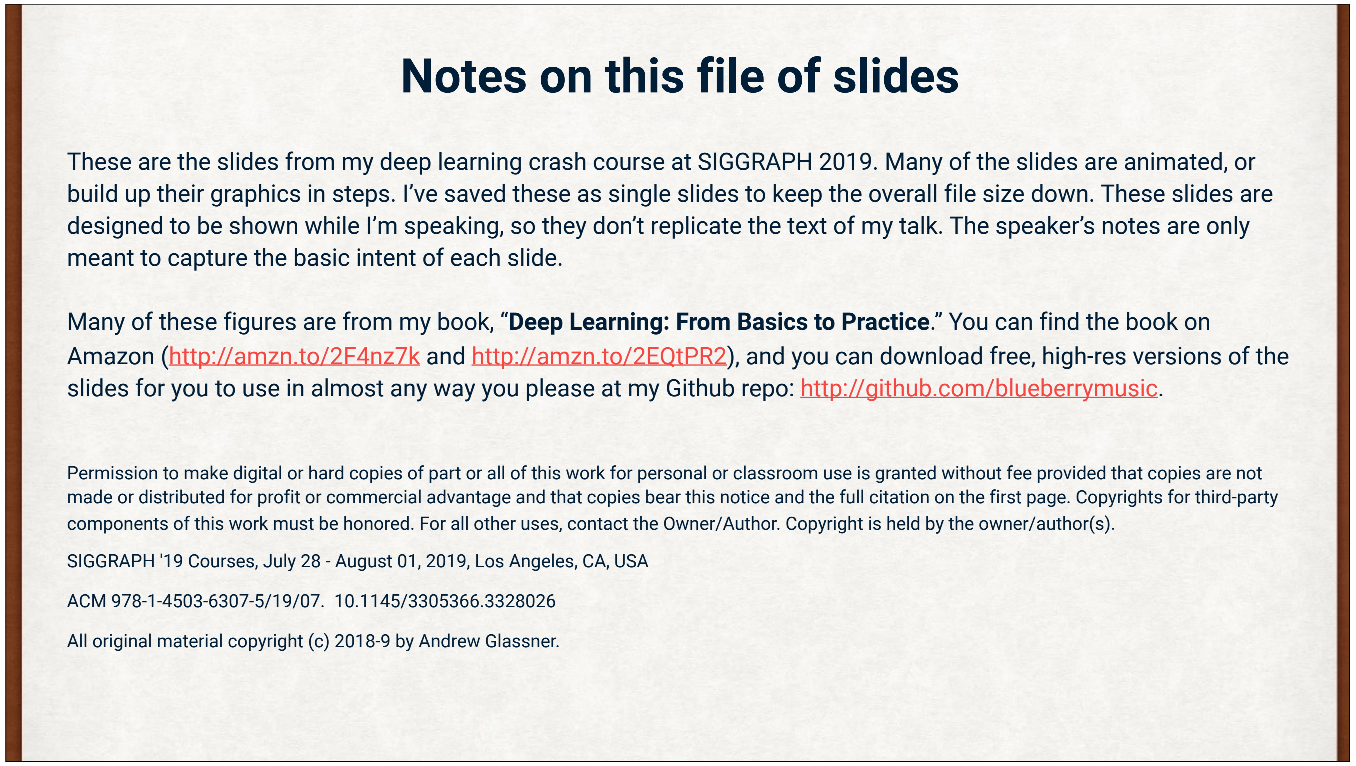
DEEP LEARNING: A CRASH COURSE

thrive
SIGGRAPH2019
LOS ANGELES • 28 JULY - 1 AUGUST

Andrew Glassner
The Imaginary Institute

Concepts, terminology, structures, no math, no code. Free open-source libraries do the hard work. My background: consultant, writer, director, etc.



Notes on this file of slides

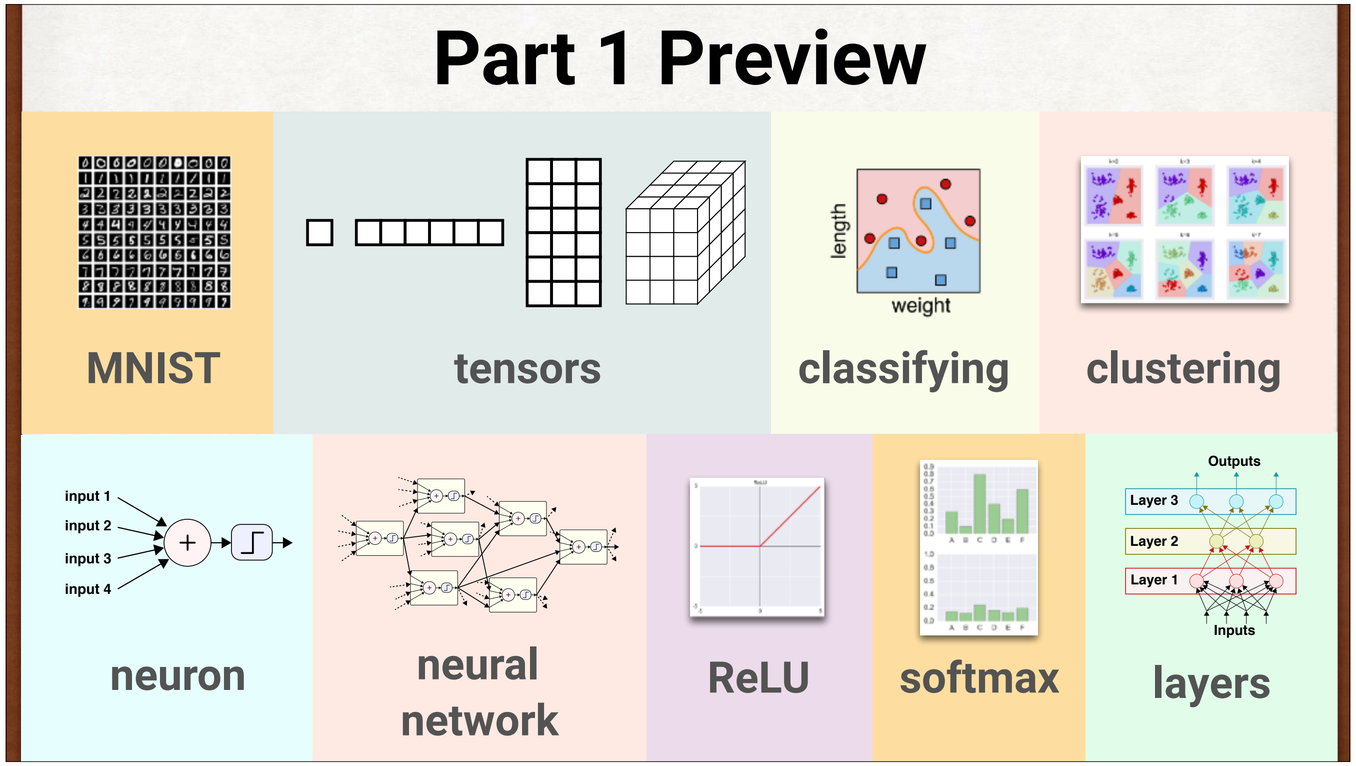
These are the slides from my deep learning crash course at SIGGRAPH 2019. Many of the slides are animated, or build up their graphics in steps. I've saved these as single slides to keep the overall file size down. These slides are designed to be shown while I'm speaking, so they don't replicate the text of my talk. The speaker's notes are only meant to capture the basic intent of each slide.

Many of these figures are from my book, "**Deep Learning: From Basics to Practice**." You can find the book on Amazon (http://amzn.to/2F4nz7k and http://amzn.to/2EQtPR2), and you can download free, high-res versions of the slides for you to use in almost any way you please at my Github repo: http://github.com/blueberrymusic.

SIGGRAPH '19 Courses, July 28 - August 01, 2019, Los Angeles, CA, USA

ACM 978-1-4503-6307-5/19/07. 10.1145/3305366.3328026

Notes for readers.



Part 1 Preview

MNIST

tensors

classifying

clustering

neuron

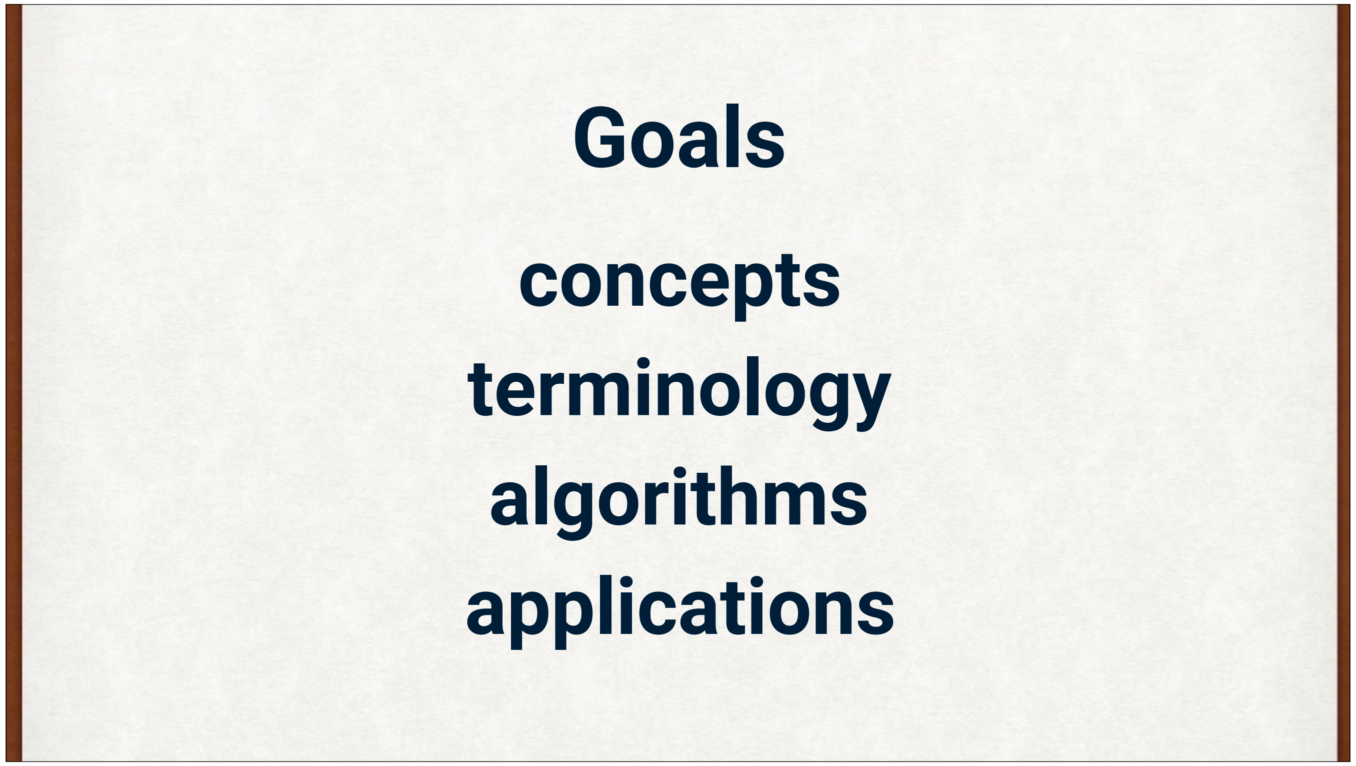
neural network

ReLU

softmax

layers

The topics for Part 1 of 4.



