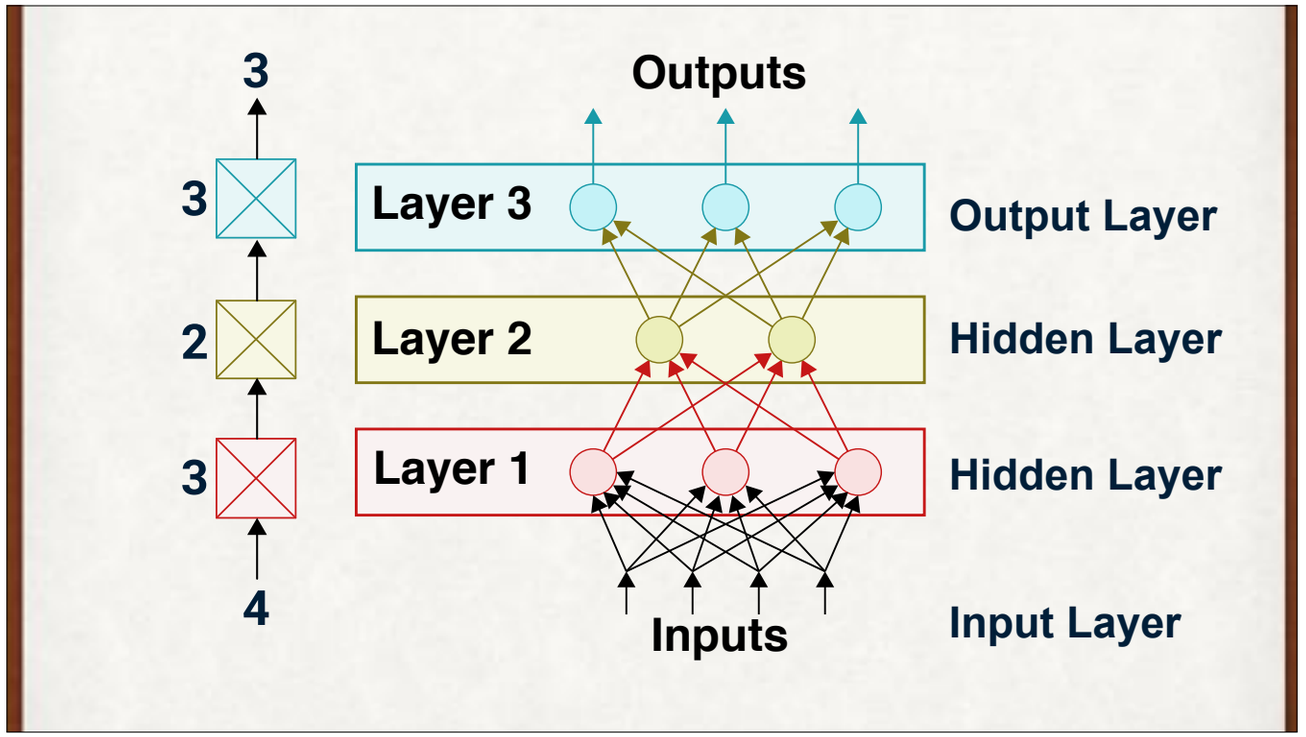
Here Layer 3 is fully-connected as well.



Drawing the network left to right.



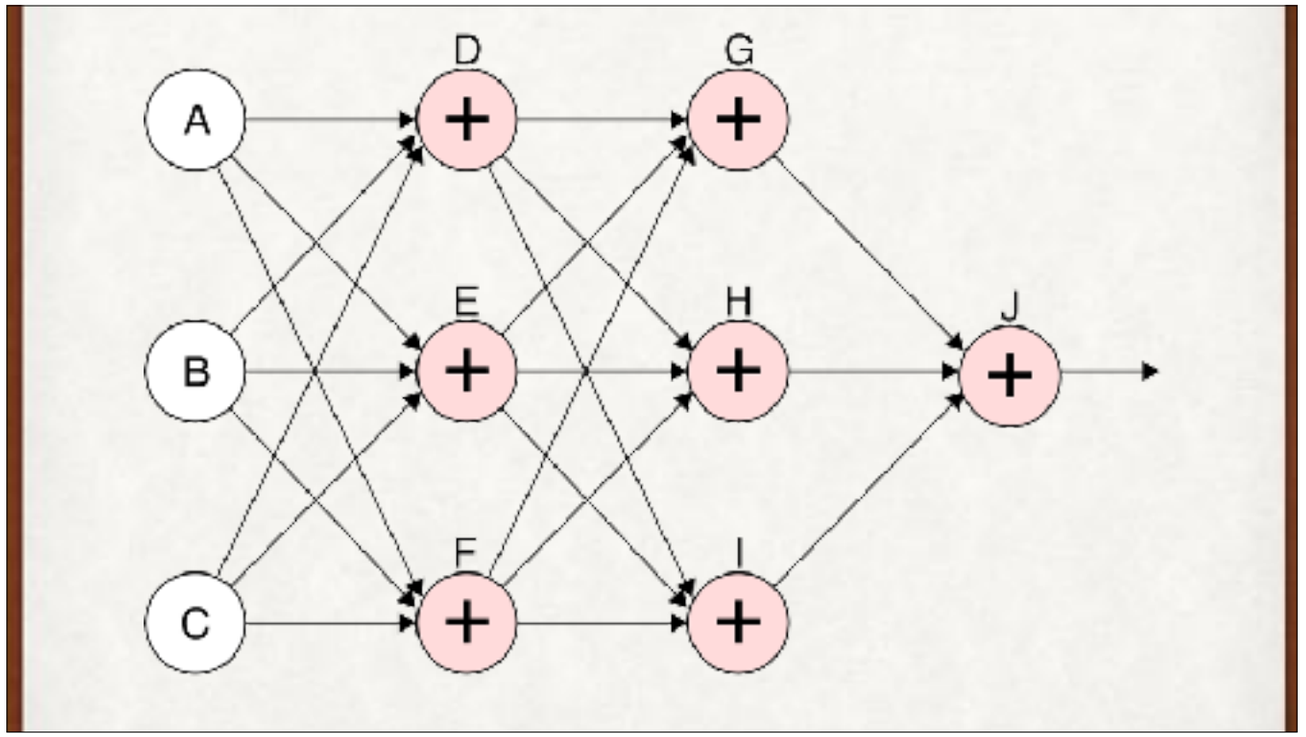
These are my icons for fully-connected layers. This is much easier to draw and interpret at a glance. The number of neurons is drawn beneath each icon.



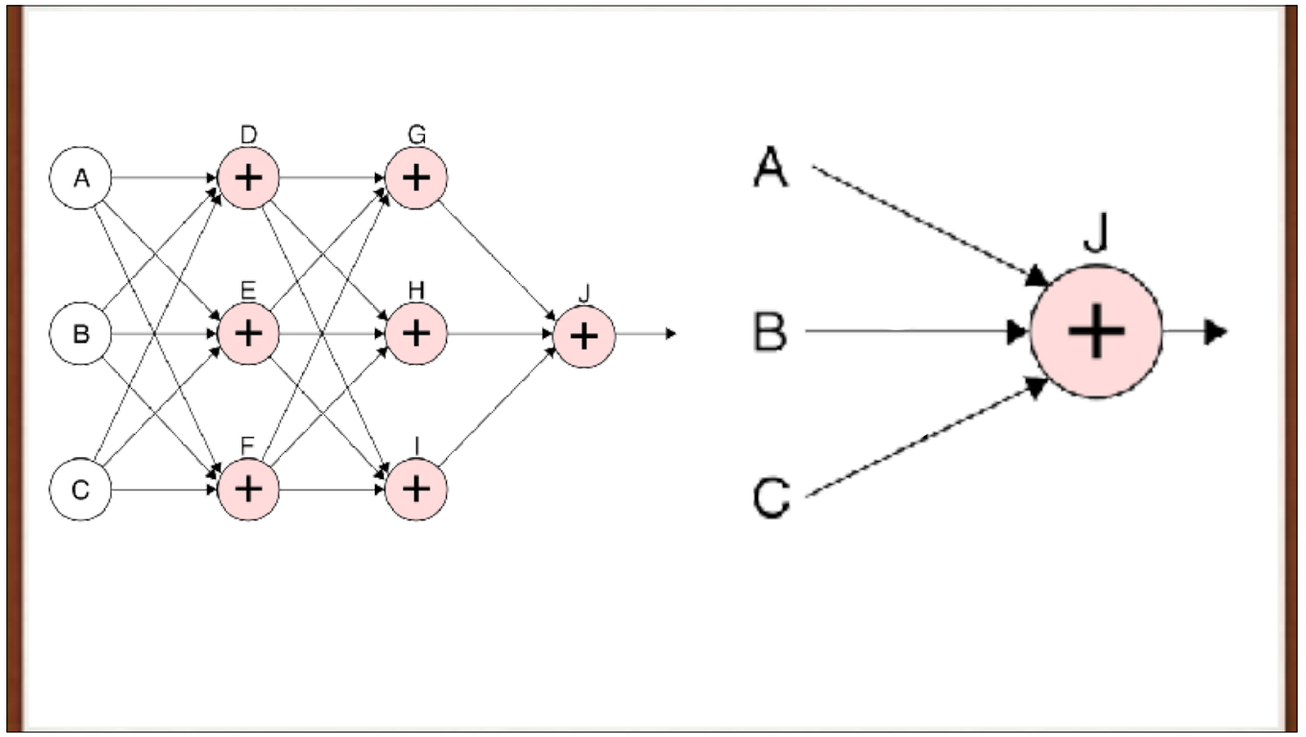
Here we have 3 **fully-connected** or **dense** layers of neurons. Here we're drawing data flowing bottom to top. The input layer is just a buffer, and is not counted. Layers 1 and 2 are "hidden" if we imagine looking at this network from the top or bottom, where we could only see the input and output layers. Layer 3 is the output layer. Note the asymmetry in the notation: the output layer has neurons and is counted. The input layer has no neurons (it's just a place to hold the data), and is not counted as a layer.

Preventing collapse

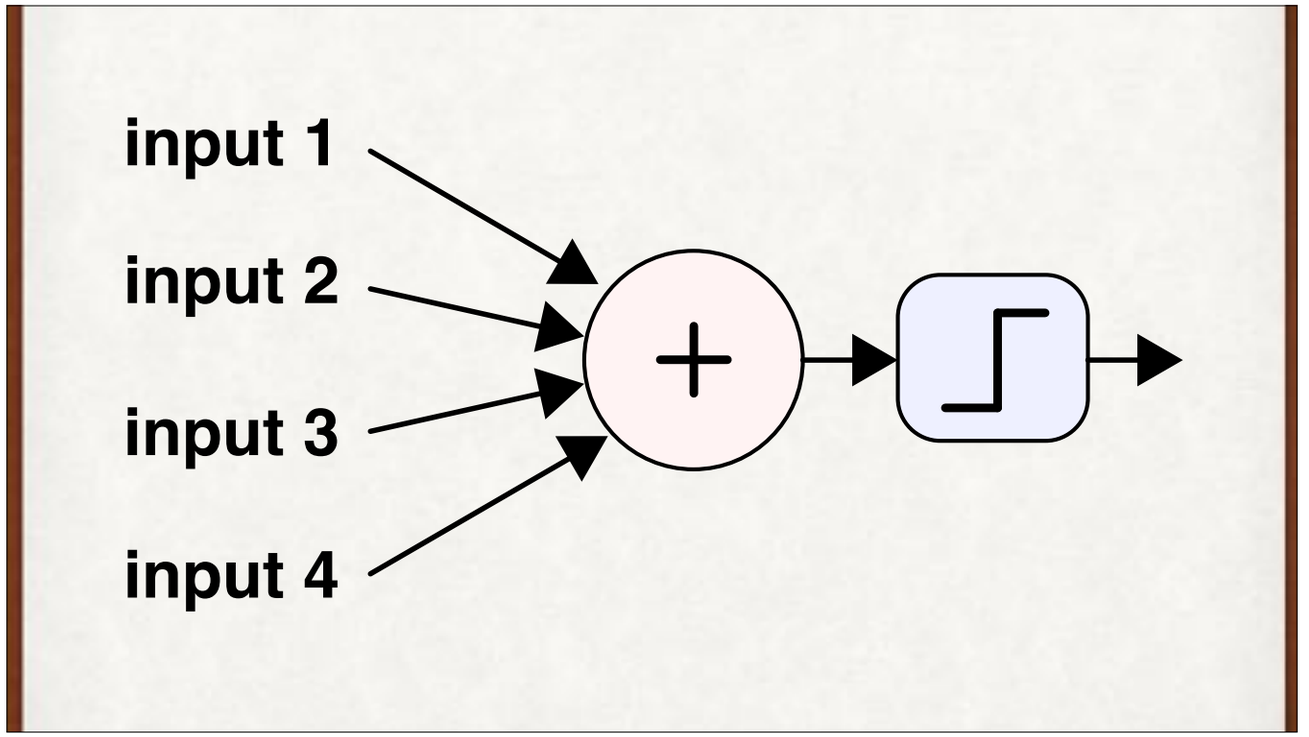
Networks without activation functions (the thresholds we've seen) can "collapse." That's bad, because it reduces them to just a single neuron. How's that happen?



Here’s a network of neurons. For the moment, there’s no threshold on these. So values come in, get weighted, summed, and that’s the output.



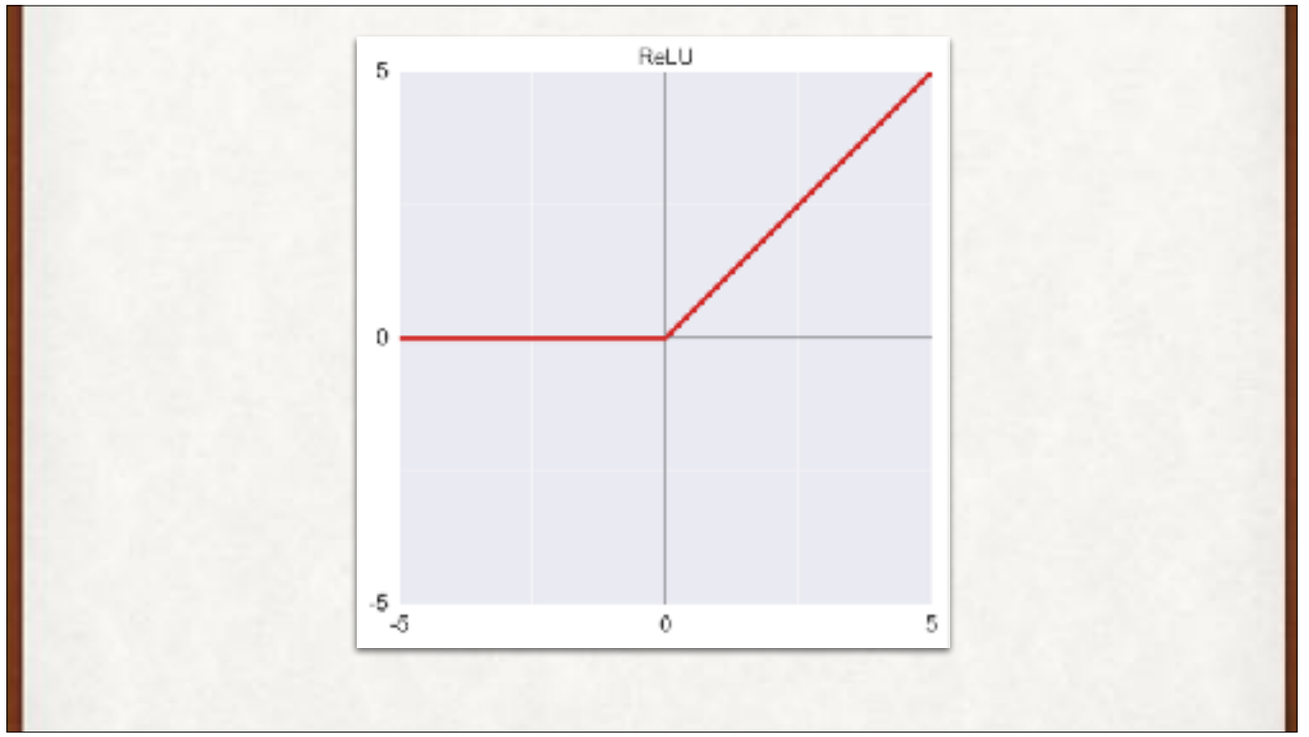
If we do the algebra, the whole thing “collapses.” The whole network is equivalent to this one neuron on the right. Our network is no more powerful than a single neuron. Not only is that a waste of time and resources, it’s also not a very “smart” network.



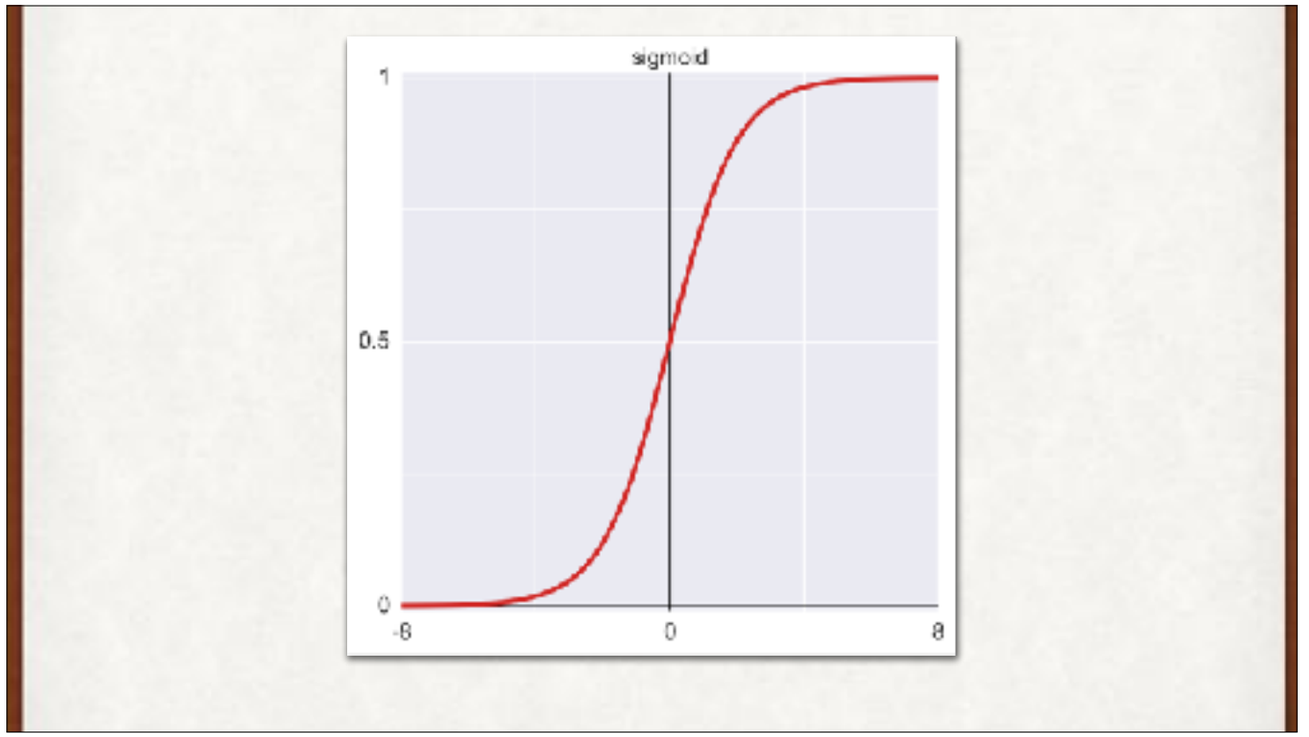
The threshold at the end is the “non-linearity” that prevents collapse. Without the threshold, we have a single linear equation, and one neuron. With the threshold, we have a network of neurons that don’t combine and collapse.

Activation functions

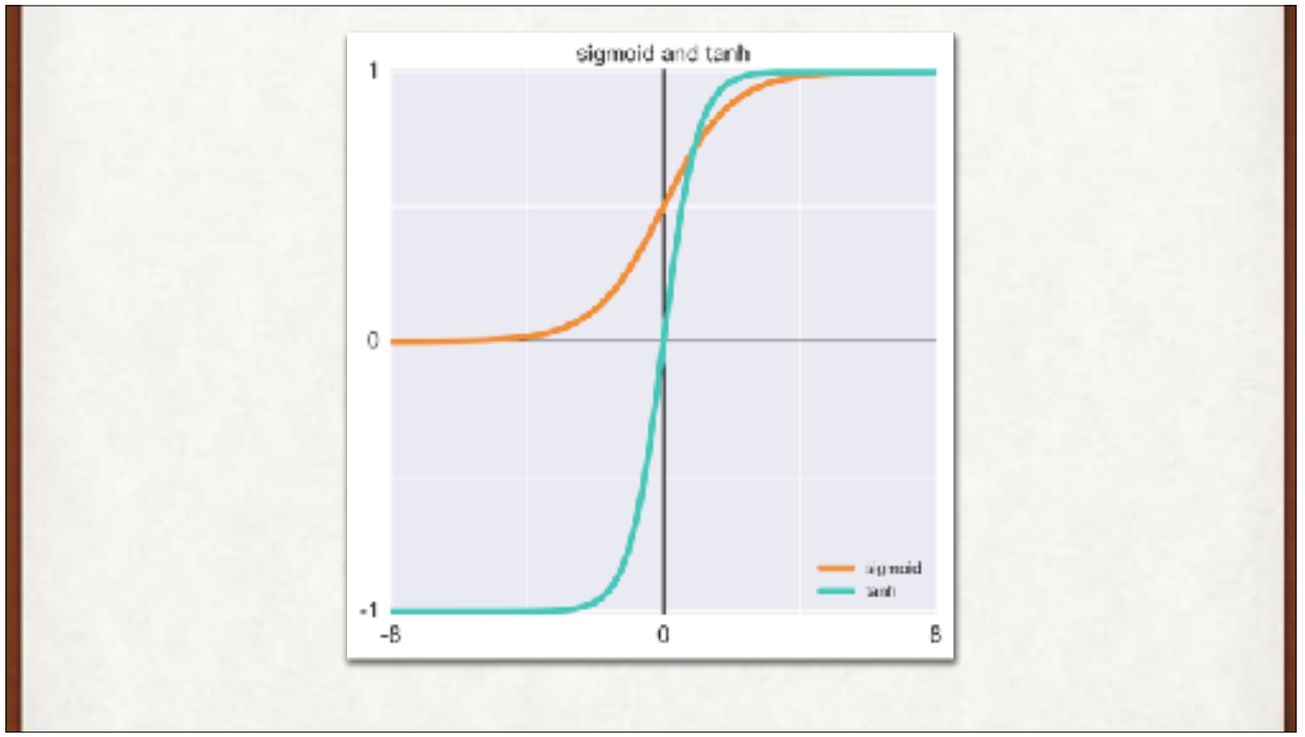
The threshold is also called an “activation function”. Lots of non-linear activation functions have been proposed and experimented with.



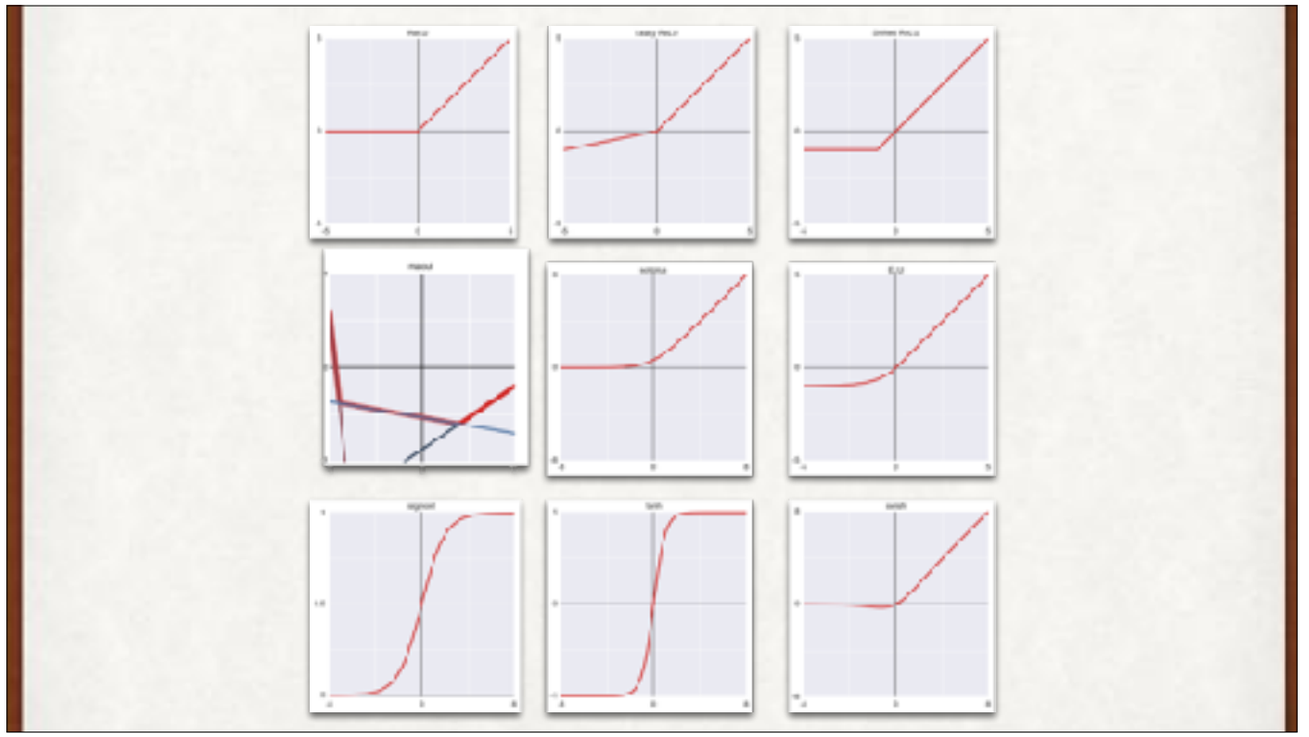
A Rectified Linear Unit (ReLU) is the most popular, and often the default choice in deep learning libraries.



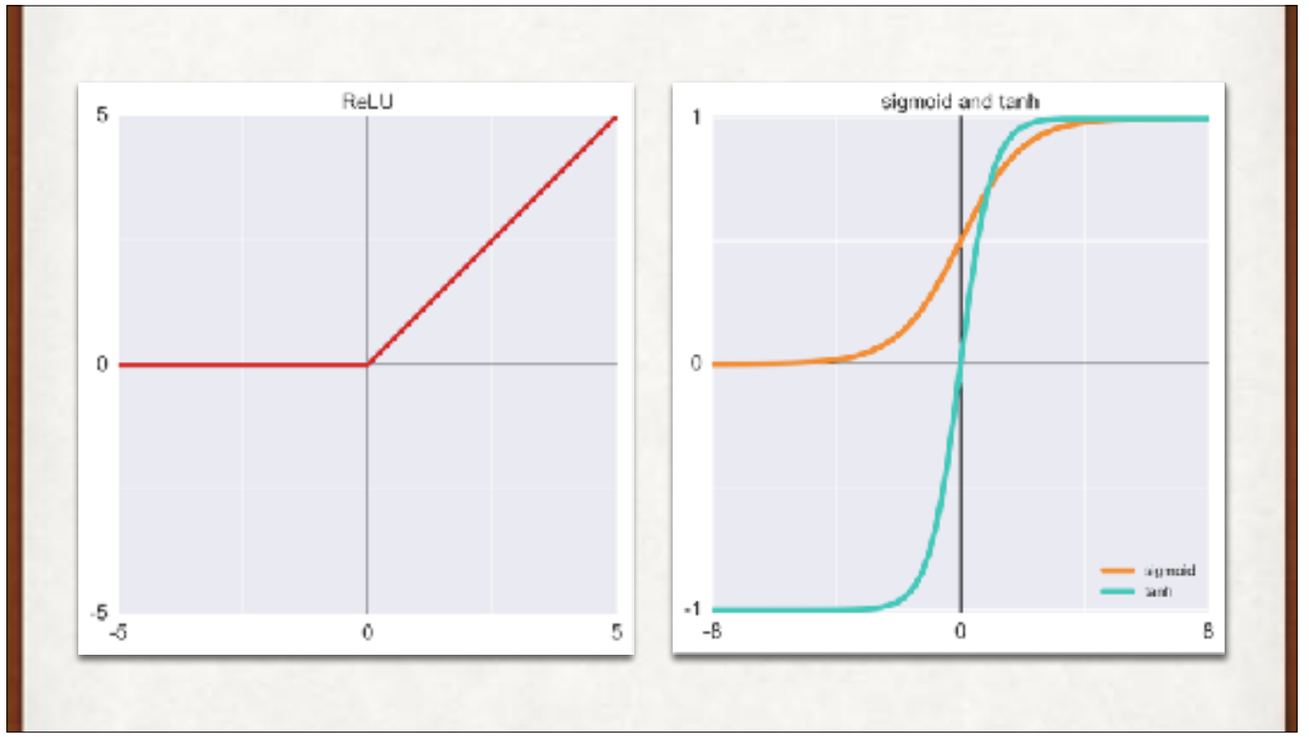
Sigmoid curve. It’s like a threshold, but smooth and continuous. We sometimes say it “squashes” the input, which can be any real number, to the range [0,1].



tanh, or hyperbolic tangent, or inverse-tangent. It’s a lot like the sigmoid, but a little steeper, and outputs [-1, 1] rather than [0, 1]



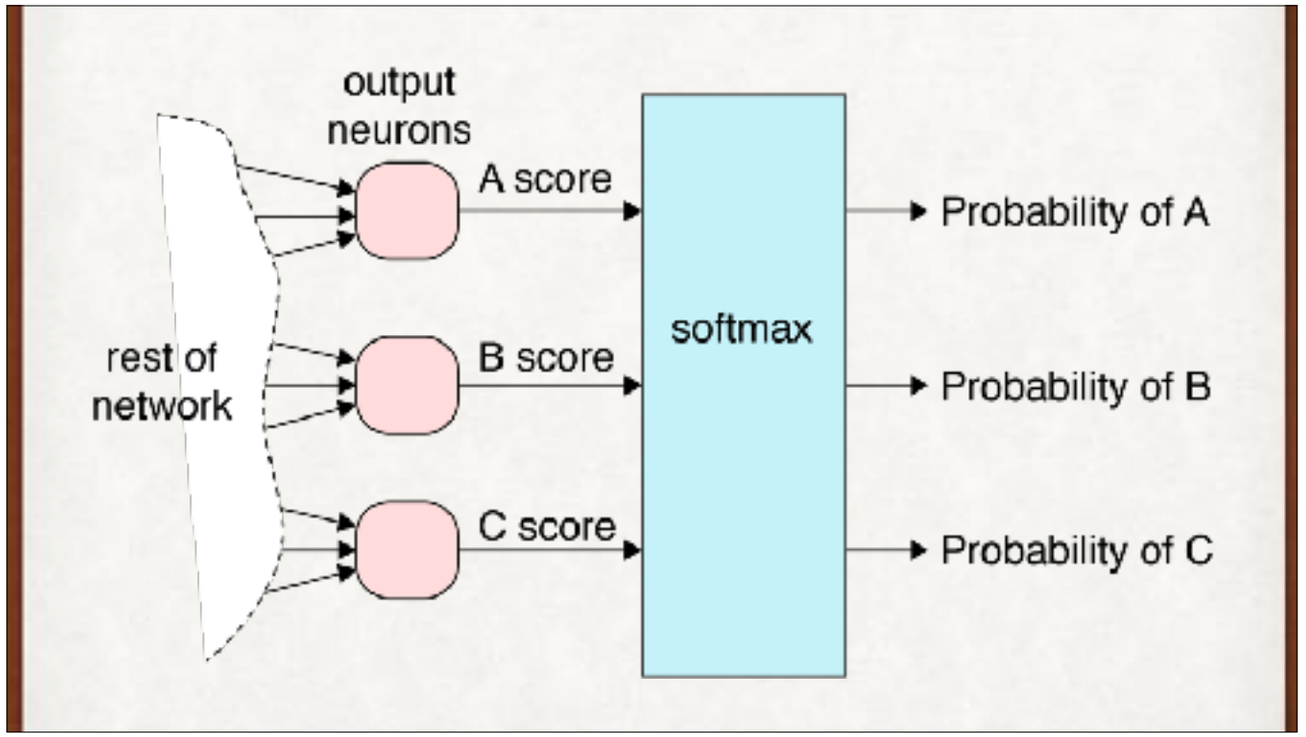
An activation function menagerie. And there are more!



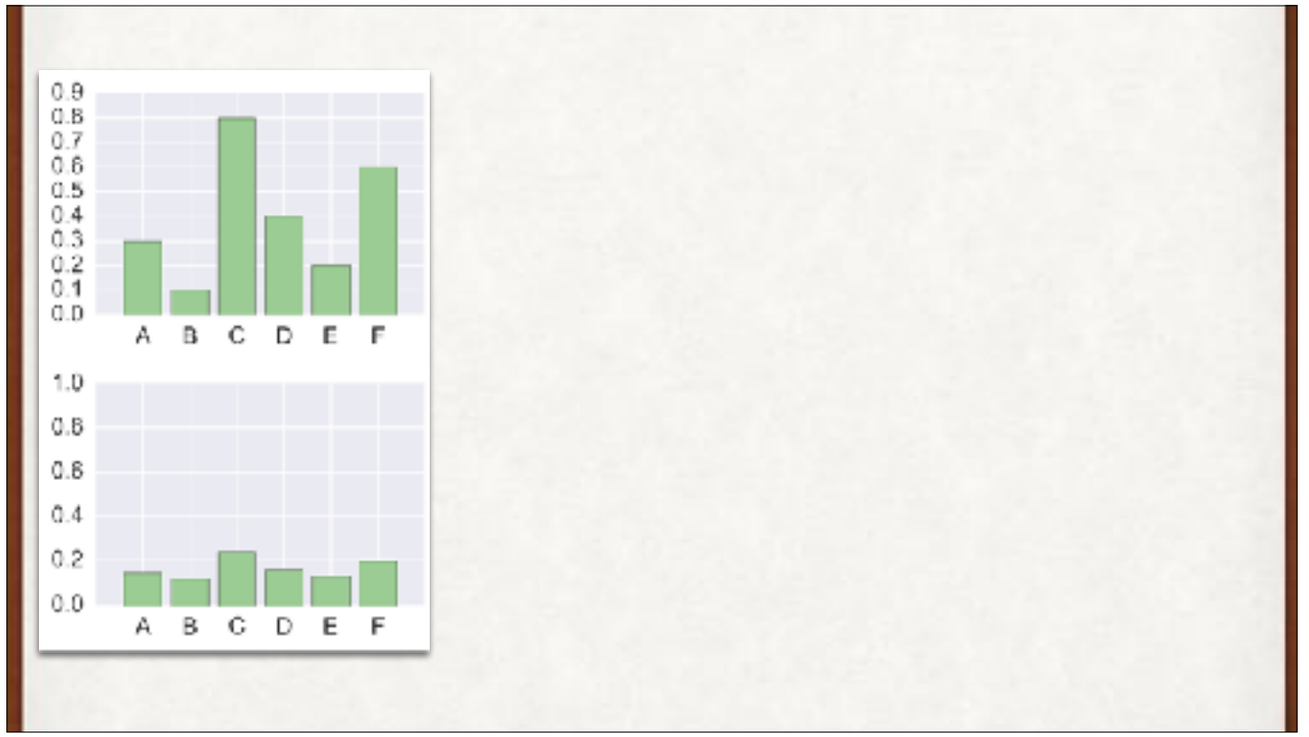
Picking the right one is partly theory, partly hunch and experience, and partly experiment. Usually all the neurons on any given layer share the same activation function. These are the most popular. The output layer of a classifier uses…

softmax

softmax. This is a little piece of mathematics. Not quite an activation function, because it takes as inputs all the outputs of the output layer neurons simultaneously.



Unlike the functions we've seen so far, softmax takes in multiple inputs - usually all the outputs of the last layer of the network. It munches on them and produces a probability for each class.



The results of softmax. Top, a set of inputs to softmax. Bottom, the output. The sorting stays the same from biggest to smallest, but each value is adjusted by a different amount. The outputs all add up to 1. Up top, we can only say that category C is more likely than, say, B, but we can’t say by how much. The numbers are not easily interpreted as telling us anything except the order in which the different classes are being assigned. Beneath, we have actual probabilities, so we can say that C is twice as likely as B, and maybe 30% more likely than D. We can’t say much about the meanings of the heights of the bars before softmax, except to identify their sorting order.

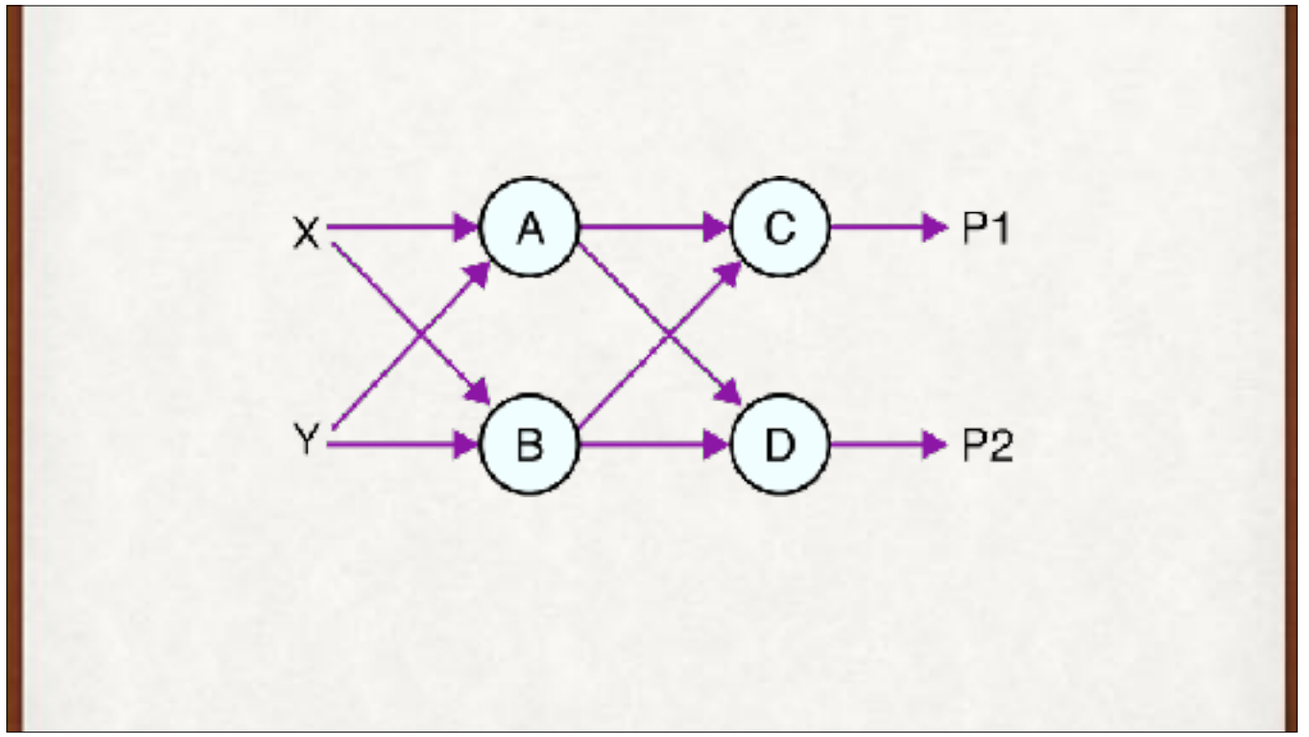
The results of softmax. Top, a set of inputs to softmax. Bottom, the output. The sorting stays the same from biggest to smallest, but each value is adjusted by a different amount. The outputs all add up to 1.

The results of softmax. Top, a set of inputs to softmax. Bottom, the output. The sorting stays the same from biggest to smallest, but each value is adjusted by a different amount. The outputs all add up to 1.

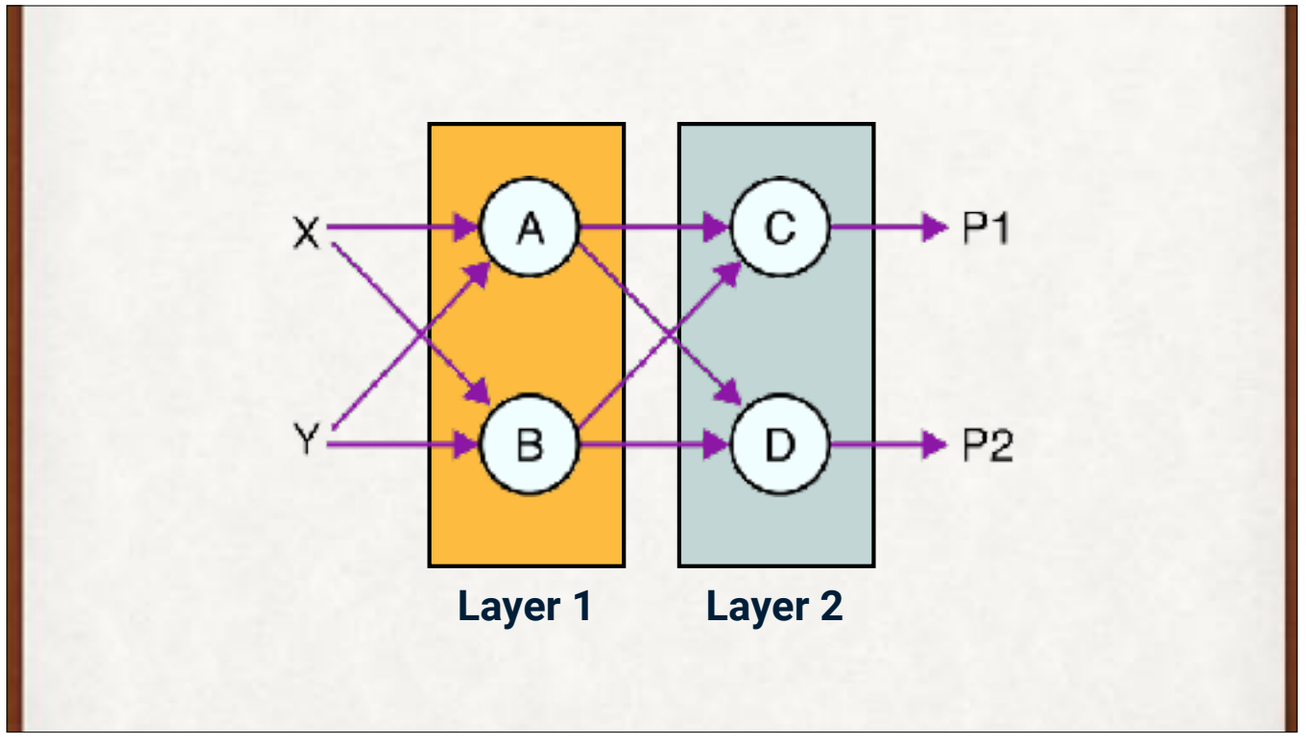
A cleanse of our mental palette. Some visual sherbet between courses. Let’s talk about something new.

Fixing network mistakes

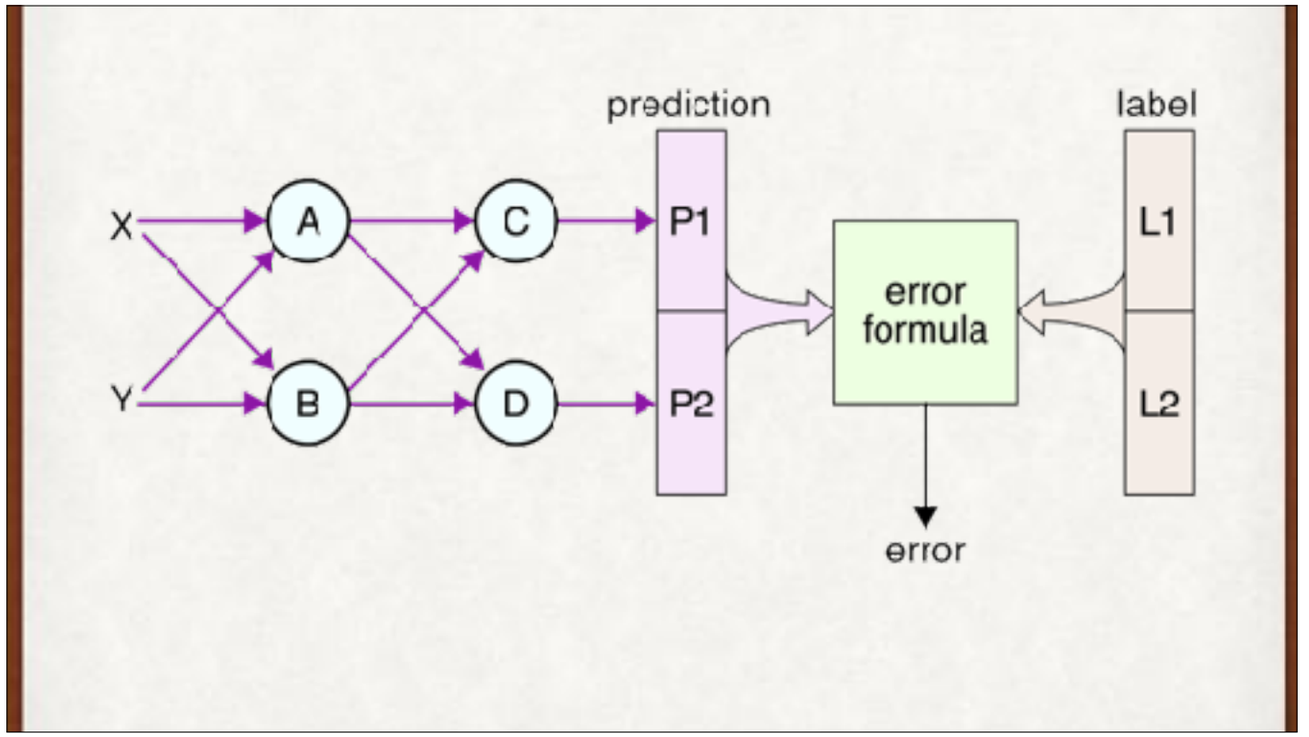
To learn from mistakes, we first have to know when we’ve made one.



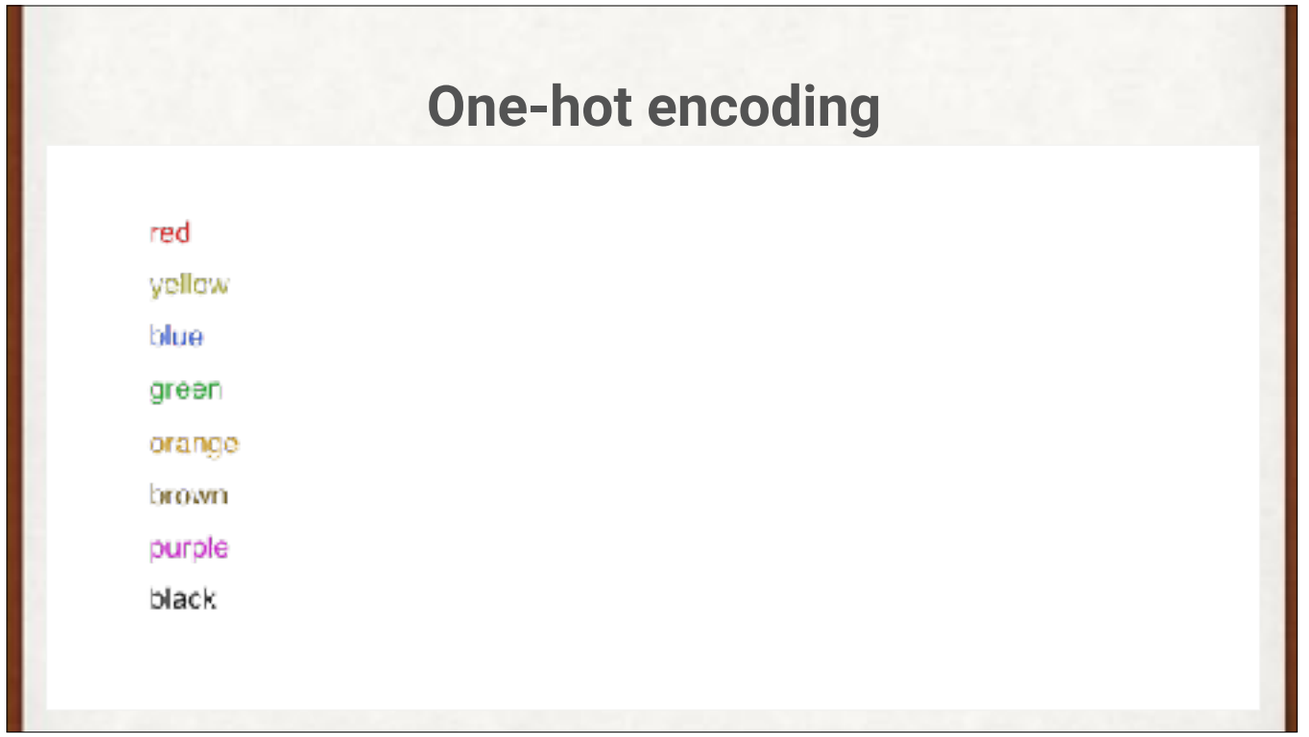
A tiny neural network. Let’s imagine there’s a softmax at the end.



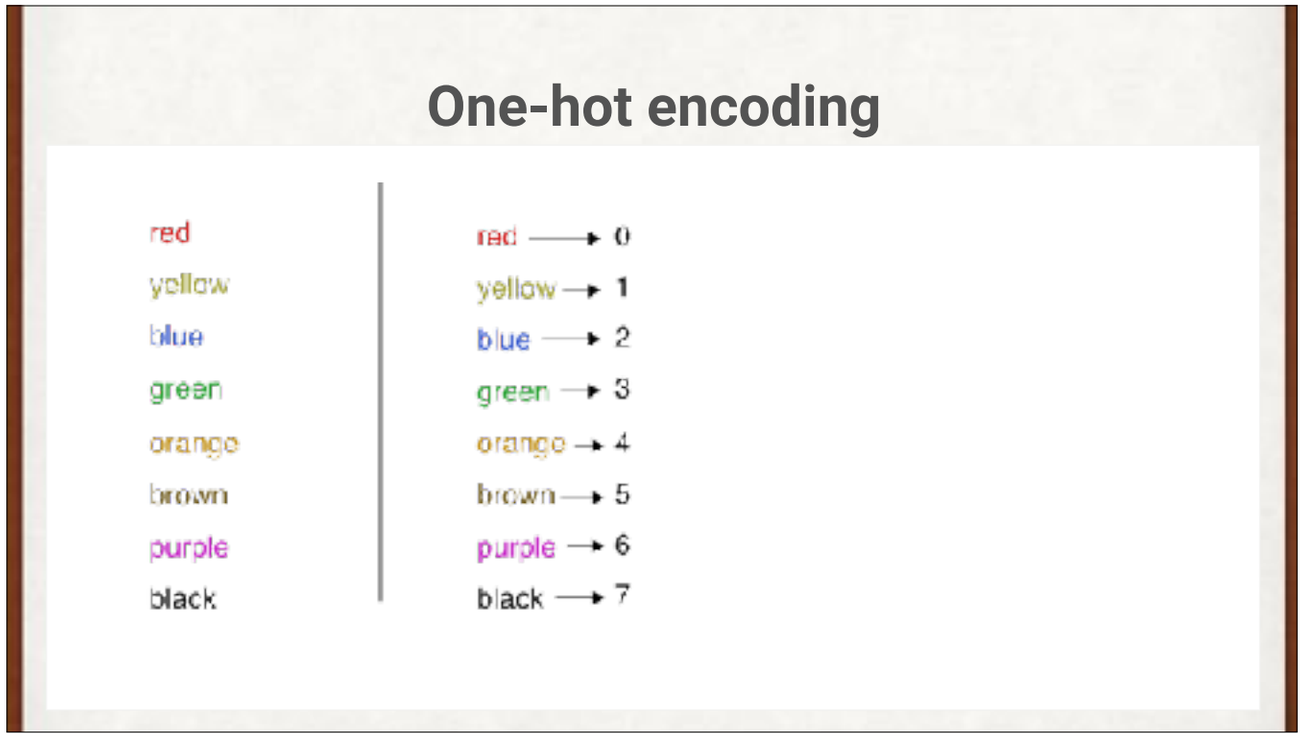
A tiny neural network.



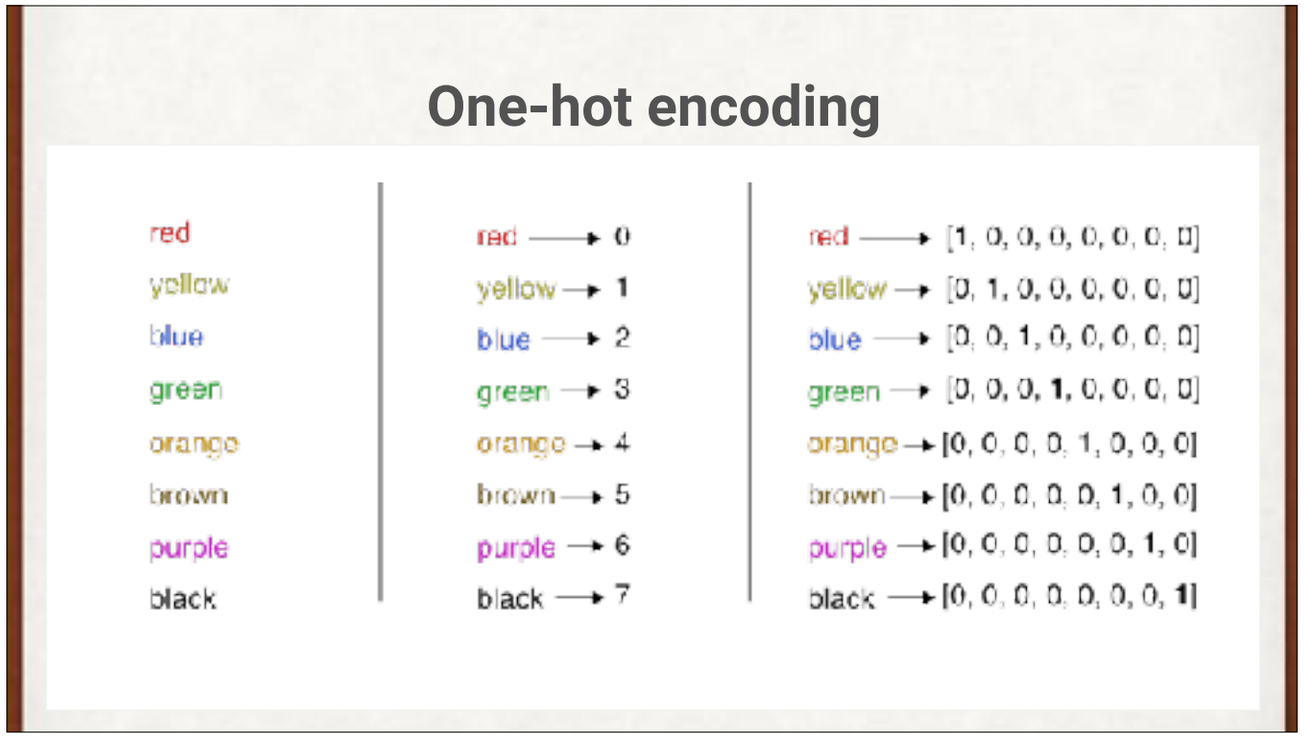
The output of the net is the “prediction”, here for two classes. Let’s suppose that we have softmax applied to the outputs of C and D. We compare the predicted probabilities to the correct answer in the “label” to find our error.



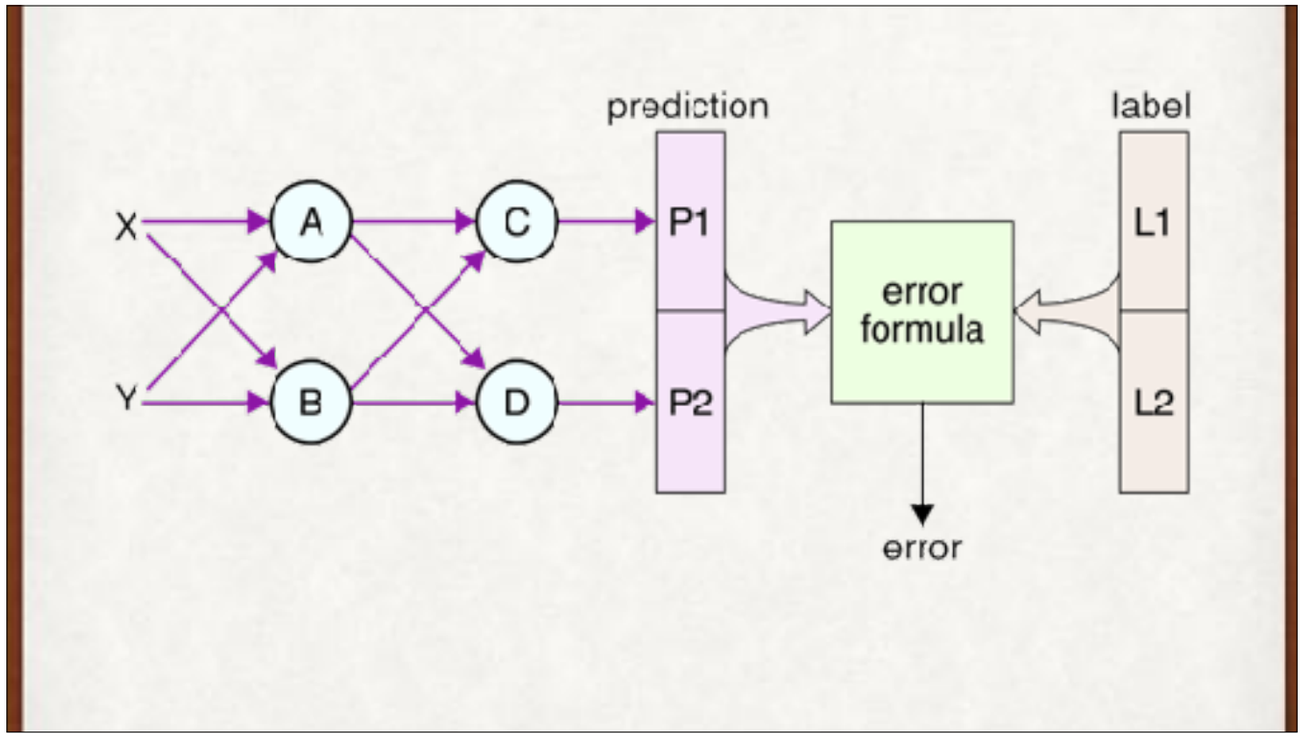
Since the output is a list of probabilities, we can convert our label into the same thing. Here, the probability of each class is 0, except for the correct class, which is 1. We call this one-hot encoding, which lets us turn numbers into labels. These are the 8 colors in the very first box of Crayola crayons.



Since the output is a list of probabilities, we can convert our label into the same thing. Here, the probability of each class is 0, except for the correct class, which is 1. We call this one-hot encoding, which lets us turn numbers into labels.



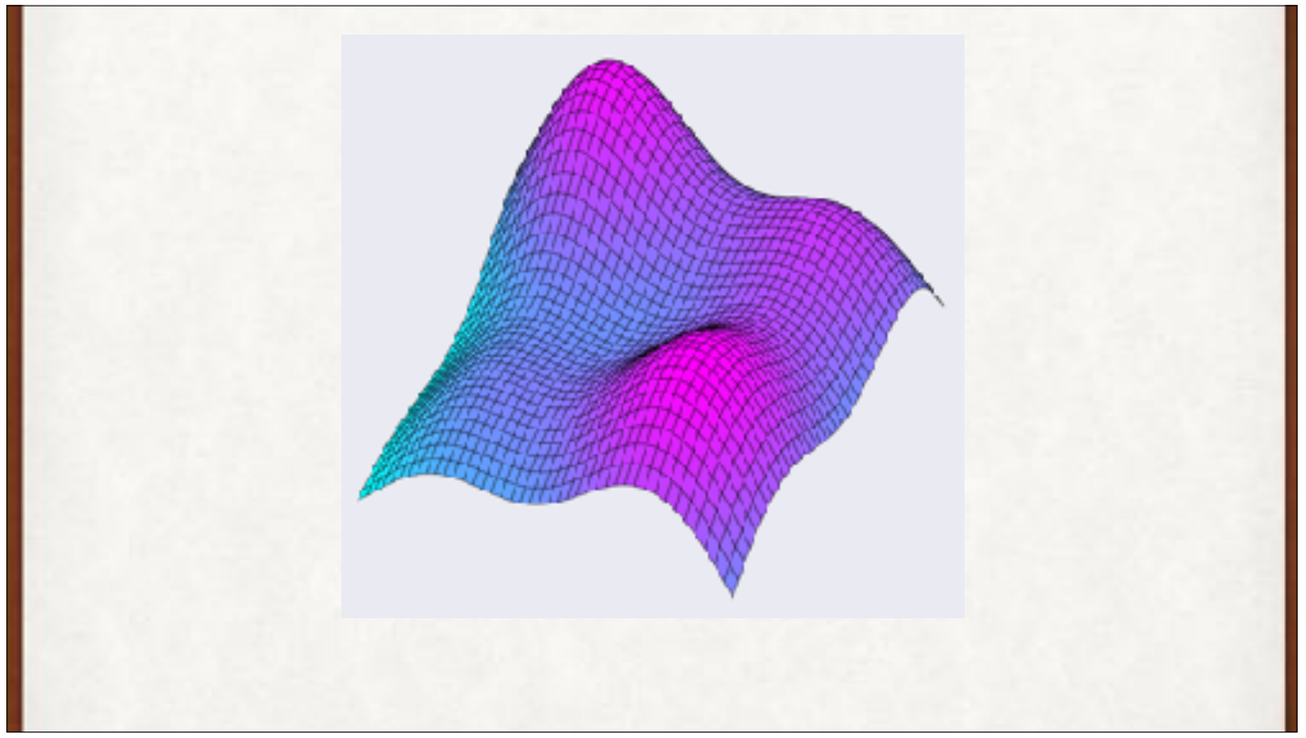
Since the output is a list of probabilities, we can convert our label into the same thing. Here, the probability of each class is 0, except for the correct class, which is 1. We call this one-hot encoding, which lets us turn numbers into labels.



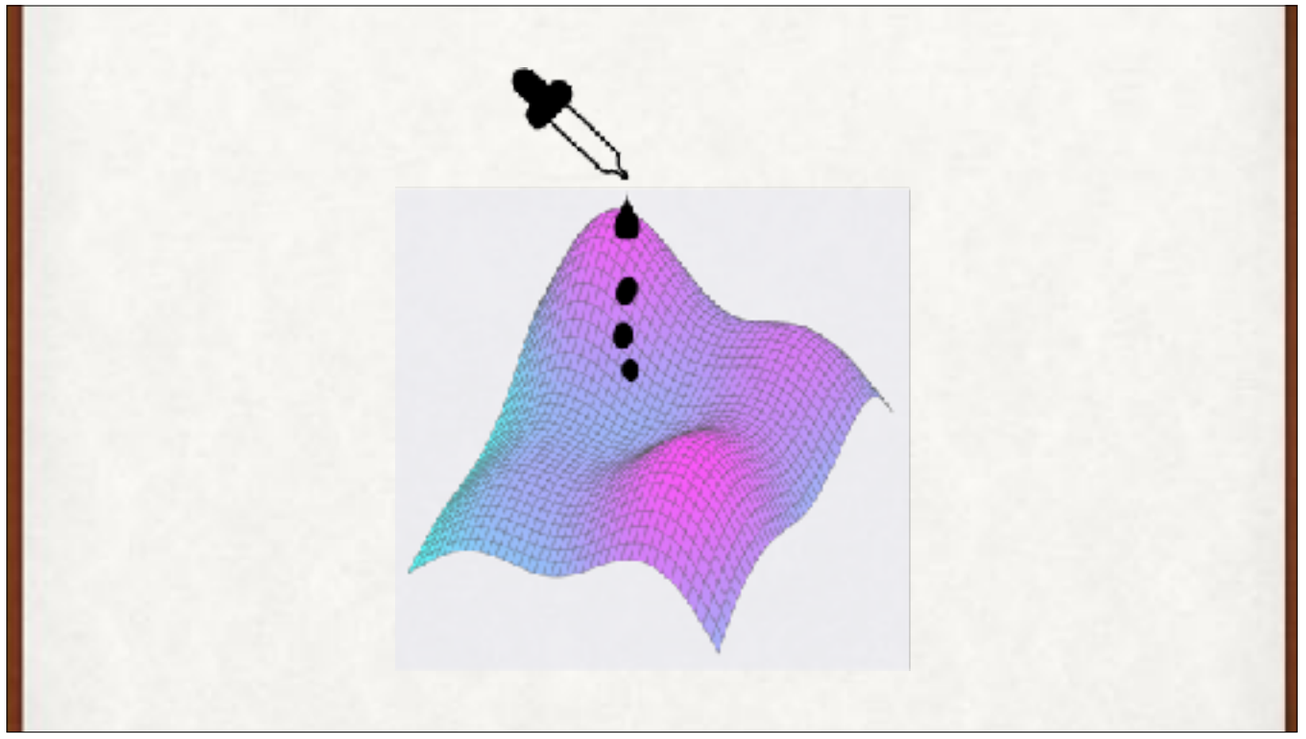
Now we can compare our 2 output values with the 2 values in the label to get our error.

Learning from mistakes: Gradient Descent

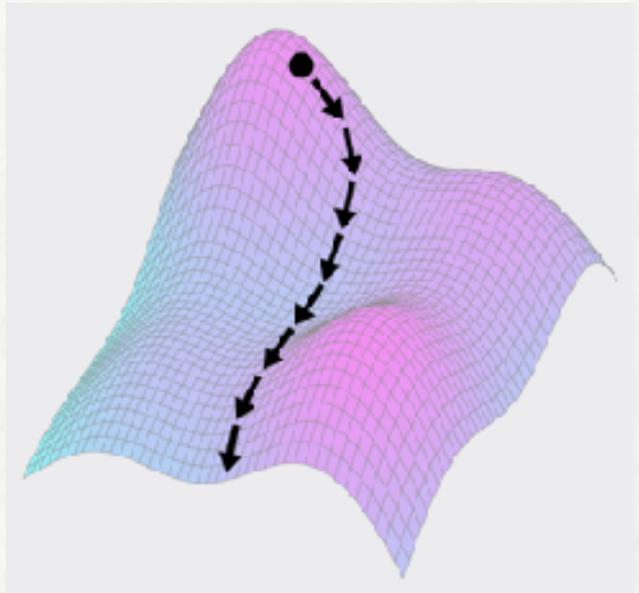
Learning from misteaks is a good idea. Learning from misstakes. From mistakes. We'll look at the error as a curve, where we plug in the weights in our network, and get back an error. We'll want to adjust the weights to make the error go down. The standard, and super useful, way to look at this is to consider the error to be a curve (or surface), and then we look for the weights that result in the smallest value for this curve (or surface).



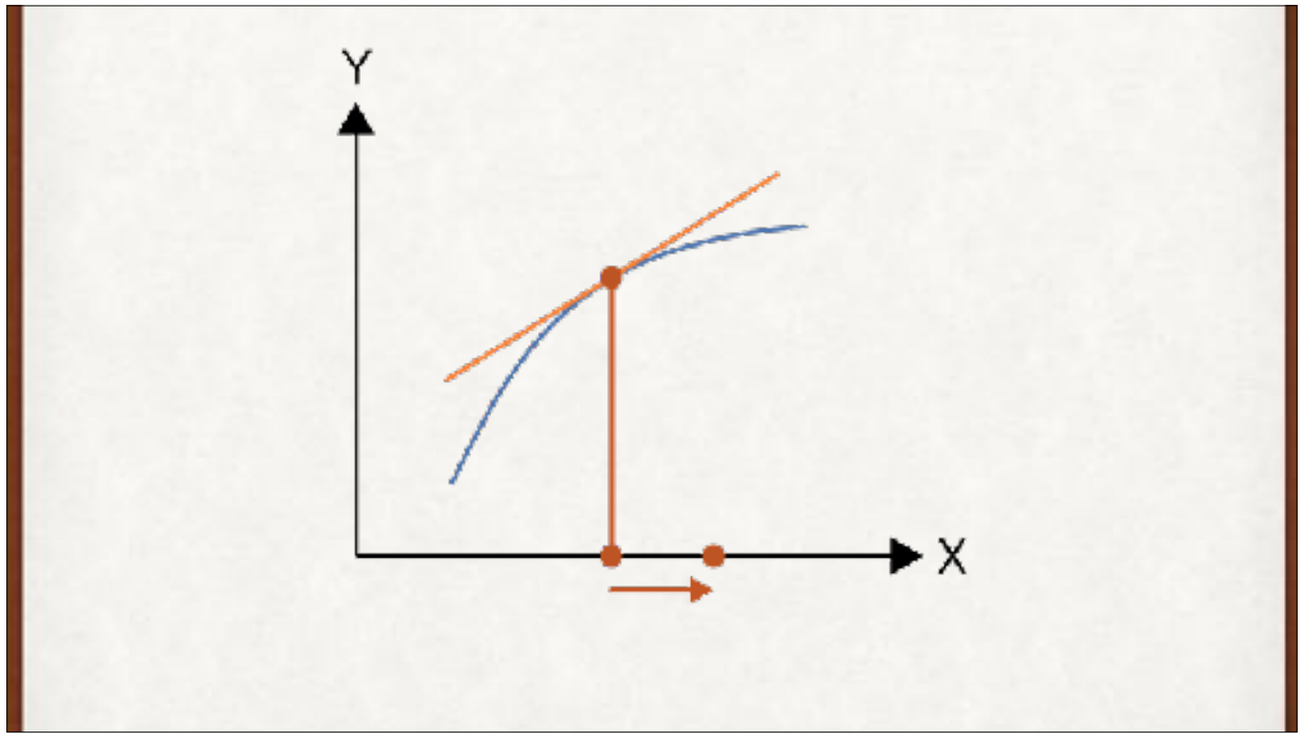
It works in 3D (with 2D surfaces, or more properly, height fields, since there’s just one value above for each (x,y) point below the surface).



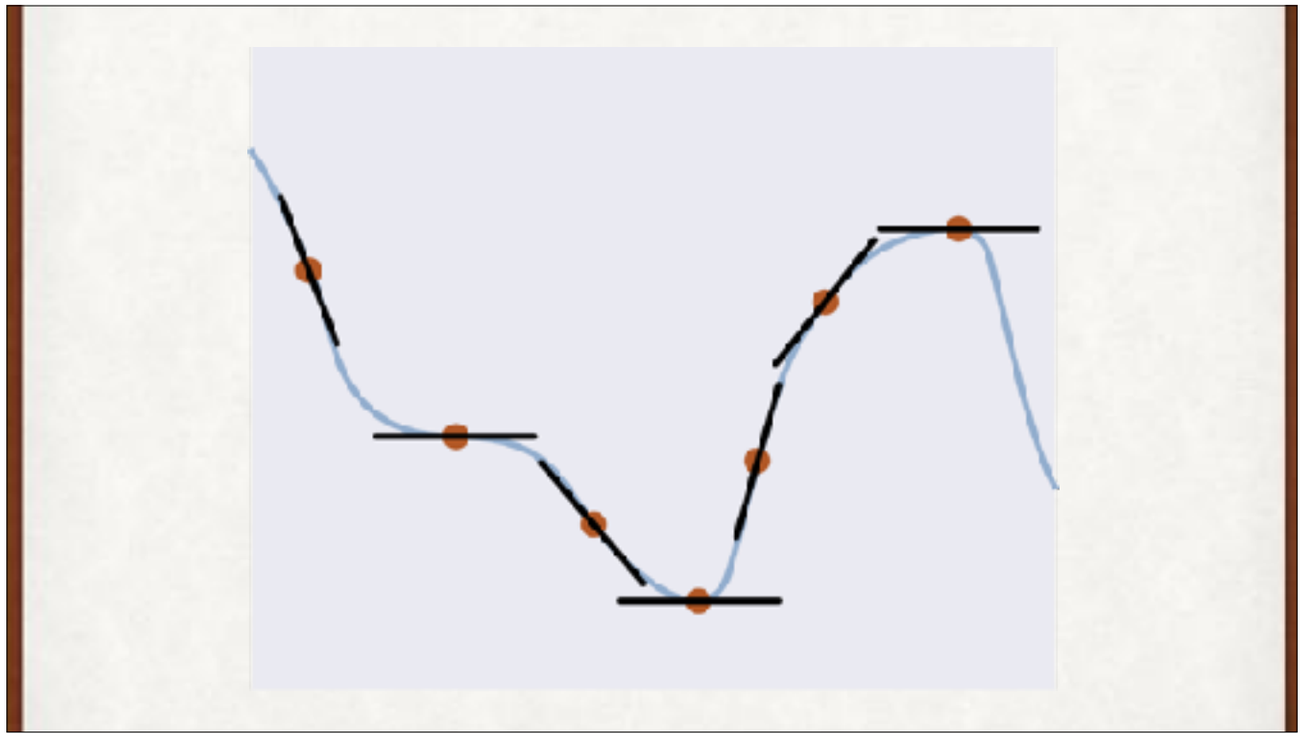
A natural example of finding the gradient is watching the path of water dropped on the surface.



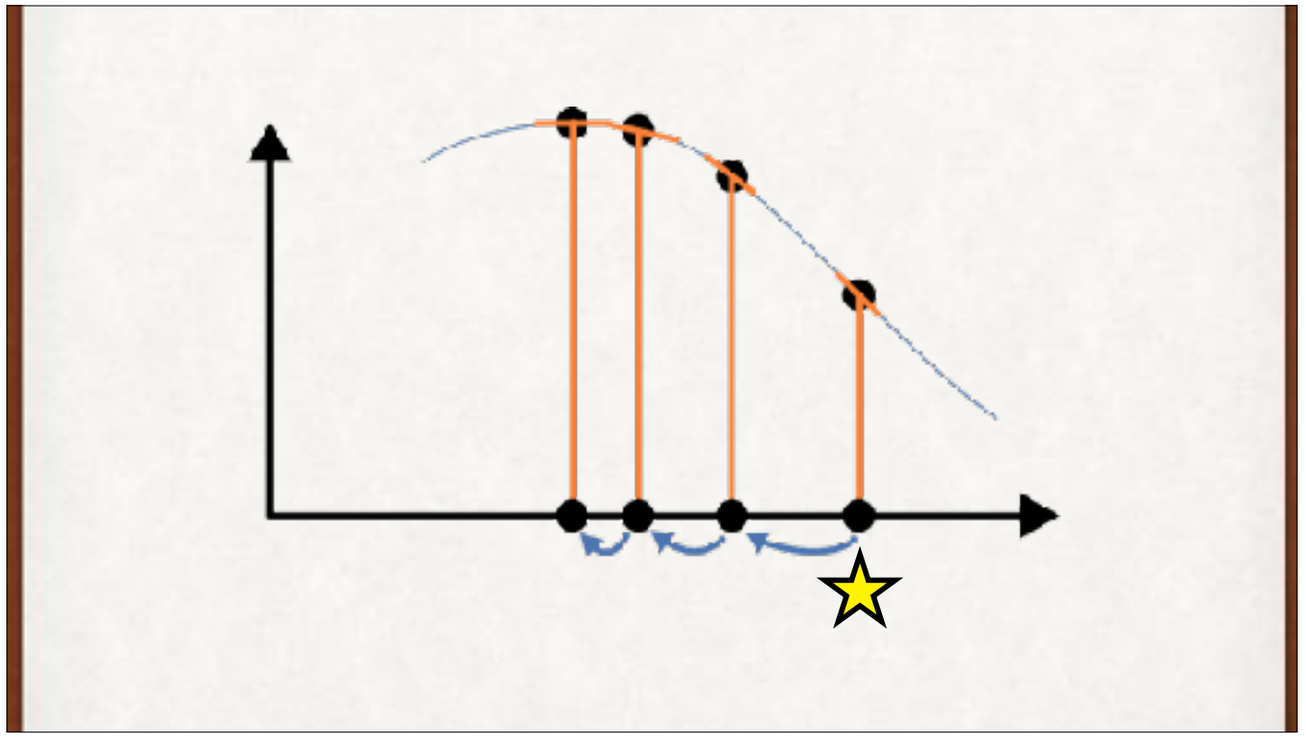
The water will follow the fastest route downhill. That’s the same as following the negative gradient (the gradient points to the direction of steepest ascent, so it’s opposite, or negative, is in the direction of steepest descent).



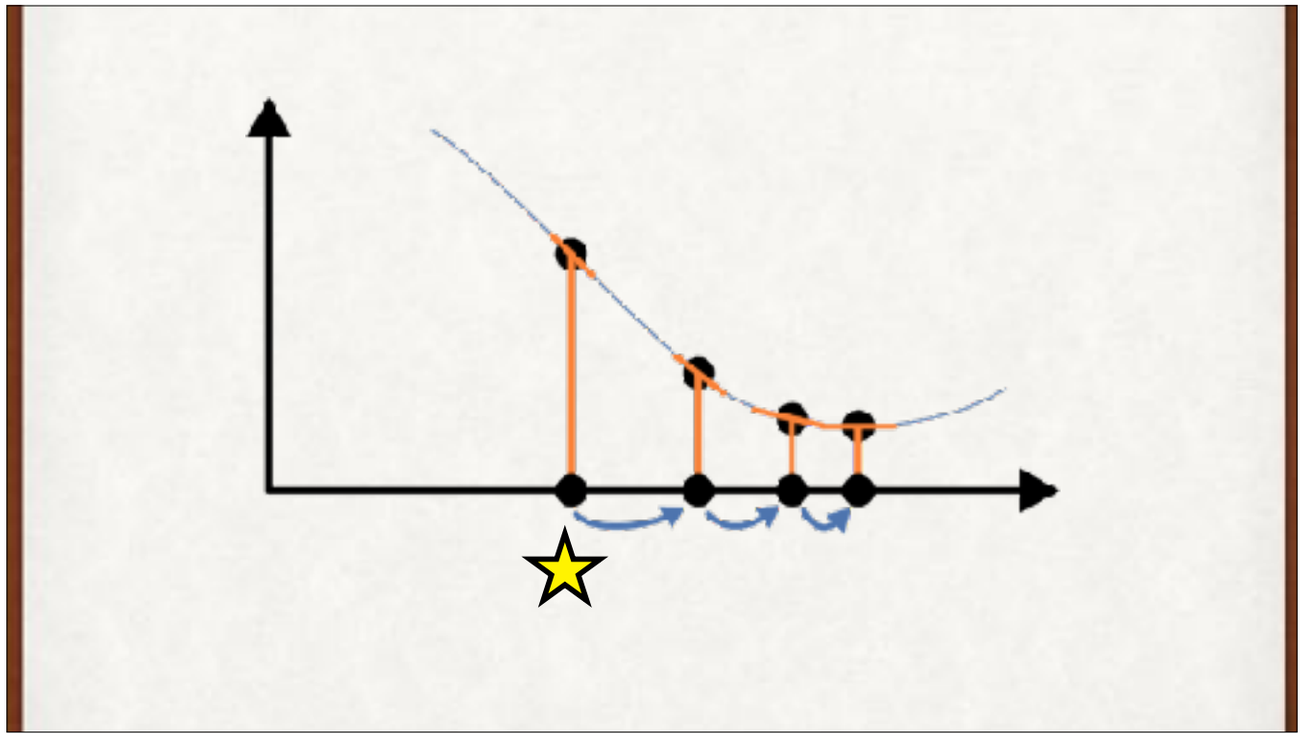
The derivative of a 2D curve can be thought of as the slope (orange) at a point on the curve.



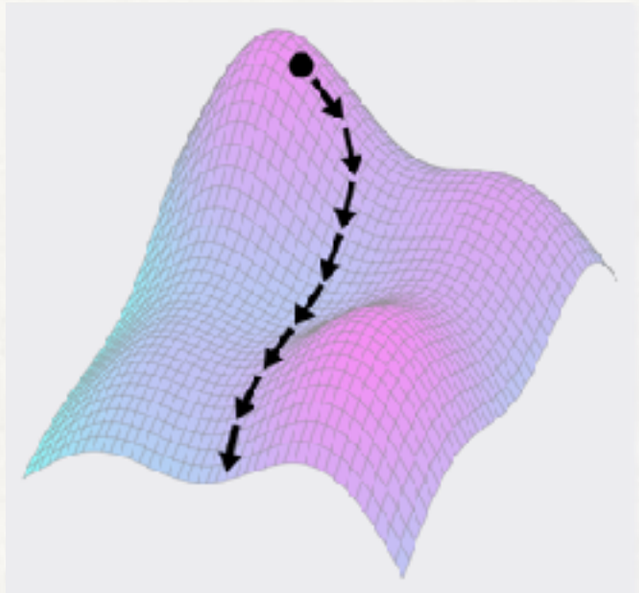
A bunch of points on a curve with their derivatives. We usually draw the length of the line to correspond to the magnitude of the derivative, but here they’re all the same so they’re easier to see.



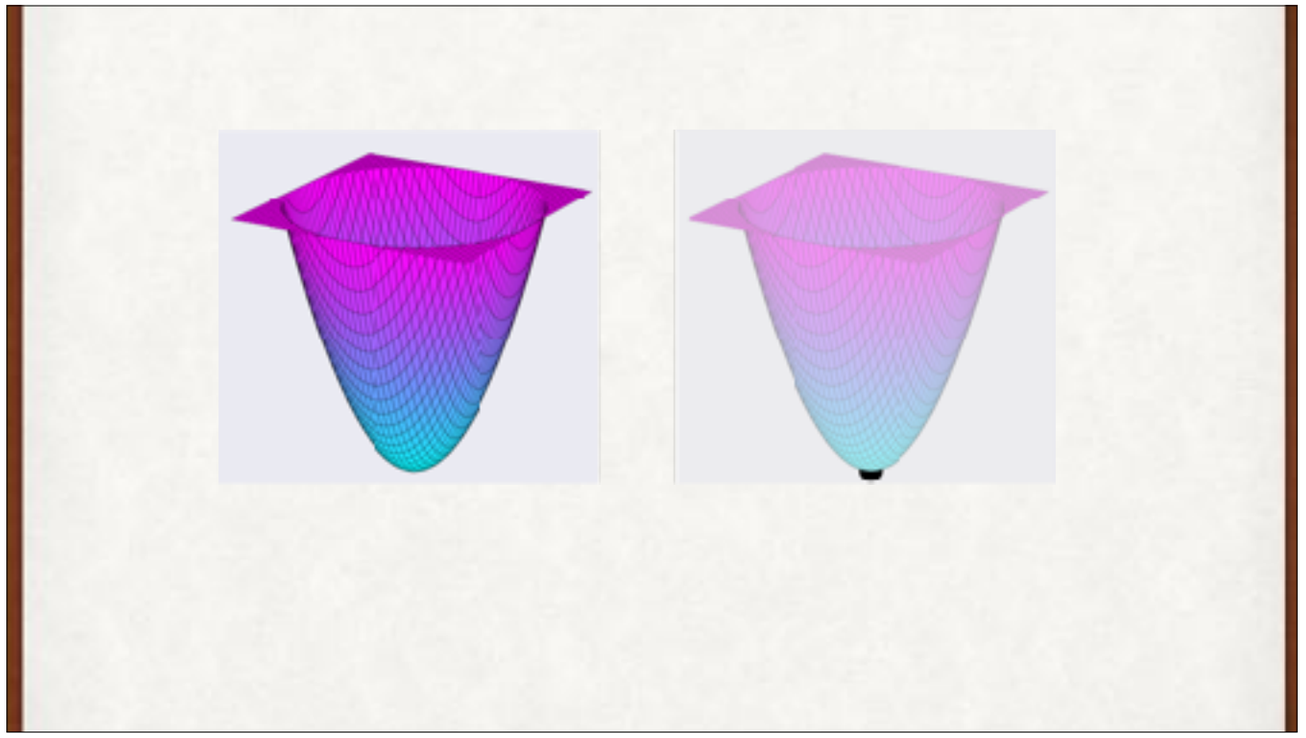
To find a larger function value than where you are, move in the direction of the derivative. We start at the star, where the derivative has a negative value. So we move left, and find another negative value for the slope, but it’s smaller in magnitude (that is, not so steep). So we move left a little bit less. Repeat until the slope becomes flat, telling us we’re at the top of the curve.