Goals

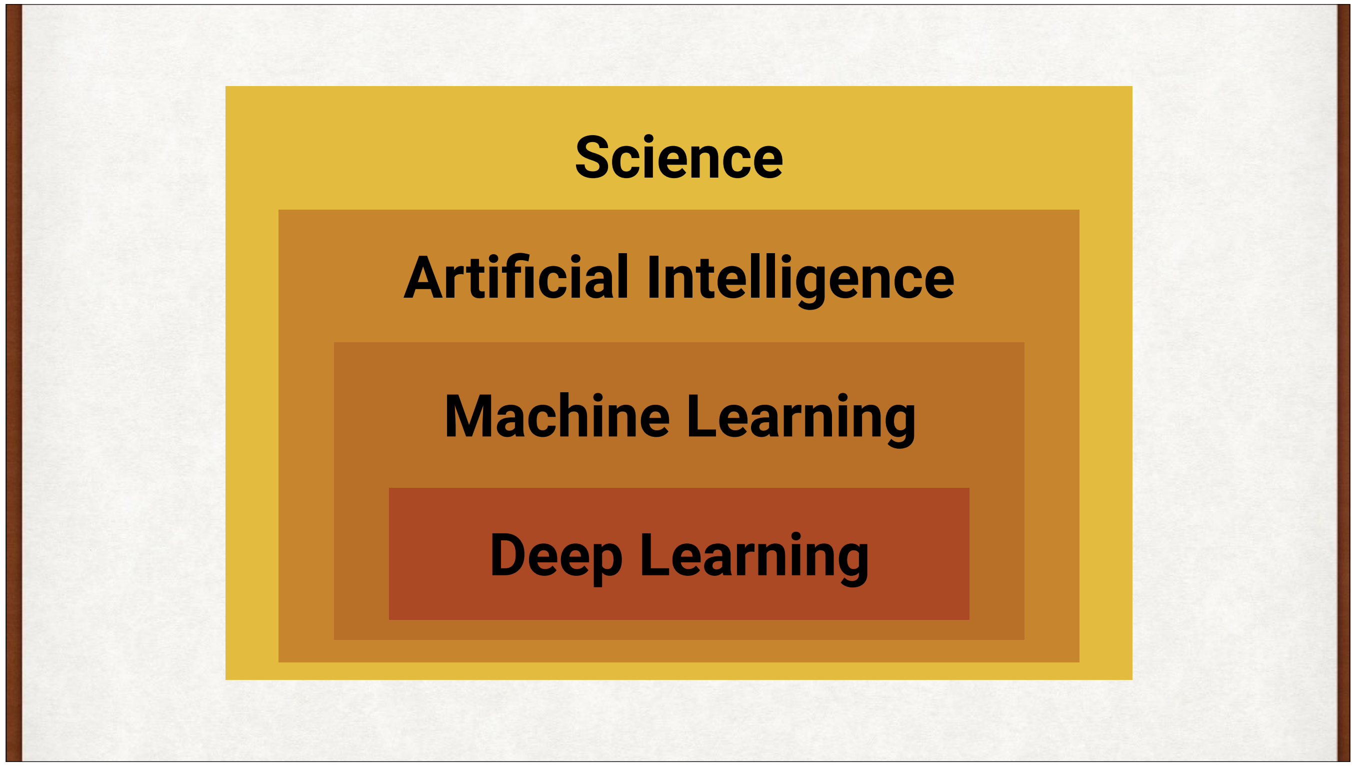
concepts

terminology

algorithms

applications

Our goals for the course.



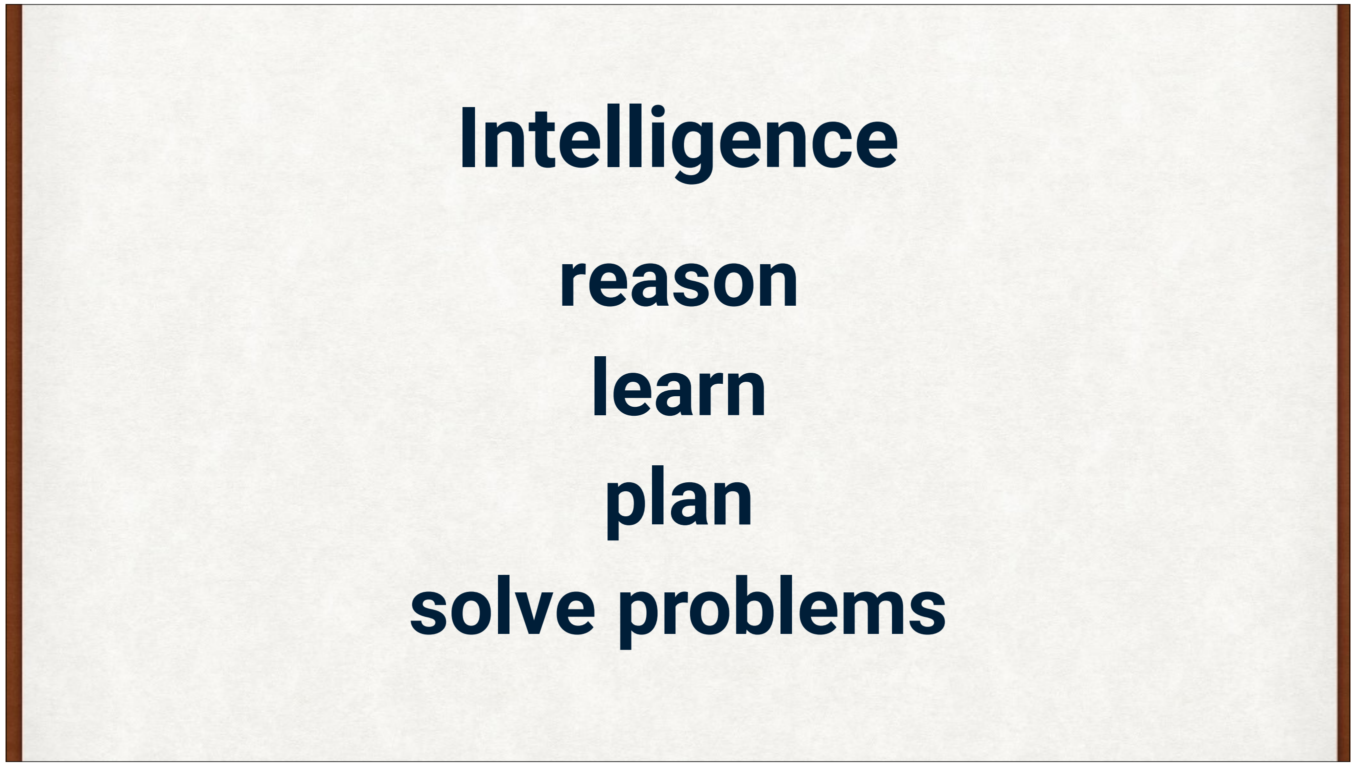
Science

Artificial Intelligence

Machine Learning

Deep Learning

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. Do ML & DL have “intelligence” in them?



Intelligence

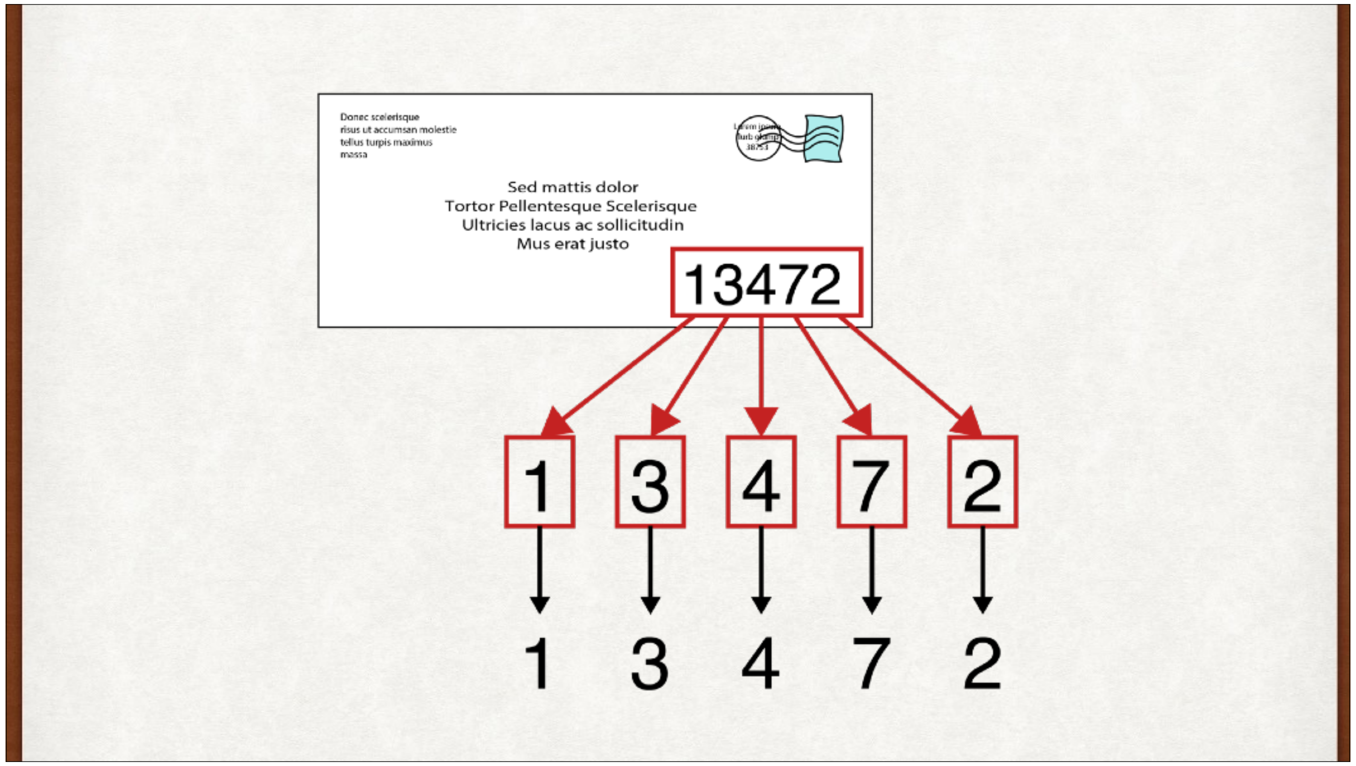
reason

learn

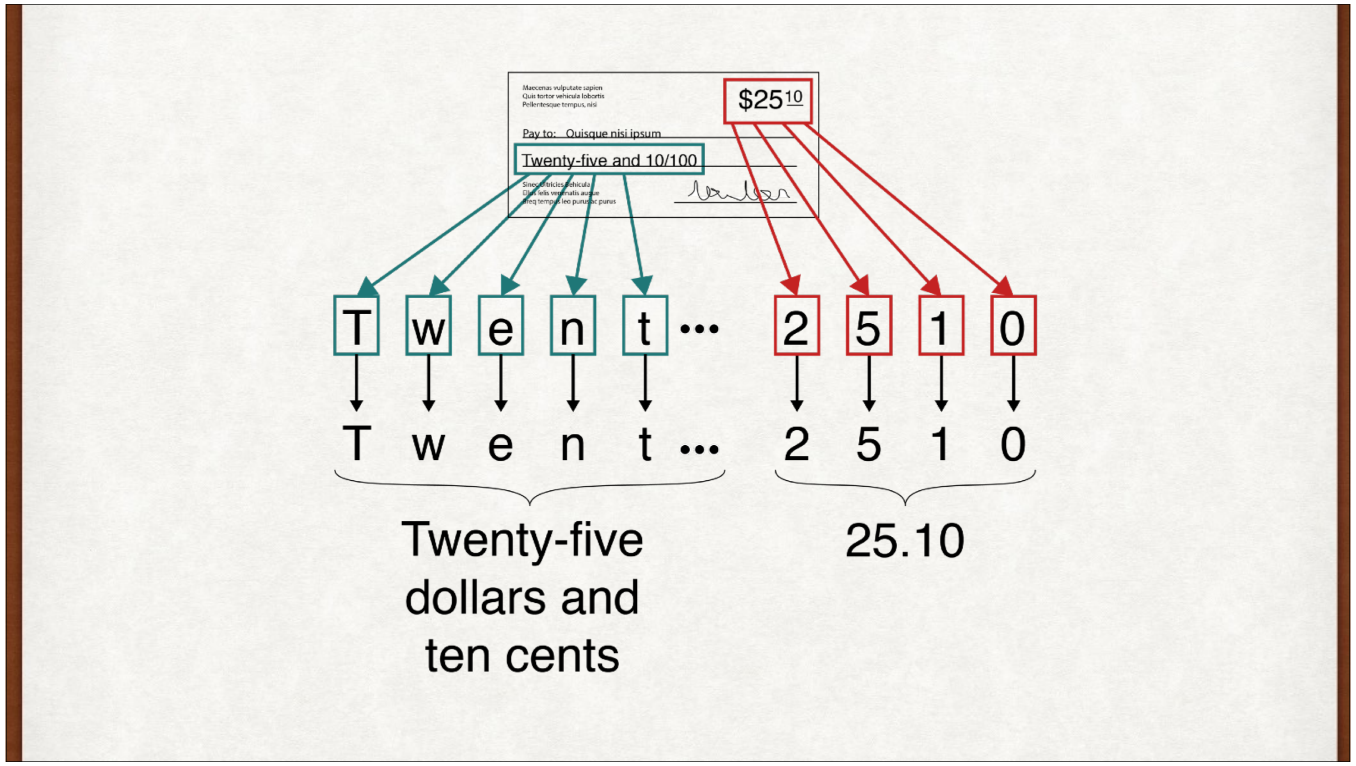
plan

solve problems

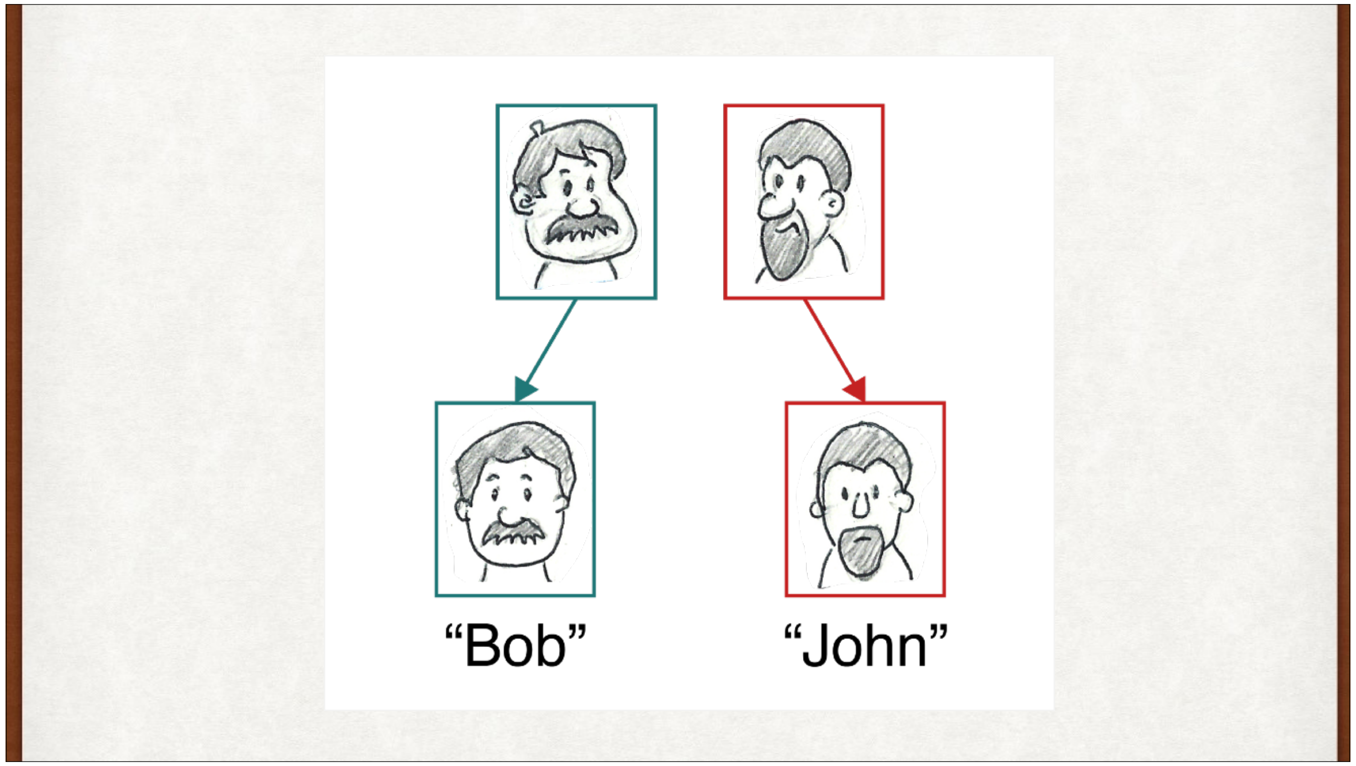
Roughly, some ideas that are characteristics of intelligence. Nope, none of this in ML & DL (well, some learning). They’re like the spherical horse. These algorithms don’t have semantic/structural/domain knowledge the way we usually those terms.



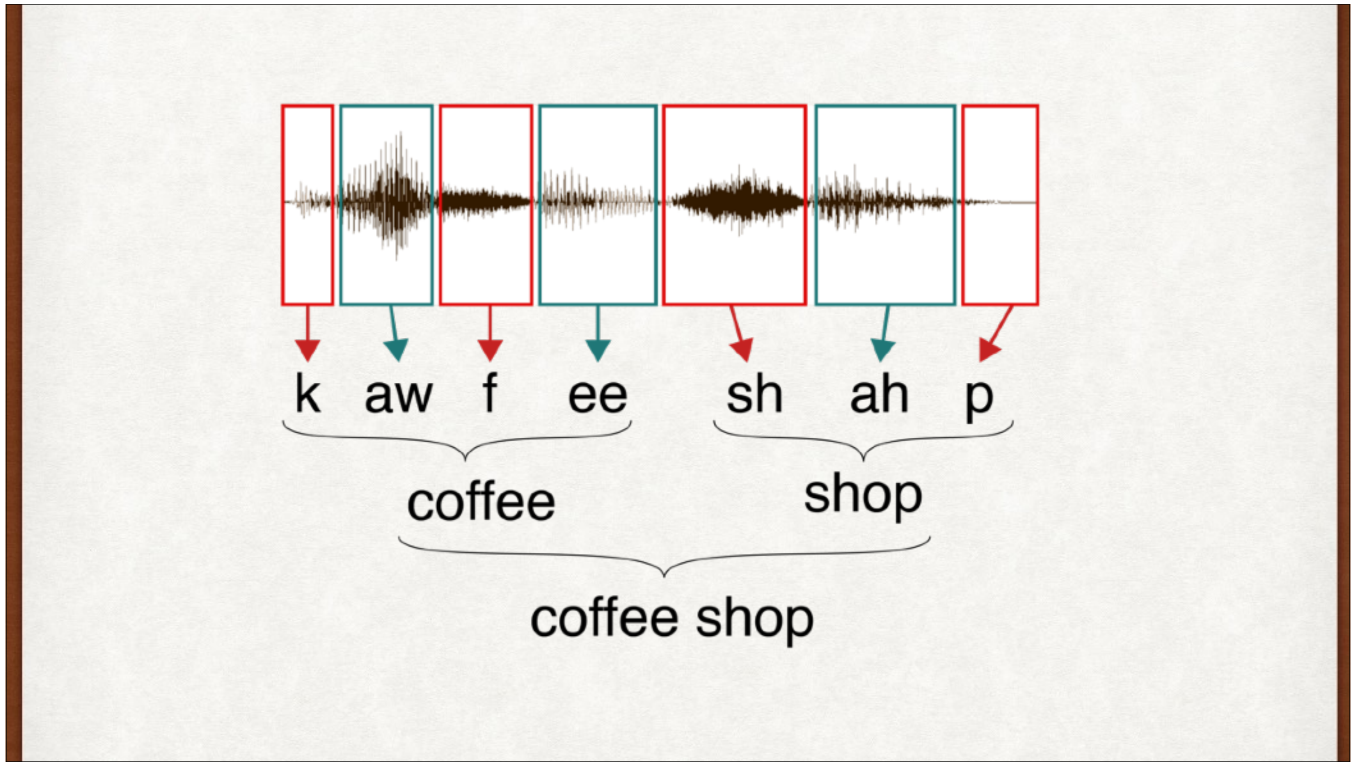
Some ML tasks. Reading digits and letters, recognizing faces.