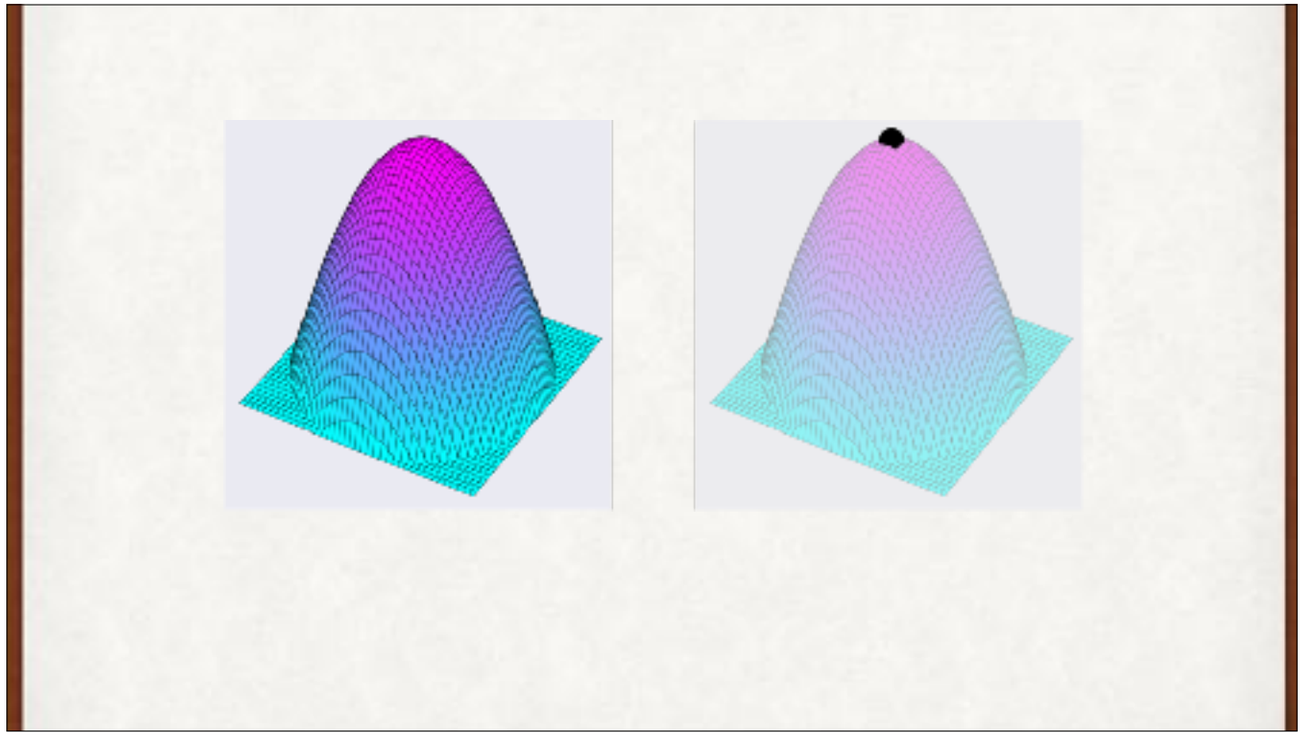
To find a smaller value, move opposite to the derivative. Again, start at the star. The derivative is positive, so move right, where steps are proportional to the size of the derivative.



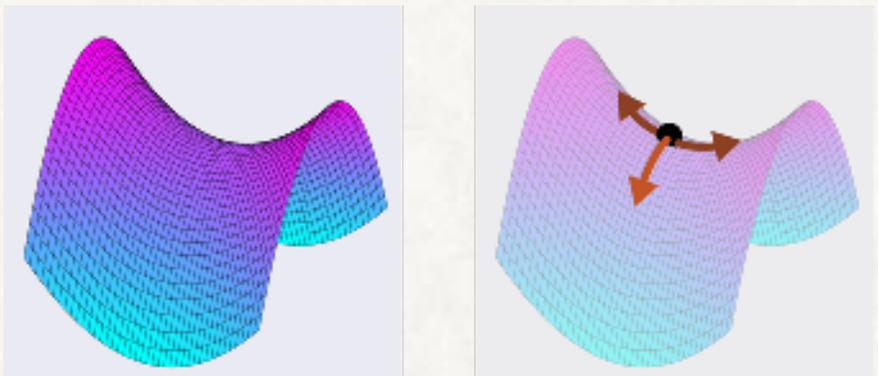
The water will follow the fastest route downhill. That’s the same as following the negative gradient (the gradient points to the direction of steepest ascent, so it’s opposite, or negative, is in the direction of steepest descent).



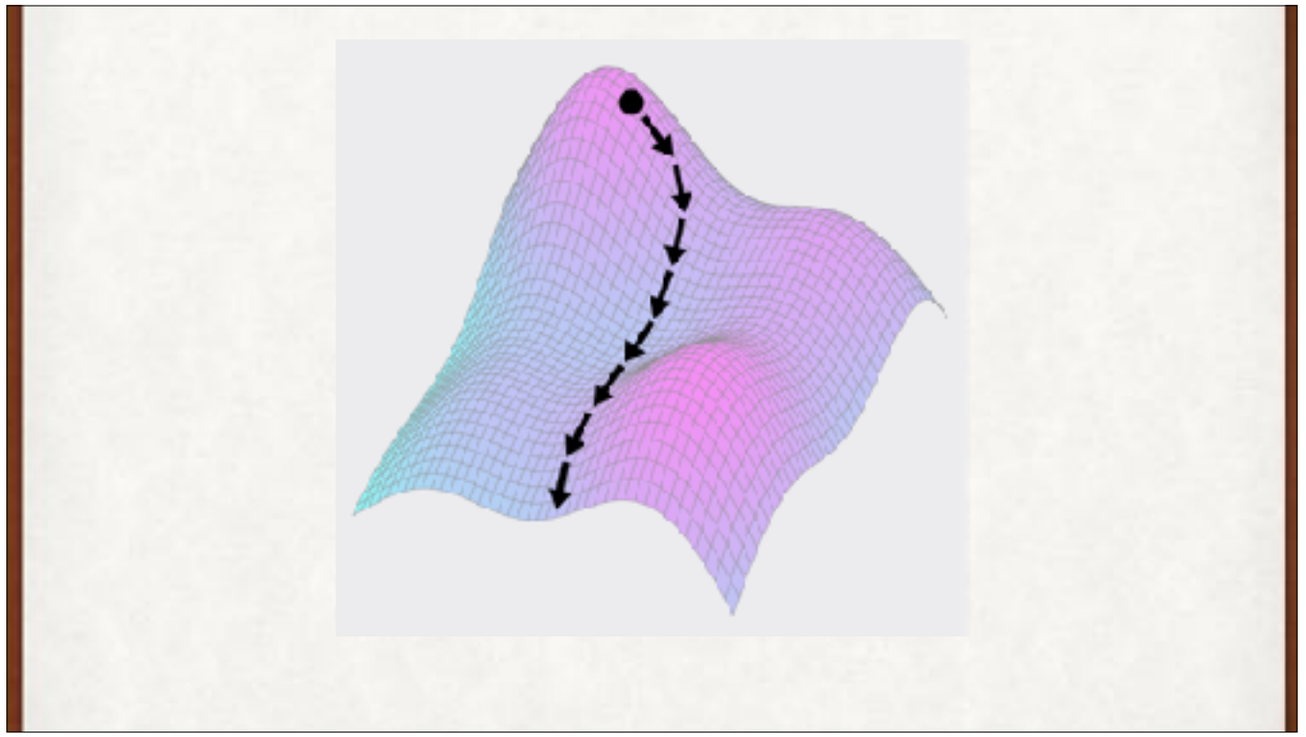
There’s no gradient at the bottom of a bowl. In that immediate neighborhood, the surface is flat.



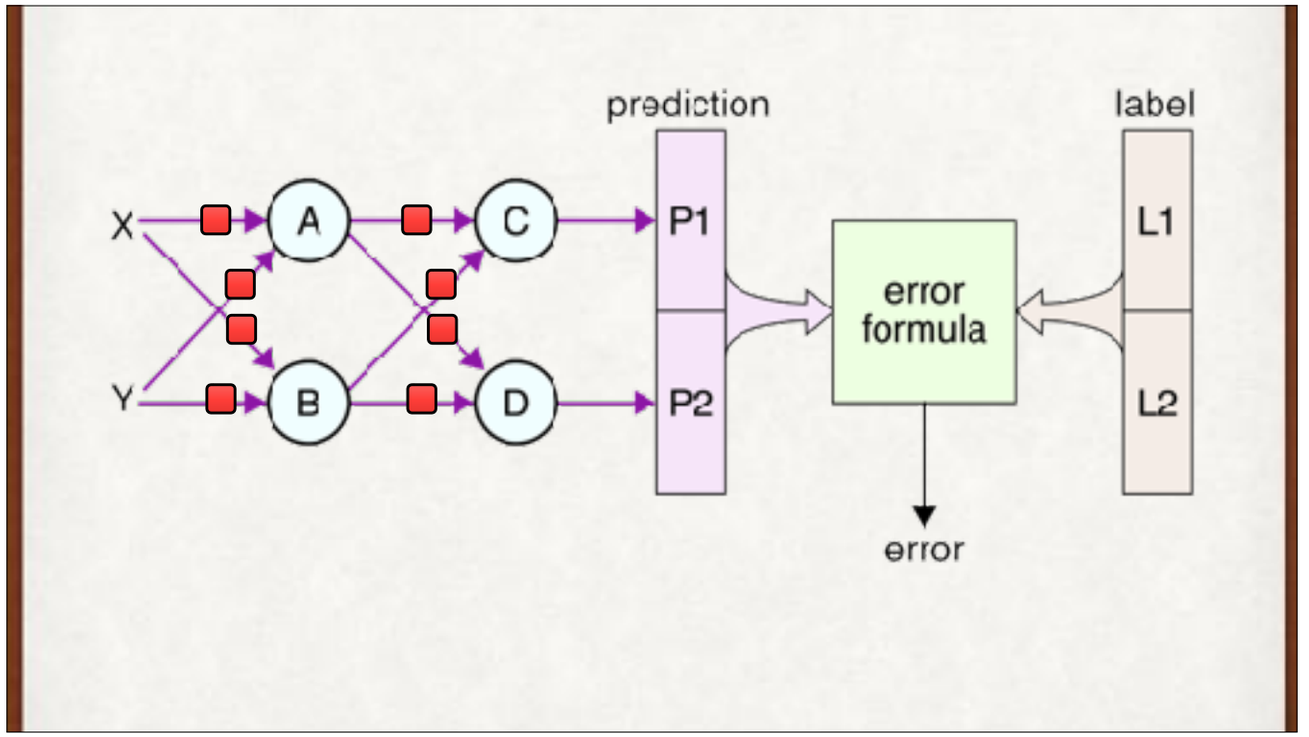
Same thing at the top of a mountain.



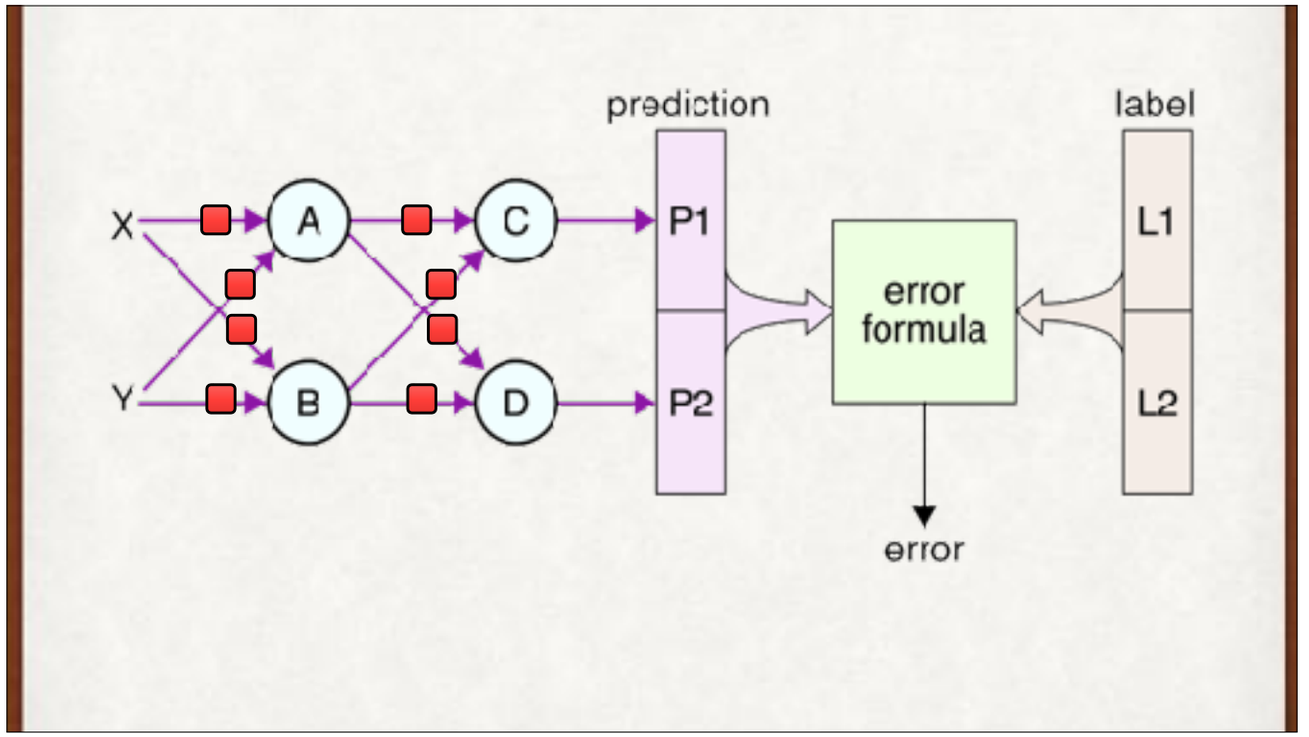
Saddles show up in 3D. They have a zero gradient right in the middle, but positive on one side and negative on the other.



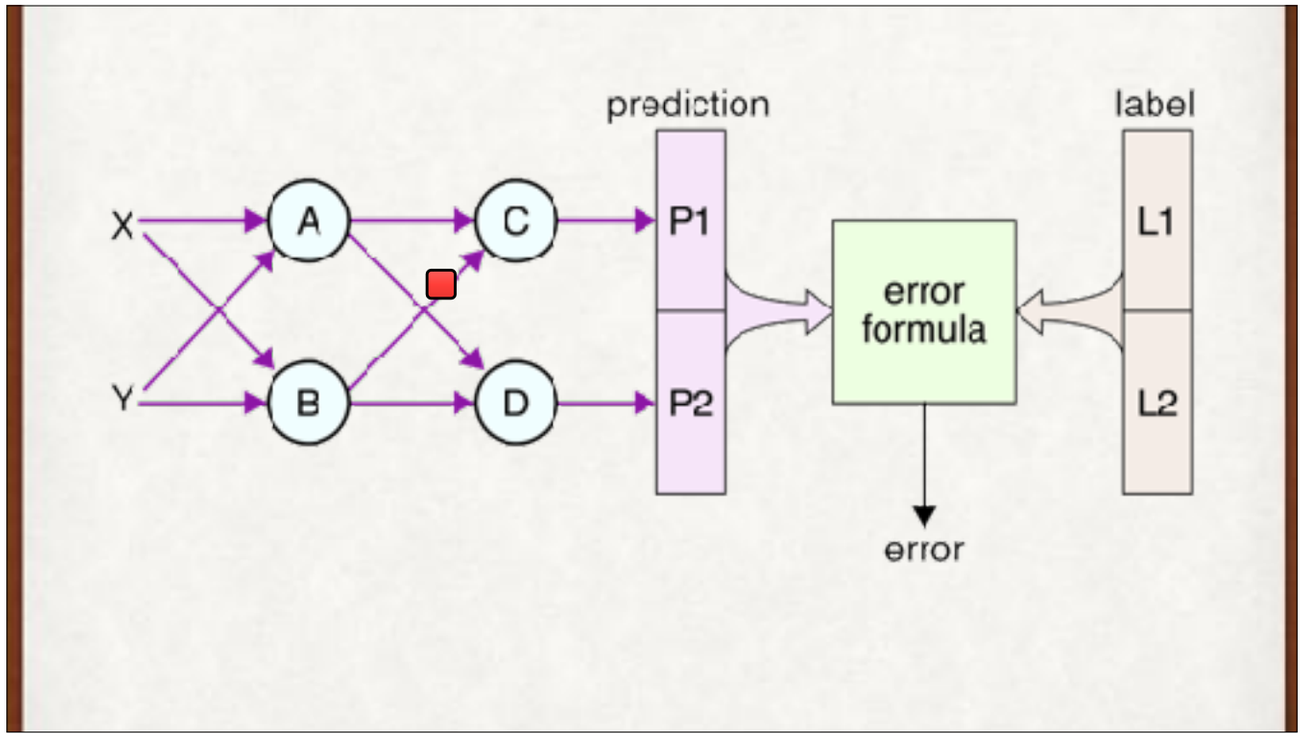
Here’s the key idea again. Follow the gradient in reverse, and you head downhill. If the height of this sheet is f(x,y), then we can find the values of x and y so that f(x,y) is at its smallest (at least in this region) by following this path. If the height of the sheet is the error in our network for given weights x and y, then this helps us find weights that make the error smaller.



So we found an error. How do we use this error curve/surface stuff to change the weights in the network?



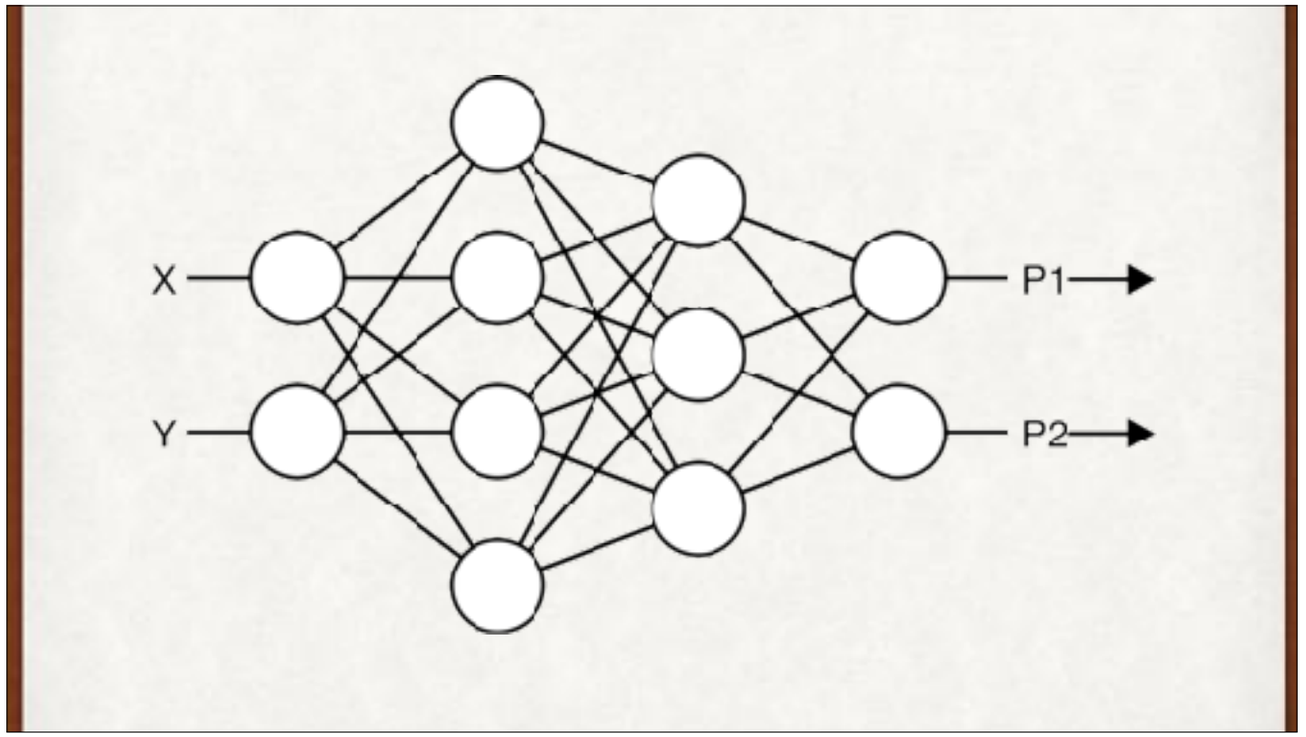
So we found an error. How do we use this error curve/surface stuff to change the weights in the network?



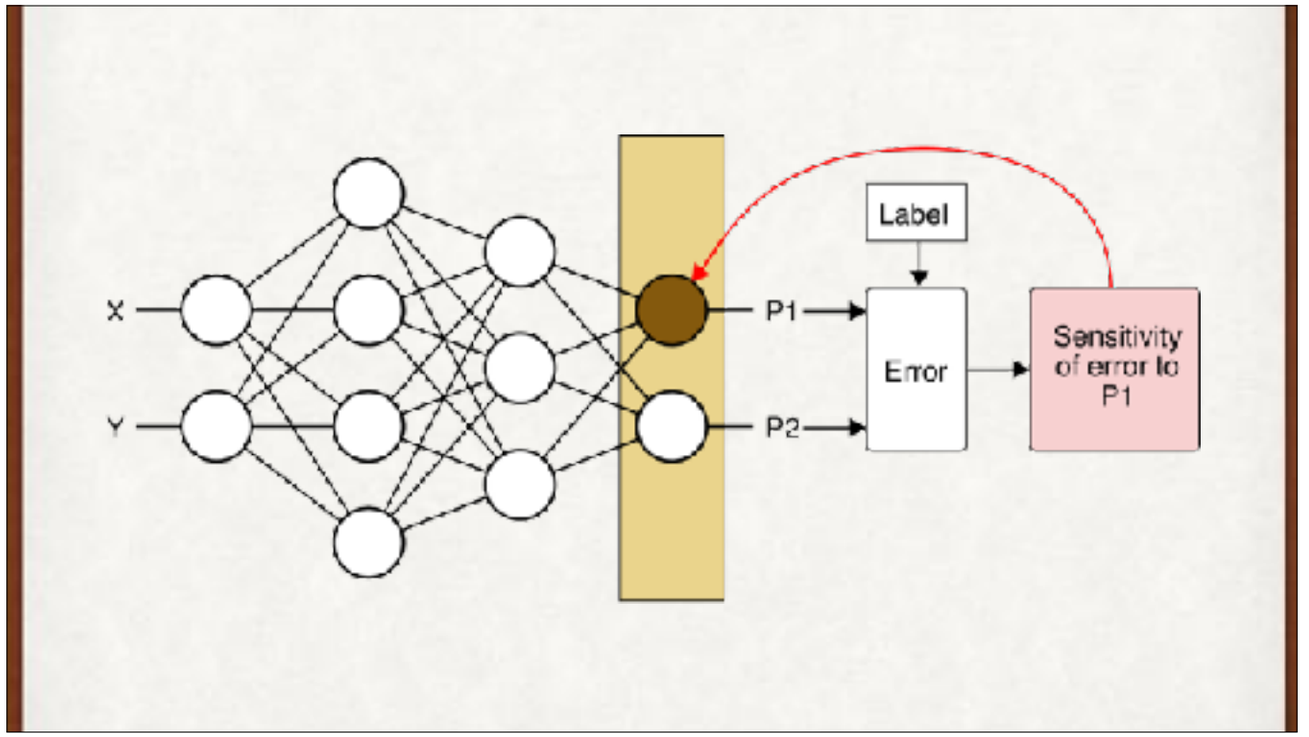
Let’s focus on this one weight, modifying the output of B as it comes into the summation in C.

Backpropagation

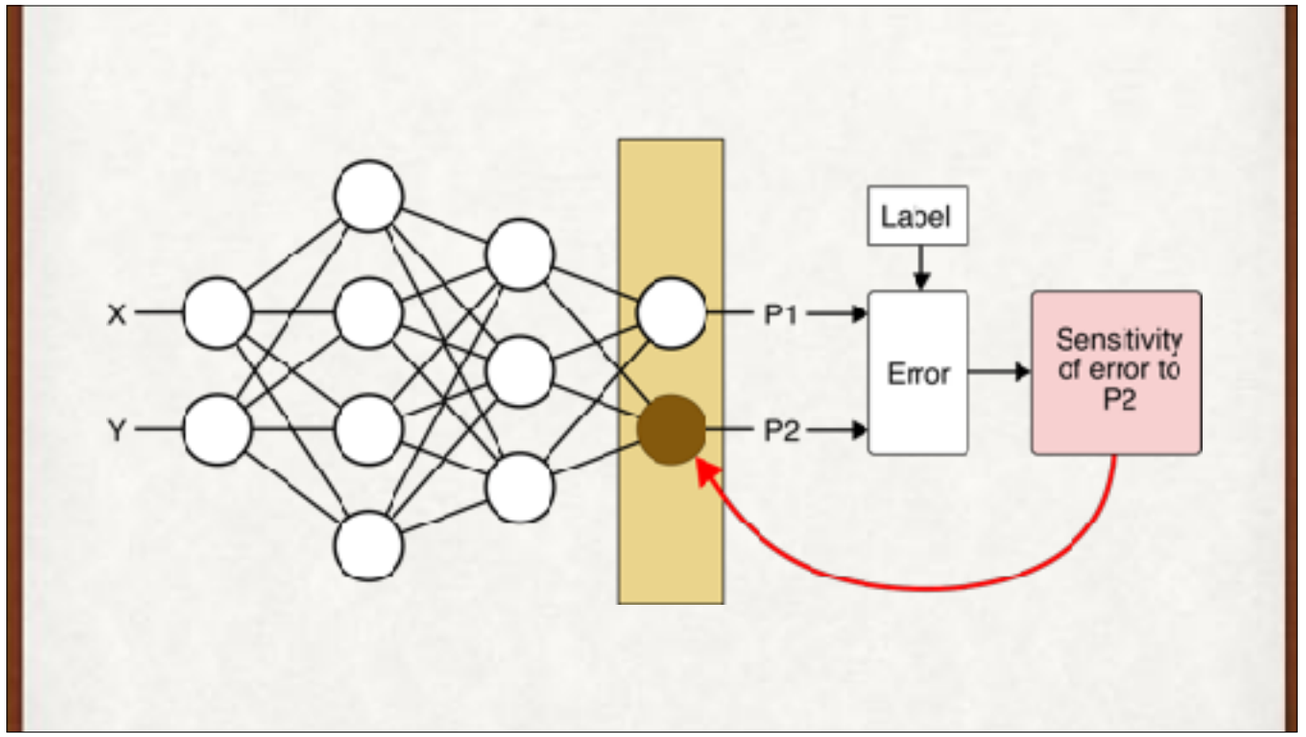
We fix errors in two steps. First, we find the gradients, which tells us whether each weight in the network should be increased or decreased. Then we use those gradients to update the weight values. For the first step, we use this algorithm: backpropagation. It’s simple but subtle. It’s well worth knowing how this works, as it’s critical to deep learning. We’ll just glance at it briefly, because time is short.



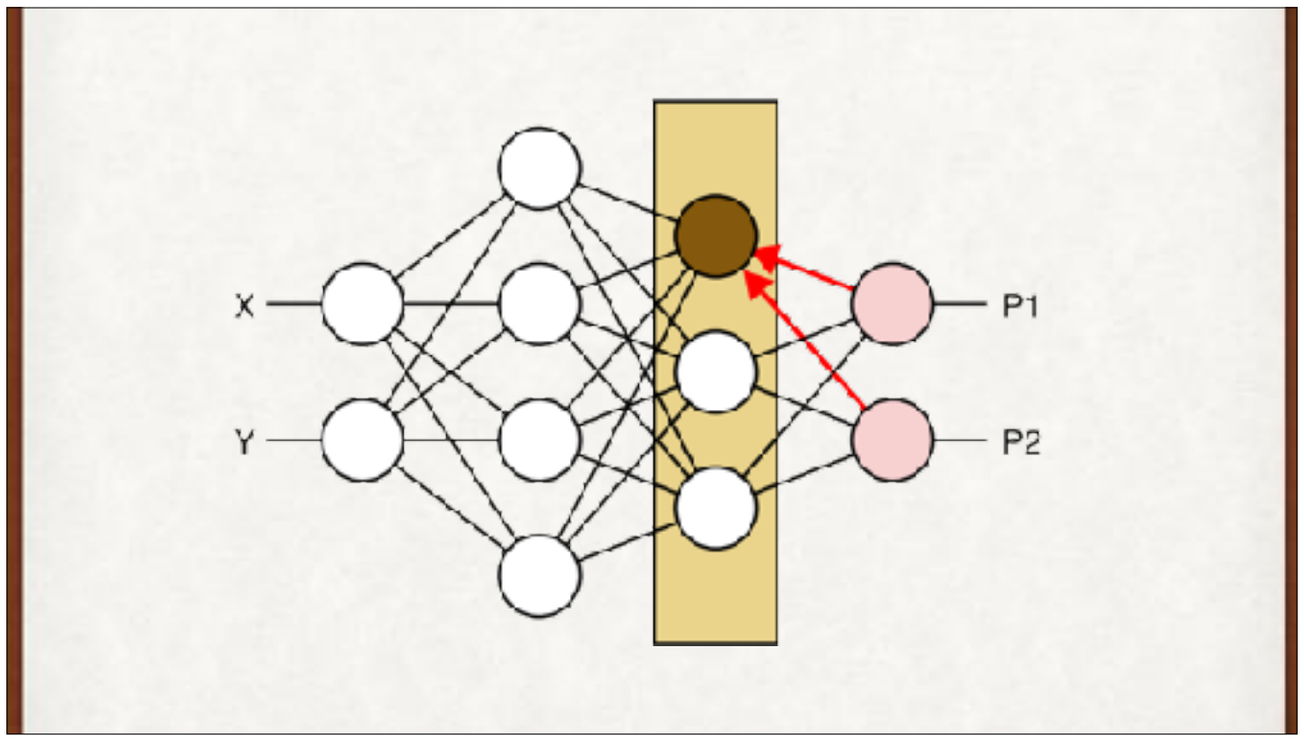
Suppose this is our 4-layer network.



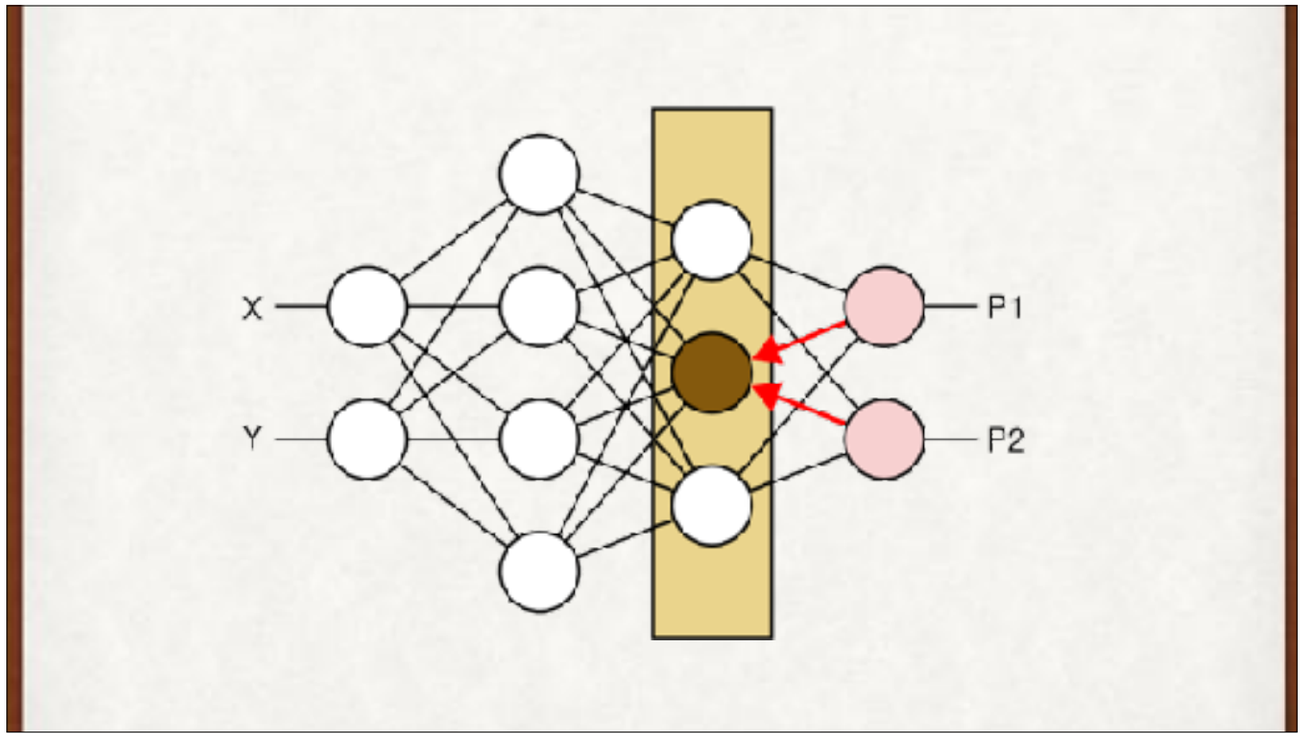
We find how the error changes as P1 changes. That is, the derivative, or gradient, of the error, with respect to P1. We can determine how to change the weights in the brown neuron so that the error (that is, the mismatch between the output and the label) is reduced.



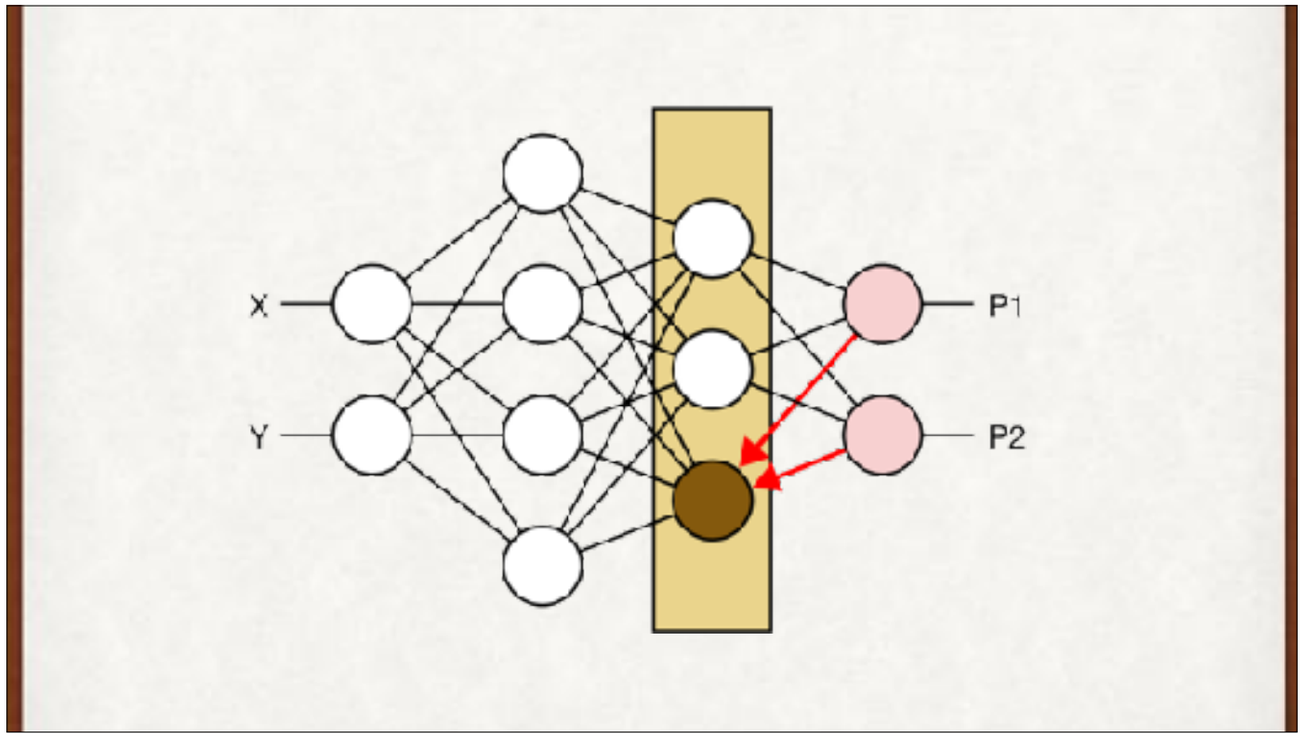
We do that for all the neurons in the output layer. Here, P1 and P2.



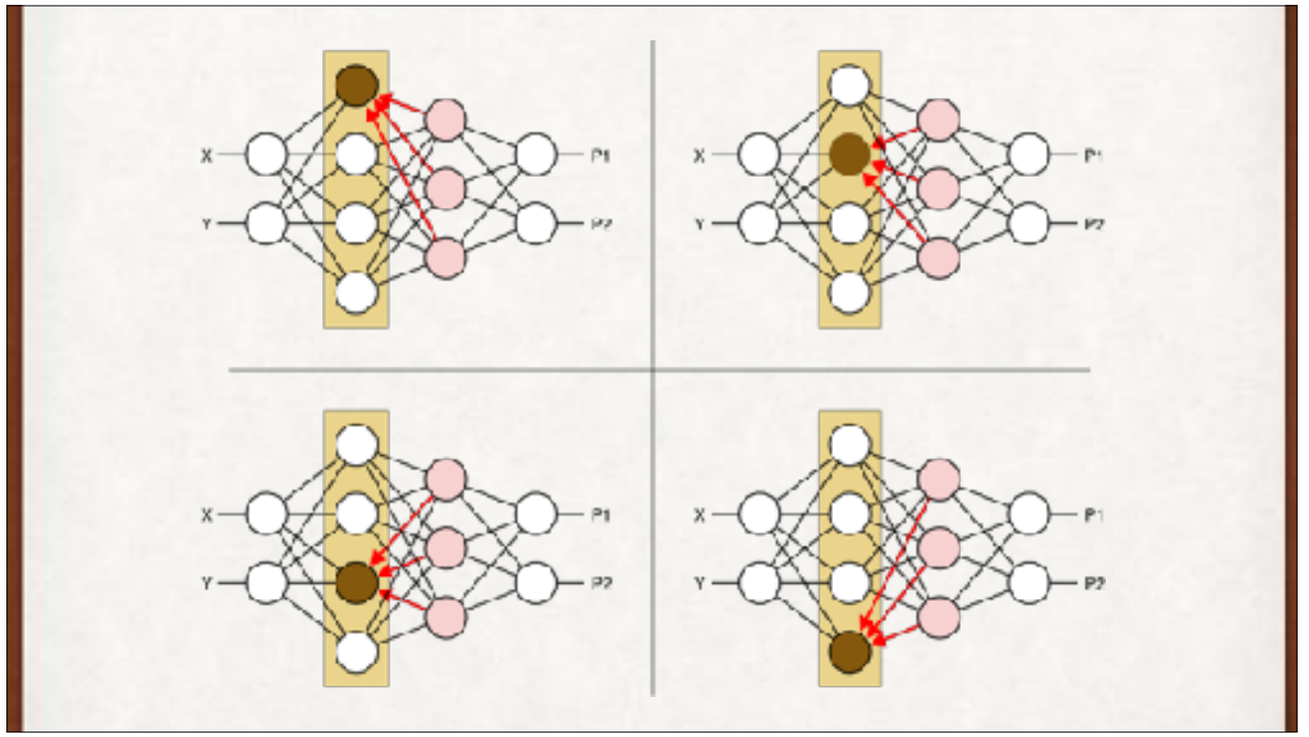
Now we can use those changes to propagate, or push, the change information (that is, the gradient of the error) towards the start of the network, or backwards (backwards propagation = backpropagation = backprop). The change information in the output layer tells us how to change the weights of the neurons in the next-to-final layer.



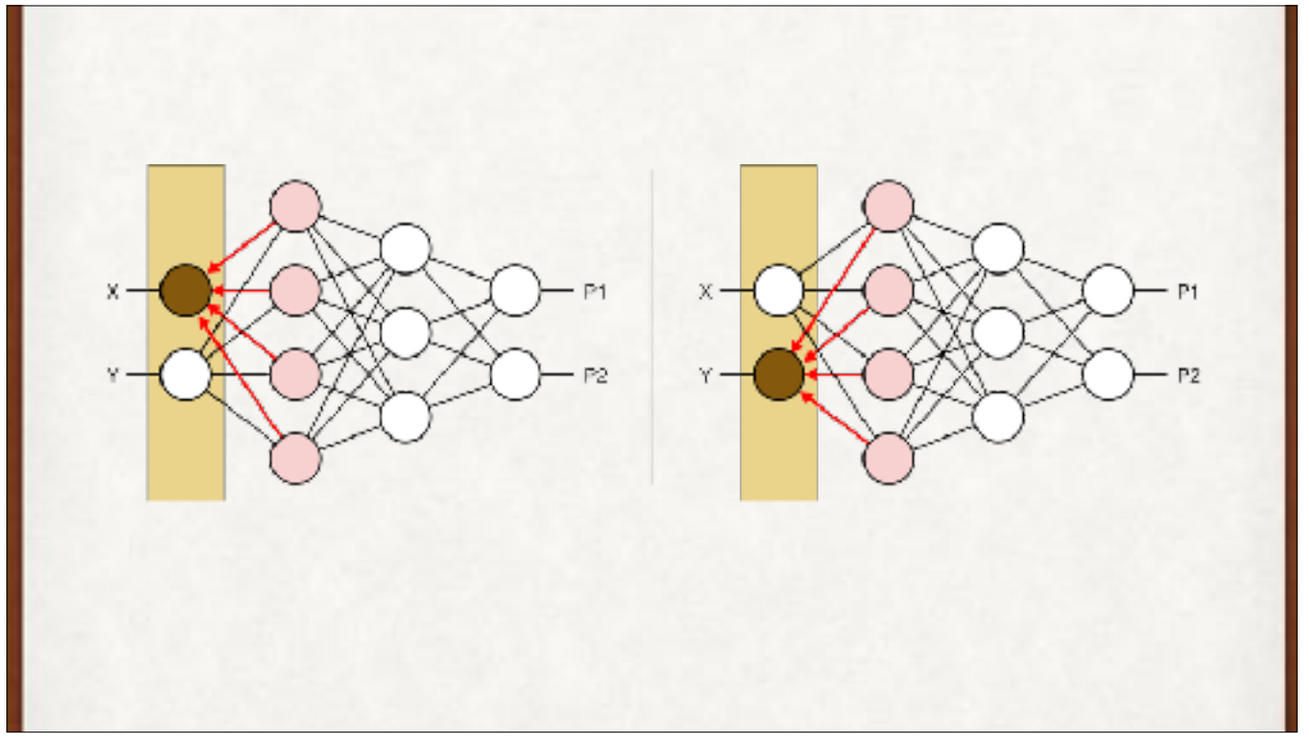
Now we can use those changes to propagate, or push, the change information towards the start of the network, or backwards (backwards propagation = backpropagation = backprop). The change information due to the final layer tells us how to change the weights of the neurons in the next-to-final layer.



Now we can use those changes to propagate, or push, the change information towards the start of the network, or backwards (backwards propagation = backpropagation = backprop). The change information due to the final layer tells us how to change the weights of the neurons in the next-to-final layer.