Maecenas vulputate sapien
Quis tortor vehicula lobortis
Pellentesque tempus, nisi
$25 10
Pay to: Quisque nisi ipsum
Twenty-five and 10/100
Sinec Ultricies Behicula
Ellus felis venenatis augue
Breq tempus leo purus ac purus
T w e n t …
2 5 1 0
T w e n t …
2 5 1 0
Twenty-five dollars and ten cents
25.10

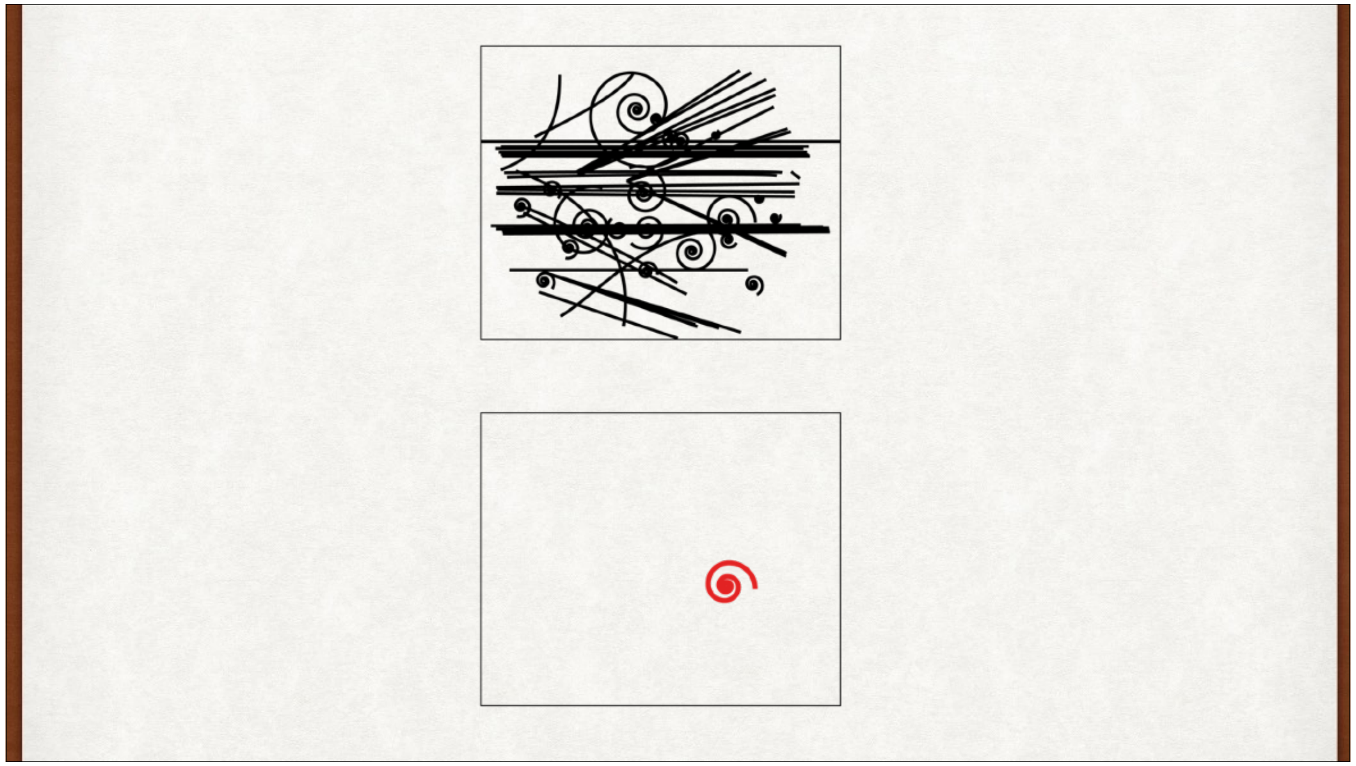
Some ML tasks. Reading digits and letters, recognizing faces.



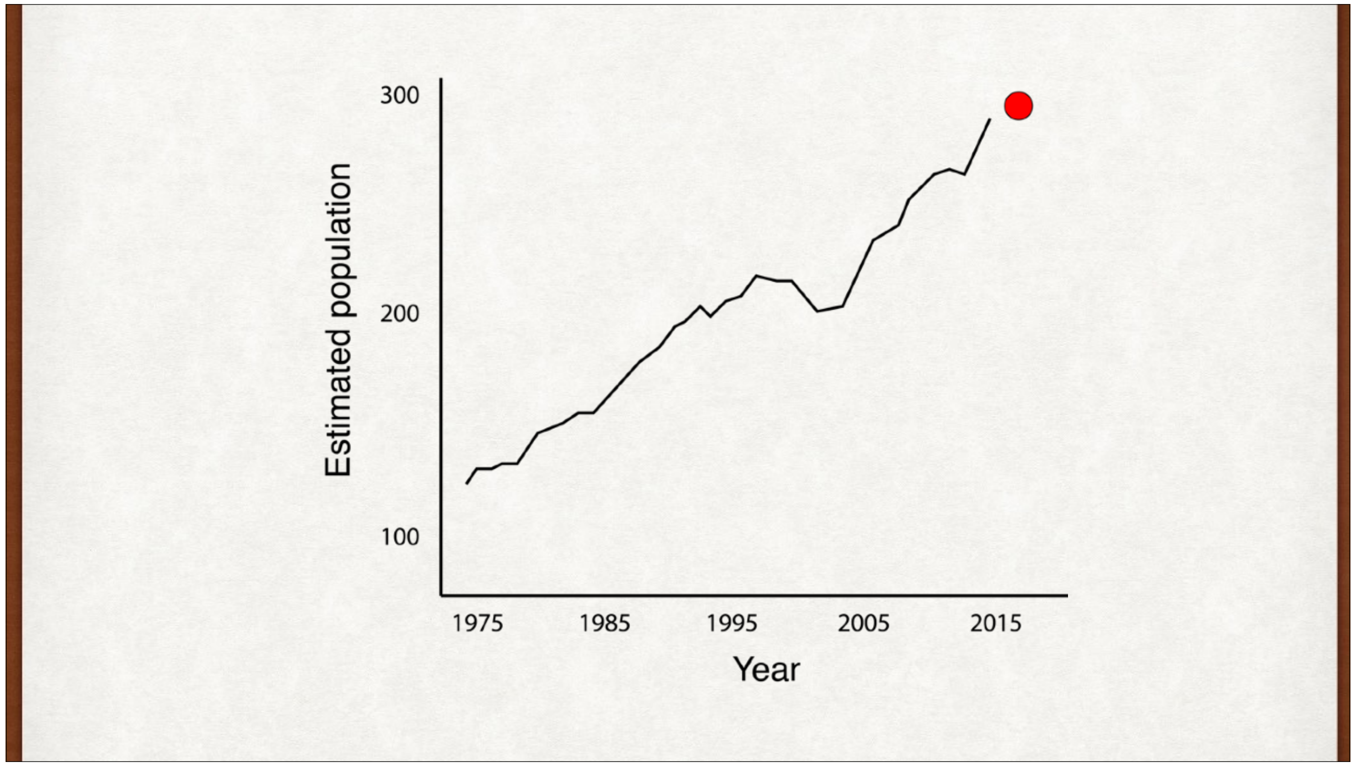
Some ML tasks. Reading digits and letters, recognizing faces.



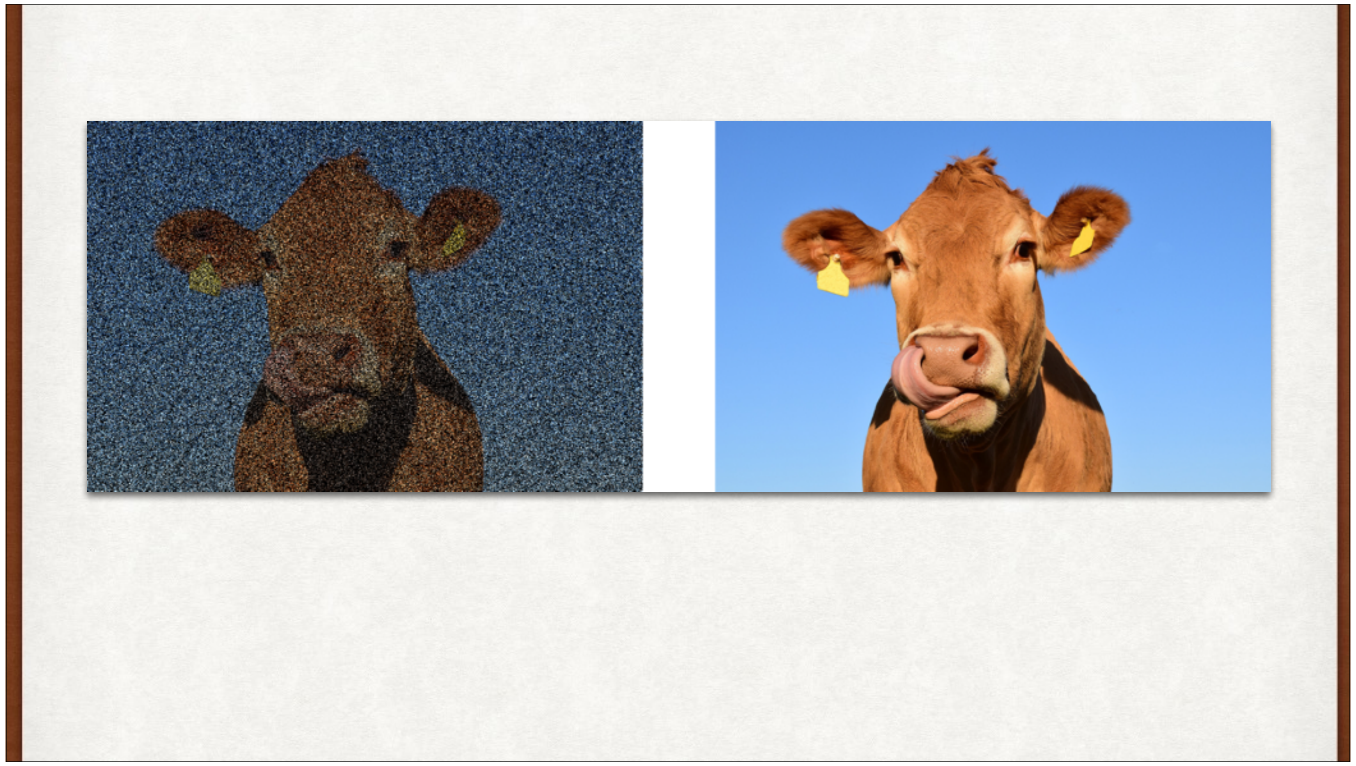
More ML tasks. Speech to words, finding a needle in a haystack, predicting missing data.



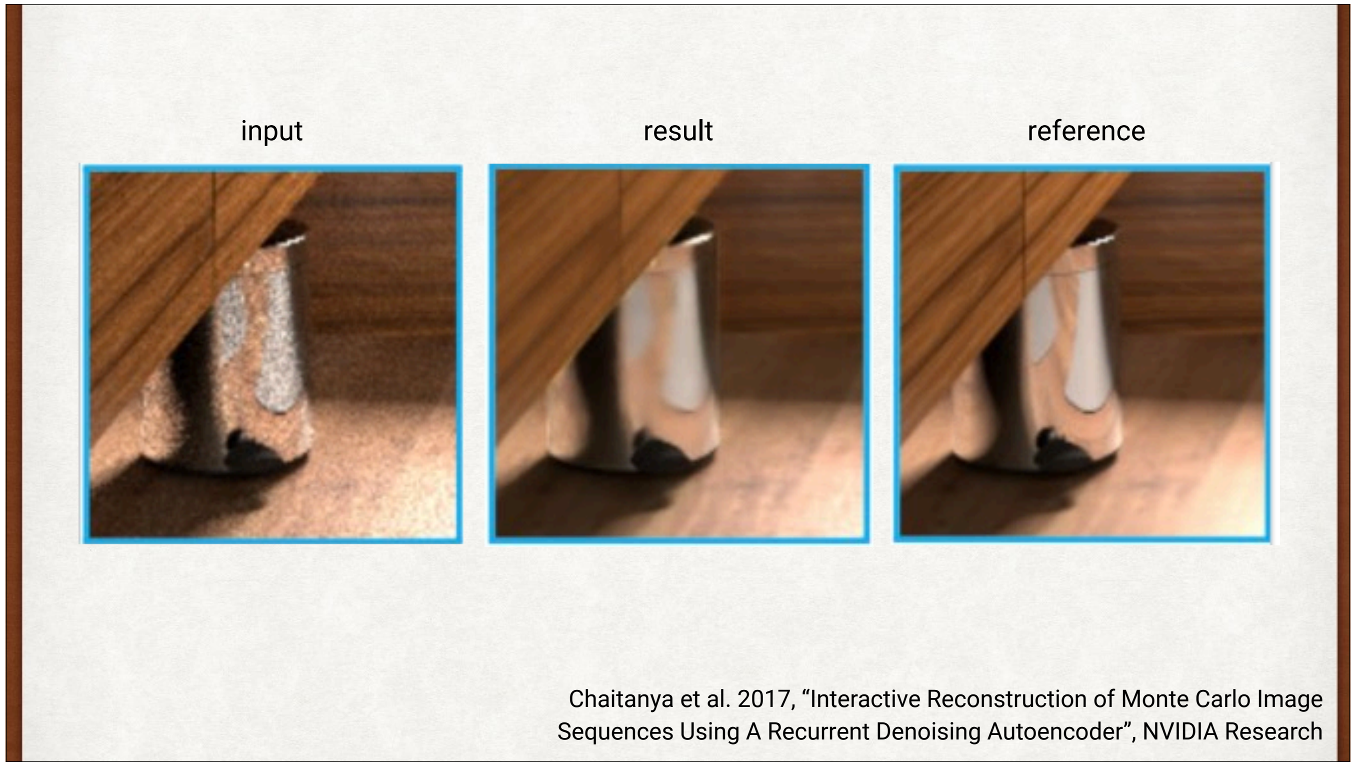
More ML tasks. Speech to words, finding a needle in a haystack, predicting missing data.



regression to mediocrity -> regression to the mean



Applications: Denoising, from the left to the right



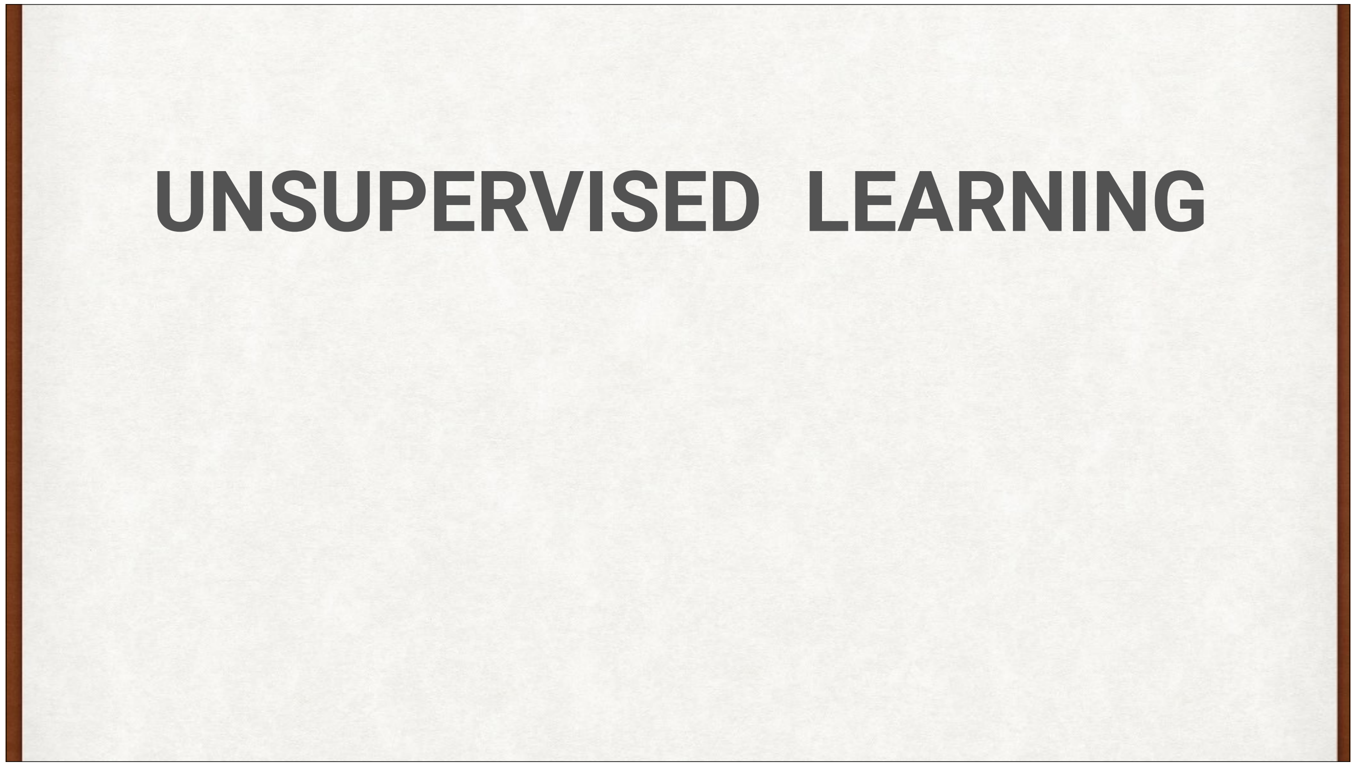
input

result

reference

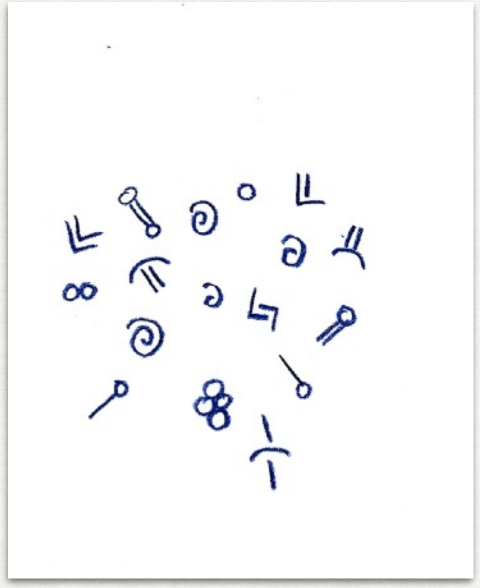
Chaitanya et al. 2017, “Interactive Reconstruction of Monte Carlo Image Sequences Using A Recurrent Denoising Autoencoder”, NVIDIA Research

Close-up reconstructions from the autoencoder.