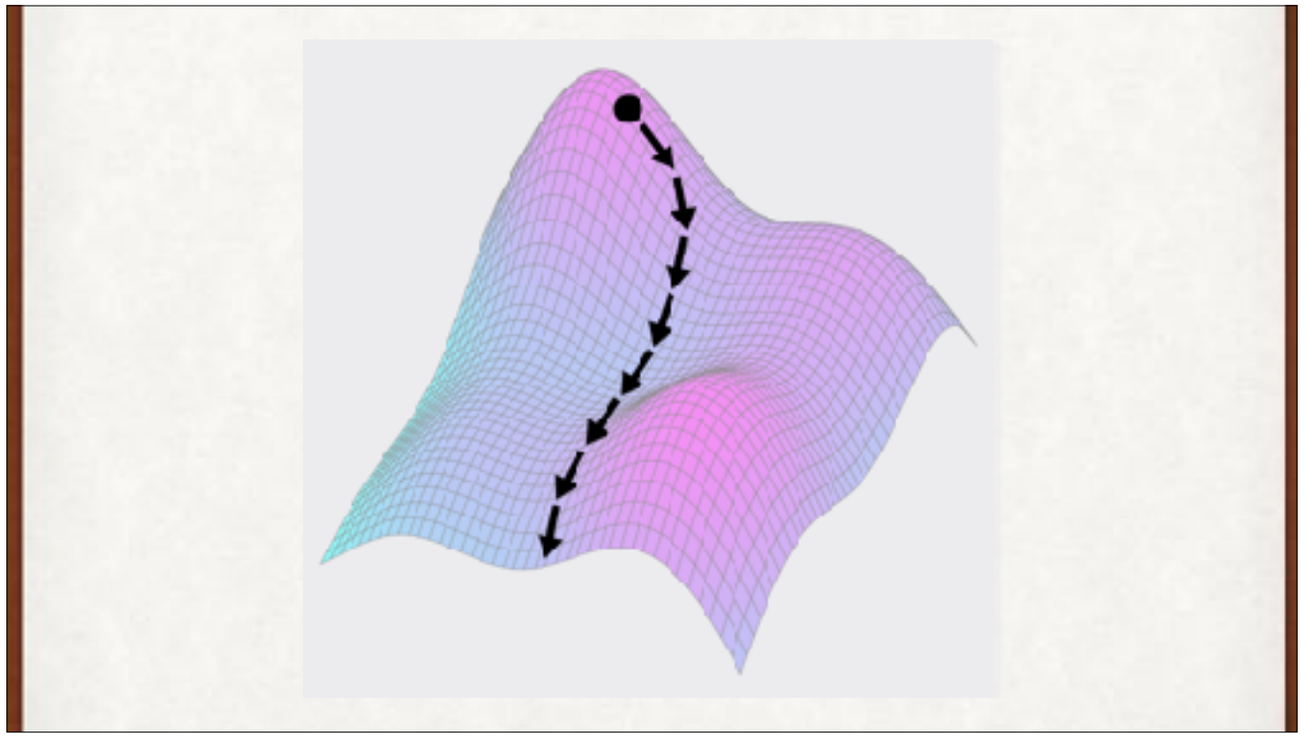
And we keep doing it. The changes in the third layer tell us how to modify the weights to the neurons in the second layer.



And then we get to the first layer, and now know how to modify the weights. The next step is to use this gradient information to actually change the weights. Much more information on backpropagation is in my book (still without any math beyond some multiplication and addition). It’s worth knowing about!

Learning
with
gradient descent

Now that backprop has given us the gradient of the error for every weight, we can use this to change every weight to produce a smaller output. Since we're using the gradient to move downwards on the error surface, this is reasonably called gradient descent.

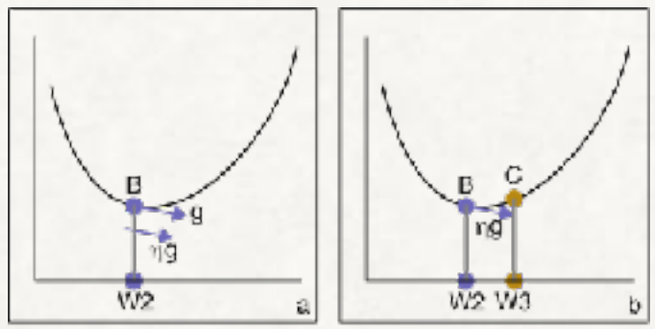


Recall the strategy for minimizing error: follow the reverse gradient of the error function to go downhill. The values below the error surface are the weights. So when we get the weights in position so that they’re under the lowest part of the error surface we can find, we’ve trained our network as well as we can.

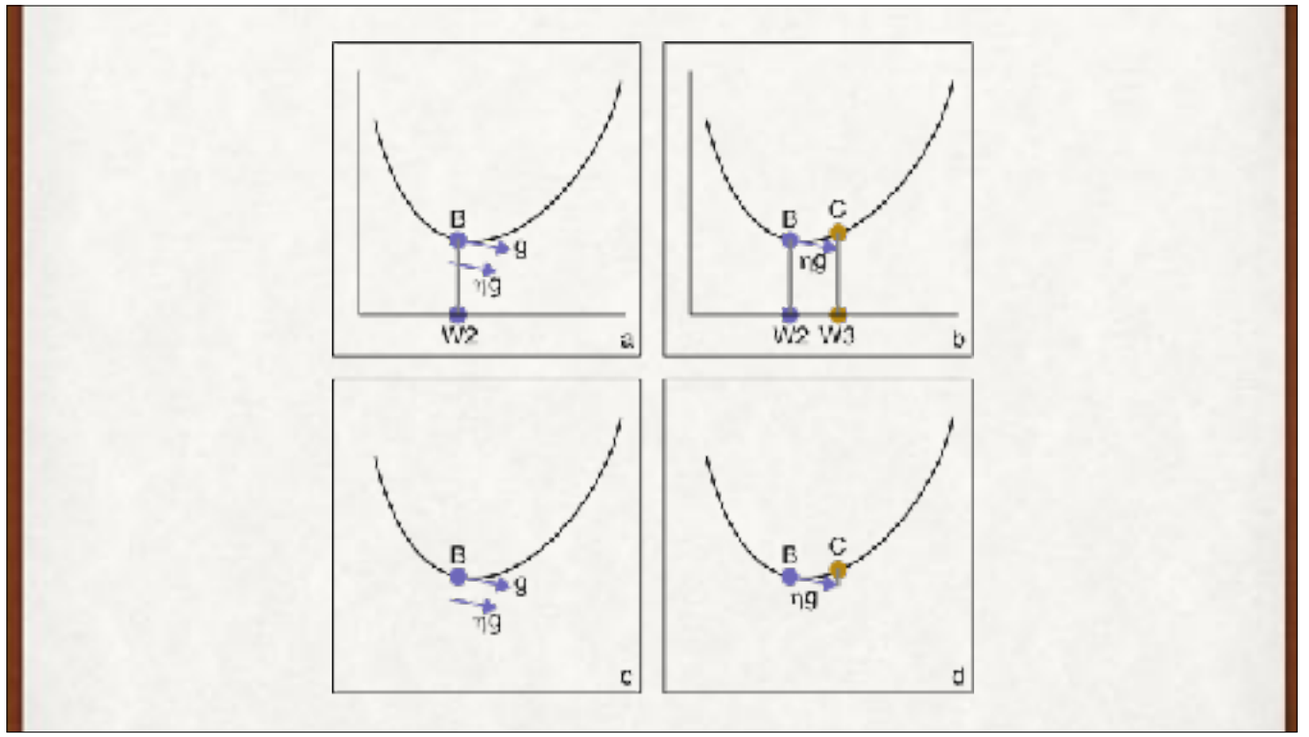
learning rate

$$\eta$$

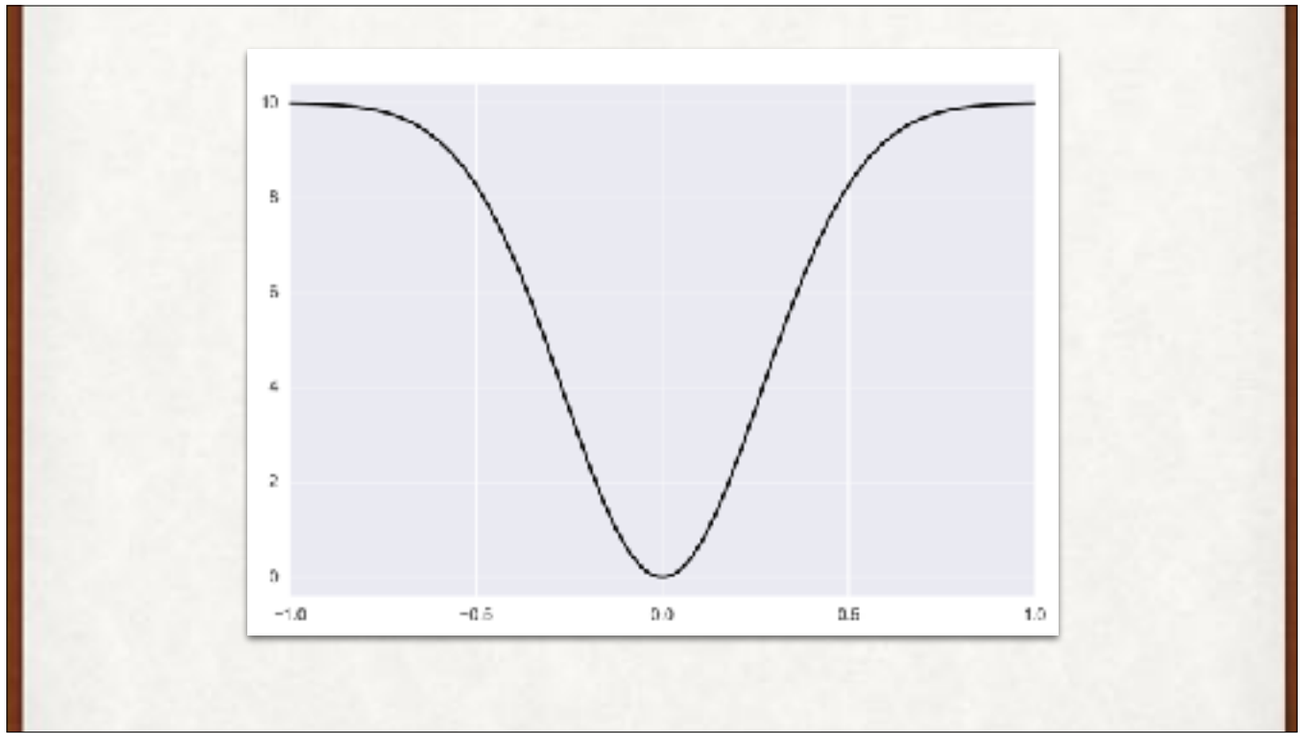
The amount by which we change each weight is governed by a real number called the learning rate, often represented by the Greek letter eta. This is a real number, usually around 0.2 or less. It’s a parameter to the network, but one we provide, rather than one that’s learned. Such parameters are called **hyperparameters**. This is probably the most important hyperparameter of any network. As we’ll see, a value that’s too big means the network won’t learn well. A value that’s too small means the network will learn at a glacial speed.



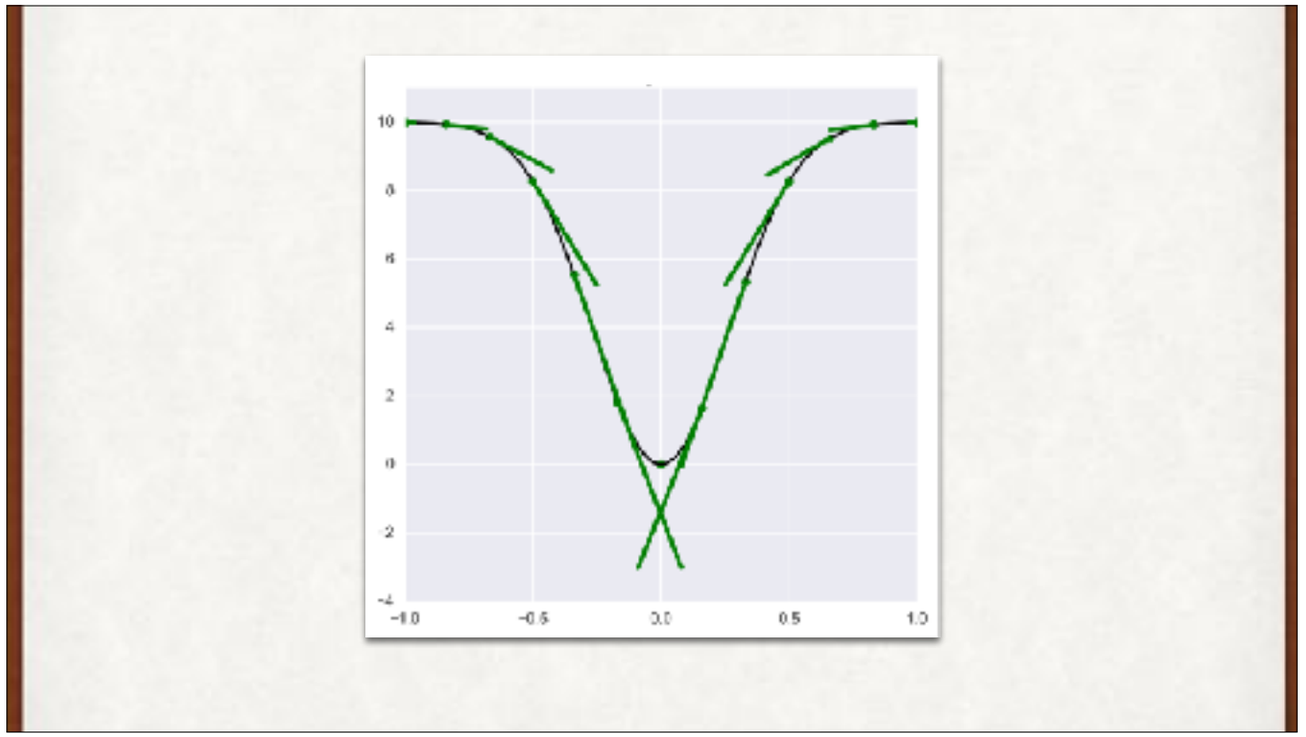
Here’s the idea for improving weight W2. Its error value is point B on the error curve, with gradient g. Adding a scaled amount of g takes us to point C, and corresponding weight W3. We moved too far in this example, so our error actually went up a little.



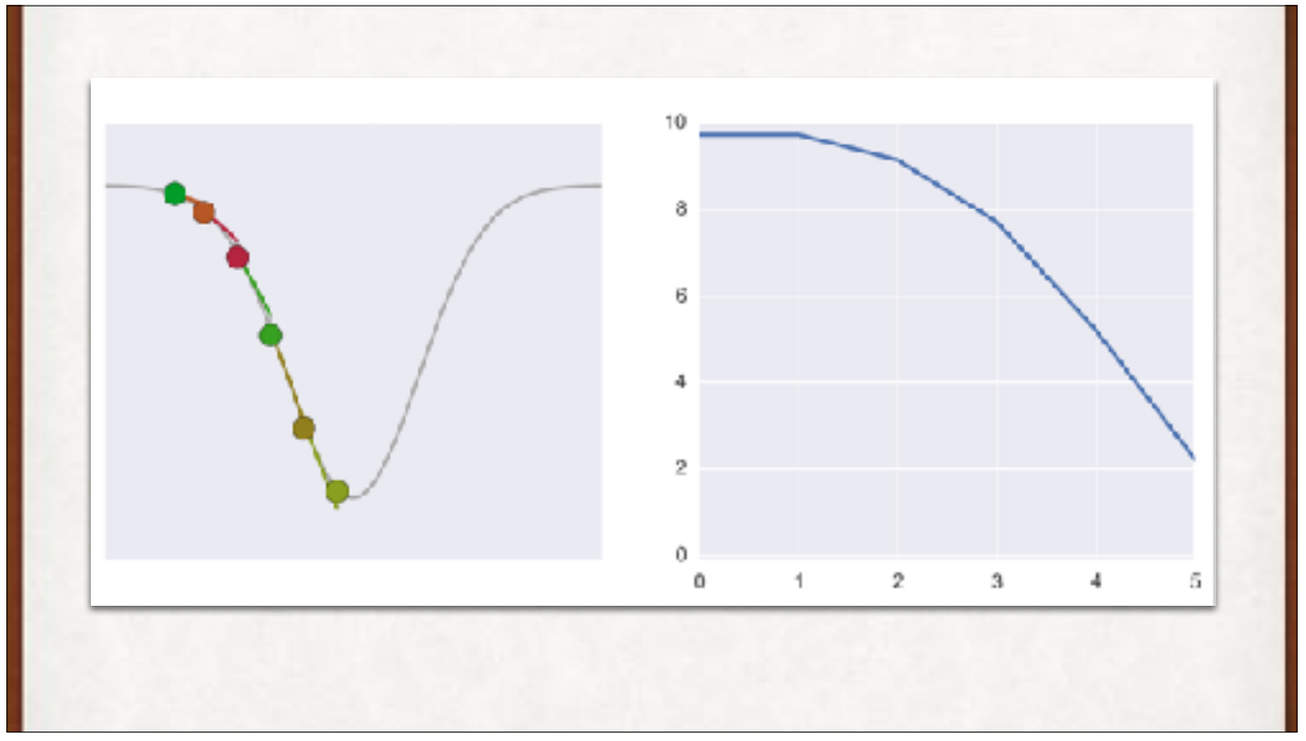
As shown at the bottom, usually we leave out the explicit axes and weights, understanding them to be there, so we can focus on what’s happening with the error curve.



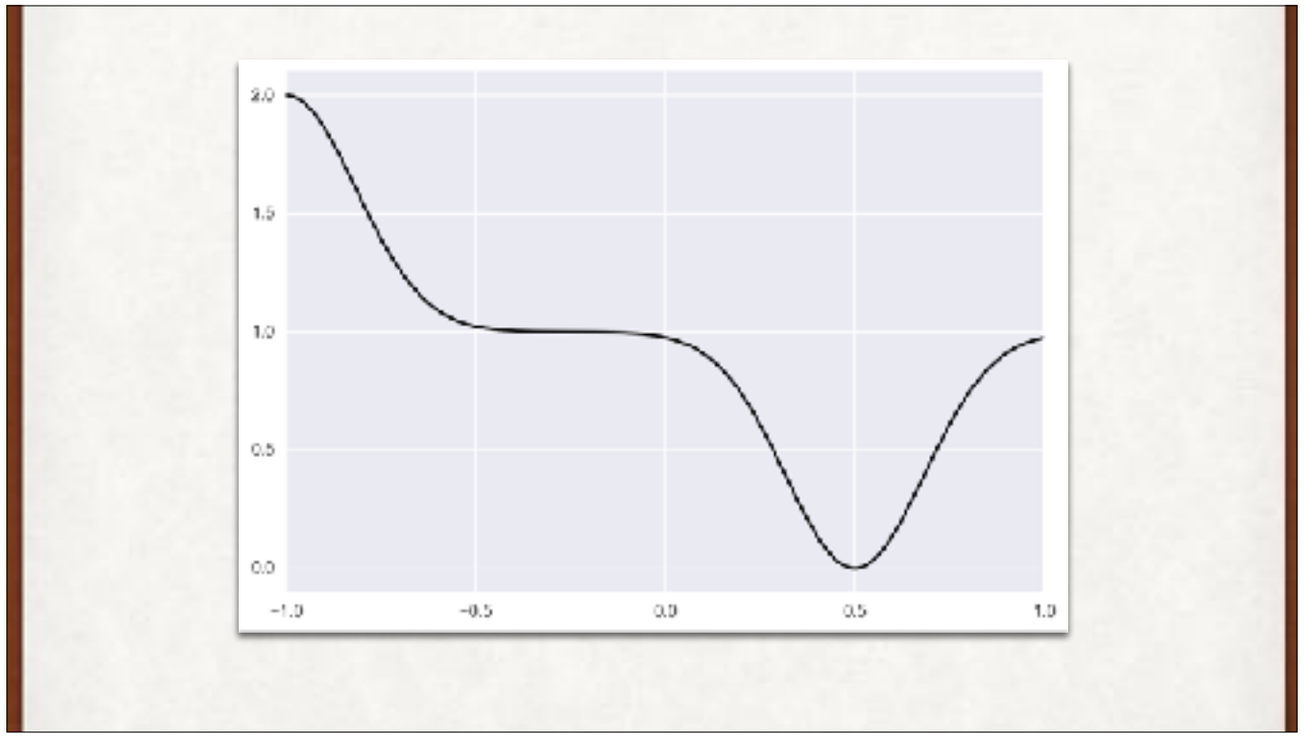
Suppose this is our error curve. We want to get to the bottom.



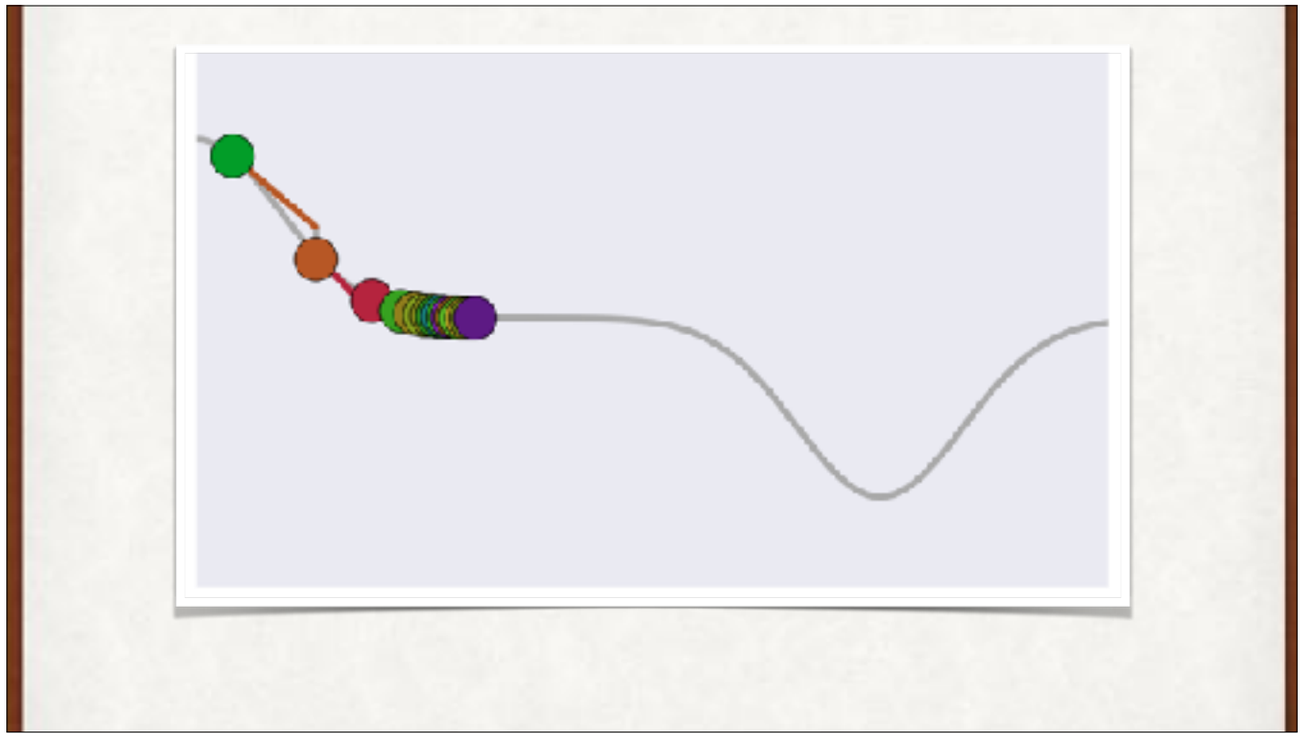
Every point has a gradient, except the bottom of the bowl, where the derivative (or gradient) is zero.



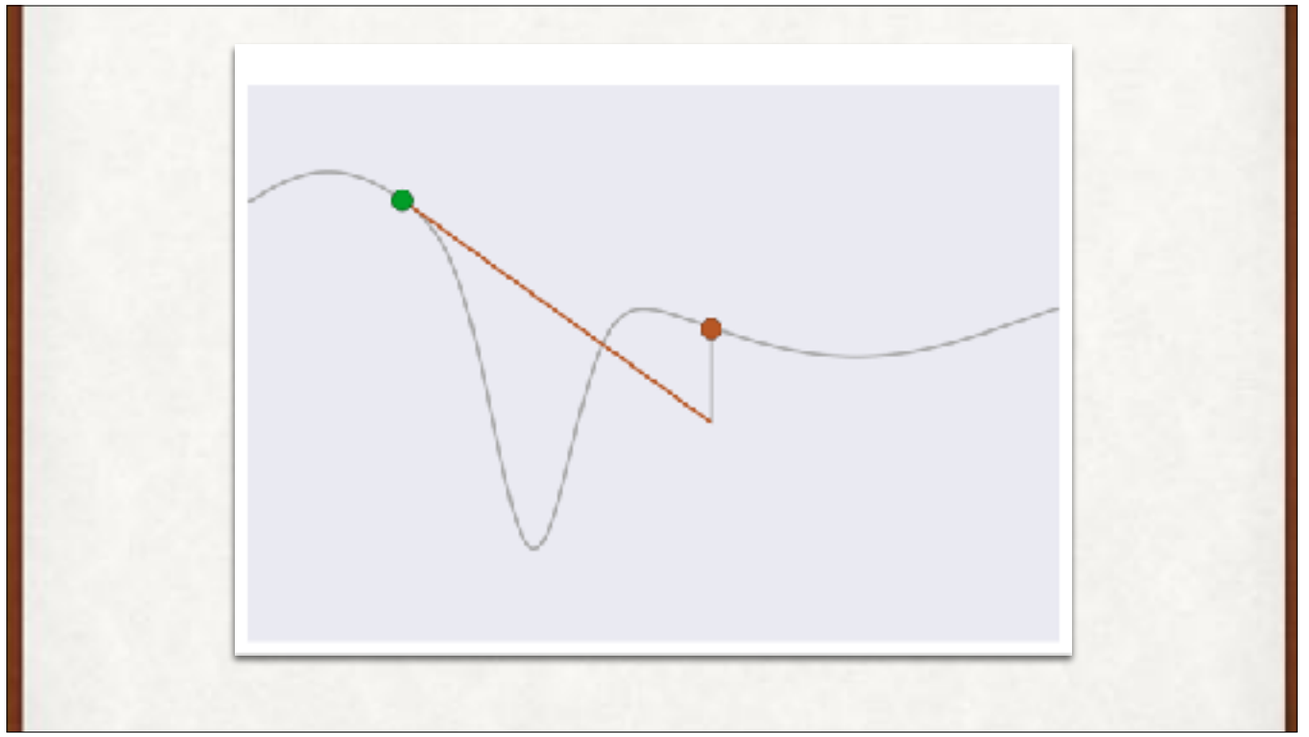
Starting at the far left, the left image shows successive values as we move the value of the weight to follow the gradient downhill. On the right is the error after each step.



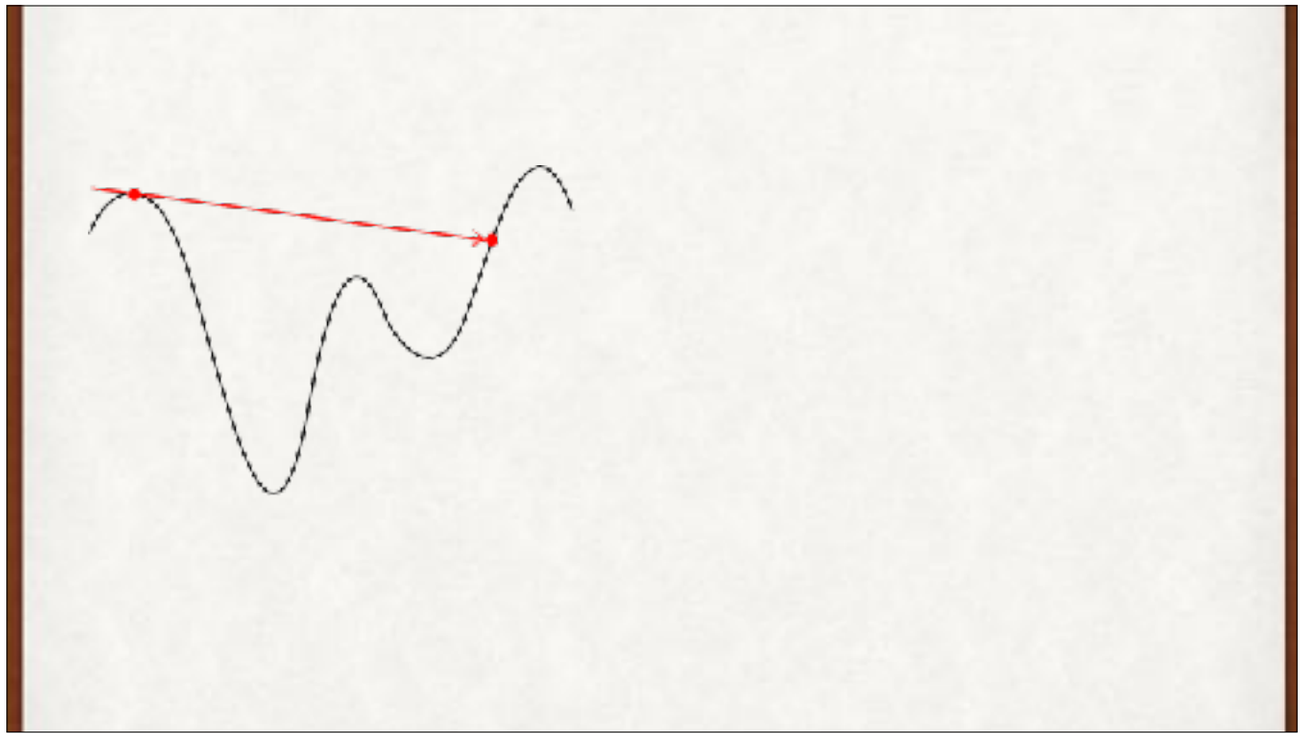
A more interesting error curve.



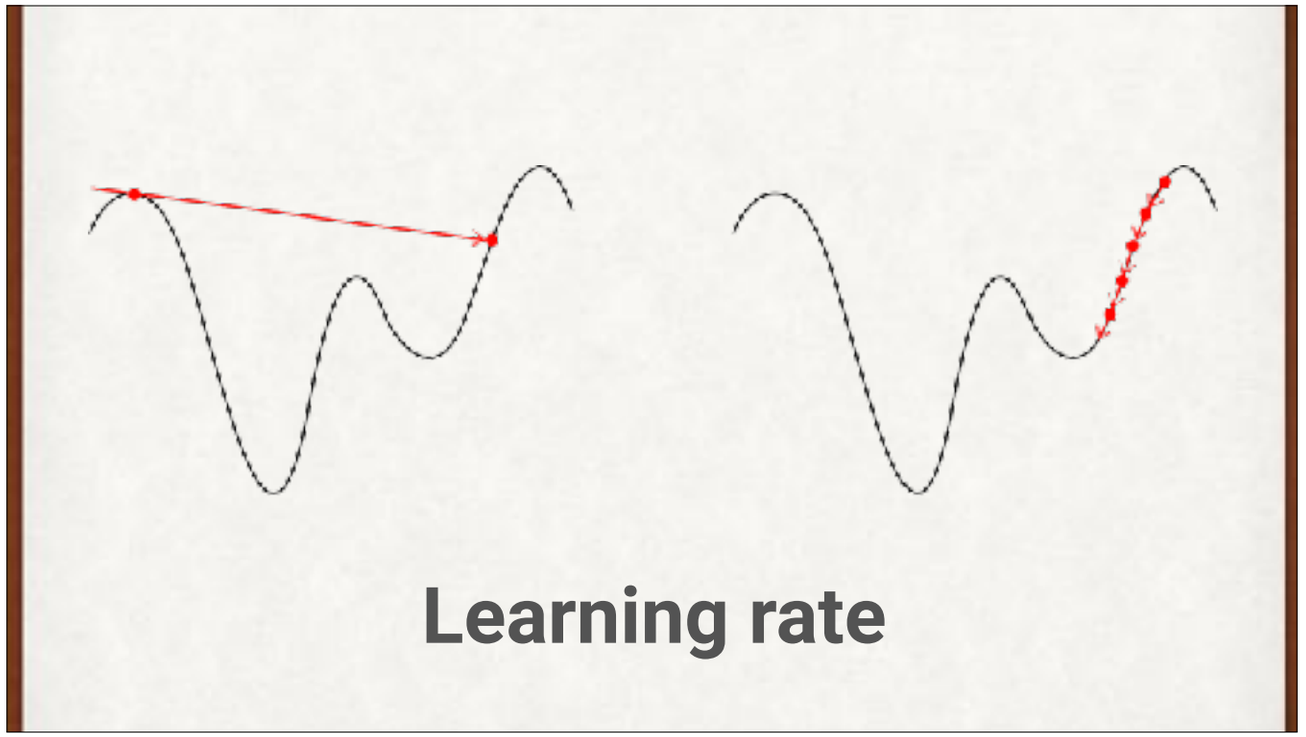
Starting with the green dot at the far left, and… uh-oh. The gradient goes to zero on the flat region, and we get slow down…. until we’re stuck. Zero gradient, zero progress. No more learning!



We could take bigger steps, but that’s not always productive. Here we pop right over the minimum, and further steps will move us to the right.

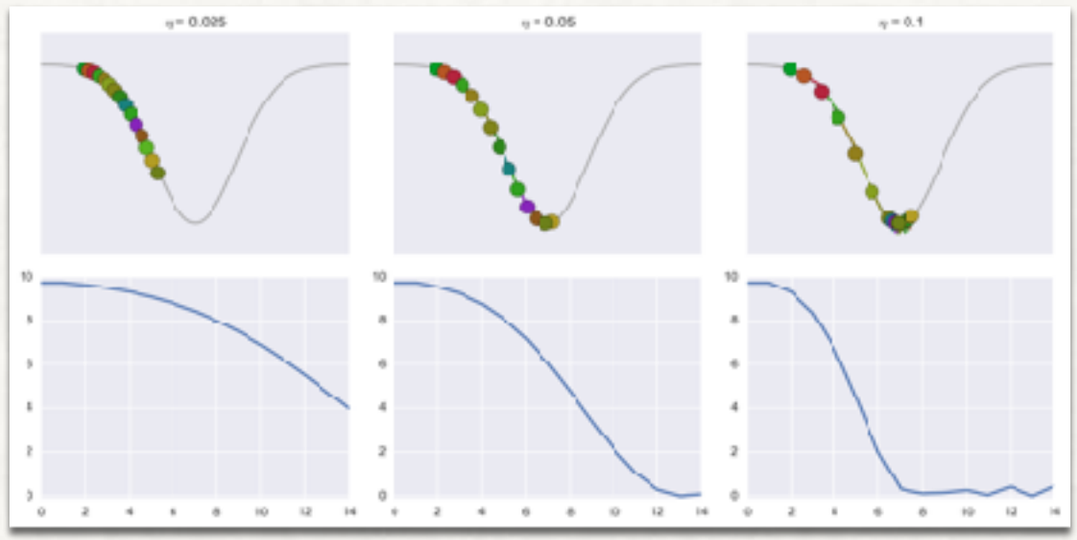


Steps too big and we overshoot our minimum. Steps too small and we take forever to move downhill, and can get stuck in a local minimum when a larger minimum is nearby. Both are problems.



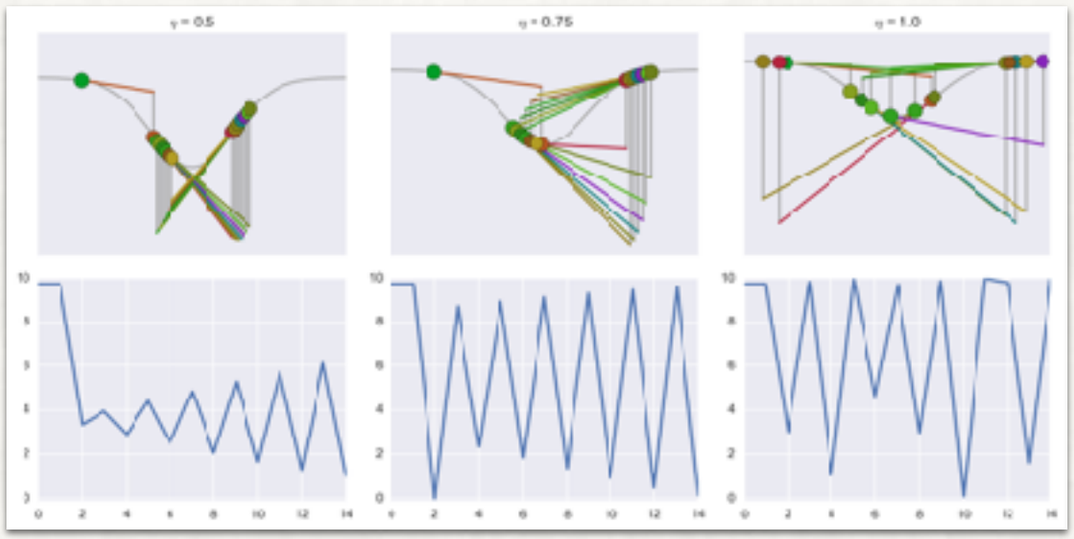
Steps too big and we overshoot our minimum. Steps too small and we take forever to move downhill, and can get stuck in a local minimum when a larger minimum is nearby. Both are problems.

Small learning rates

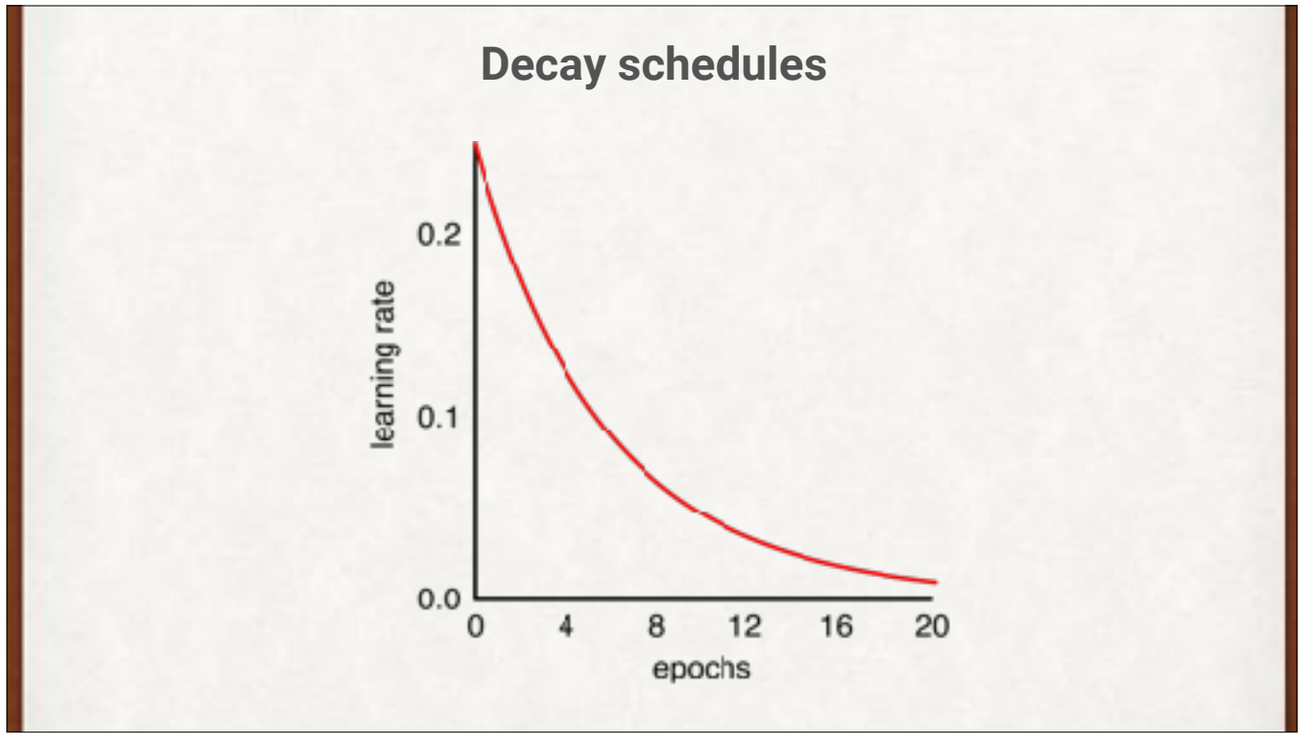


When the step size, or learning rate (Greek lower-case eta), is small, we can take a long time to get where we’re going. Top, starting with the green dot at the far left, with different values for the step size. Directly beneath, the error associated after each iteration.