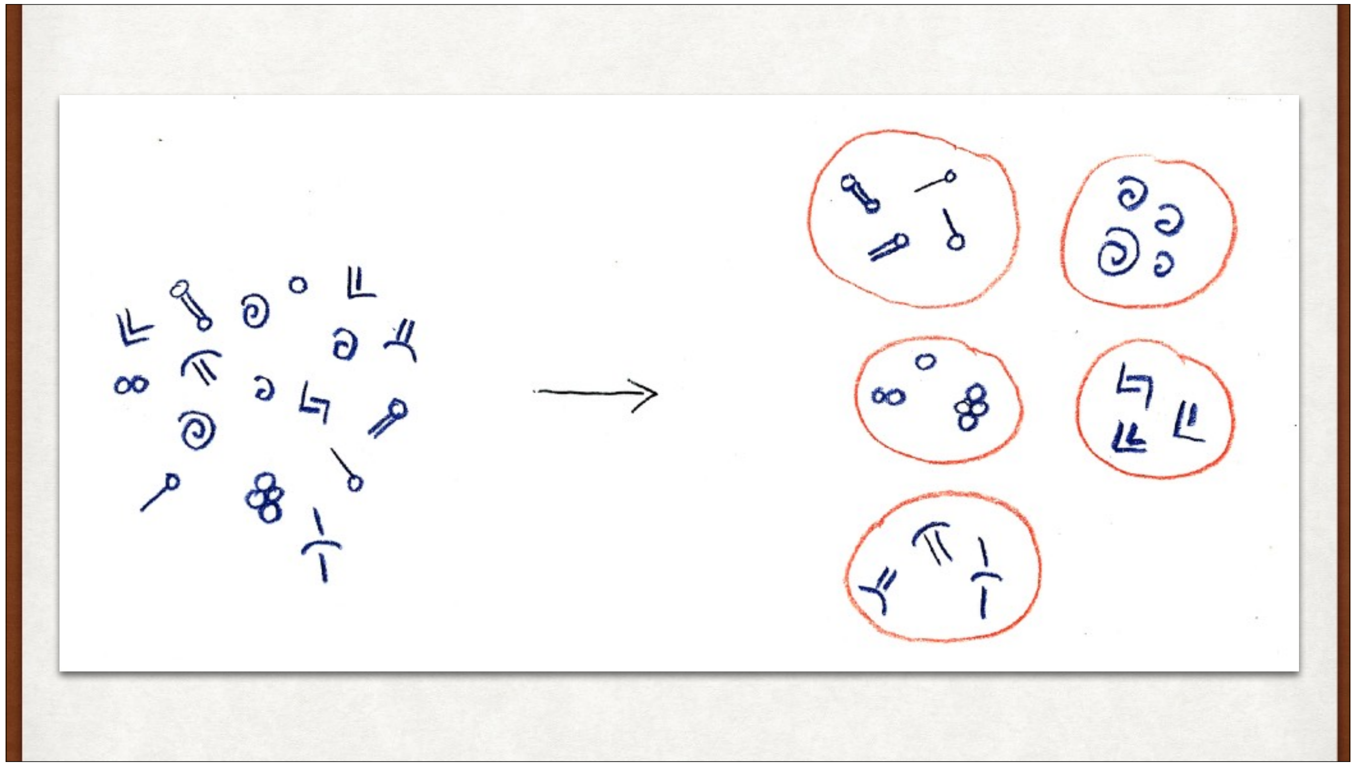


UNSUPERVISED LEARNING

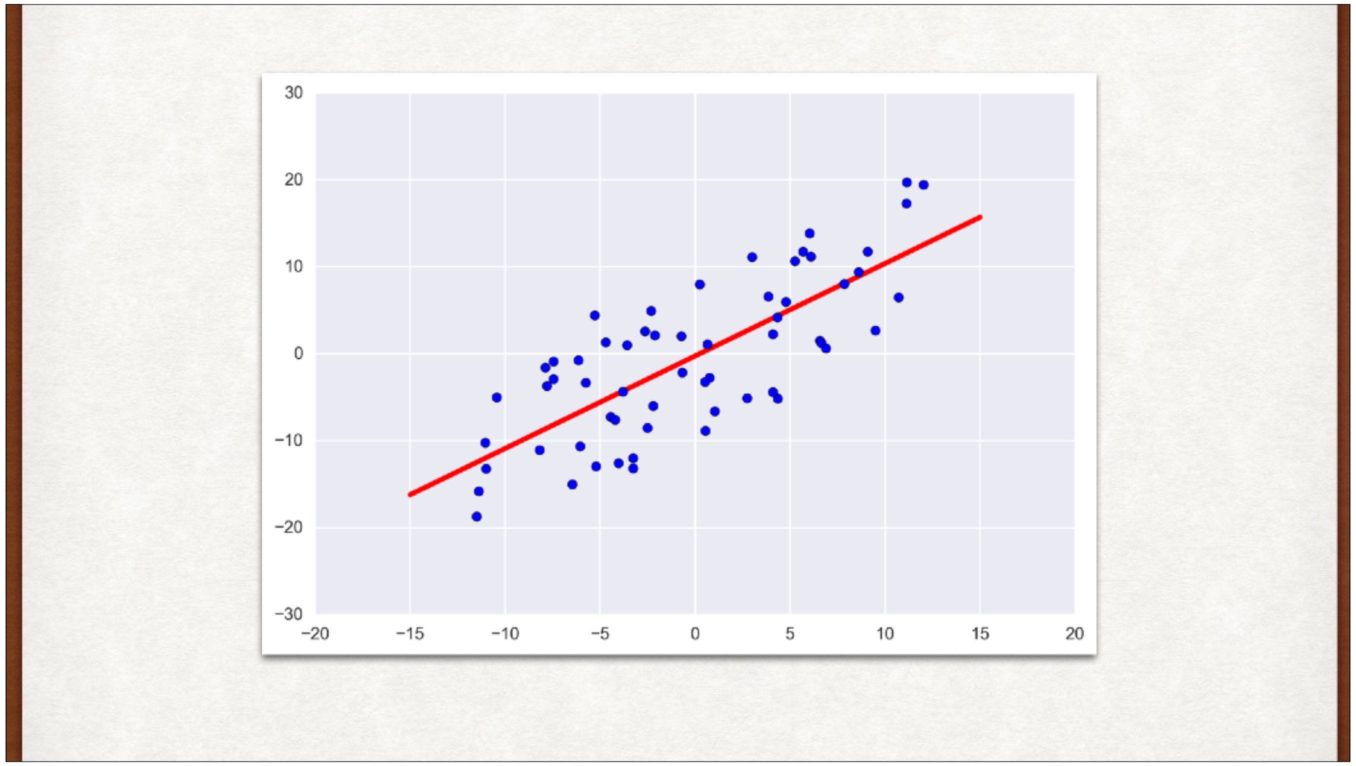
Unsupervised learning is when we have a bunch of data without labels. Usually we want to group or cluster them somehow or figure out something about their relationships.



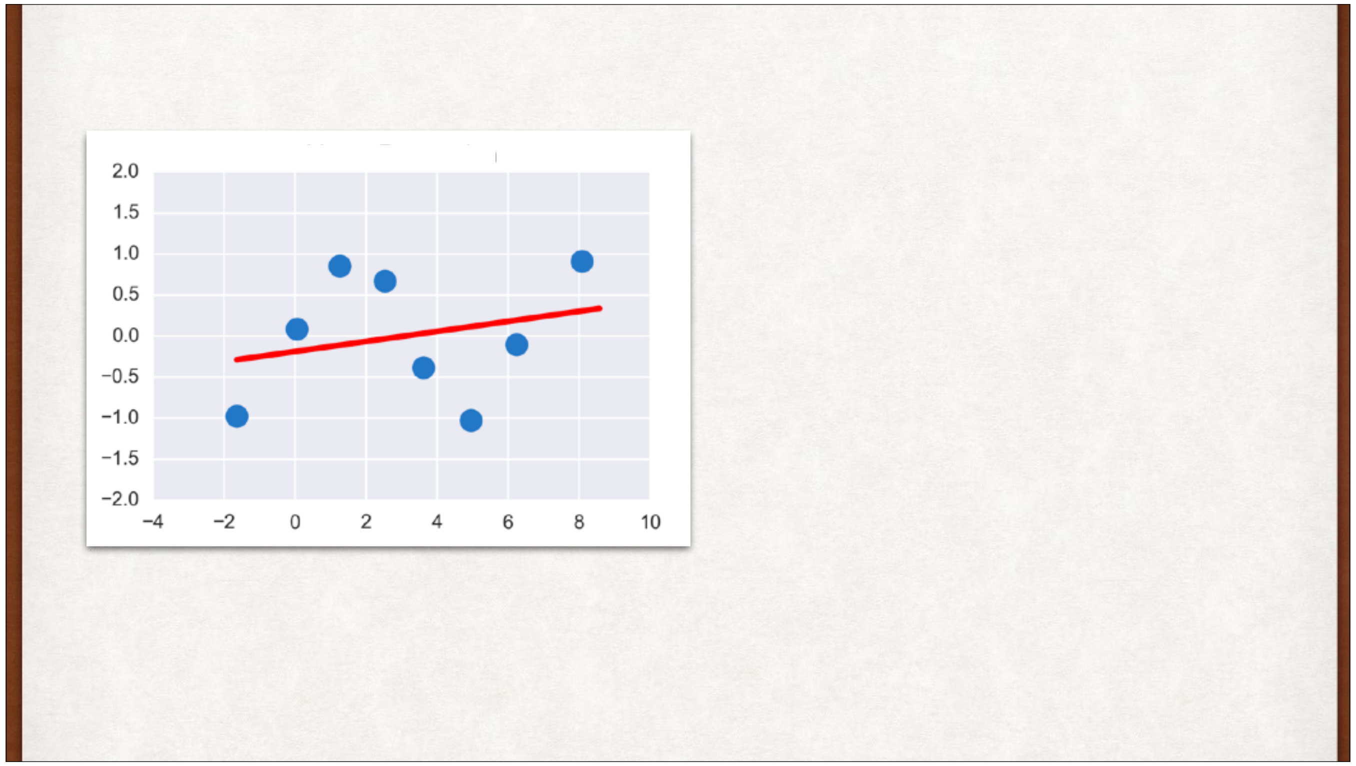
Discovered pottery markings. Cluster them into similar groups, without any help from us.



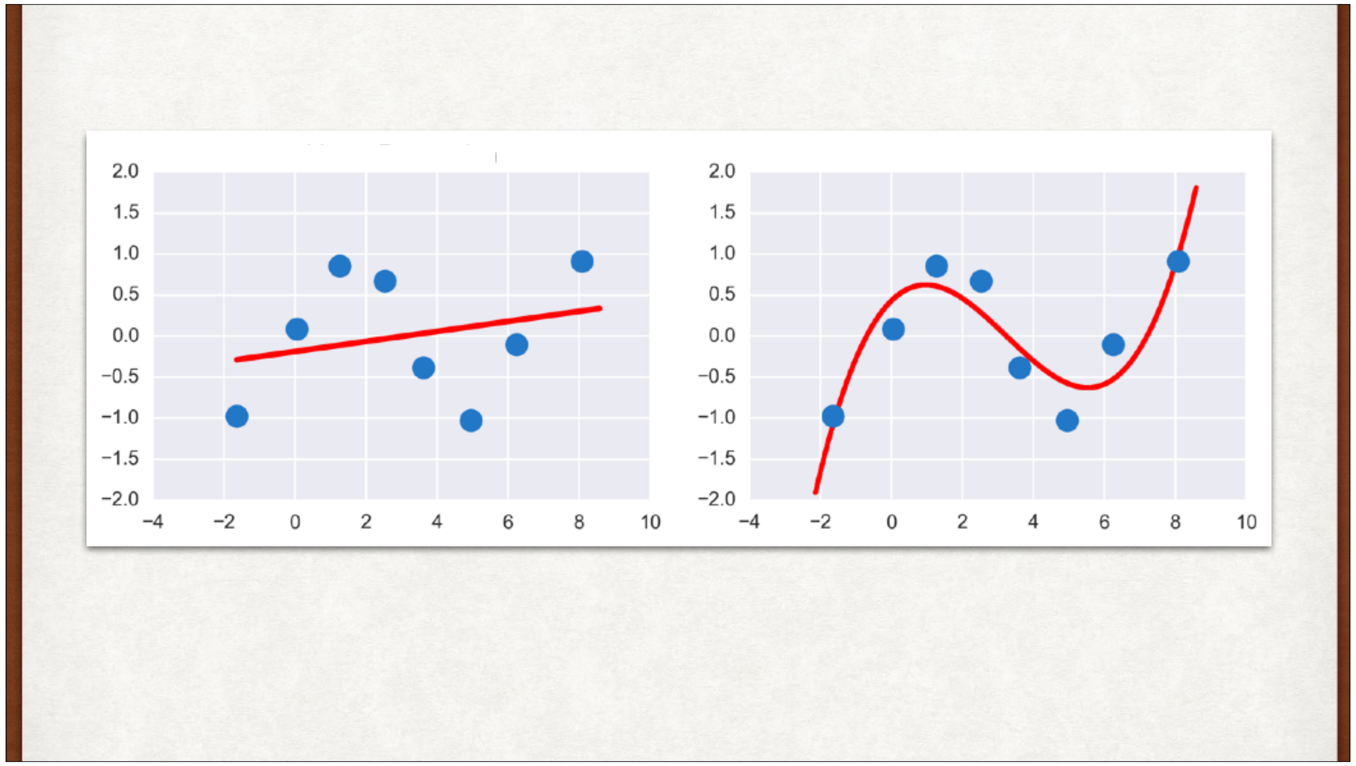
Discovered pottery markings. Cluster them into similar groups, without any help from us.



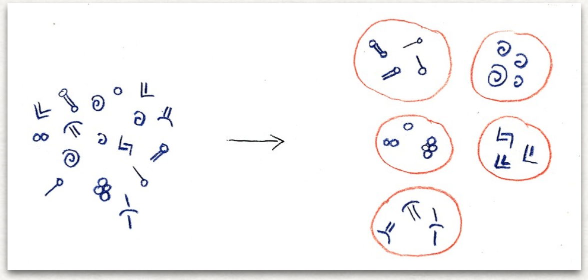
Linear regression: find the best line that approximates the data.

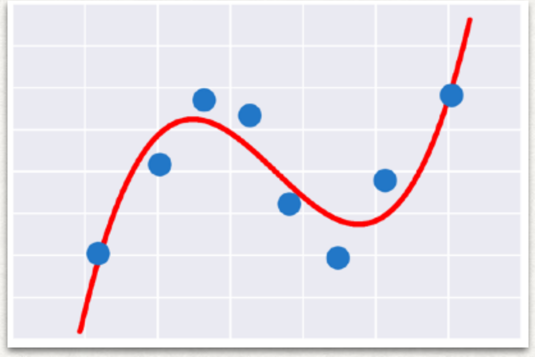


Sometimes a curve fits the data better than a line.

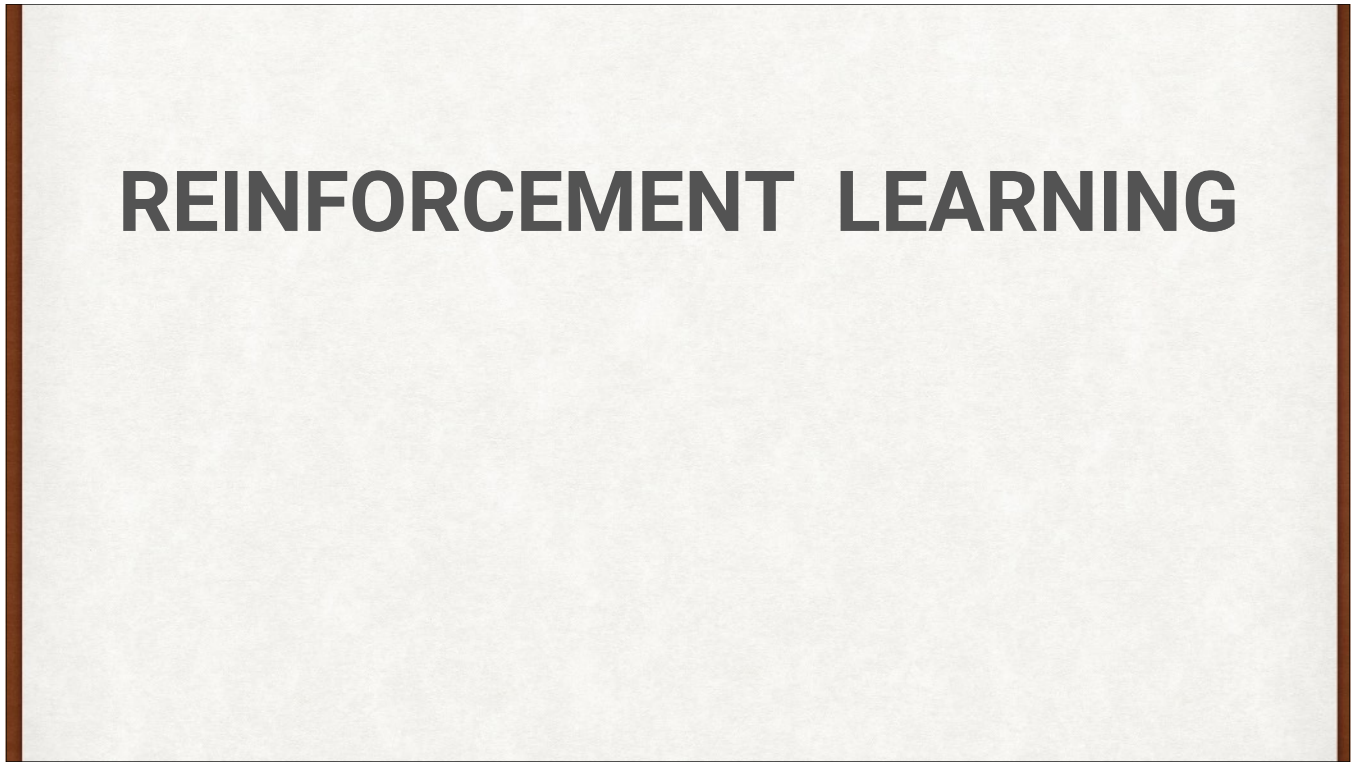


Sometimes a curve fits the data better than a line.



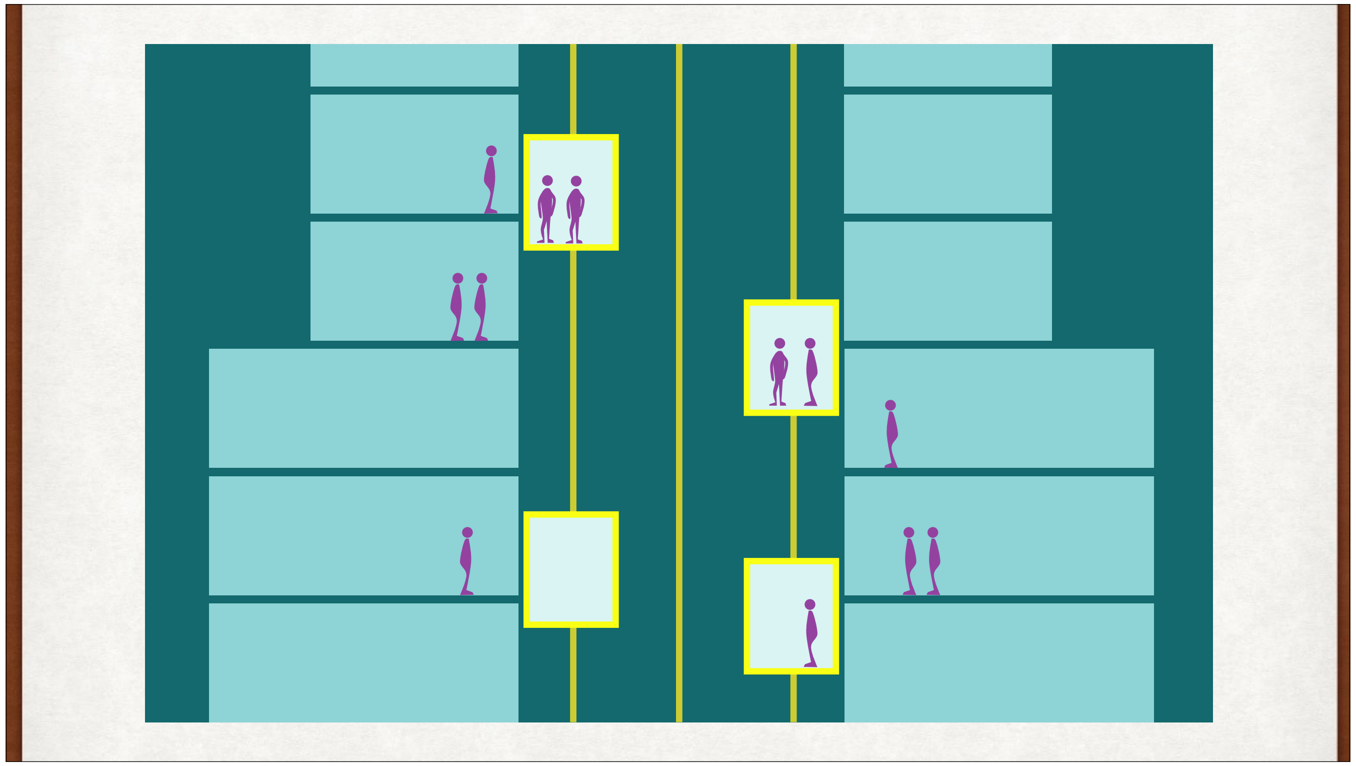


Clustering and interpolating (with classifying).