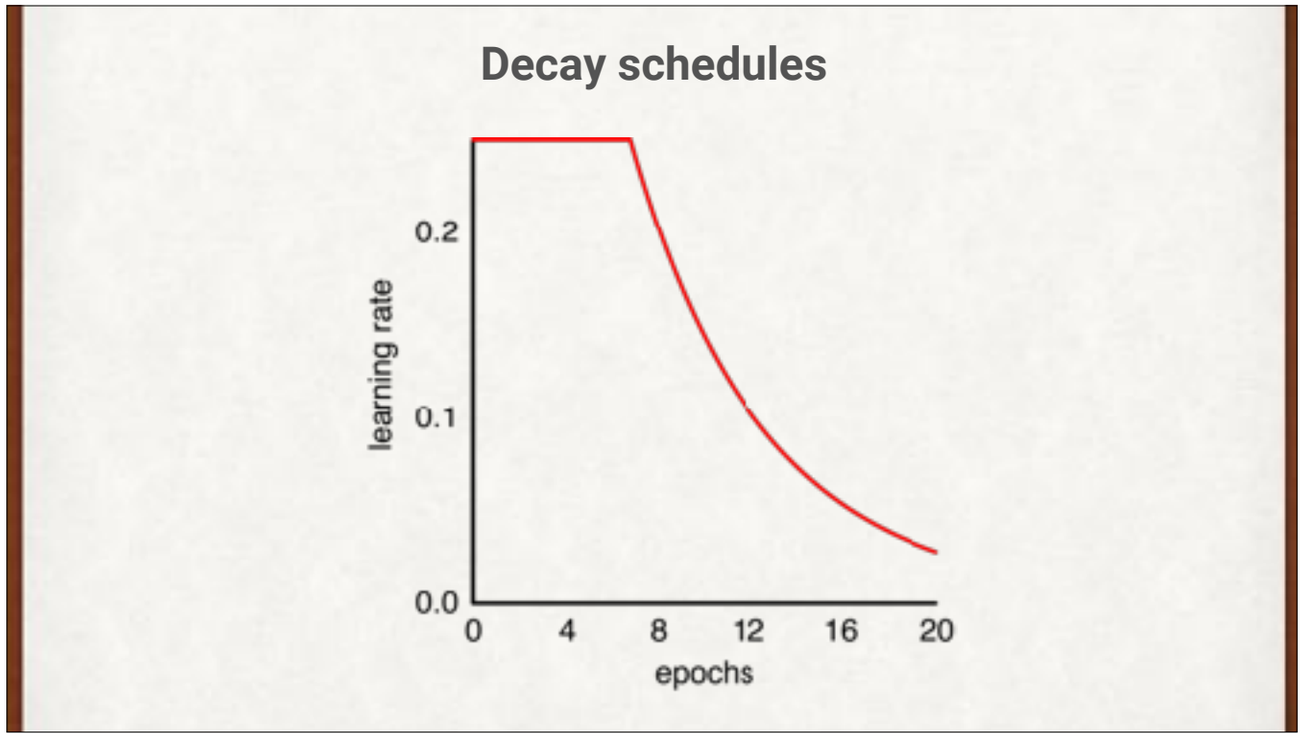
Large learning rates



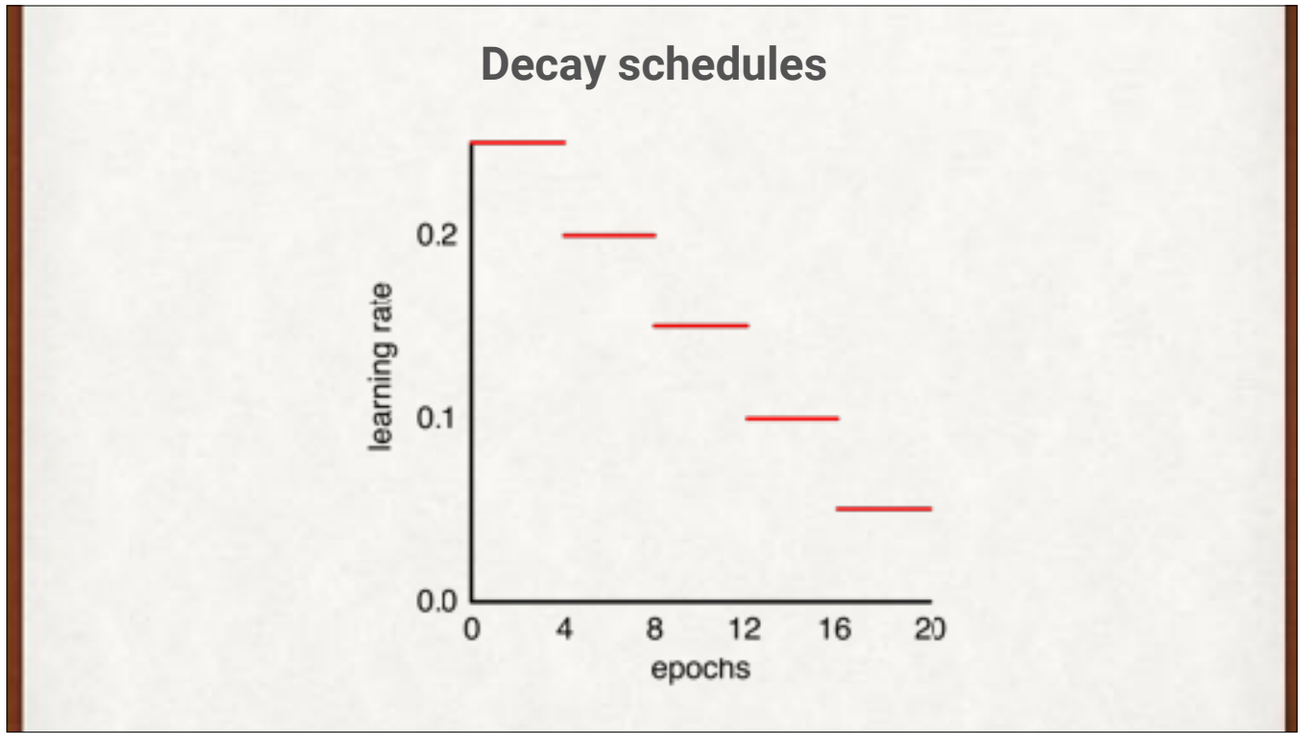
Large learning rates get into trouble in their own way. Here we’re oscillating around the minimum.



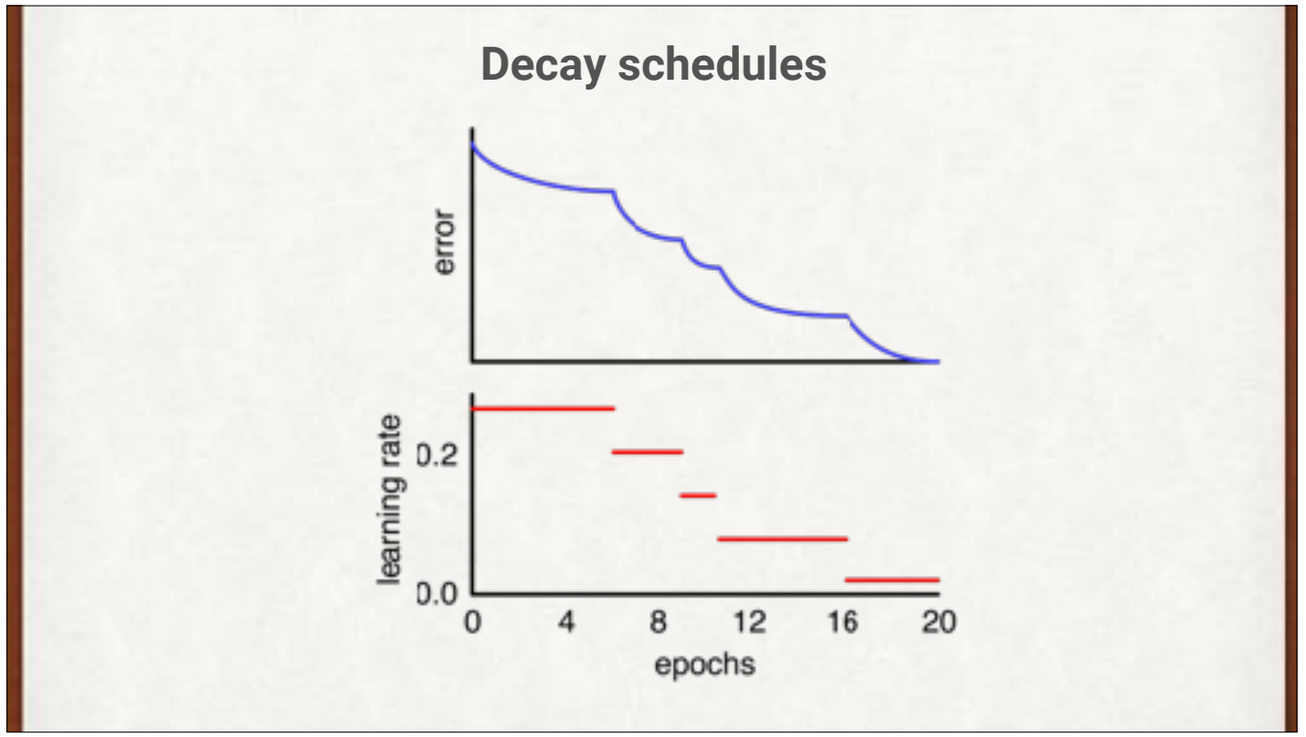
A useful strategy is to decrease, or “decay,” the learning rate over time, so we start with big steps and then get smaller. Think of using a metal detector at the beach: start with big steps, then smaller and smaller as we find the spot where the metal is hiding. Here are different strategies, or “decay schedules,” for reducing the error as learning goes by.



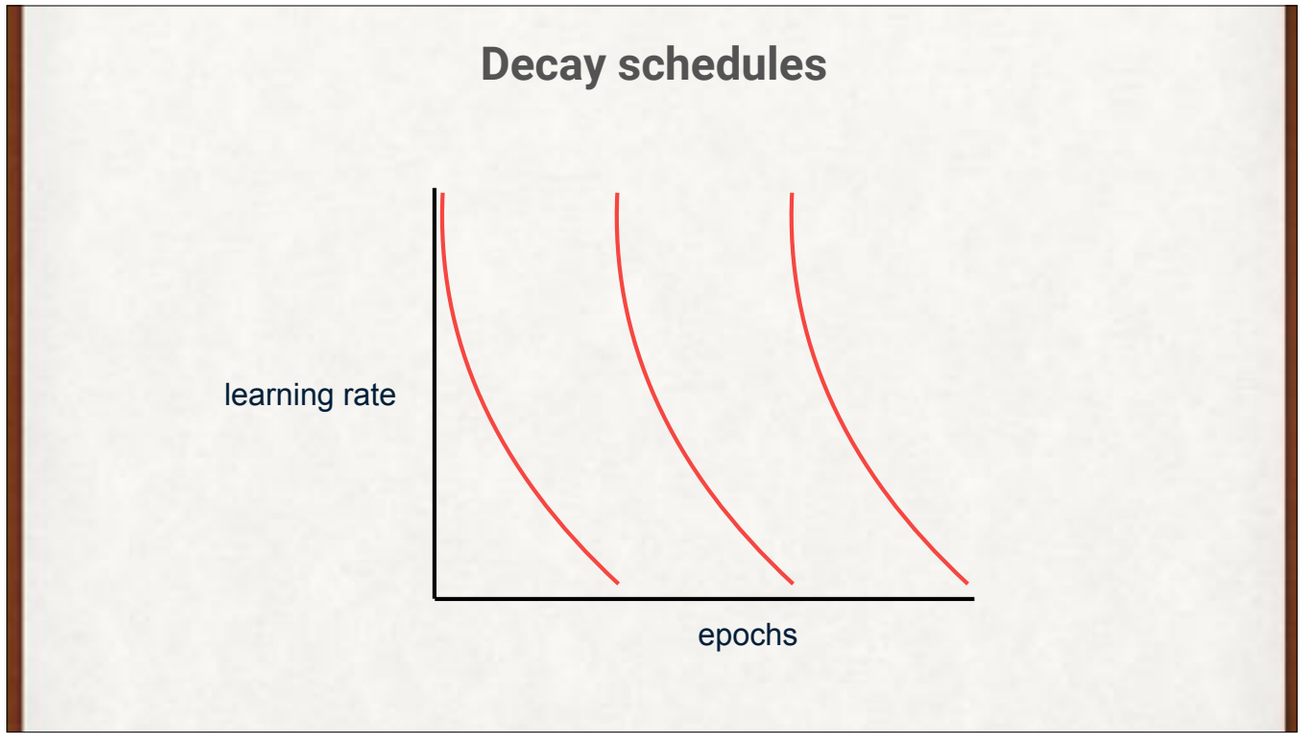
A useful strategy is to decrease, or “decay,” the learning rate over time, so we start with big steps and then get smaller. Think of using a metal detector at the beach: start with big steps, then smaller and smaller as we find the spot where the metal is hiding. Here are different strategies, or “decay schedules,” for reducing the error as learning goes by.



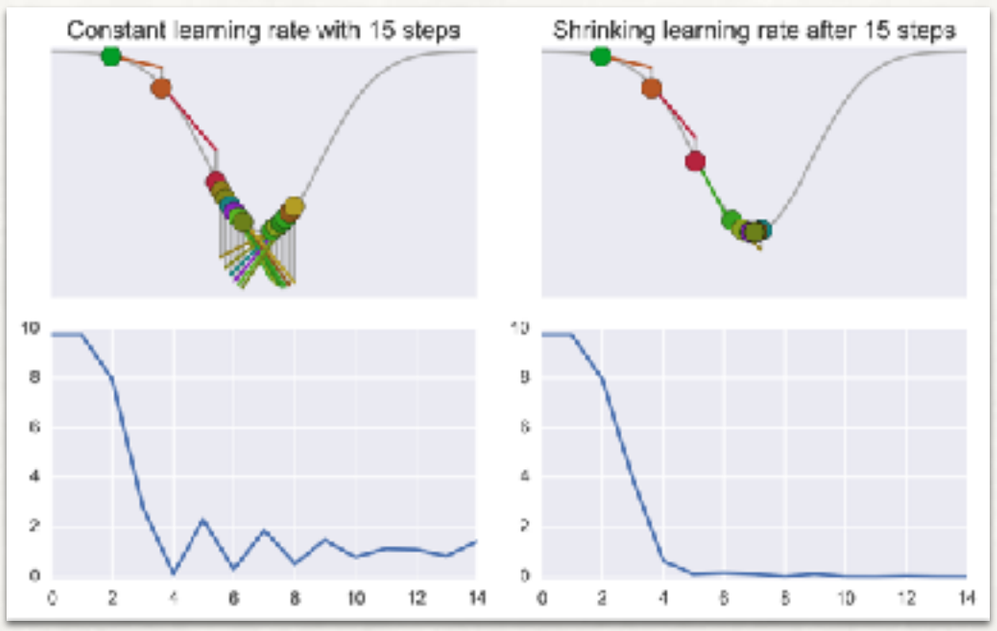
A useful strategy is to decrease, or “decay,” the learning rate over time, so we start with big steps and then get smaller. Think of using a metal detector at the beach: start with big steps, then smaller and smaller as we find the spot where the metal is hiding. Here are different strategies, or “decay schedules,” for reducing the error as learning goes by.



A useful strategy is to decrease, or “decay,” the learning rate over time, so we start with big steps and then get smaller. Think of using a metal detector at the beach: start with big steps, then smaller and smaller as we find the spot where the metal is hiding. Here are different strategies, or “decay schedules,” for reducing the error as learning goes by.

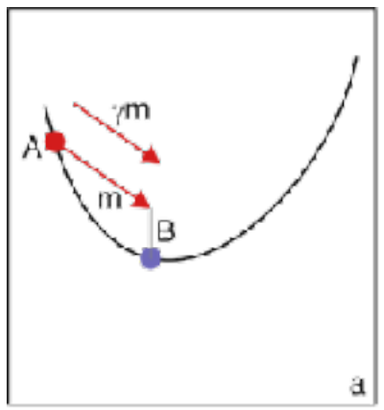


A useful strategy is to decrease, or “decay,” the learning rate over time, so we start with big steps and then get smaller. Think of using a metal detector at the beach: start with big steps, then smaller and smaller as we find the spot where the metal is hiding. Here are different strategies, or “decay schedules,” for reducing the error as learning goes by.



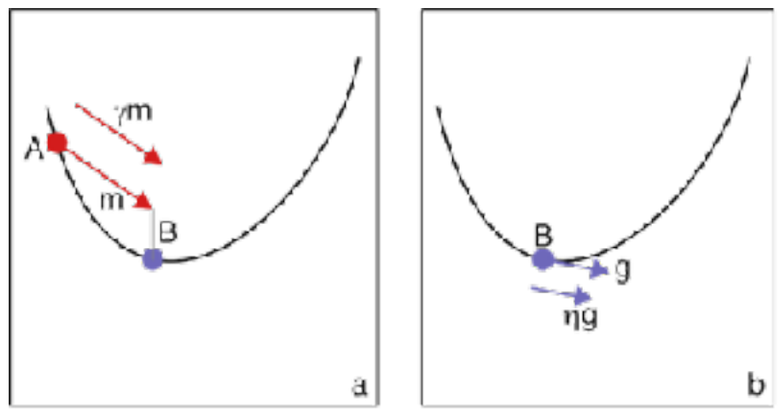
Shrinking over time works well here, and in most other situations.

“Momentum”



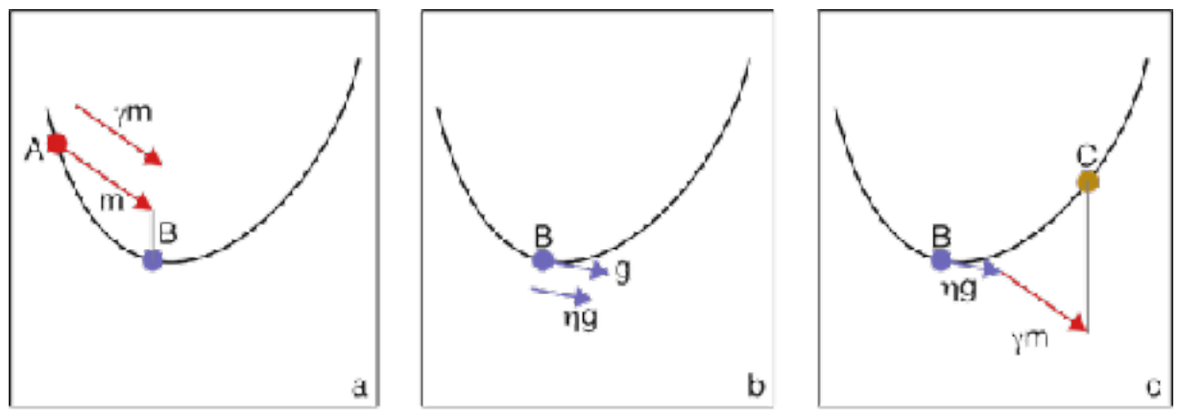
We can make gradient descent more efficient. Momentum (a physicist would call it inertia) says we should use a little bit of the last motion in the new motion. Here, m is the change from some previous value that got us to A. We scale that by gamma to get point B, which is just temporary. We find the gradient (g) at B, and scale that by the learning rate. Adding together (gamma * m) and (eta * g), we get point C, and that’s our next point (we forget about B). In this case we overshot, but gradient descent with momentum is usually more efficient than without.

“Momentum”

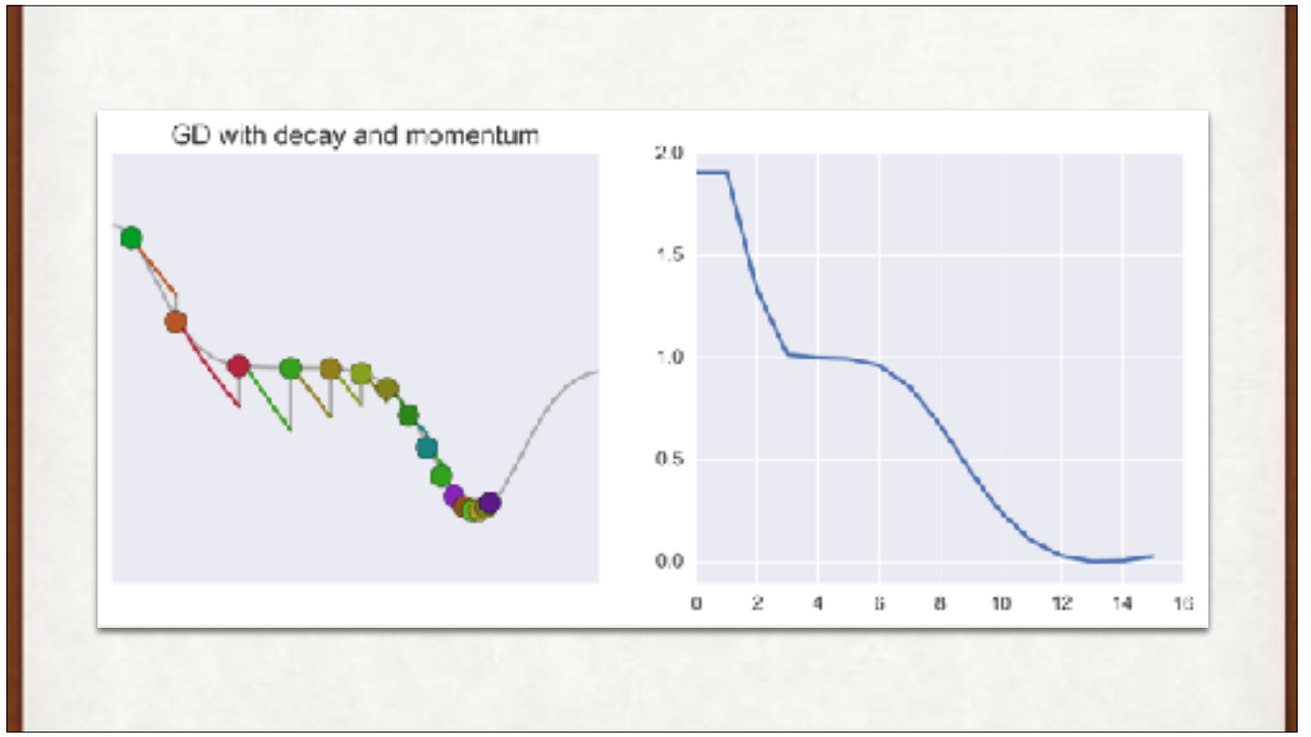


We can make gradient descent more efficient. Momentum (a physicist would call it inertia) says we should use a little bit of the last motion in the new motion. Here, m is the change from some previous value that got us to A. We scale that by gamma to get point B, which is just temporary. We find the gradient (g) at B, and scale that by the learning rate. Adding together (gamma * m) and (eta * g), we get point C, and that’s our next point (we forget about B). In this case we overshot, but gradient descent with momentum is usually more efficient than without

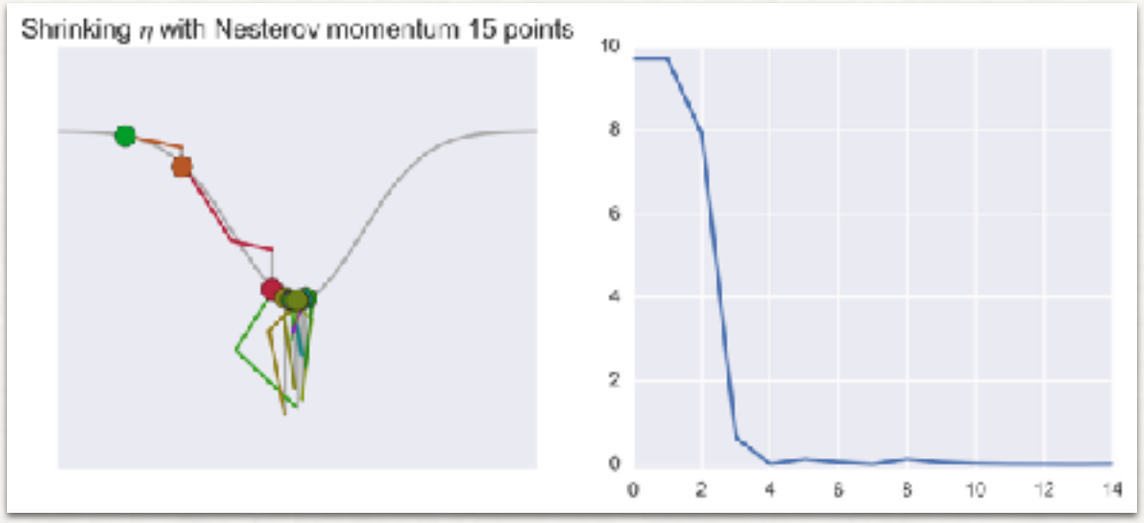
“Momentum”



We can make gradient descent more efficient. Momentum (a physicist would call it inertia) says we should use a little bit of the last motion in the new motion. Here, m is the change from some previous value that got us to A. We scale that by gamma to get point B, which is just temporary. We find the gradient (g) at B, and scale that by the learning rate. Adding together (gamma * m) and (eta * g), we get point C, and that’s our next point (we forget about B). In this case we overshot, but gradient descent with momentum is usually more efficient than without



Momentum gets us over the plateau (as long as it’s not too long). Notice that the momentum decreases as we move across the flat zone, but in this case it was enough to get us to the drop.



Nesterov momentum is a little more complicated, but works even better than plain momentum. It’s a cool idea because it combines information from where we’ve been (momentum) with information from where we’re going, to find out where to go now. The details aren’t hard, but by now we have the general idea: Gradient descent, good. Tweaking gradient descent, better.

Convergence

We almost never get “clean” data to train our systems. Making sure our data is consistent and well-formatted (or “clean”) is essential, and often a big piece of the overall work.

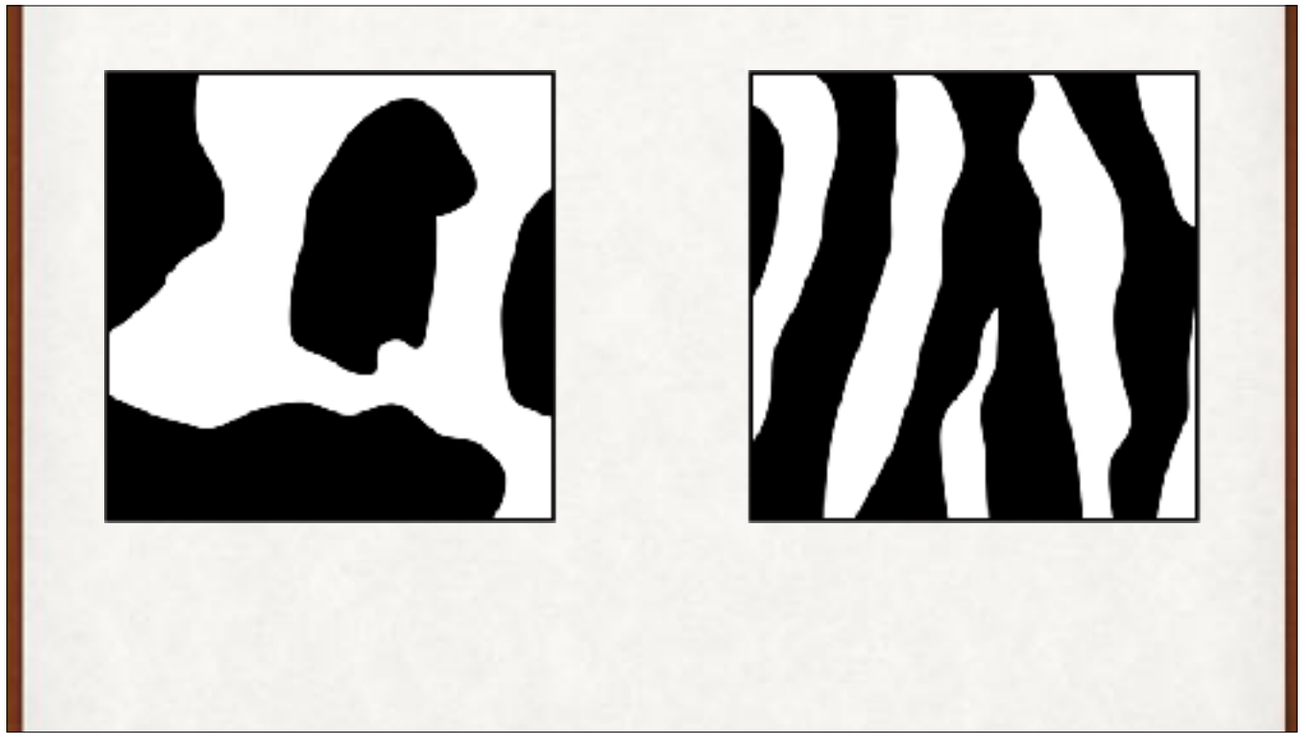
Take a breath. Clear your mind. Go to the happy place. We’re switching to another new topic.

Data

We almost never get “clean” data to train our systems. Making sure our data is consistent and well-formatted (or “clean”) is essential, and often a big piece of the overall work.

Preparing Data

We almost never get “clean” data to train our systems. Making sure our data is consistent and well-formatted (or “clean”) is essential, and often a big piece of the overall work.



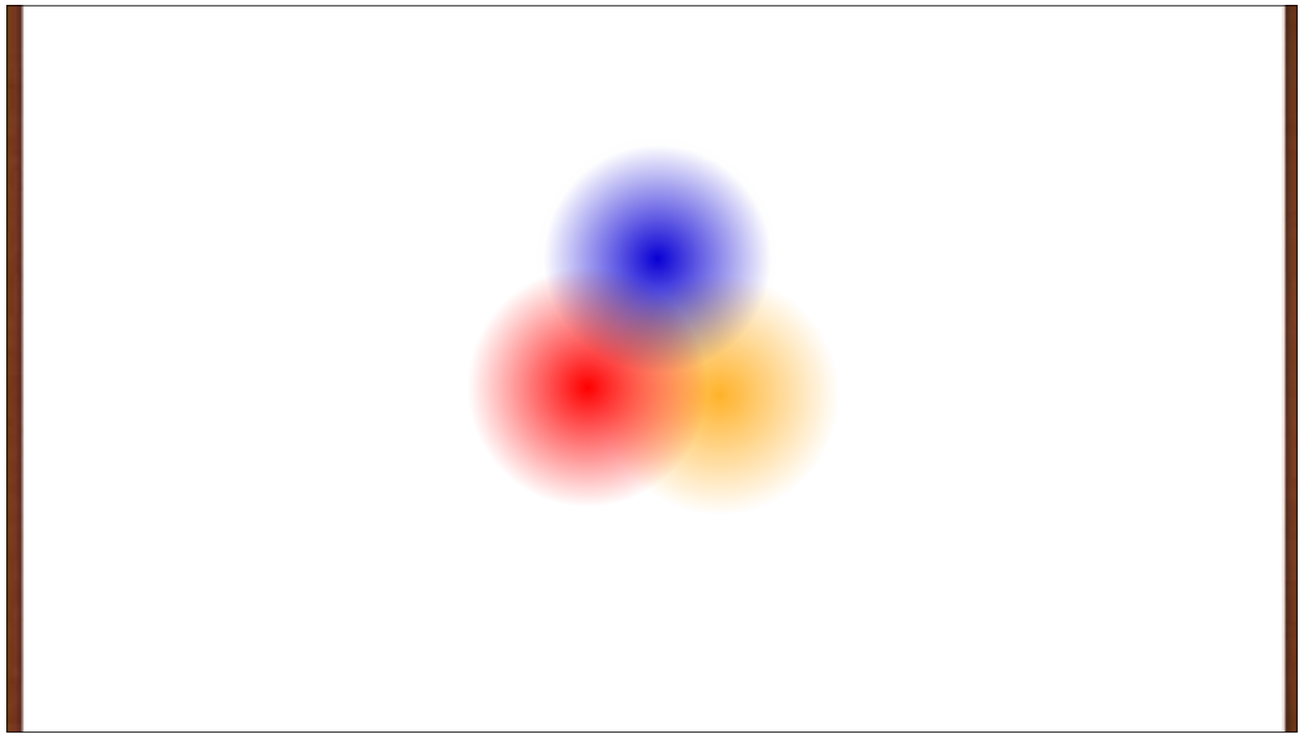
Suppose we've taught our system how to distinguish the patterns on cows and zebras by analyzing close-ups of the patterns…

…and later, some well-meaning user gives us pictures of the whole animals. These aren’t the kind of images we trained on, and the results probably won’t be great. So we must be vigilant to keep all of our data, from training to real-world use, structured consistently.

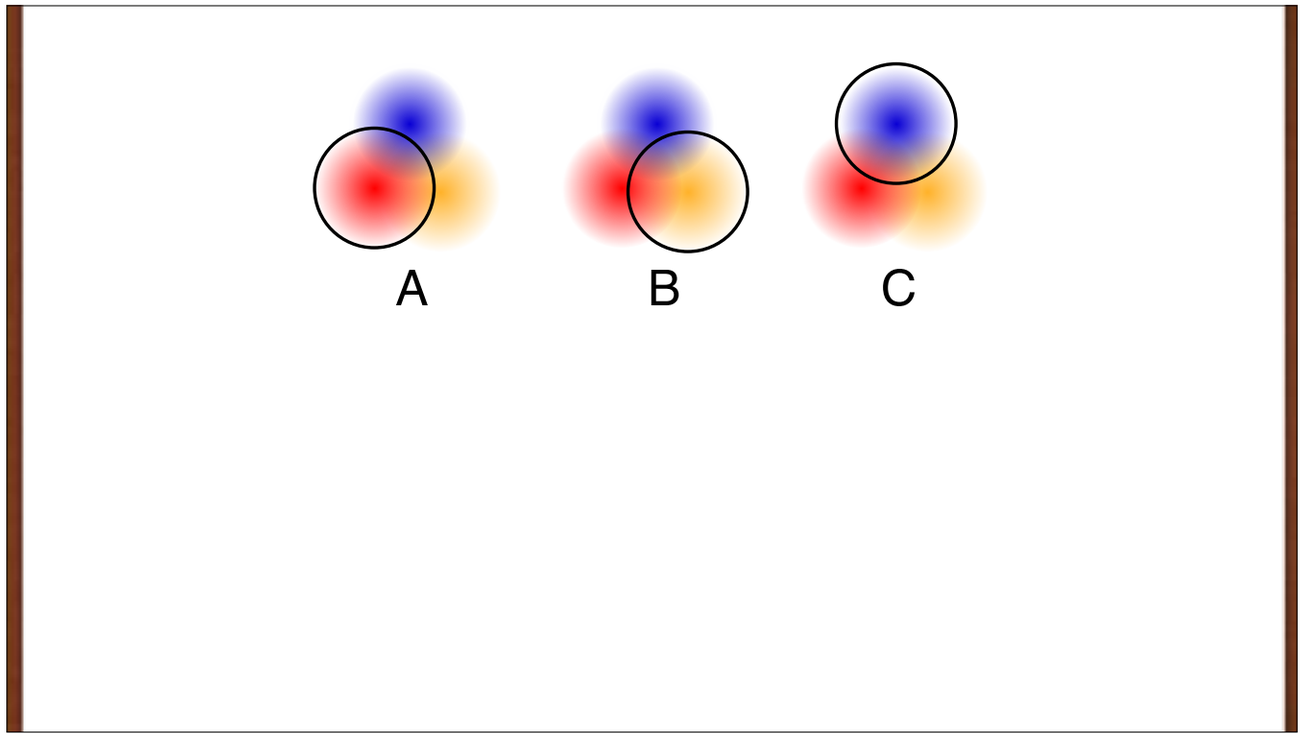
Dimensionality Reduction

It’s efficient to reduce the size of our data when we can. A good way to do that is to reduce the number of dimensions in each piece of data.

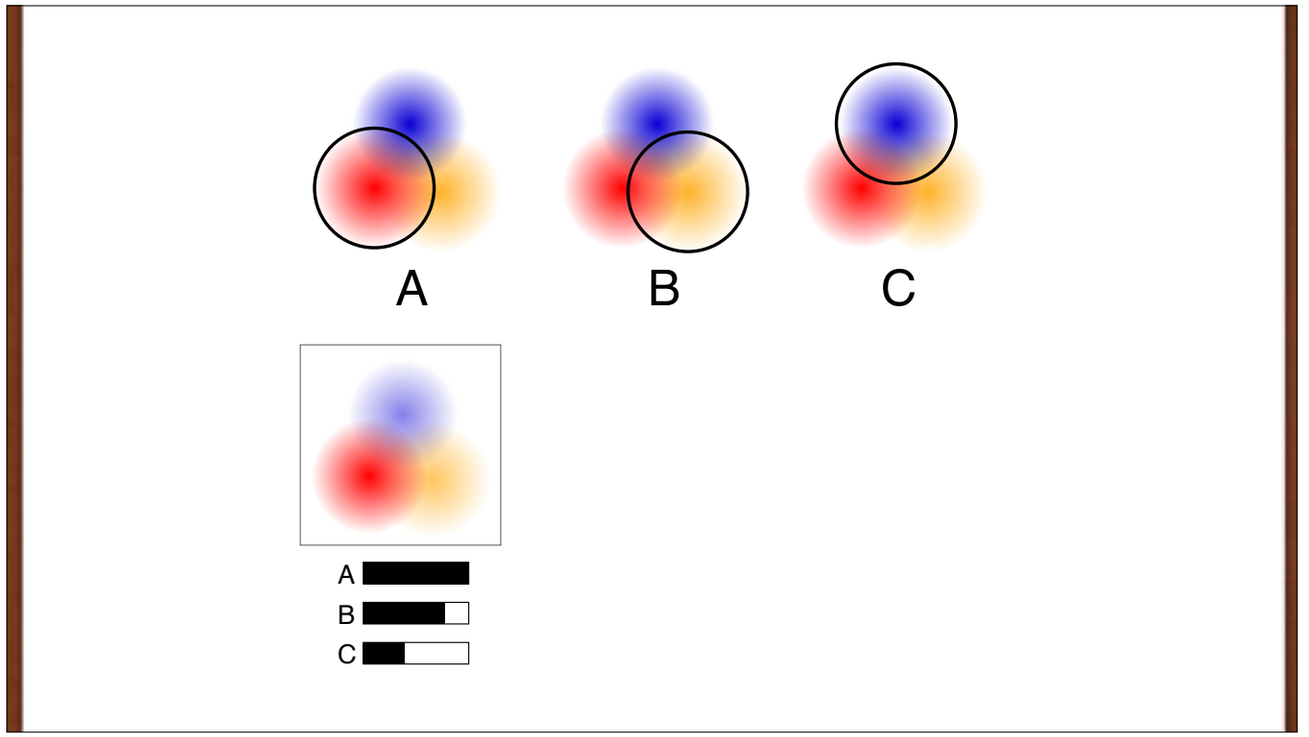
We can replace the many pixels in each image with just three numbers. Then our system only needs to read and learn about three values, not tens or hundreds of thousands.



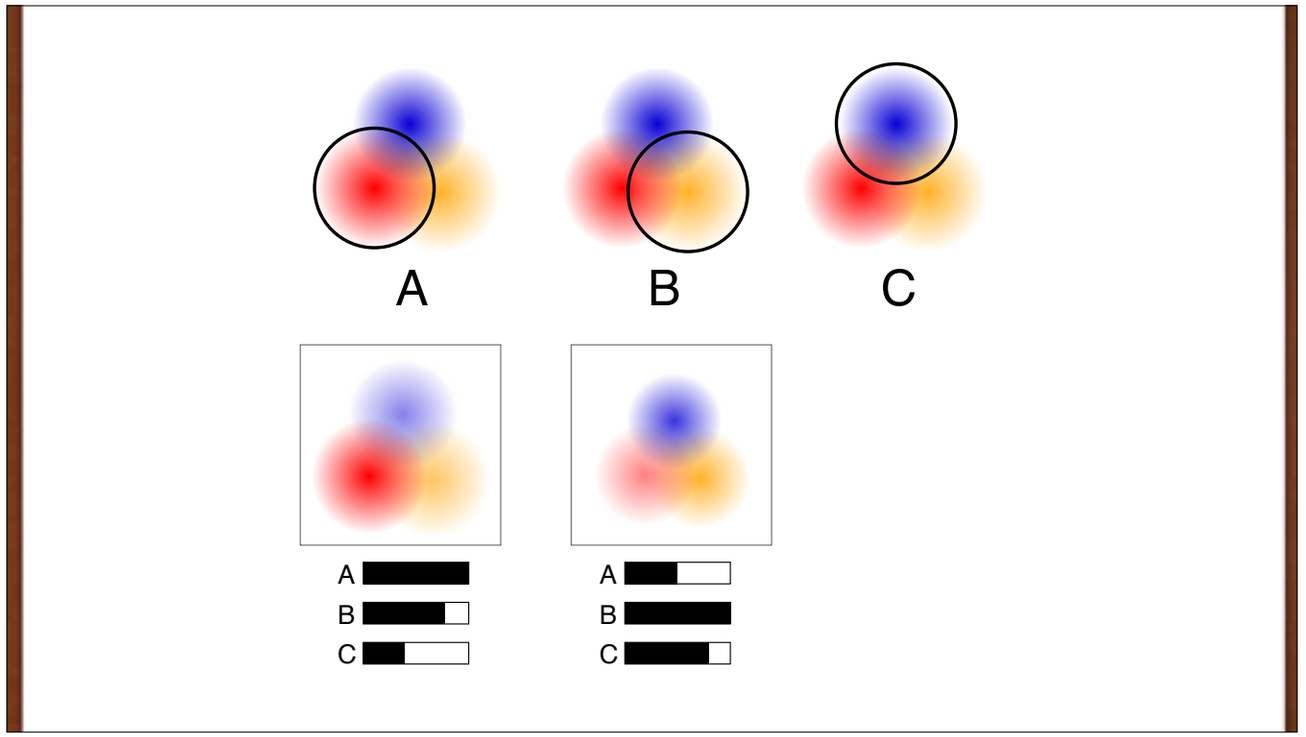
The system can determine that all of our images are built from these three components, scaled by different amounts and summed.



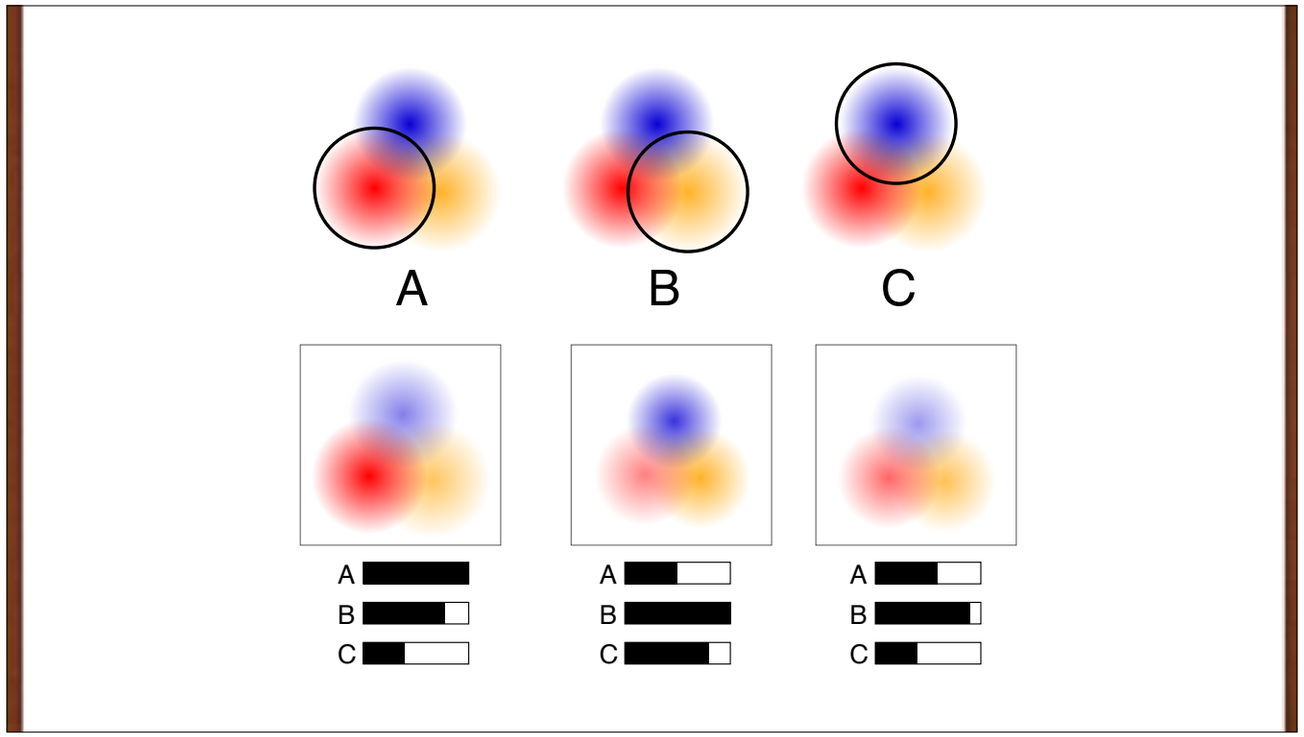
The system can determine that all of our images are built from these three components, scaled by different amounts and summed. Here are the individual components.



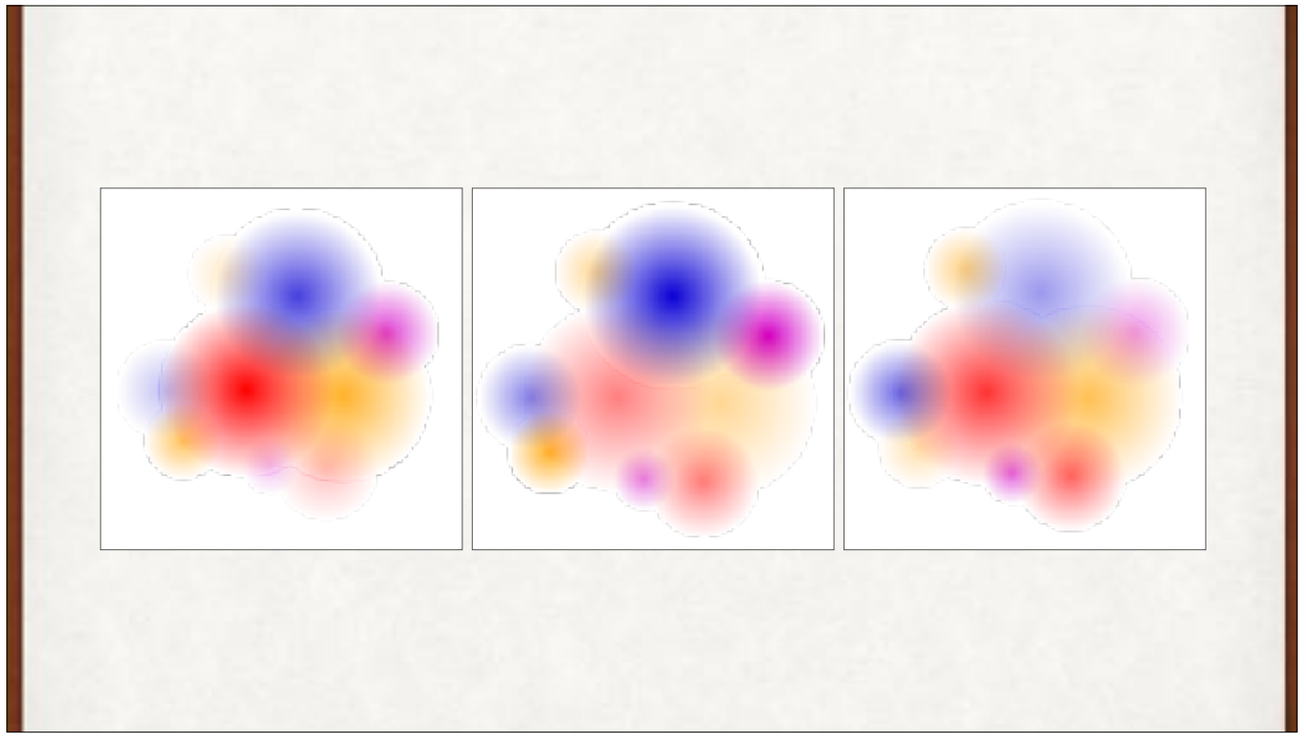
By weighting A, B, C and adding them together, we get the result. We need only 3 numbers, not entire RGB images.



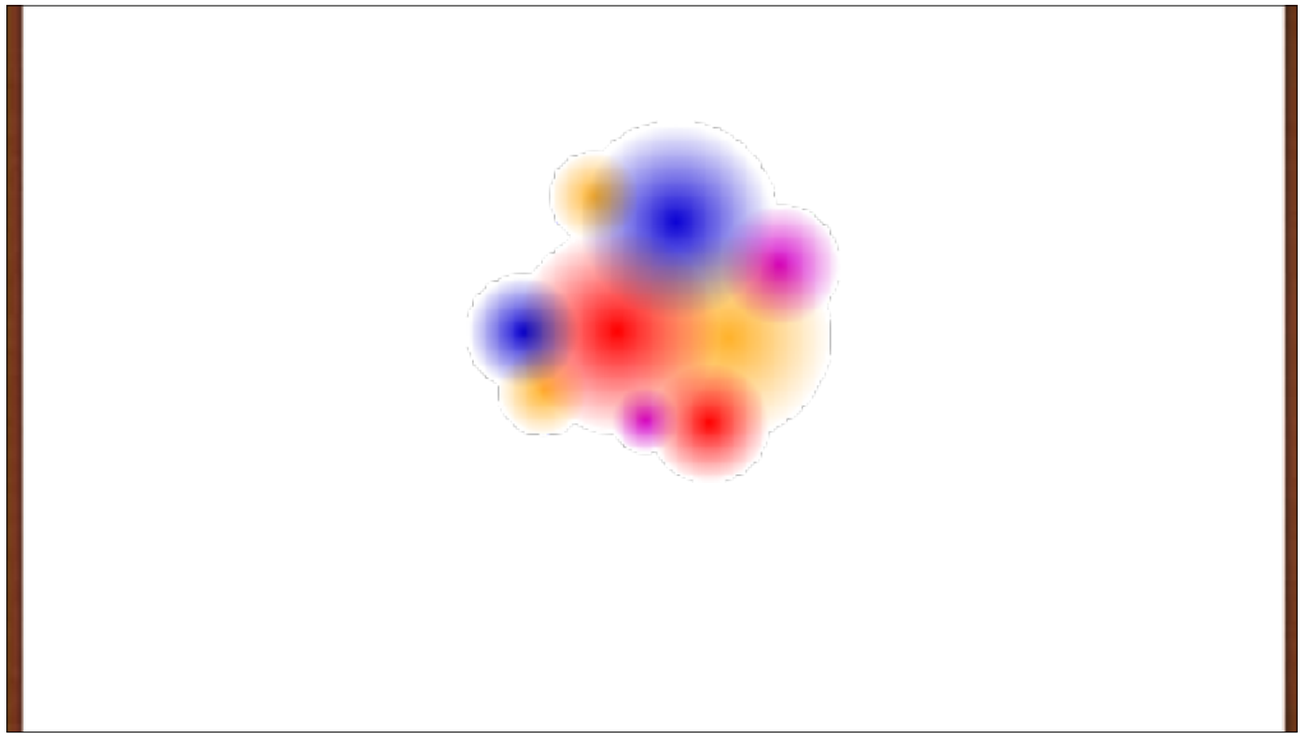
By weighting A, B, C and adding them together, we get the result. We need only 3 numbers, not entire RGB images.



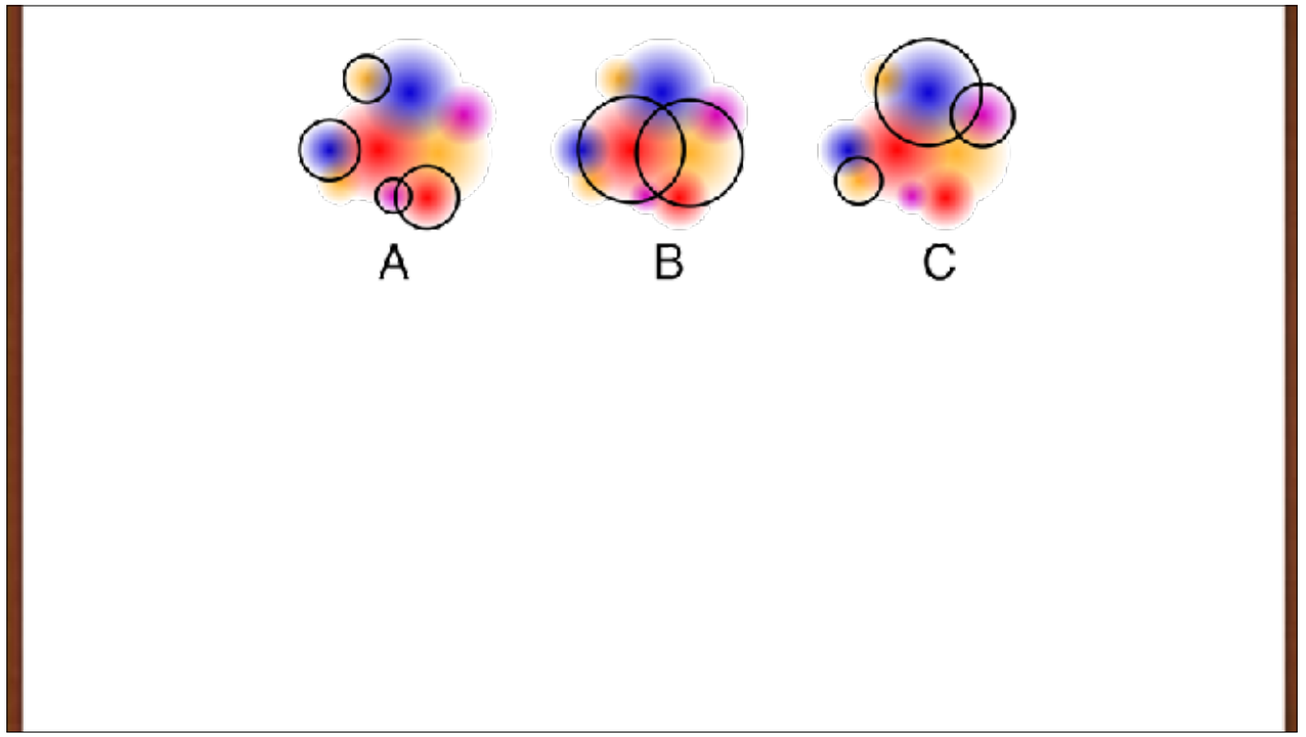
By weighting A, B, C and adding them together, we get the result. We need only 3 numbers, not entire RGB images.



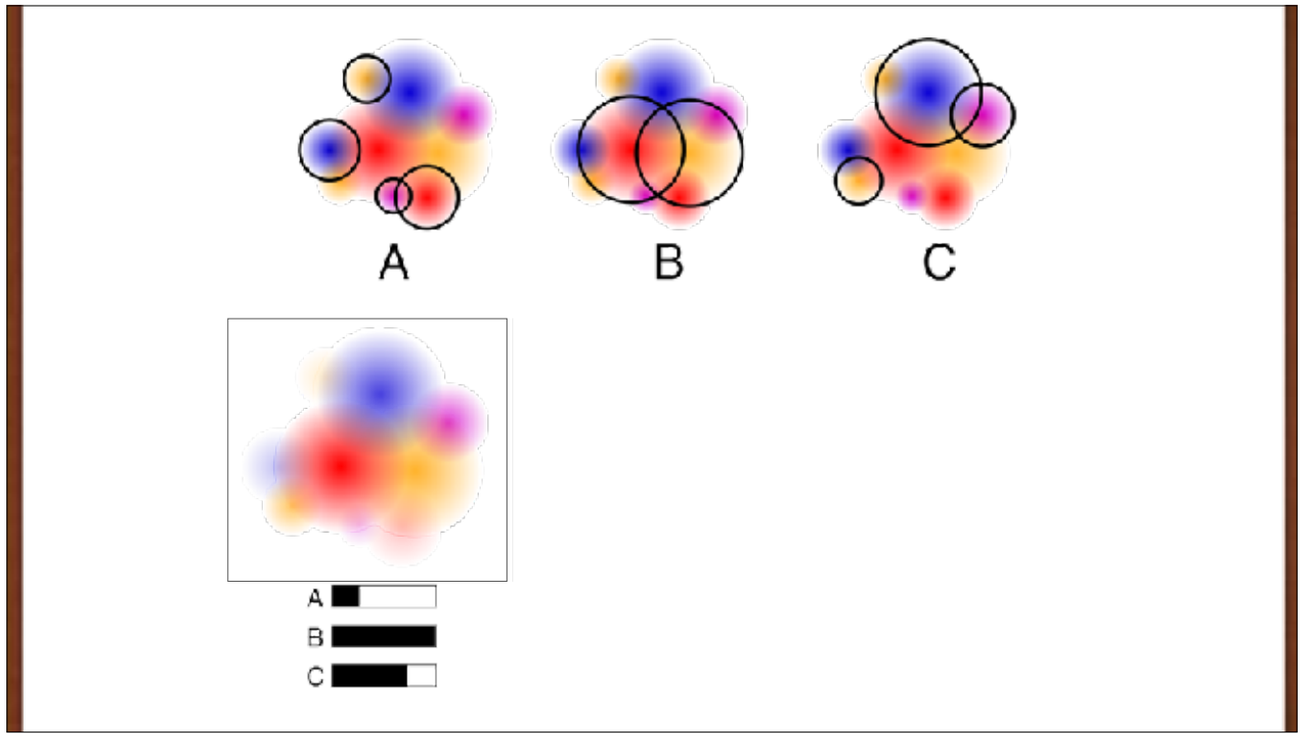
A more complex case. How about these three images?



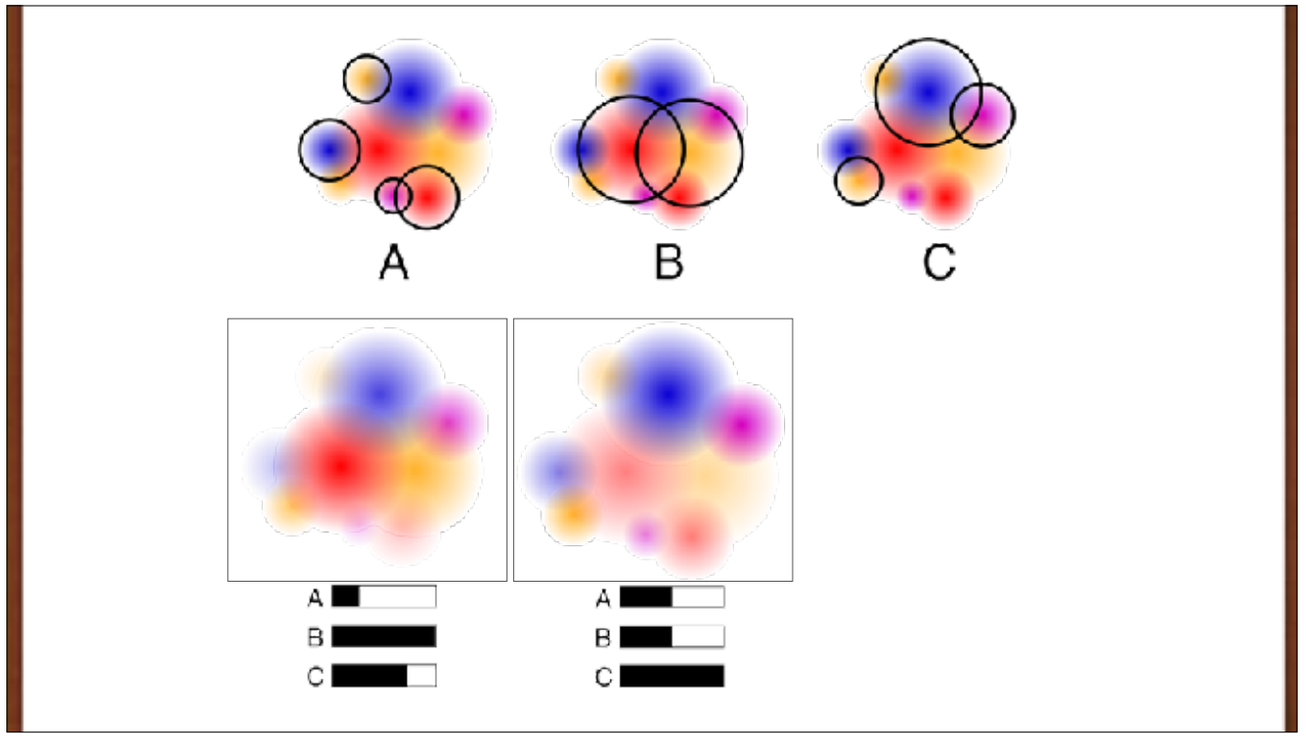
The system find that they’re all made from these components…



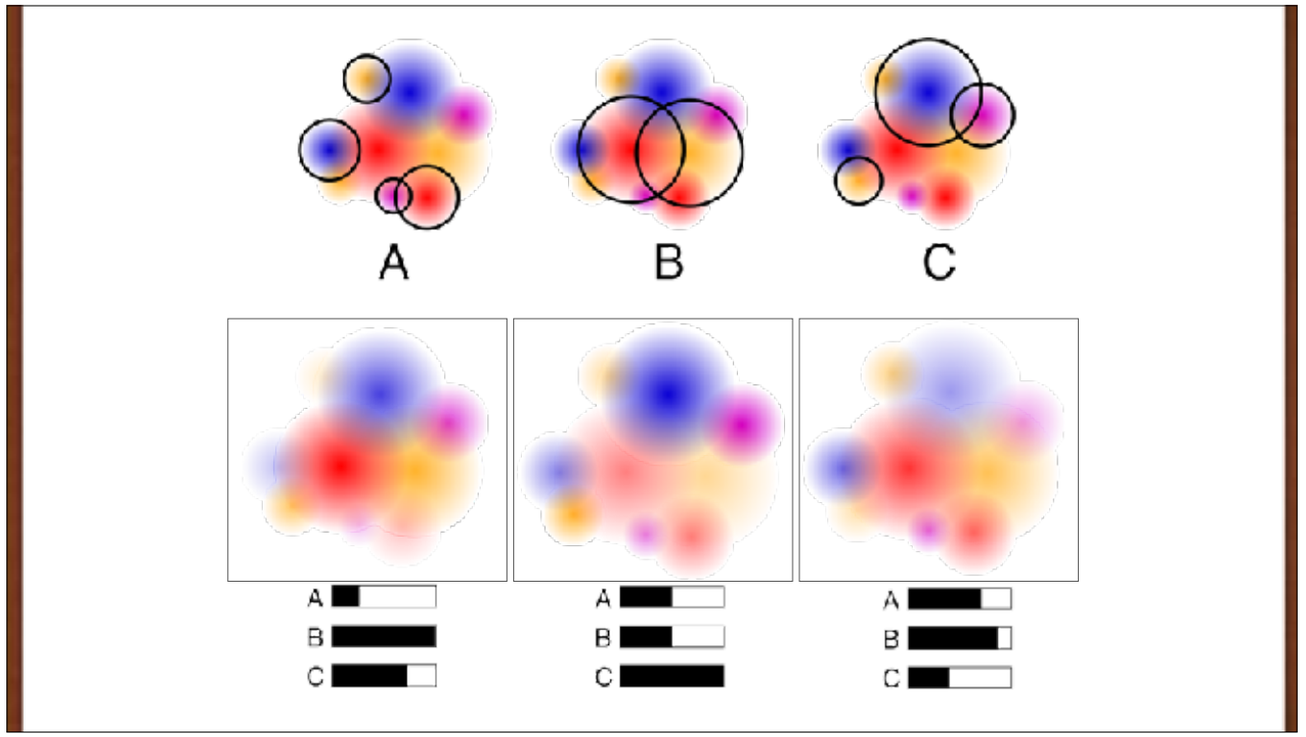
Furthermore, they’re grouped in three clusters. We just scale and sum the clusters.



Again, scale A, B, and C and sum them. Again, only 3 numbers are needed.



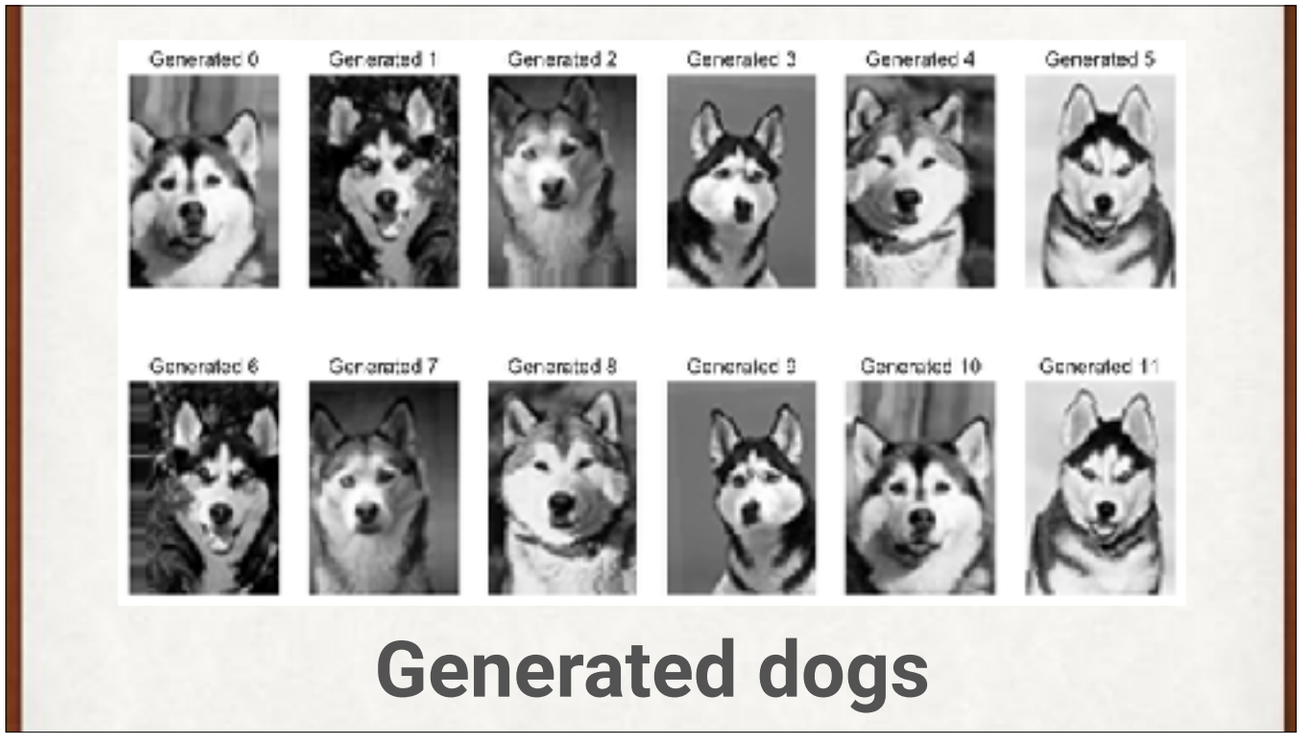
Again, scale A, B, and C and sum them. Again, only 3 numbers are needed.



Again, scale A, B, and C and sum them. Again, only 3 numbers are needed.

Starting dogs

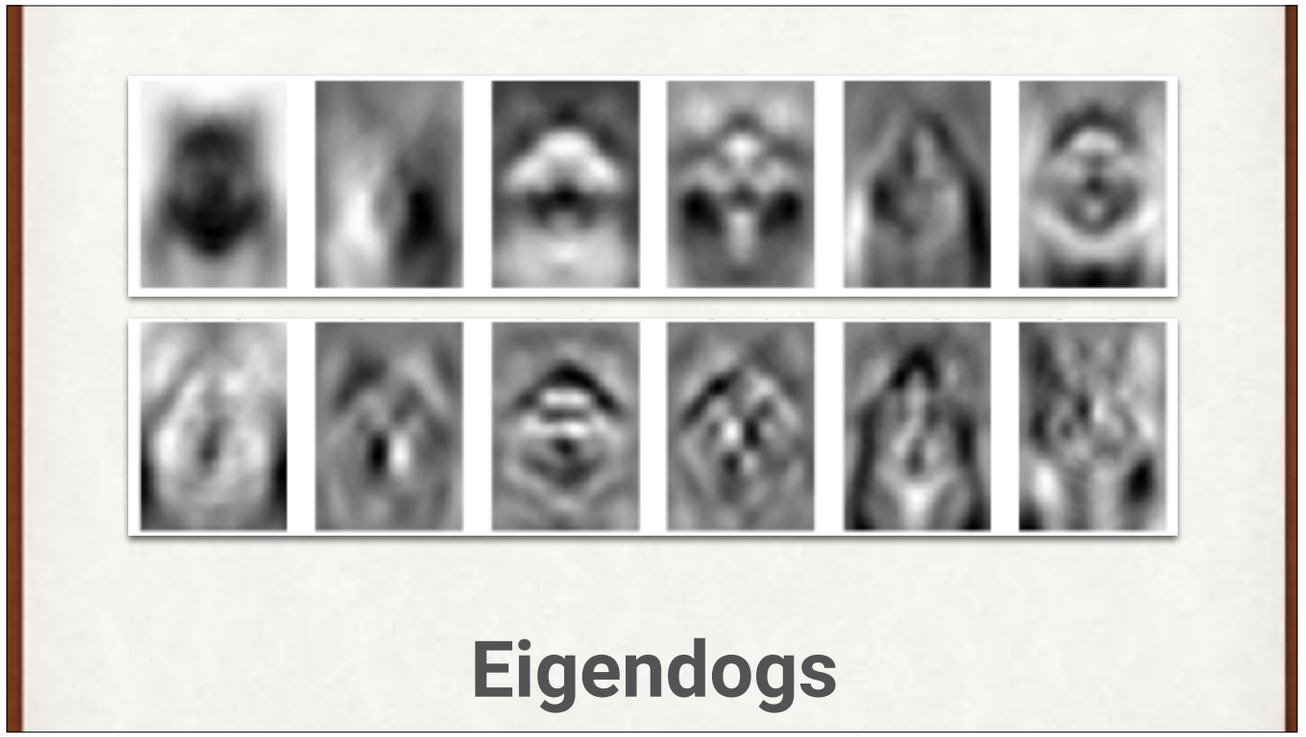
Pictures contain a lot of data. Big pictures can contains millions of numbers. We can save time and resources by crunching the images down into smaller numbers. We could do that by just making the image smaller, but when our data is more abstract, it’s hard to work out what that can mean. Luckily, there are tools to help us represent our inputs with controllable amounts of information. Let’s apply dimensionality reduction to pictures of dogs. In this example, we’ll focus on these Huskies.



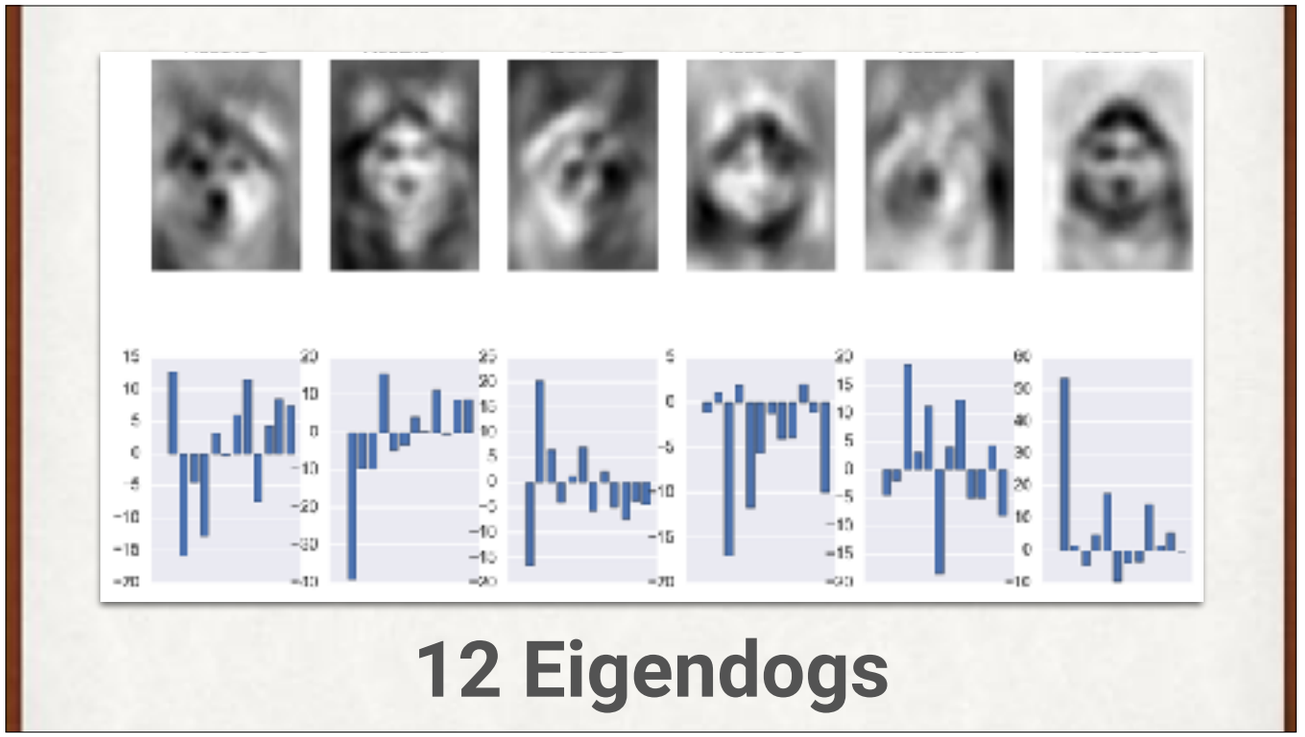
Generated dogs

Six isn’t a big training set. We can randomly rotate, flip, scale, and otherwise modify the starting data by little amounts to make more data. We can make thousands and thousands of similar, but unique, dogs.

We can apply some numerical adjustments to make our dogs easier for the system to learn from. Here’s the result. The dynamic range of each of these dog images has been reduced somewhat, but if we’re using real numbers (rather than integers from 0 to 255), we haven’t lost information (well, except for normal floating-point stuff).

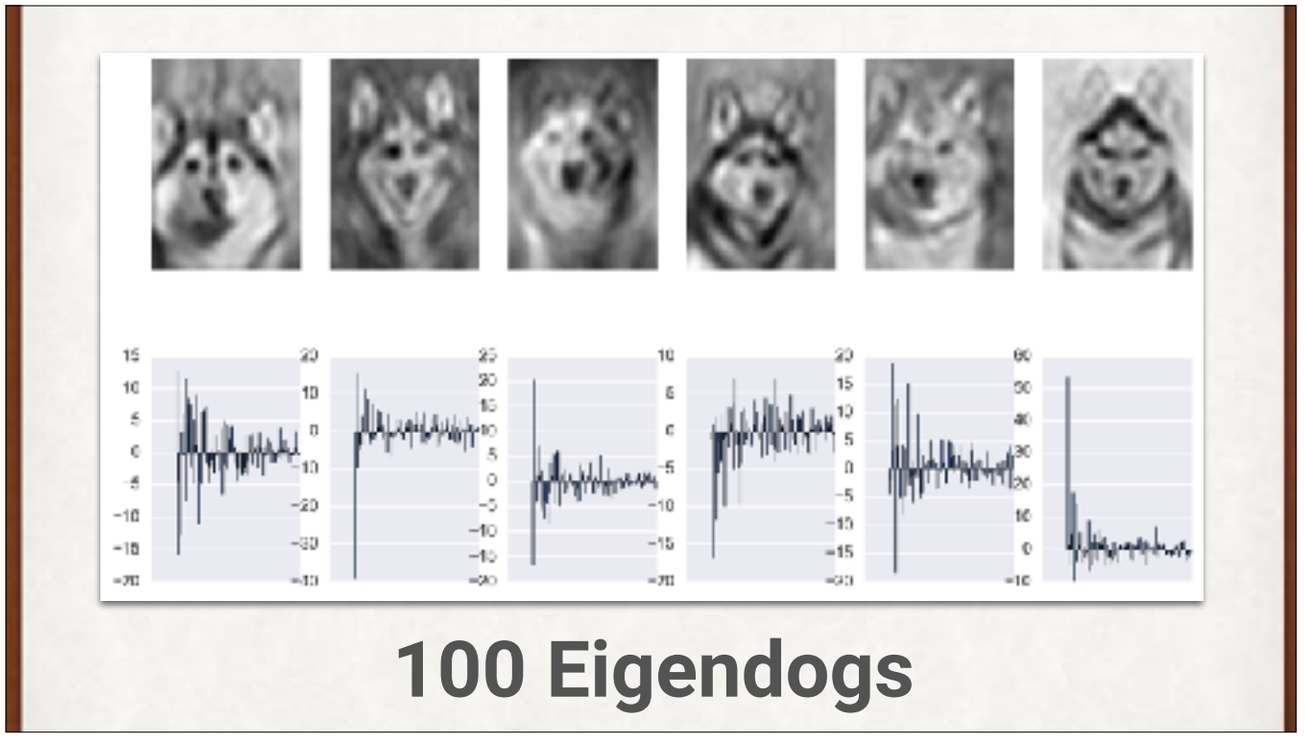


We will find the “most important” parts of the dog images with an algorithm that finds their “eigenvectors,” or important components. These are dogs, so we’ll call the eigenvectors by the much better name eigendogs. Here are the first 12 eigendogs from our data. By mixing these together with different weights, we can recreate any of the inputs. So with just 12 numbers (one for each weight), and these 12 images, we can describe every input image. Let’s do that for 6 dogs…



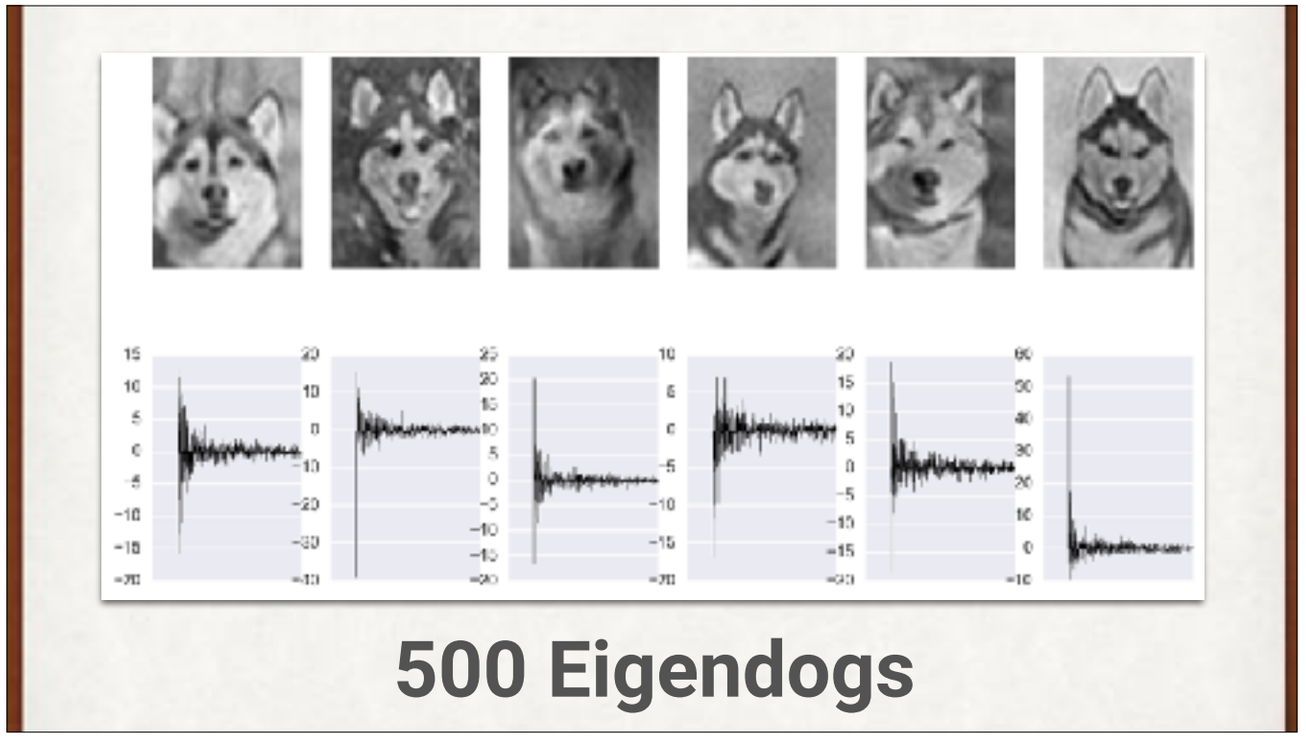
12 Eigendogs

…and we get images that are recognizable, but blurry. Using only twelve components sort-of produces doggish images, but they’re not great. Asking for more of these eigendogs means that we can get back more detail.



100 Eigendogs

100 eigendogs is more eigendogs, giving us better results.



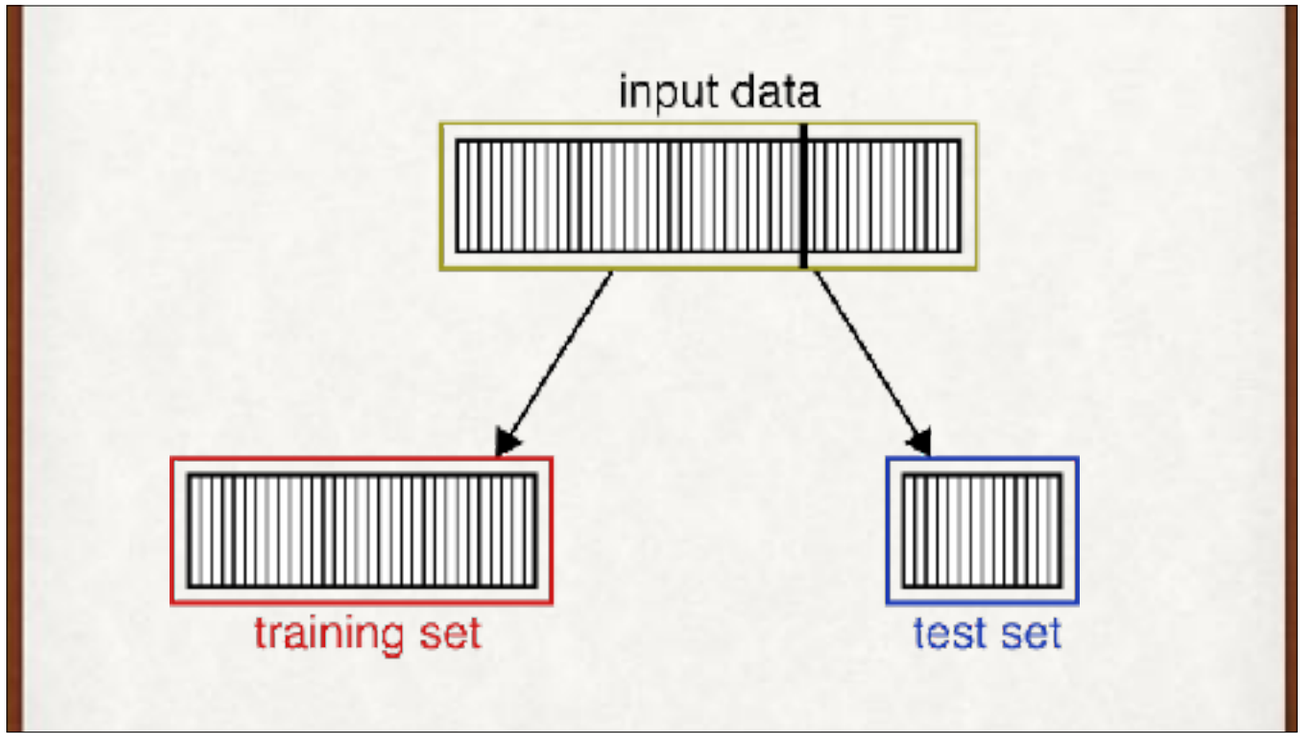
500 Eigendogs

Even more eigendogs, even better results. Even here, we need only 500 numbers per image, rather than the thousands (or more) that make up each image. We also need the 500 eigendog images. But if we have 100,000 input images, then the cost of saving the 500 eigendogs is dwarfed by the savings in using only 500 numbers for each image, rather than the value for every pixel.

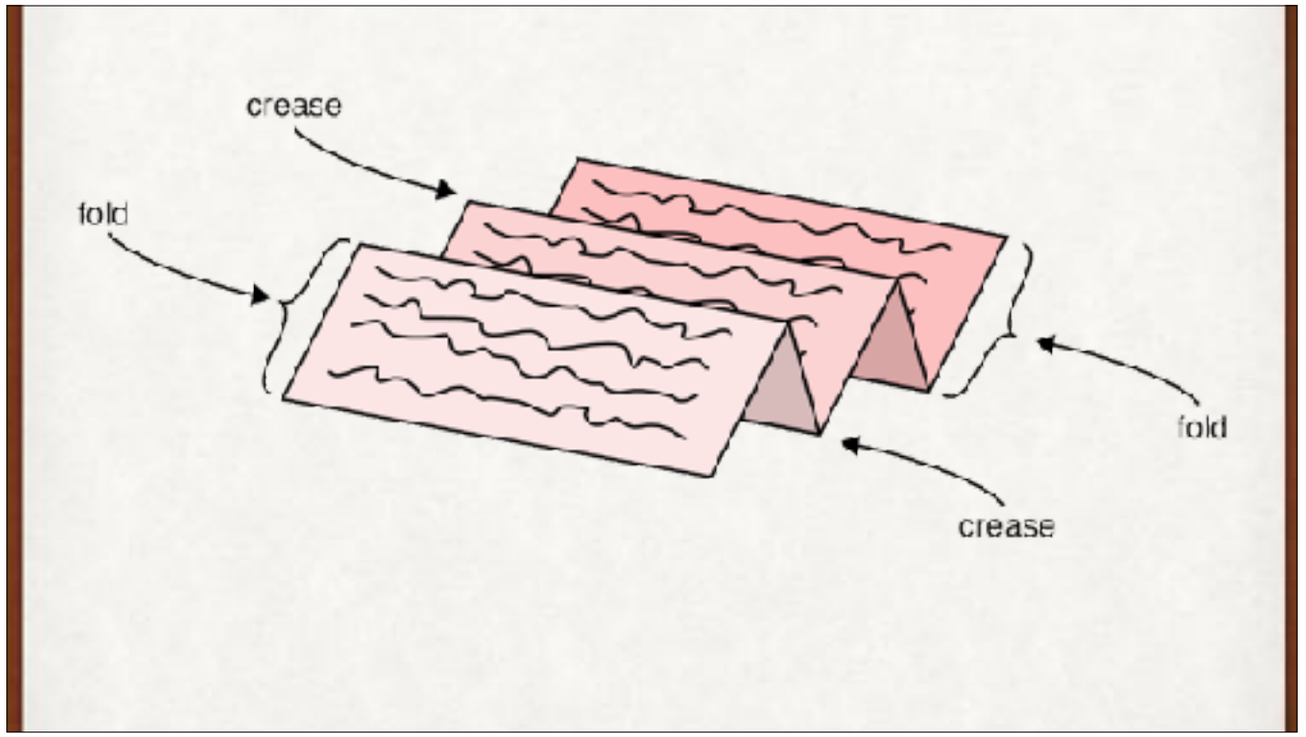
Palette cleanser. More sherbet, please!

Evaluating accuracy: Cross-validation

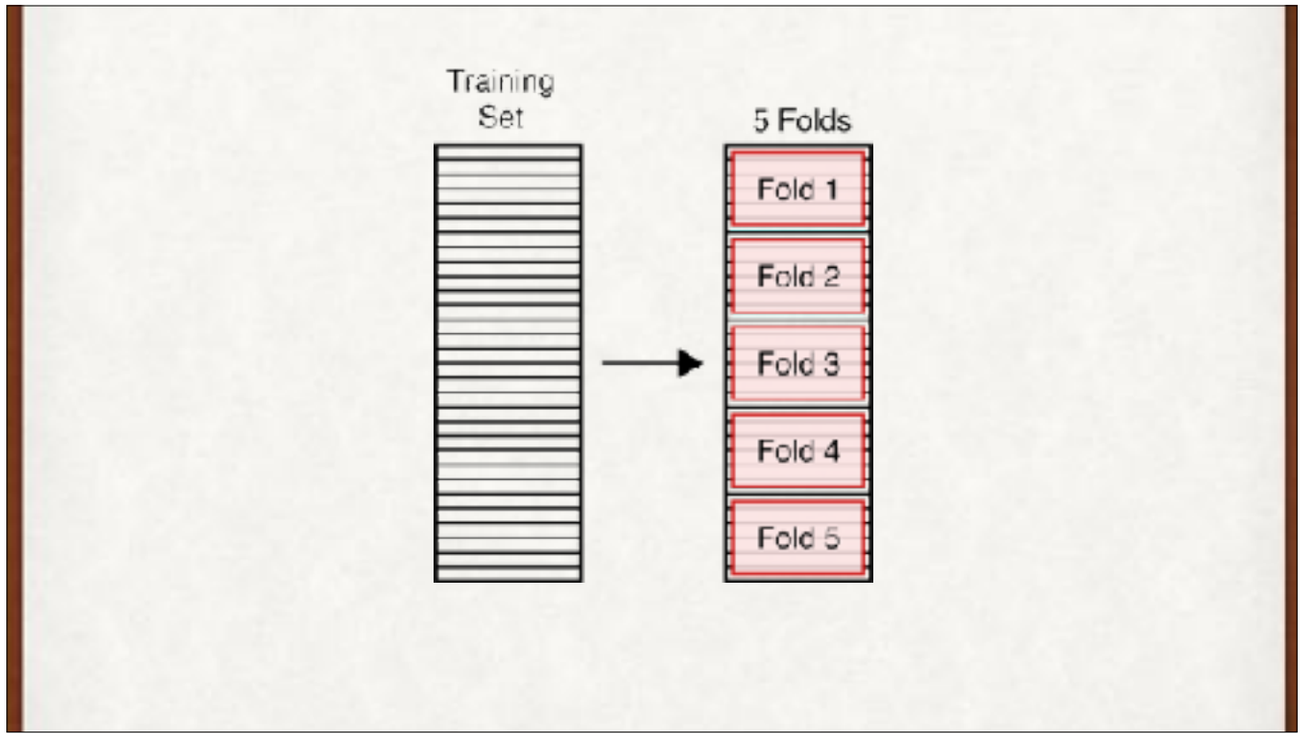
When is our system “good enough” to “deploy,” or make available in the real world? There is currently no way to predict this. The only way to know is to give it new data that it has never seen before, and see how it does. If it performs well enough, we’re set. If not, it’s up to us to somehow make it better. In this step, DL is a lot like experimental chemistry or biology: you make something using your experience and best guesses, and then try it out and see how it actually performs. Usually it’s not nearly as good as we hope, and we have to return to the drawing board. In DL, we often improve the network in tiny steps, hoping that ultimately, the accumulated changes will make it “good enough.”



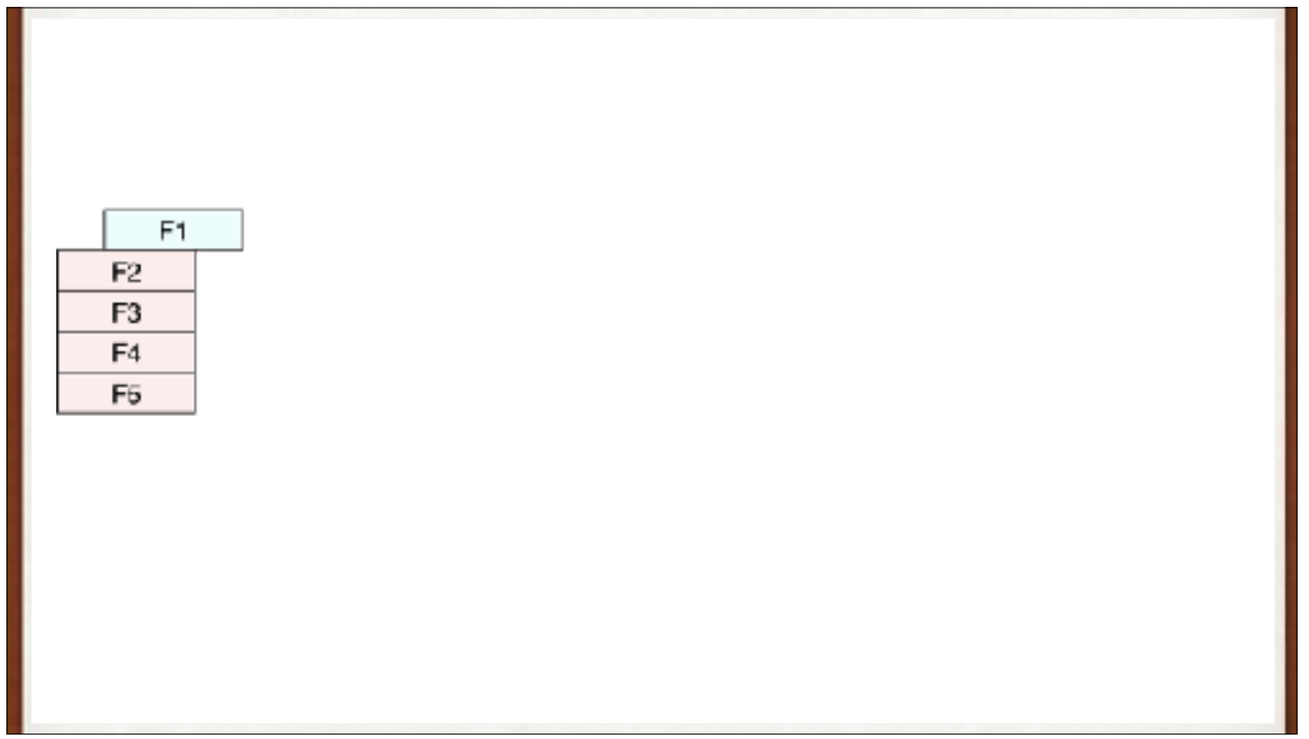
We split the incoming data into training and test sets. We train with one, and then evaluate performance at the very last step with the other. It’s essential that the system never sees the test set during training, or we get “data leakage,” and probably an overly optimistic measure of the system’s accuracy.



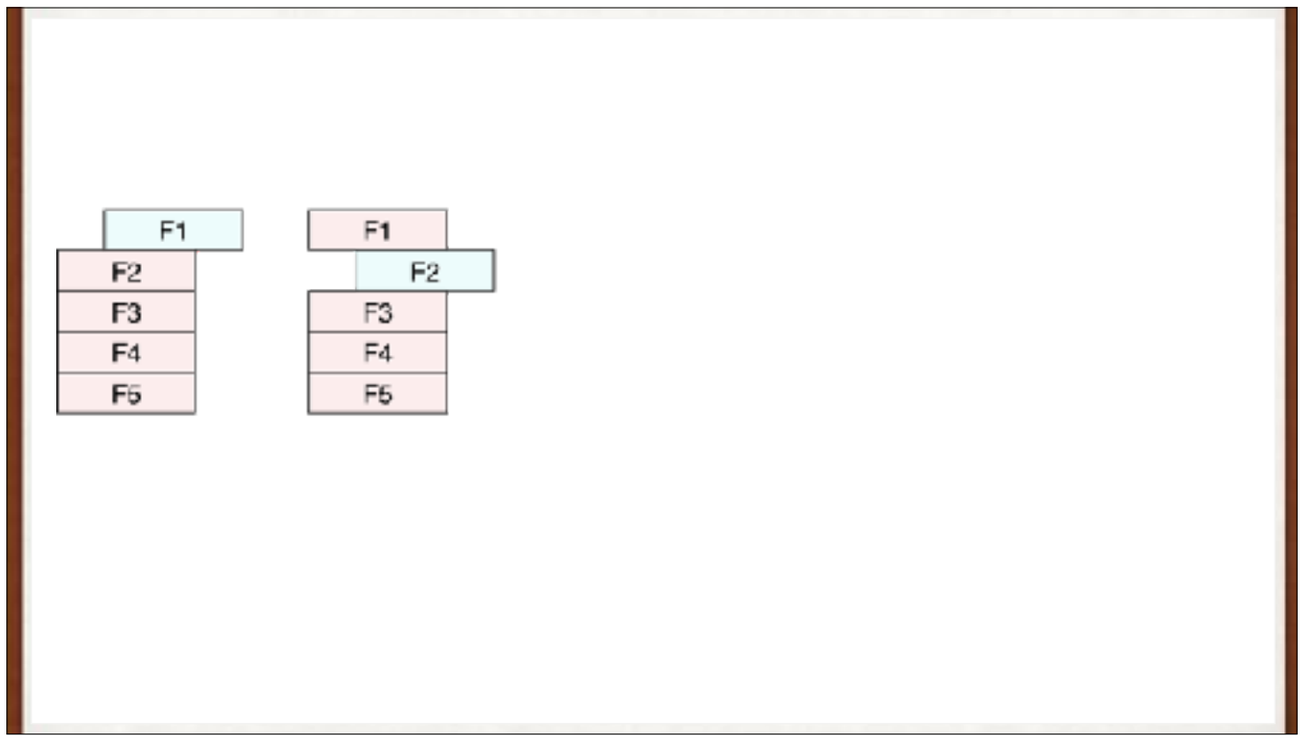
When we don’t have a lot of data, it would be a shame to set aside a separate validation set, because we want to use all of our data for training. Suppose we have 500 pictures from Pluto. We want to use every single one of them in training. And we can’t get more. To estimate accuracy, we can use cross-validation. This is a method for creating validation sets on the fly. We chop up our training data into multiple pieces, called “folds.”



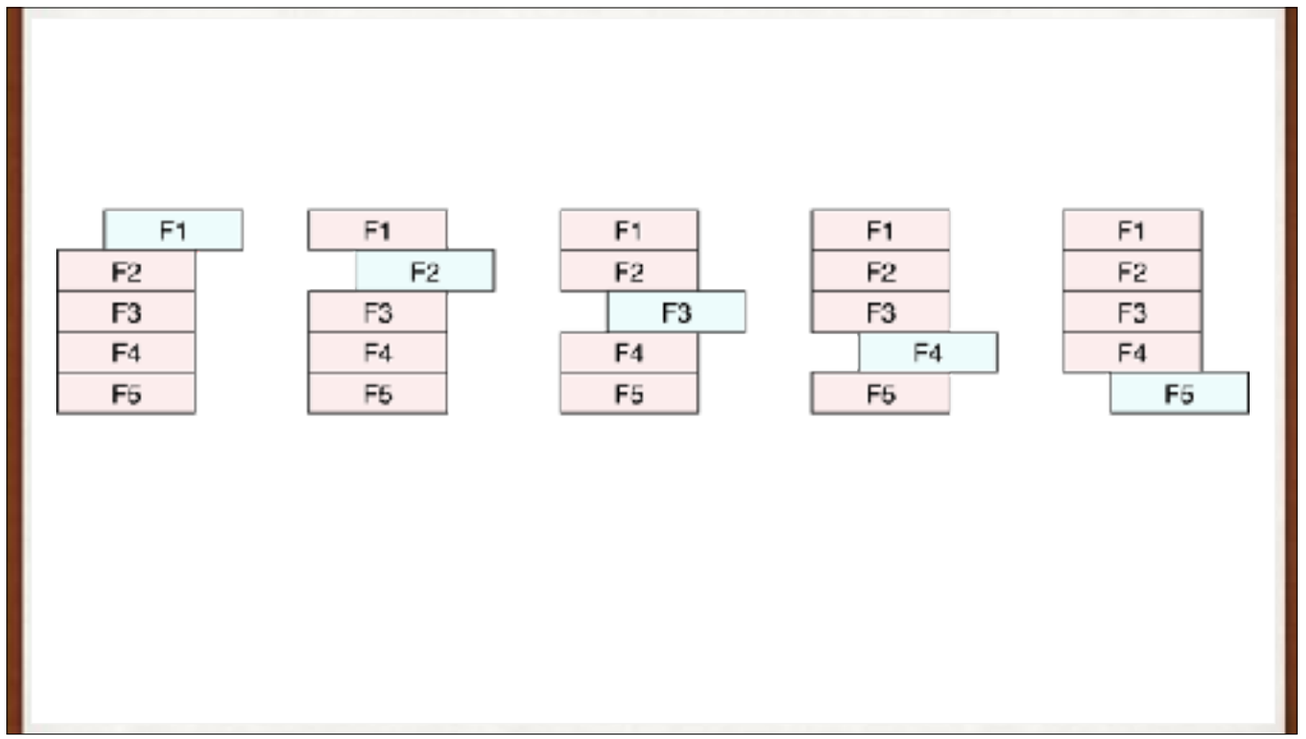
A training set split into 5 folds.



Also called **rotation estimation** or **out-of-sample testing**. To evaluate performance of our network, we'll train on folds 2 through 5, then test (or validate) with fold 1. Then we'll train with folds 1 and 3-5, and test (or validate) with fold 2. When we've done this for all 5 folds, we average the performance and that's our estimate of the system's accuracy.



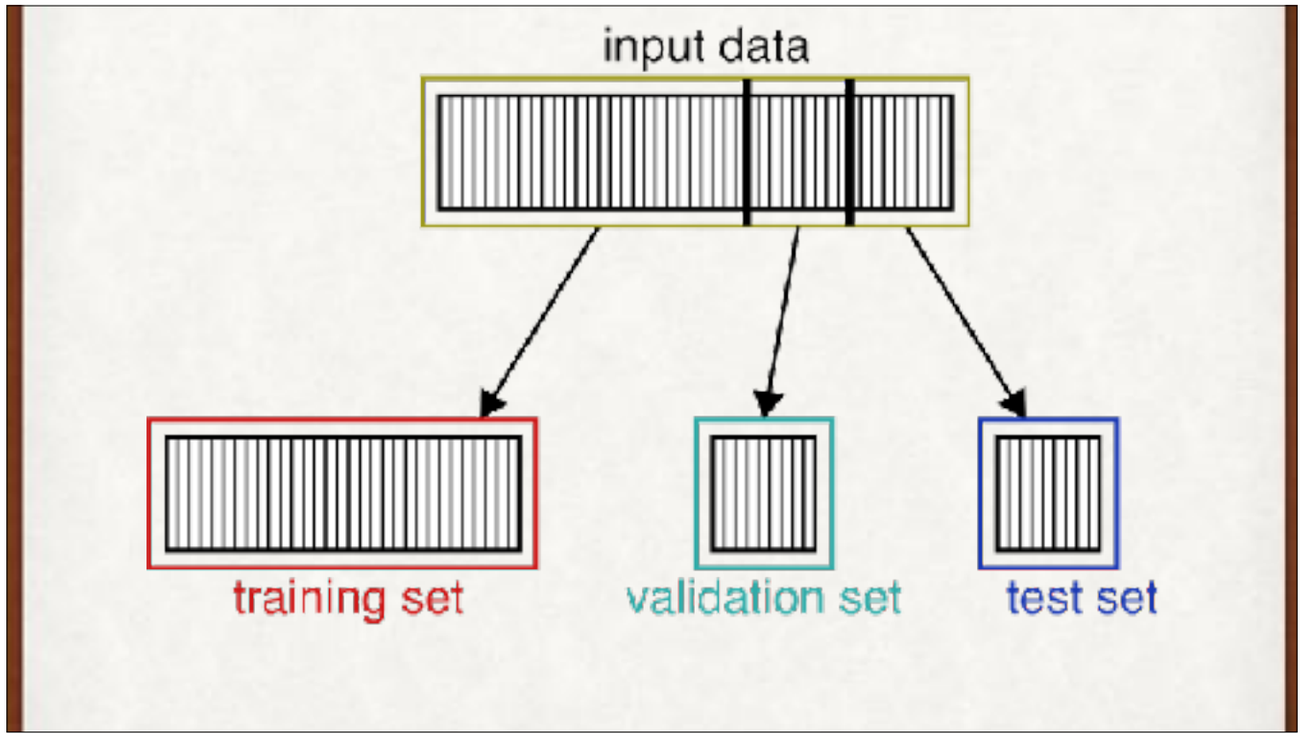
Also **rotation estimation** or **out-of-sample testing**. To evaluate performance of our network, we'll train on folds 2 through 5, then test (or validate) with fold 1. Then we'll train with folds 1 and 3-5, and test (or validate) with fold 2. When we've done this for all 5 folds, we average the performance and that's our estimate of the system's accuracy.



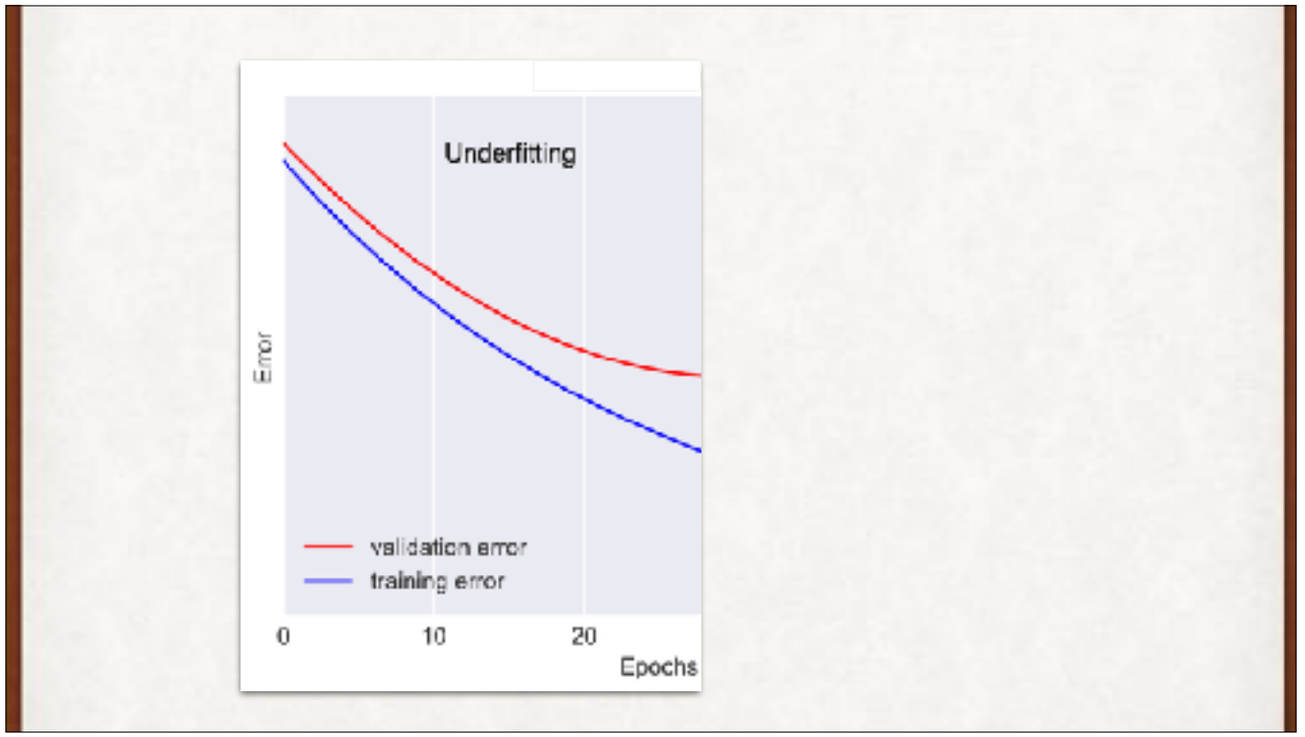
Also **rotation estimation** or **out-of-sample testing**. To evaluate performance of our network, we'll train on folds 2 through 5, then test (or validate) with fold 1. Then we'll train with folds 1 and 3-5, and test (or validate) with fold 2. When we've done this for all 5 folds, we average the performance and that's our estimate of the system's accuracy.

Training too much
(or too little)

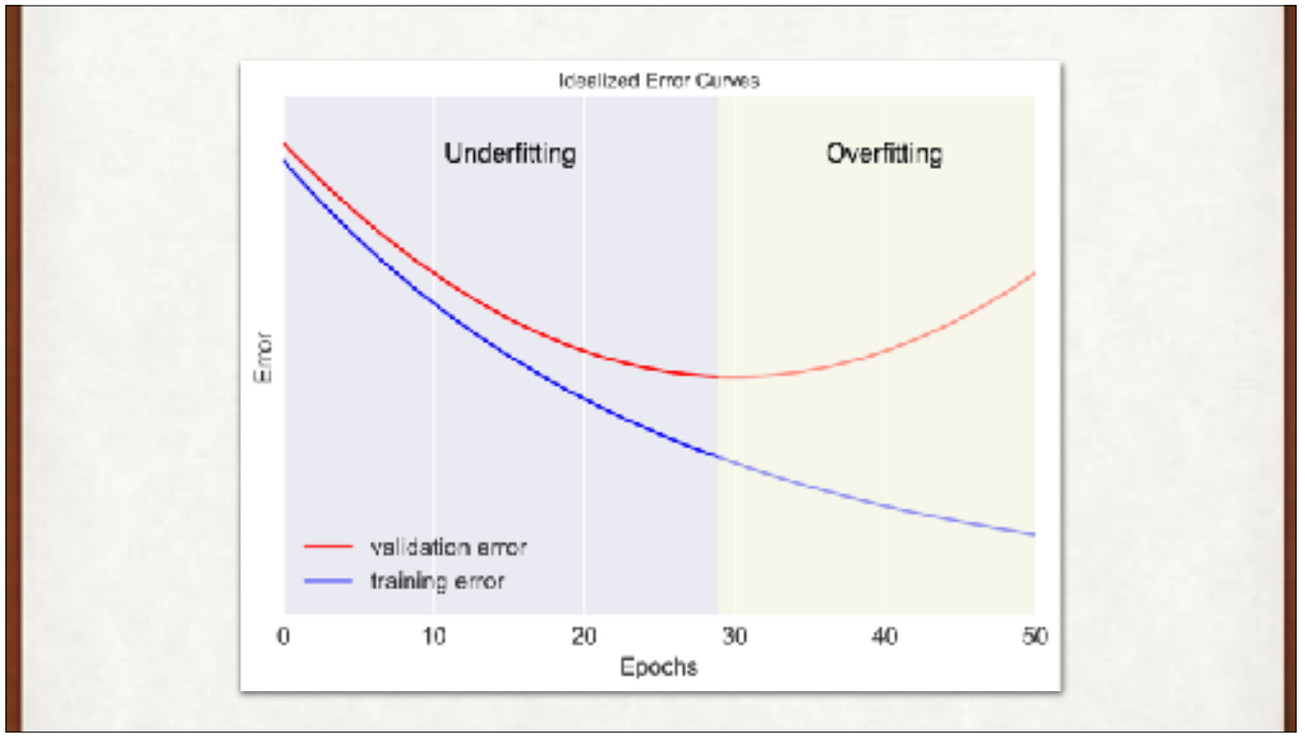
Training too little is clearly not ideal. But there’s no such thing as too much training, right? (Spoiler: wrong)



If we’re going to try a bunch of variations of our network, we can break out a separate **validation set** to let us evaluate each variation. Then we pick the one that performed best on the validation set, and evaluate its readiness for deployment with the test set.



Too little training is called **underfitting**. Too much training is **overfitting**. During overfitting, even though the training error is decreasing, the validation error is increasing. The validation error is an estimate of how well our system will perform in the real world. More training makes the system perform worse on new data, even while it's performing better on the training data. What the heck?



Too little training is called underfitting. Too much training is overfitting. During overfitting, even though the training error is decreasing, the validation error is increasing. The validation error is an estimate of how well our system will perform in the real world. More training makes the system perform worse on new data, even while it's performing better on the training data. What the heck?

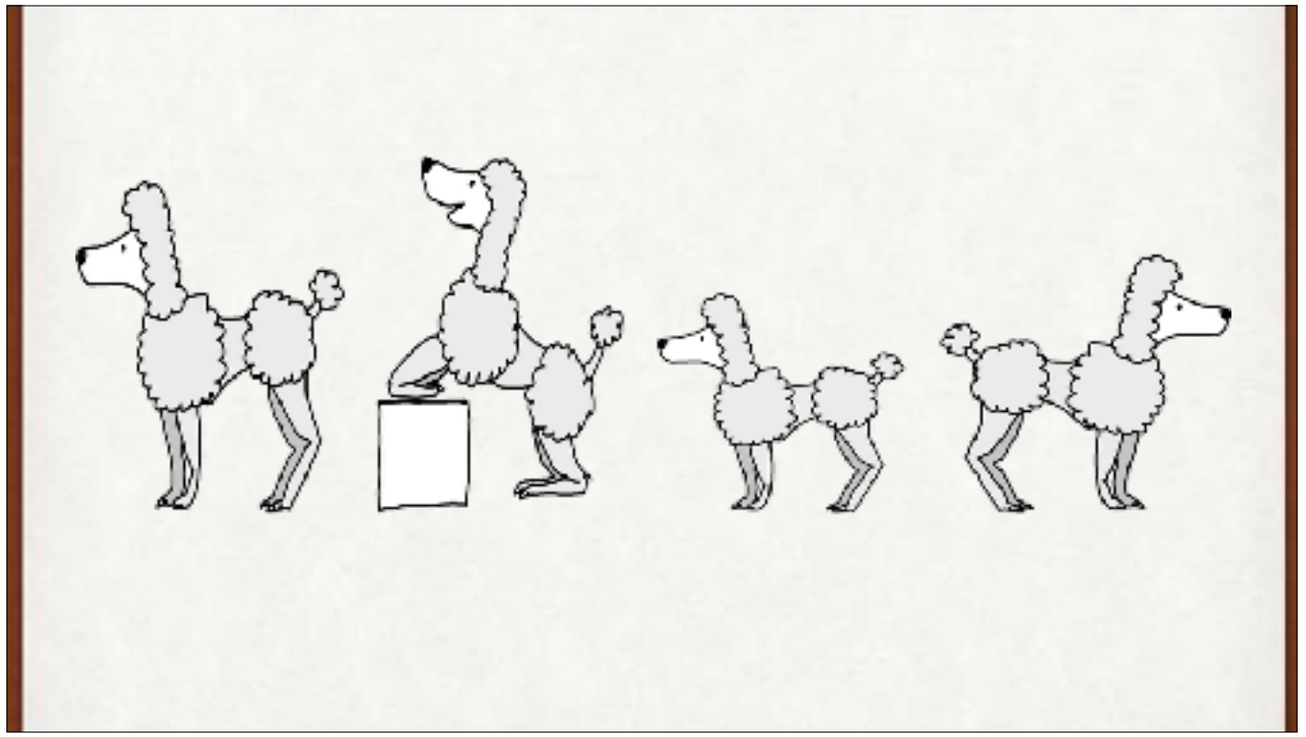
Standard Poodle

Cockapoo

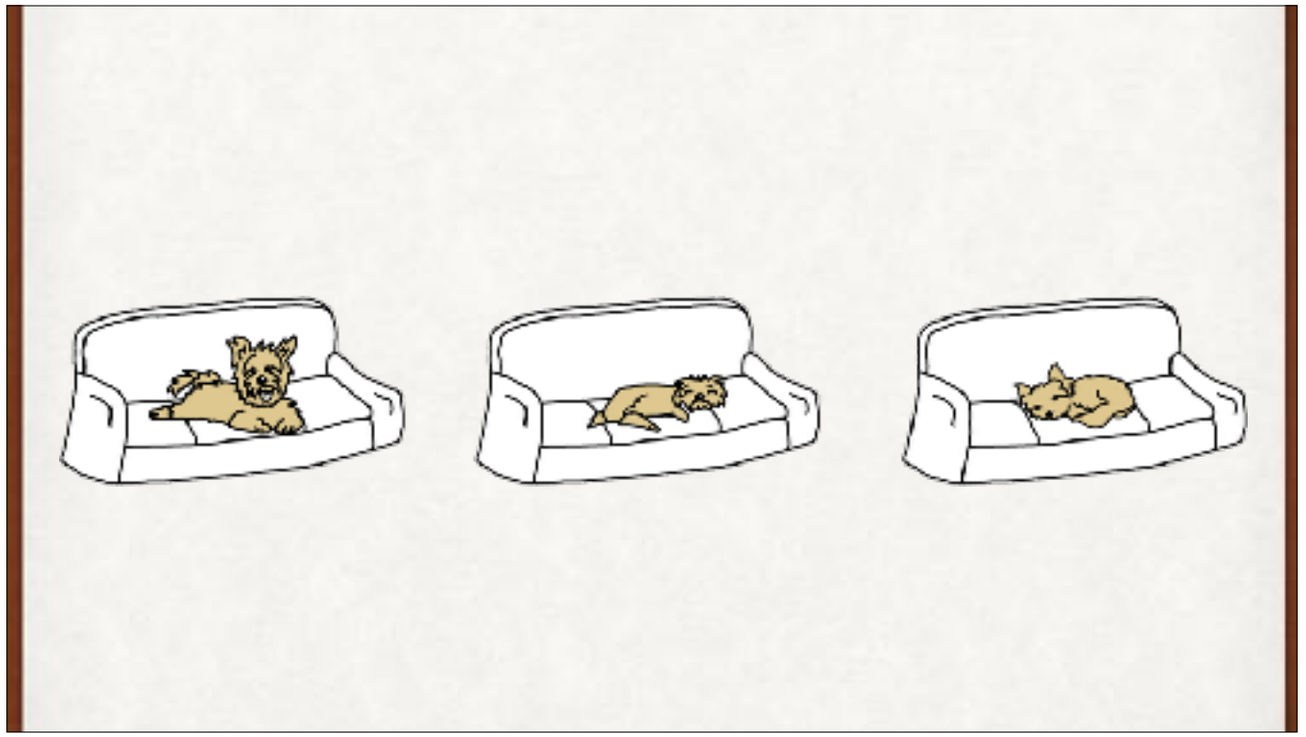
Labradoodle

Schnoodle

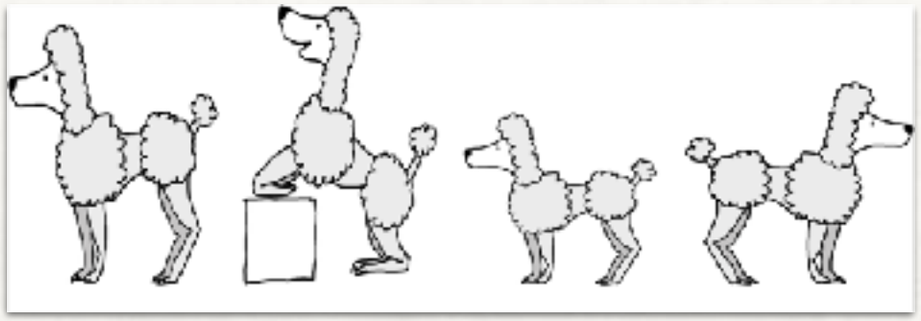
So many kinds of poodle! Let’s learn how to classify different breeds of dogs.



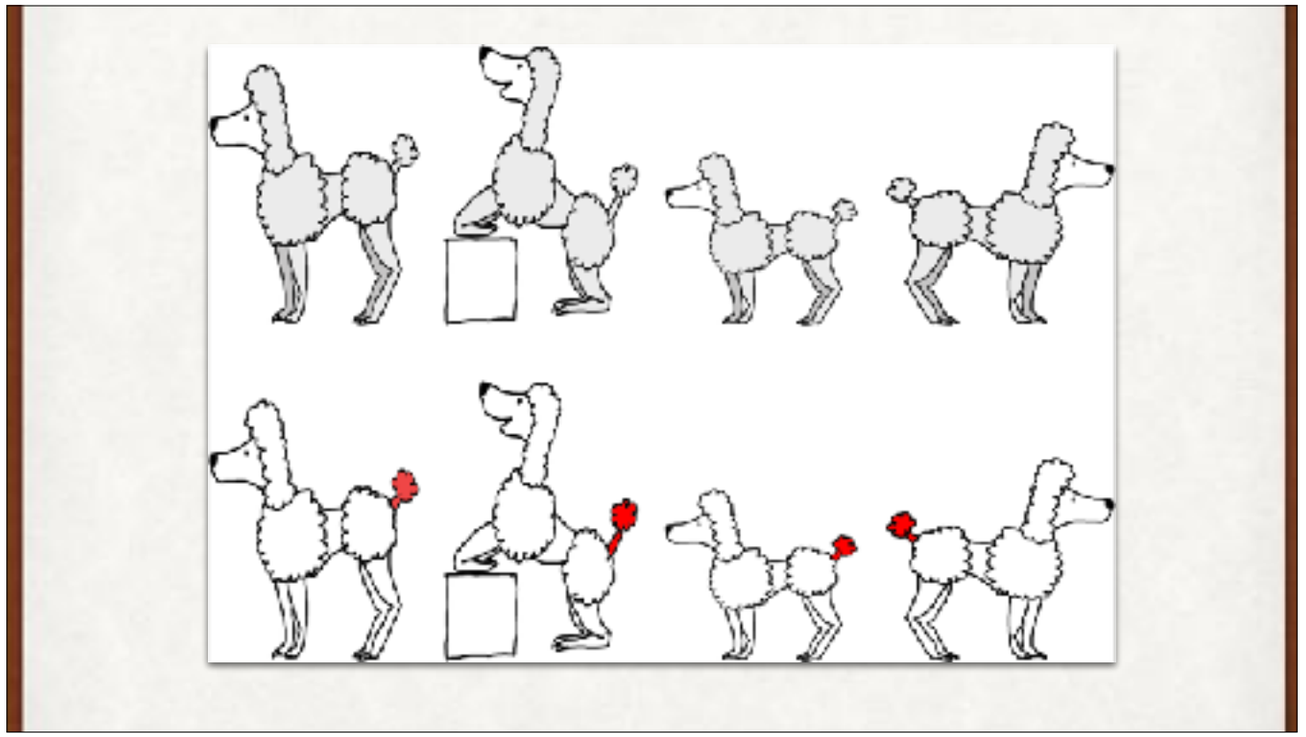
Some images of Poodles to train on.



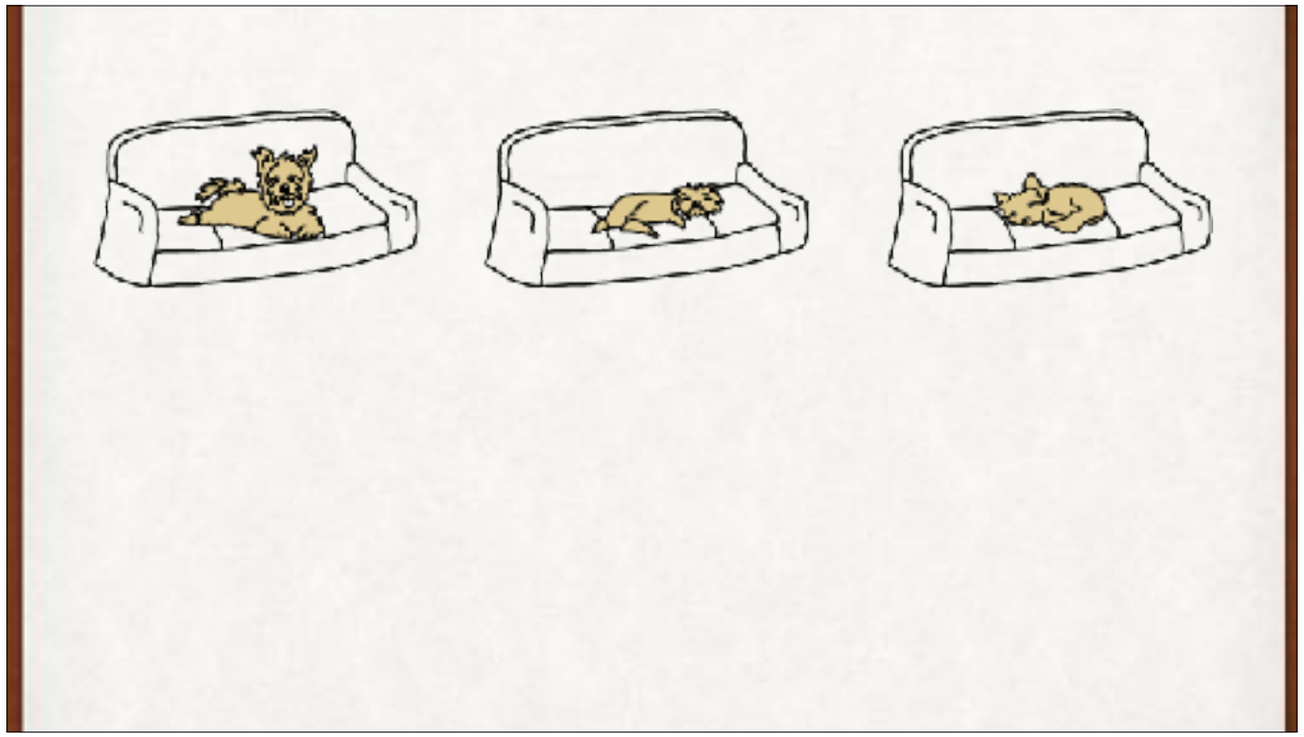
Yorkshire Terrier training data.



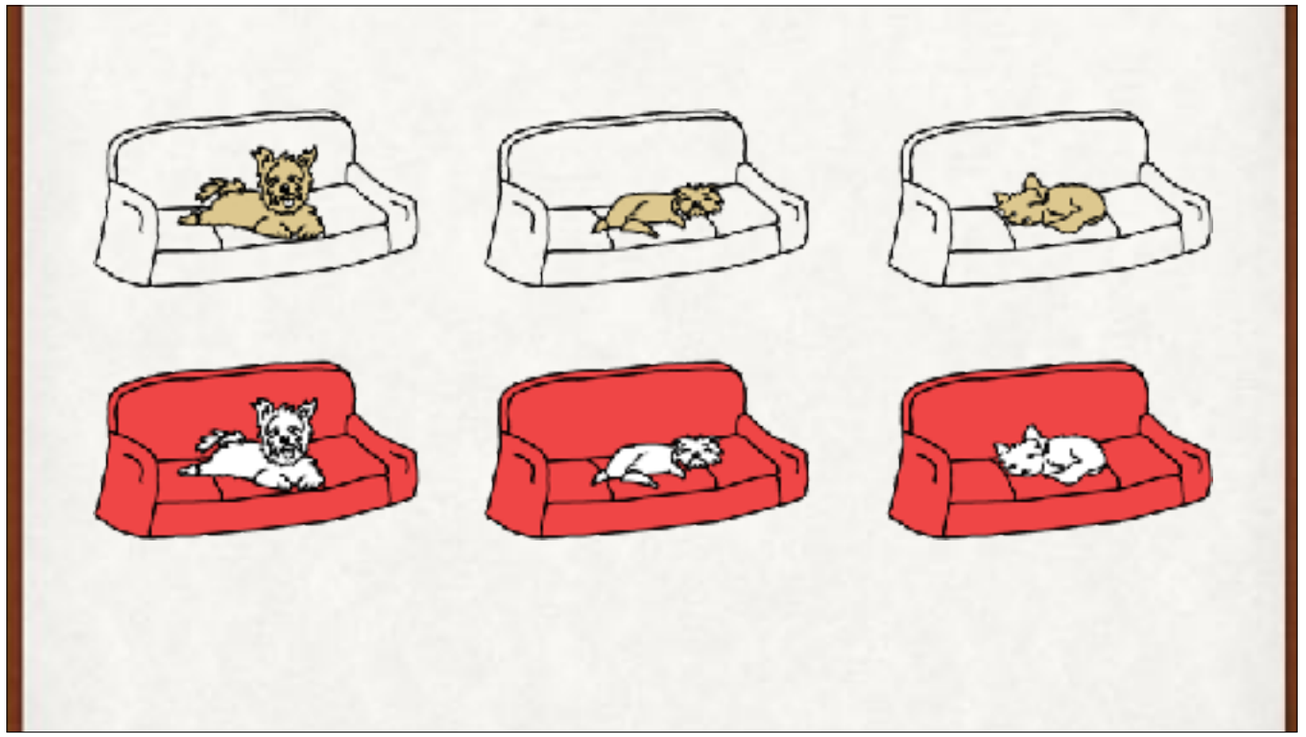
Unknown to us, the system recognizes the poodle just by the tail and fluffy ball at the end. No other dog images in our training set have this, so the system ignores everything else. This gives it great performance on the training data. But it’s just an idiosyncratic accident in the training data. We shouldn’t be generalizing from this detail.



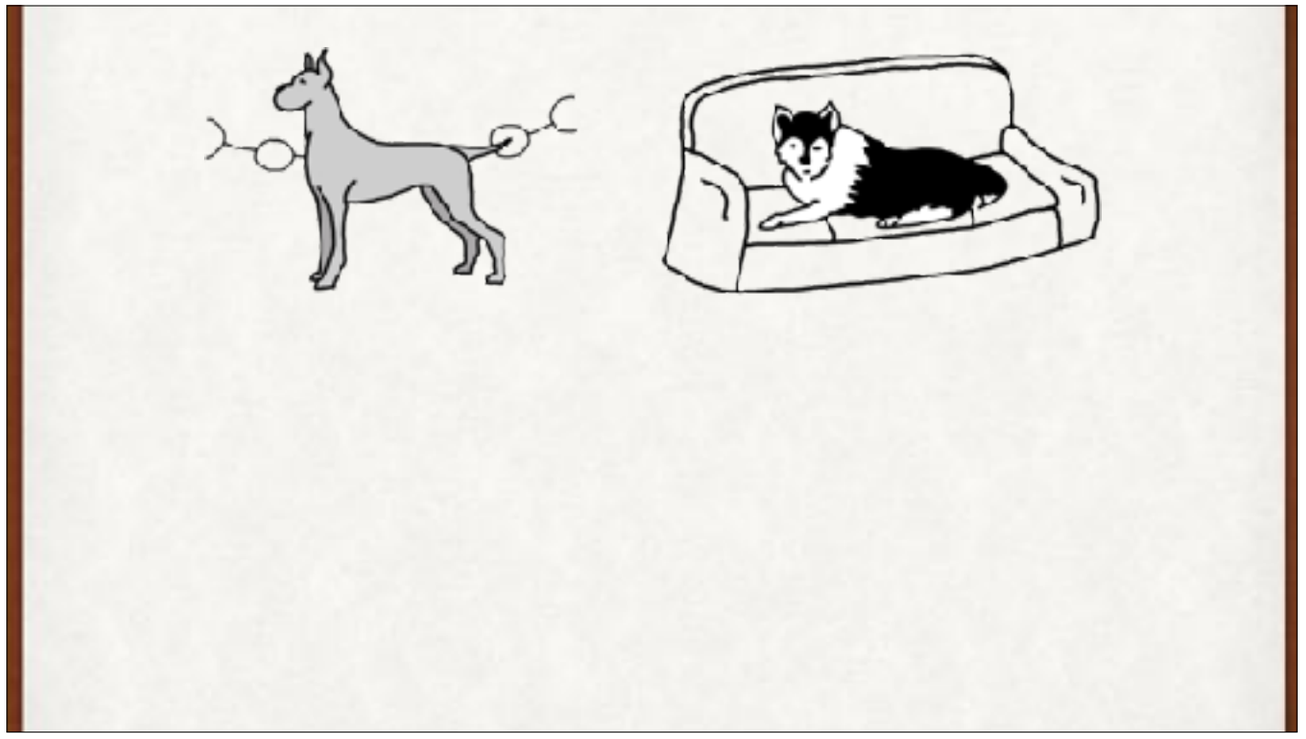
Unknown to us, the system recognizes the poodle just by the tail and fluffy ball at the end. No other dog images in our training set have this, so the system ignores everything else. This gives it great performance on the training data.



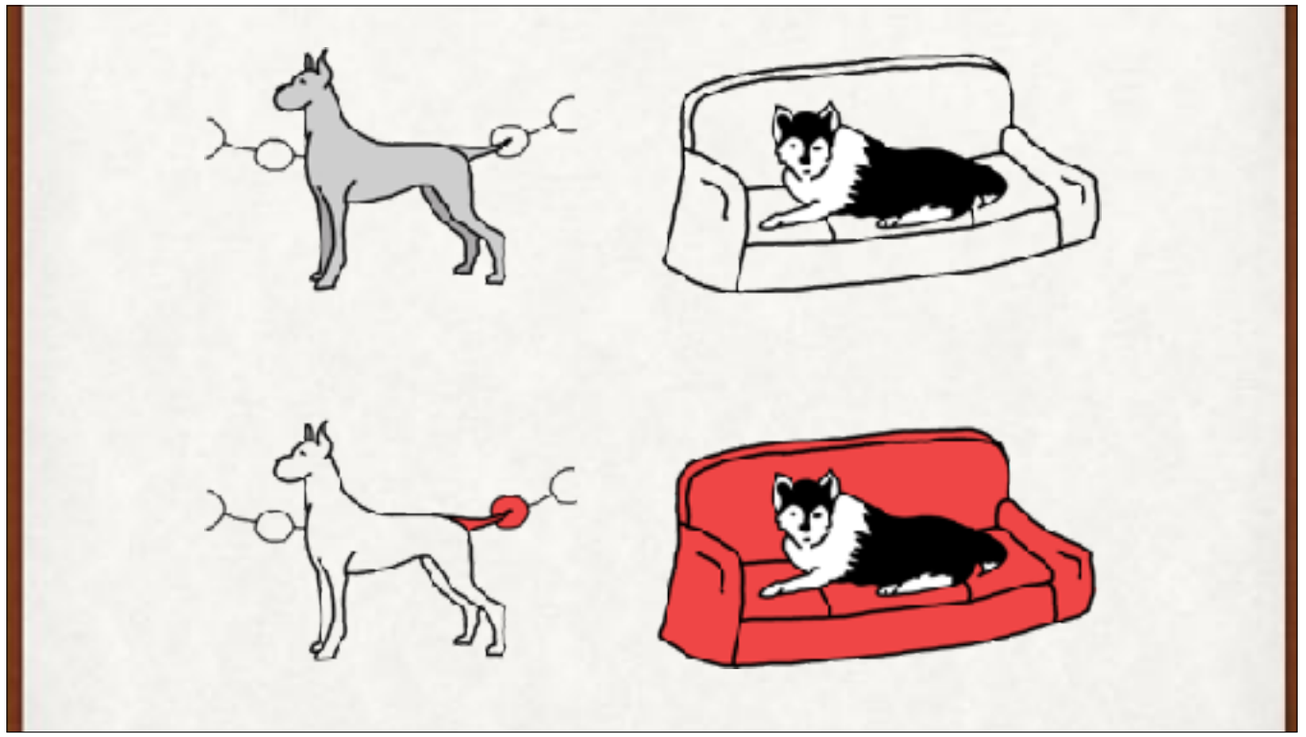
And the system isn’t recognizing Yorkshire Terriers. In our data, all Yorkies are on couches, and no other dogs are. So if the system sees a couch, it says the dog is a Yorkie. Again, that’s great for the training data, for which this just happens to be true.



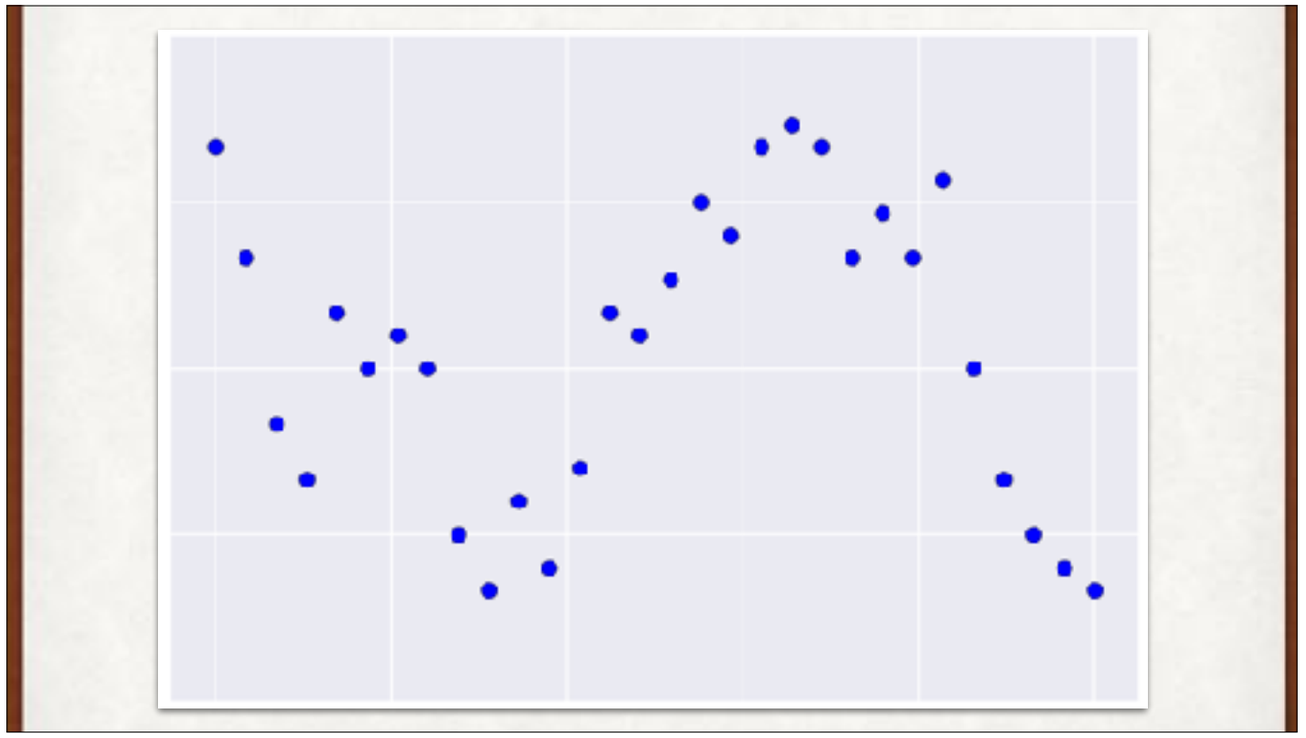
And the system isn’t recognizing Yorkshire Terriers. In our data, all Yorkies are on couches, and no other dogs are. So if the system sees a couch, it says the dog is a Yorkie. Again, that’s great for the training data, for which this just happens to be true.



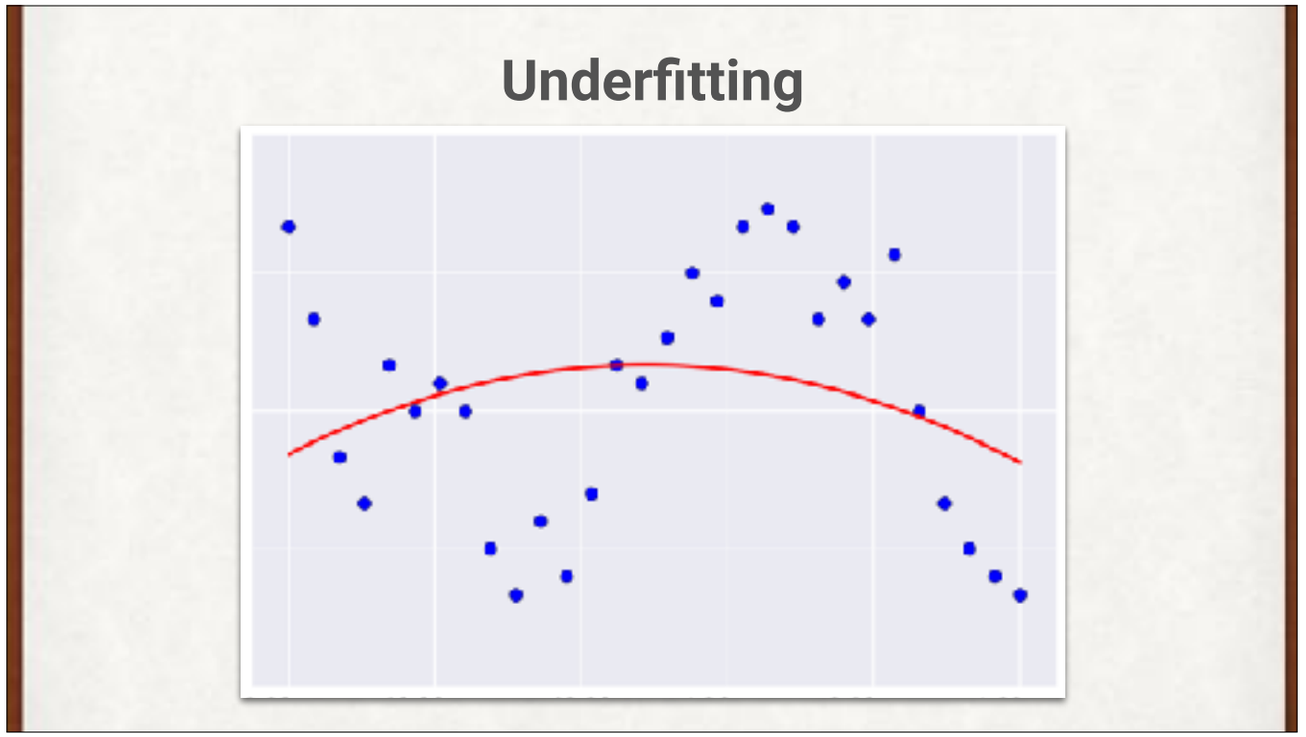
And now we get into trouble when we evaluate new images. Here, the Great Dane is in front of a holiday display with balls on a string. The system sees the tail and ball and says, “Poodle.” The Huskie is on a couch. The system sees the couch and says, “Yorkshire Terrier.” We have overtrained our system so that it’s relying on idiosyncratic details in the training data, rather than working out a general means for identifying dog breeds.



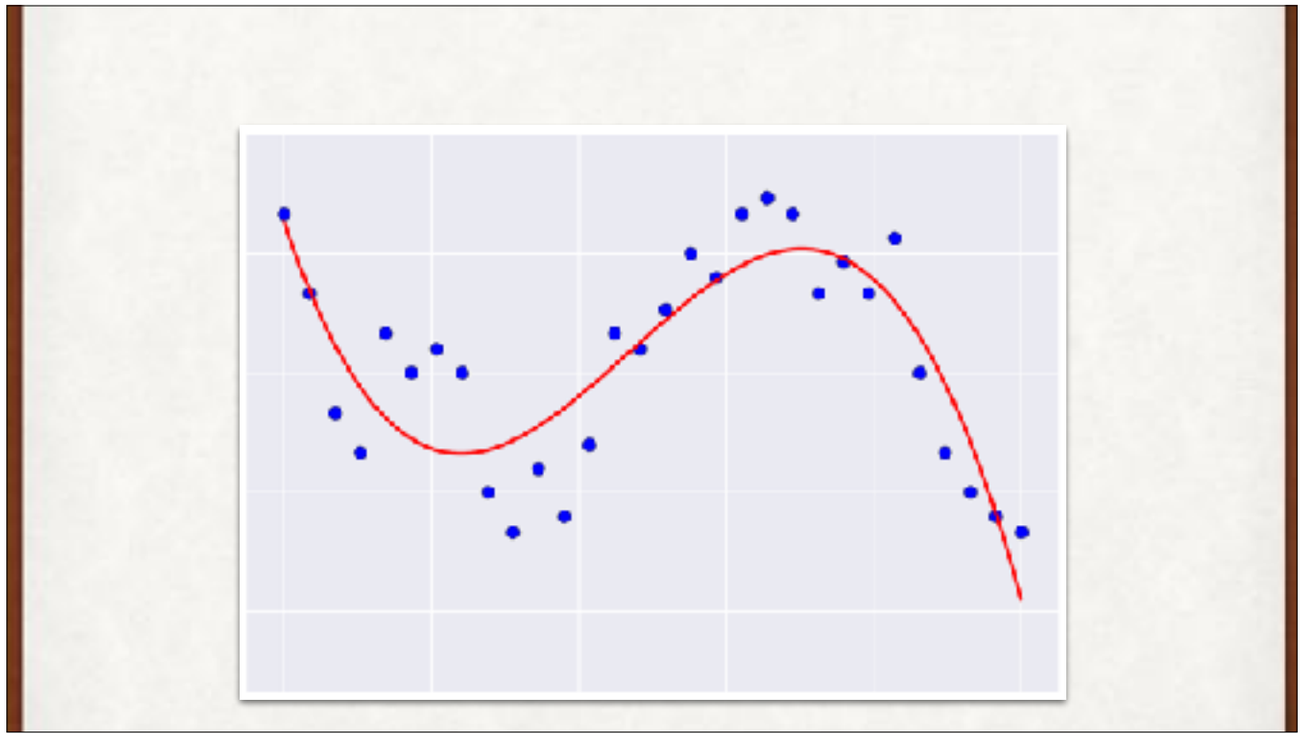
And now we get into trouble when we validate with new images. Here, the Great Dane is in front of a holiday display with balls on a string. The system sees the tail and ball and says, “Poodle.” The Huskie is on a couch. The system sees the couch and says, “Yorkshire Terrier.” We have overtrained our system so that it’s relying on idiosyncratic details in the training data, rather than working out a general means for identifying dog breeds.



A storeowner wants us to control the tempo for her background music. Here are the settings she chose on one particular day.

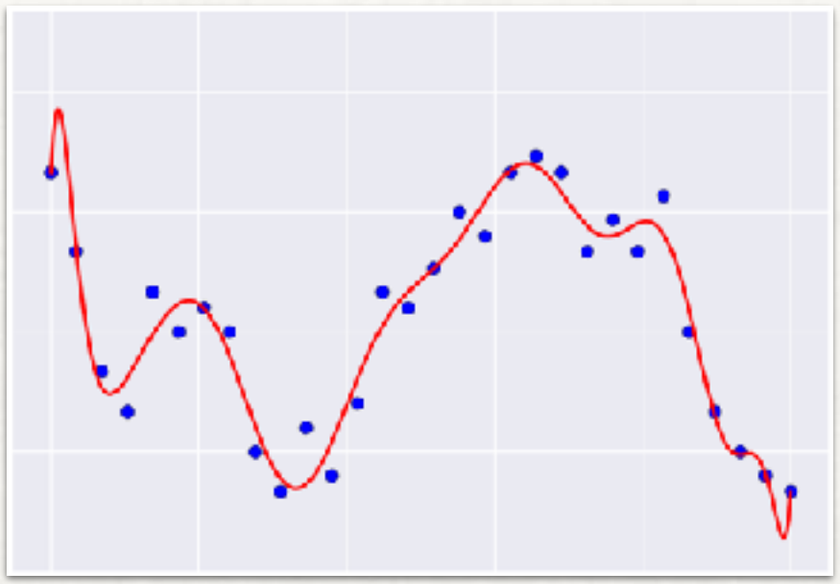


We come back the next day and automate the tempo settings. Compared to yesterday, the red curve is too simple a shape to match the data. We’re underfitting.

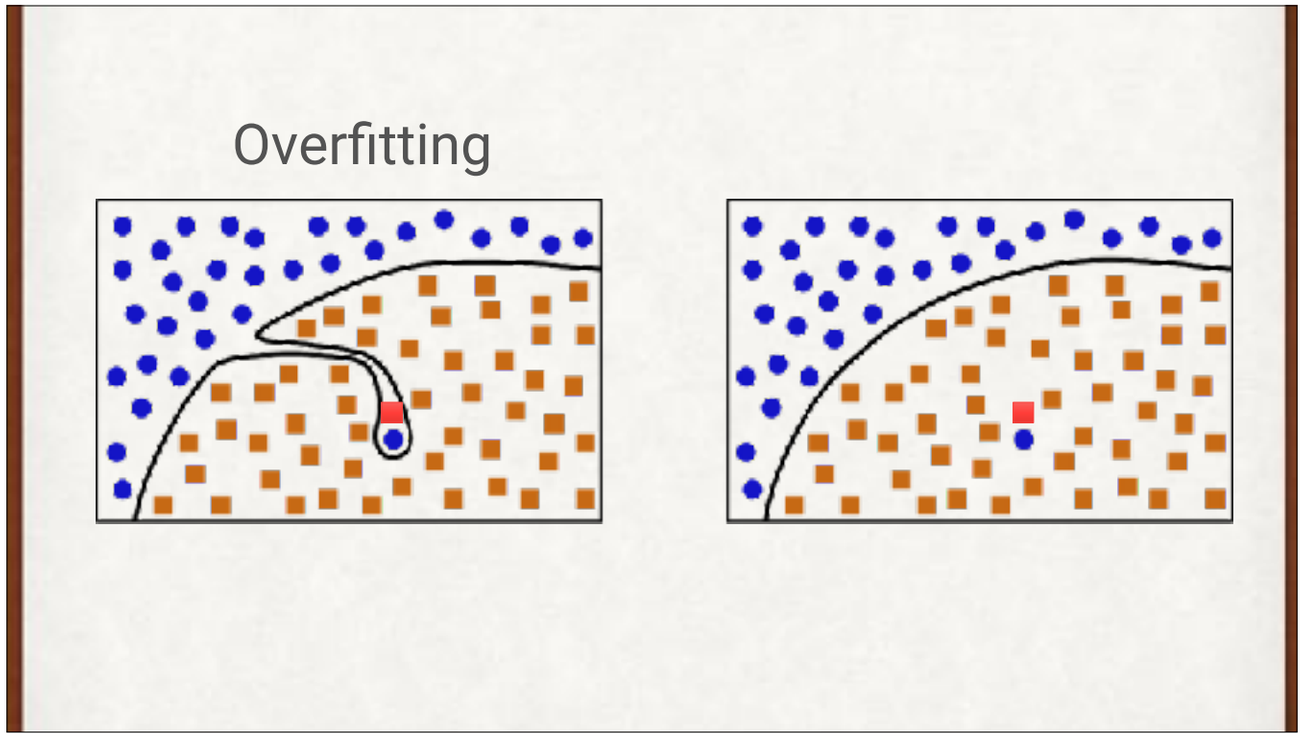


A better fit to the data.

Overfitting



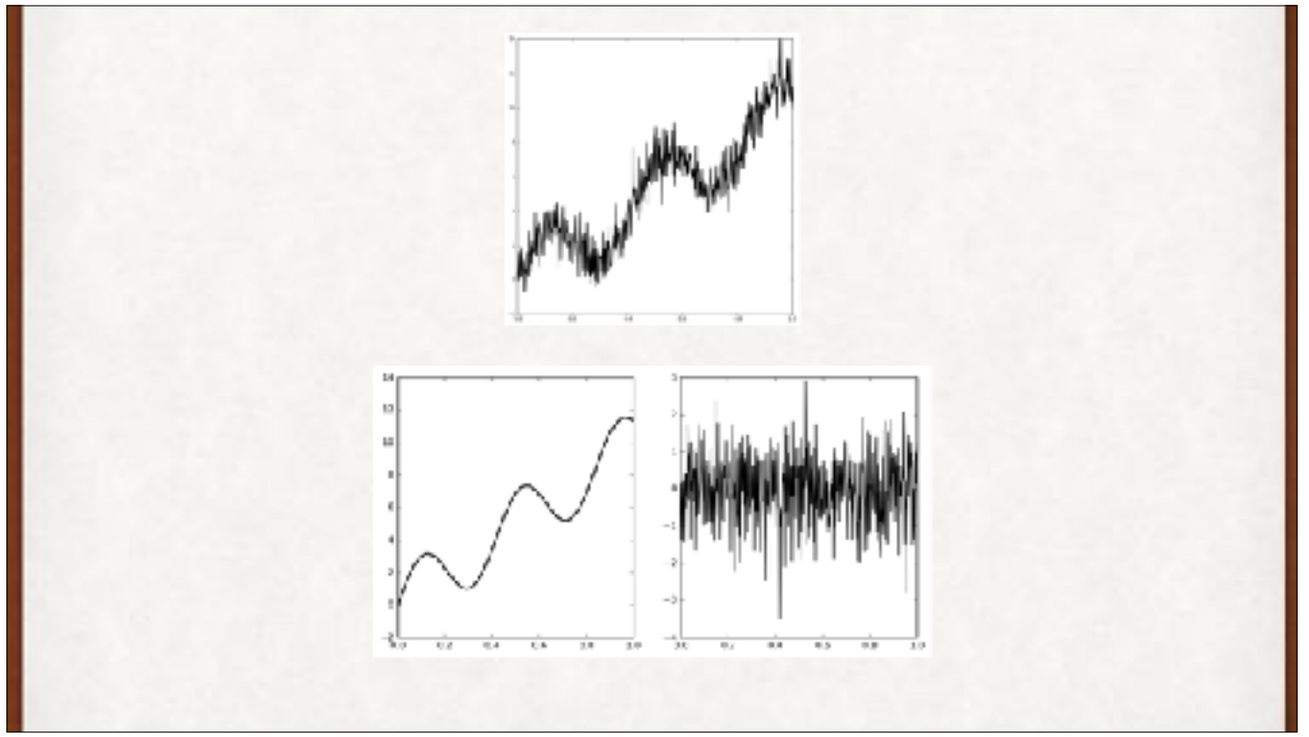
We’ve trained too much. Now we’re paying attention to irrelevant little bumps and wiggles in the data. We’re overfitting.



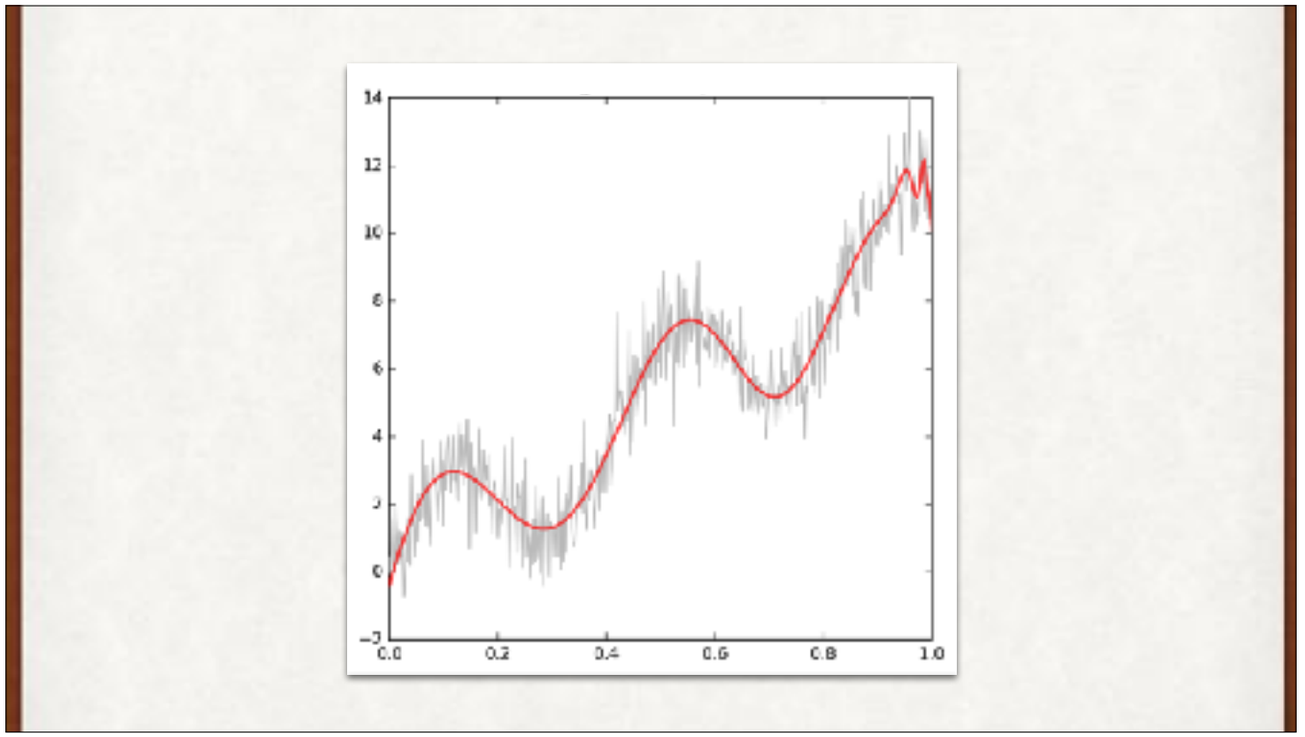
On the left, we’re probably overfitting, by paying too much attention to one weird piece of data. That’s likely to be an oddball event. On the right is (probably) a better curve. In this simple case we’re making a subjective decision that the one lone blue circle is an anomaly, but we can automate this for the general case.

Bias and Variance

When we fit data using families of curves, these ideas are important.

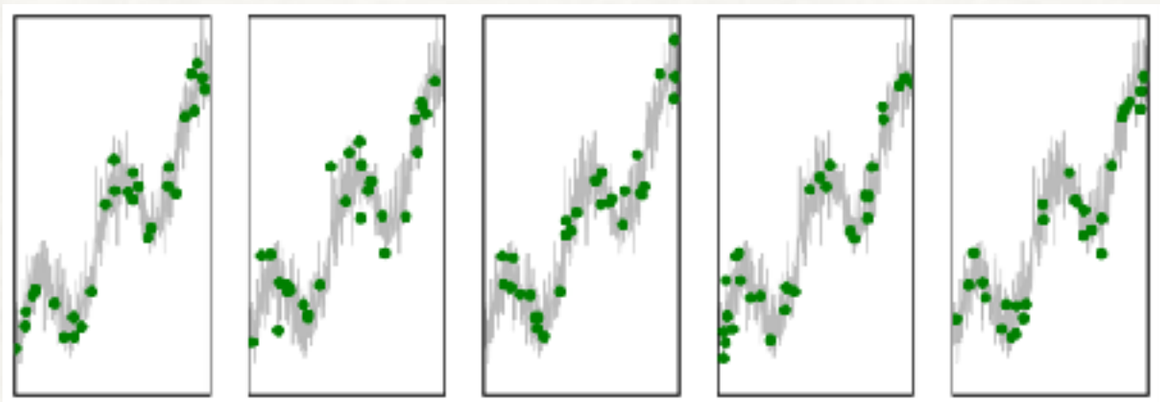


A signal (top) composed of “real data” (lower left) and noise (lower right). We want to find a curve to match the real data, not the noise.



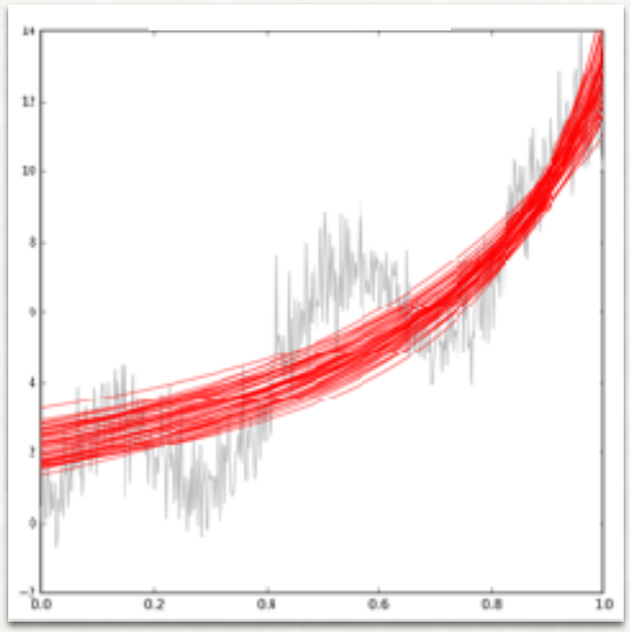
A good solution fits a relatively simple curve to the data (in this example it gets wiggly at the far right). Because the curve is simple, it can’t wiggle enough to match the noise. So this curve is probably in the sweet spot, but how can we do this automatically?

Bootstrapping



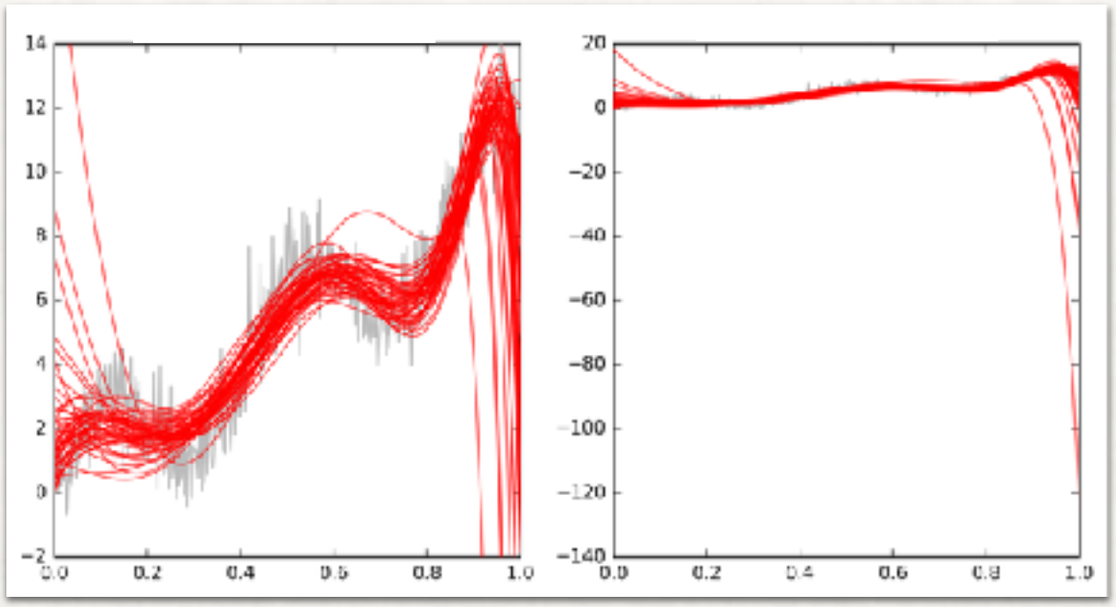
Let’s try matching our data by drawing lots of smaller, random subsets, and matching those. This is called bootstrapping.

Bias

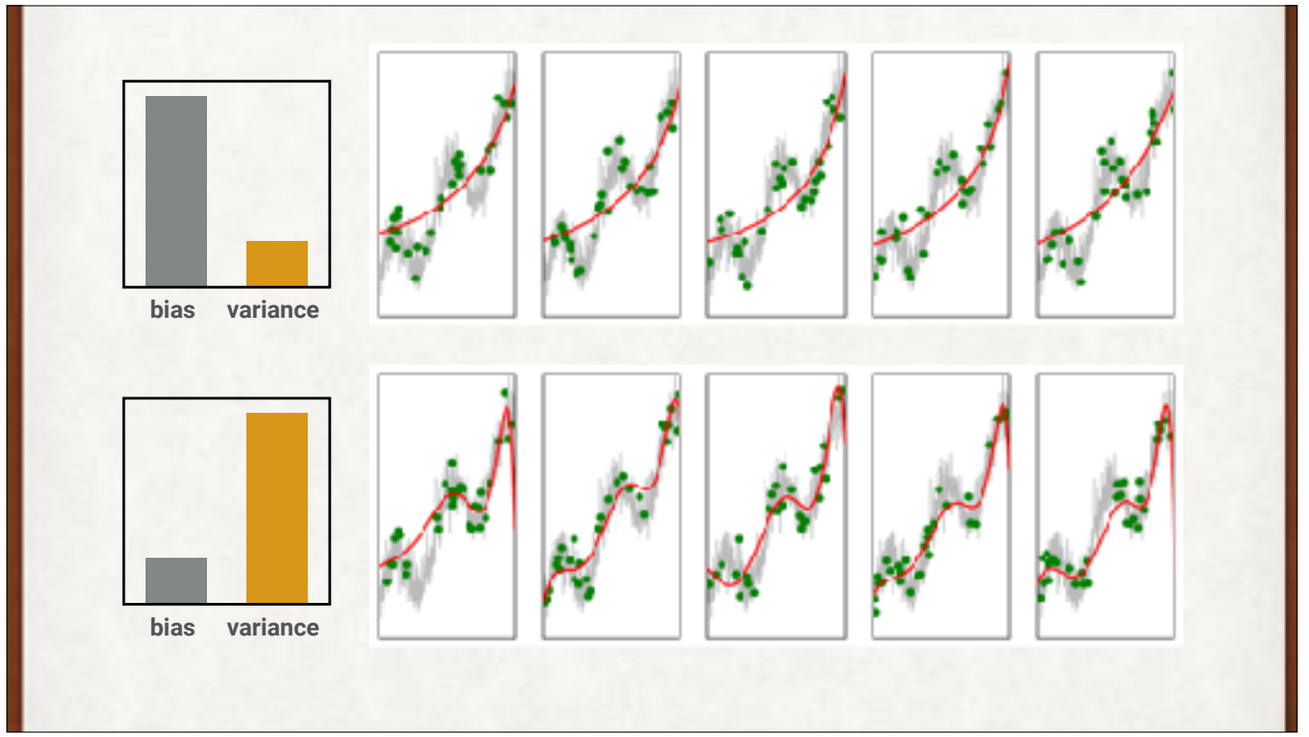


We’ll fit a simple curve to each set of bootstrapped points. Here are lots of them at once, over the original data. The curves are too simple, so we’re making a consistent error of missing the peaks and valleys. This error is inherent in our choice of curve. It’s called the bias of the family of curves.

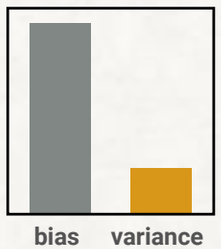
Variance



Let’s use a more wiggly family of curves. Now they come closer to the data, but they vary a lot from one curve to the next (full vertical scale on the right). We say this family of curves has high variance.

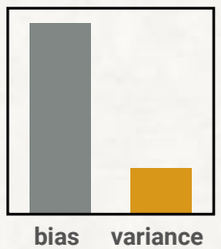


Bias and variance are tradeoffs. If we make one smaller, the other gets bigger. In other words, the curve can only have a certain amount of wiggliness. If that's low, each curve is always about the same, and we have high bias and low variance). If the wiggliness is high, each curve is different for each set of samples, and we have low bias and high variance. There's no way to make them both low at the same time.

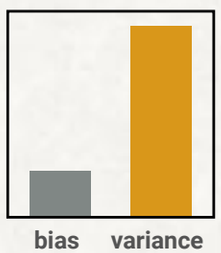


Very noisy training set

In noisy data, we might prefer high bias.

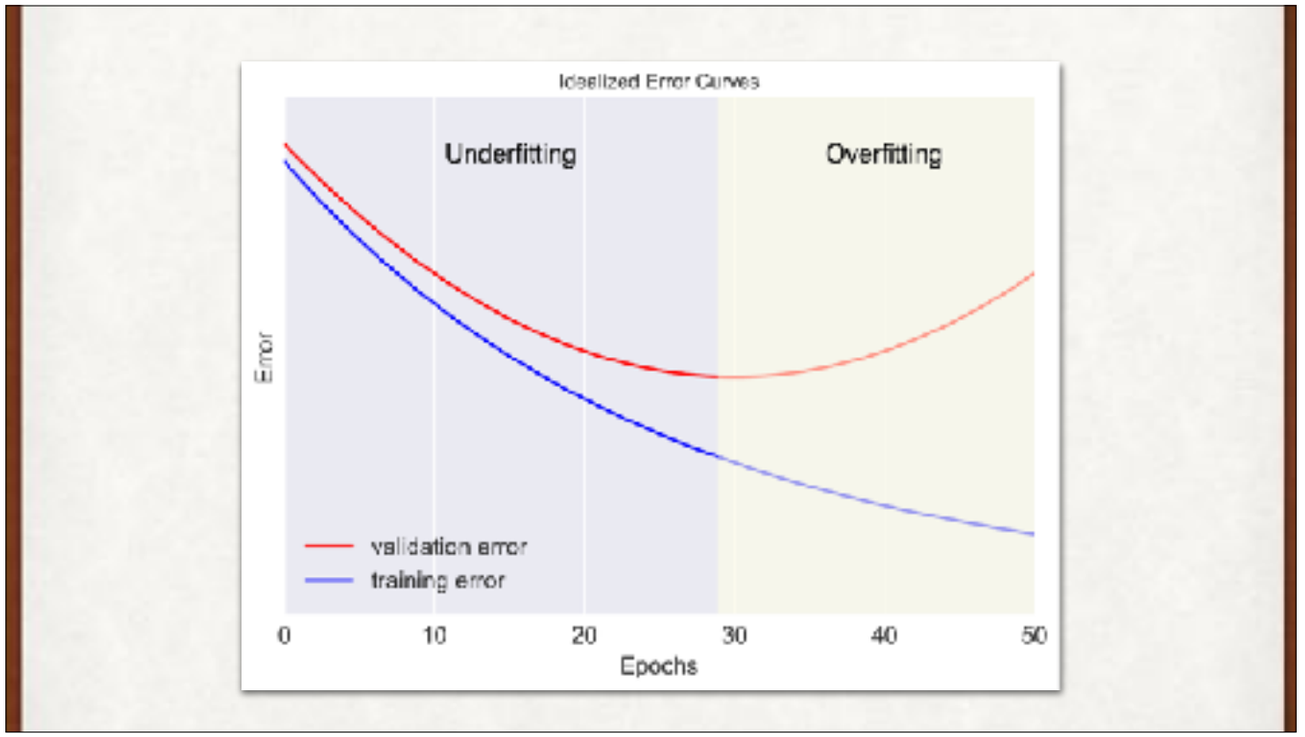


Very noisy training set



Perfect training set

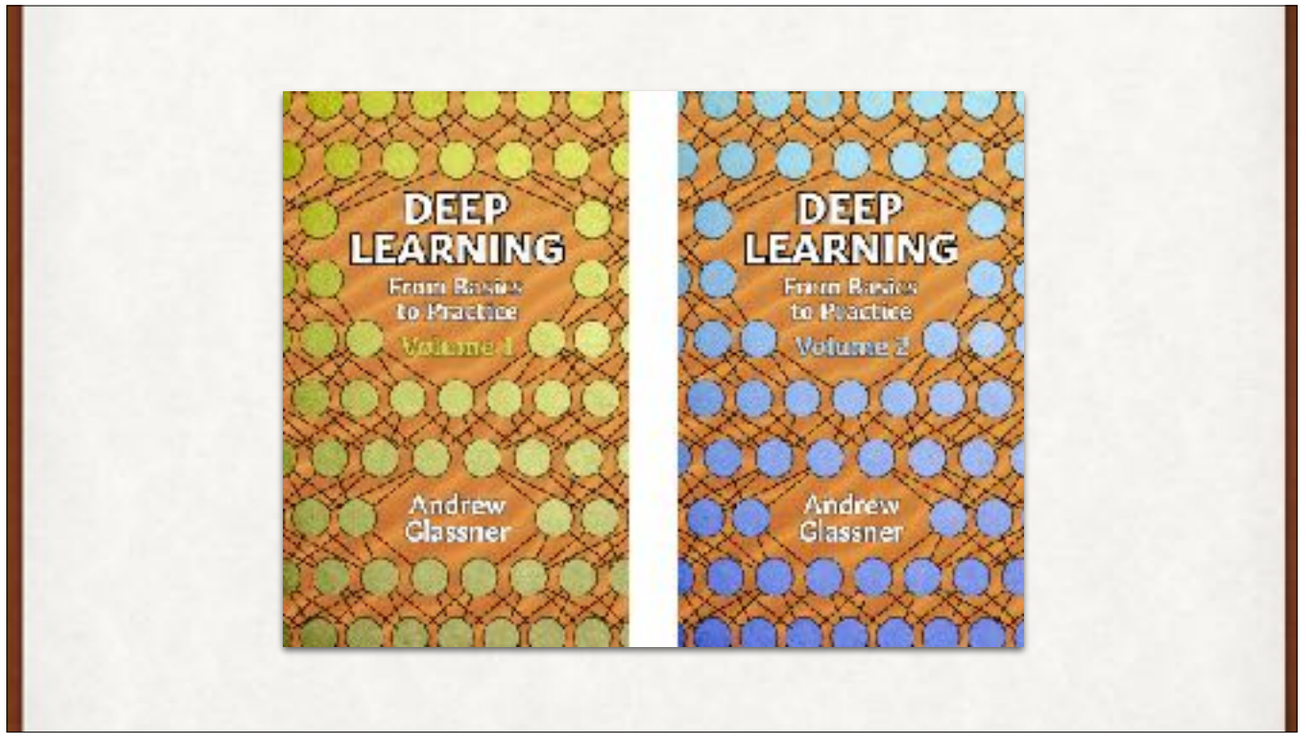
Training on perfect data, we’d want low bias, to better match the trustworthy data.



To prevent overfitting, watch the error on the validation data. The training error is still declining, because the system is getting better (too much better!) at finding details in the training data. The validation (or test) error is going up, because those details aren’t present in the validation data. In this example, stopping around epoch 29 seems to be the sweet spot.

Palette cleanser. New topic.

Here’s where we took a 15-minute break. The slides pick up again in Part 2.



This course has been adapted from my new books! There is a TON of more information in there, in a friendly style, and without math. The books are online-only and available on Amazon. They're in Kindle format, but there is a free Kindle reader for almost any device with a screen - just Google for your device and the word Kindle, and it should give you a link to a free reader. The books are at http://amzn.to/2F4nz7k and http://amzn.to/2EQtPR2

github.com/blueberrymusic

Every figure

Every Python notebook

My Github repo has every figure in the book (and thus almost every slide in this deck) available in high-res format for free, for you to use in classes, talks, or any other way you like. There are also dozens of Python notebooks to generate other figures, and to show how to implement learners of all the varieties we've discussed, and many more.

Thank you! My contact info. You can see the talk itself on YouTube at https://www.youtube.com/watch?v=r0Ogt-q956I . I'm happy to give this or related courses, seminars, and workshops based on this material. Drop me a note via email or LinkedIn! I'll post updates and related news on Twitter, along with all the other usual Twitter stuff.