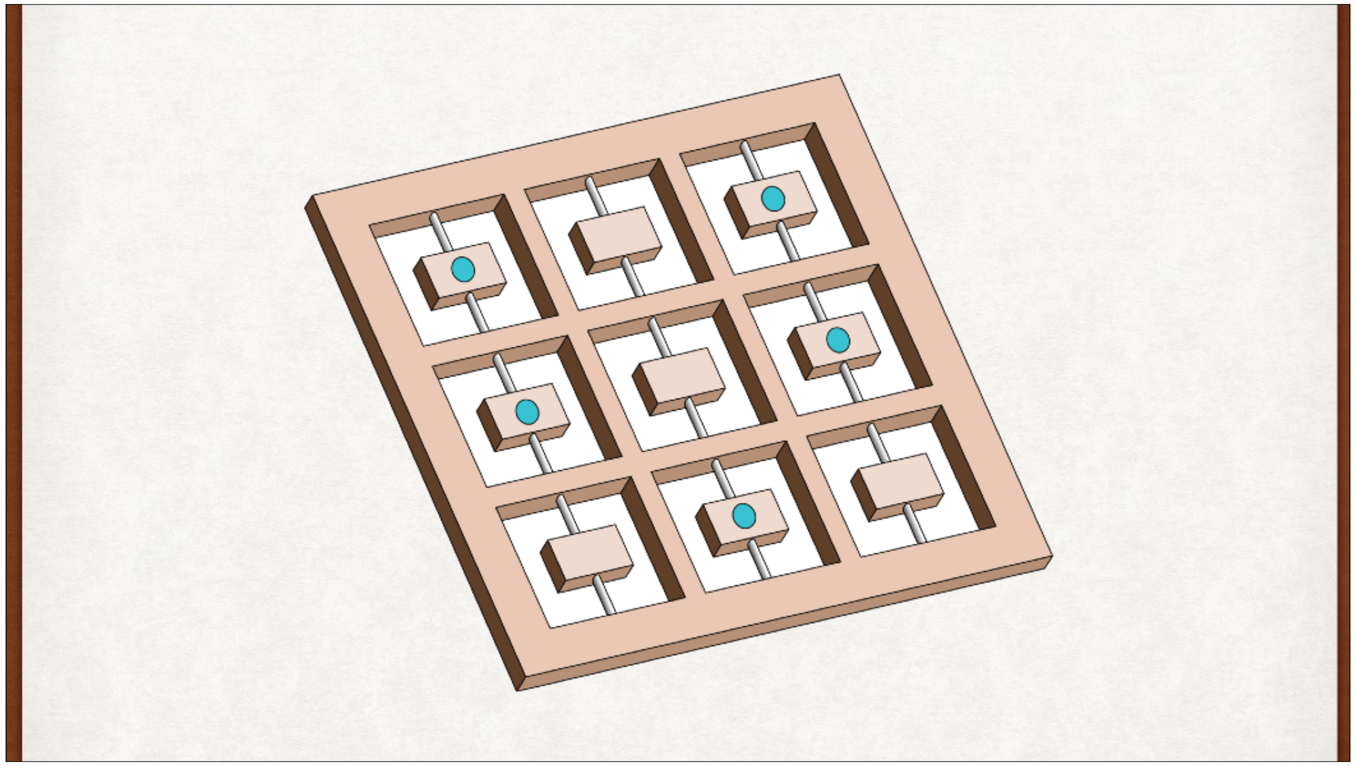


REINFORCEMENT LEARNING

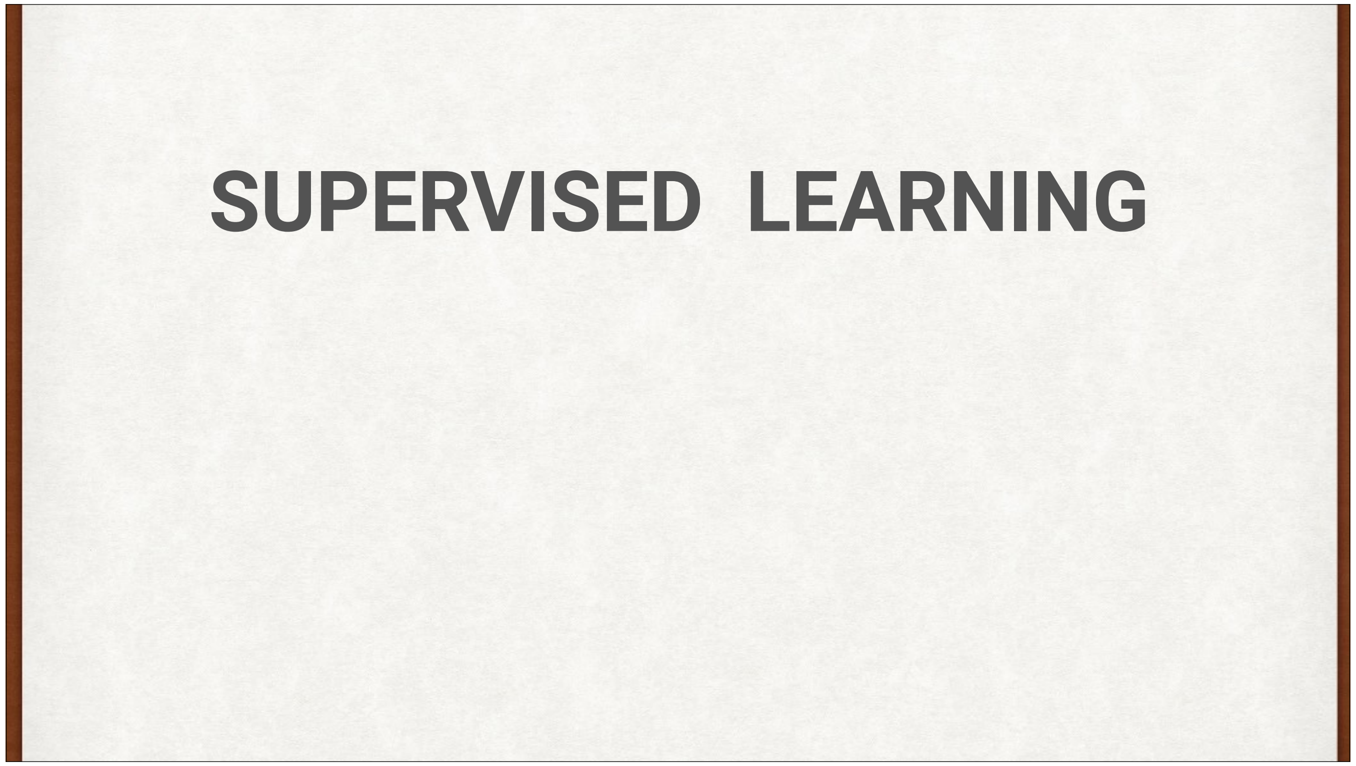
Like training a dog. Operant conditioning. Rewards for good behavior.



What’s the best schedule for a building’s elevators?

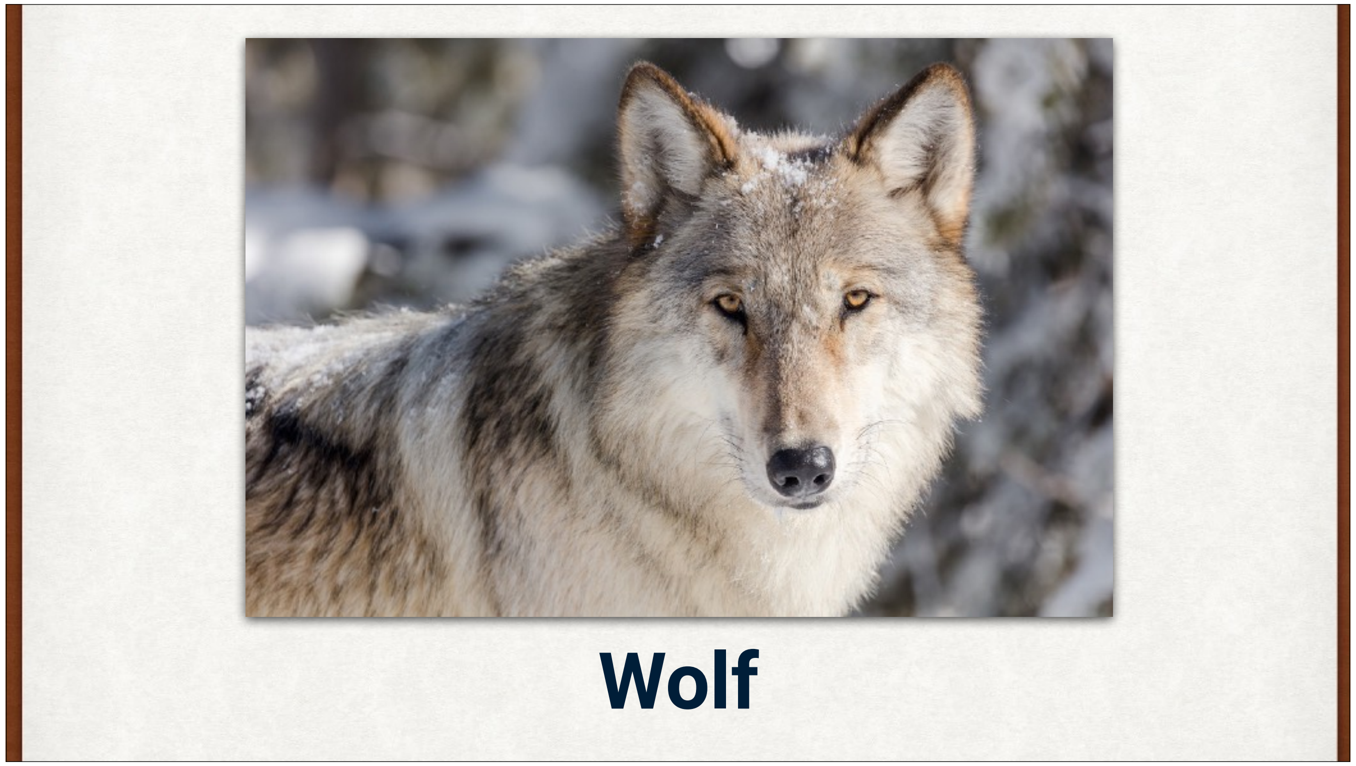


We’ll use RL to learn how to play Flippers later on. It’s a solitaire tic-tac-toe game.



SUPERVISED LEARNING

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



Wolf

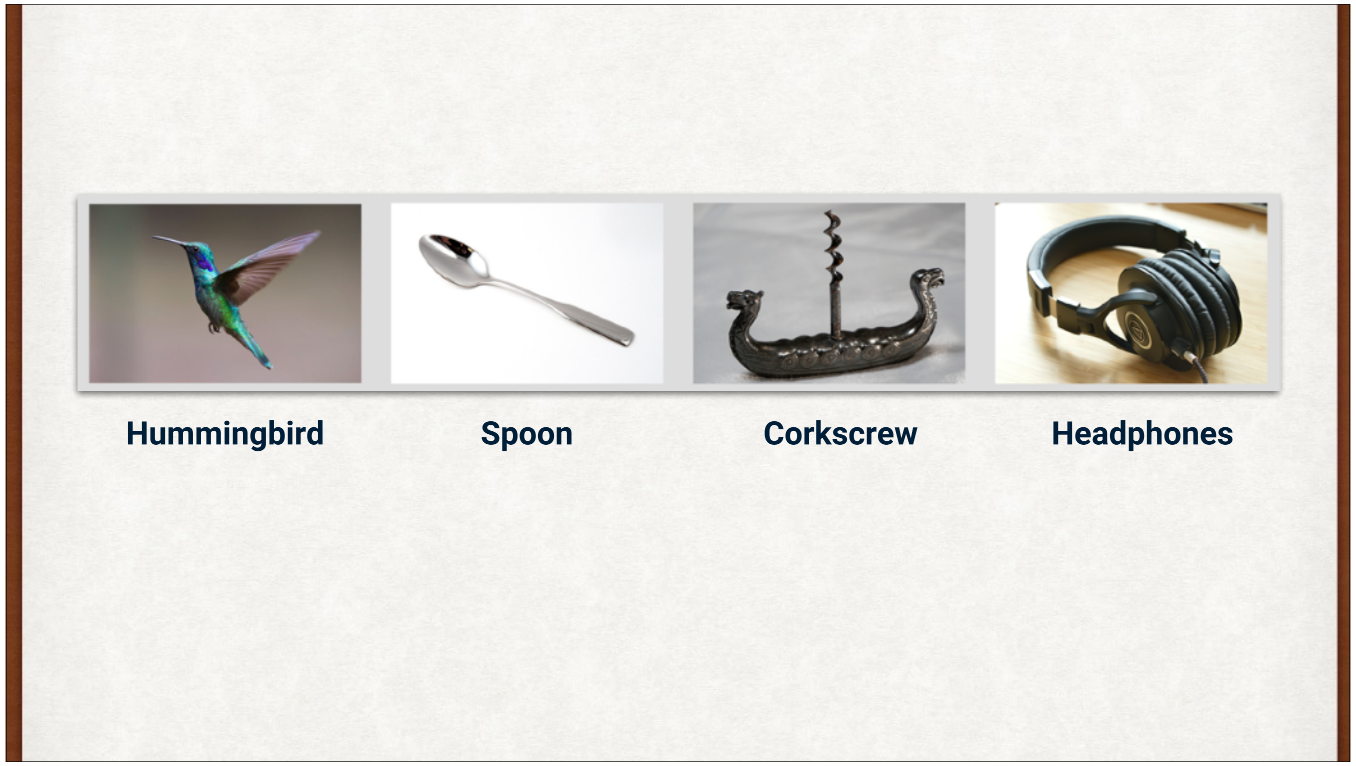
We’re pretty sure this is a wolf

https://pixabay.com/en/wolf-lone-predator-wildlife-nature-3818343/

Frog

This is pretty clearly a frog

https://pixabay.com/photos/frog-macro-amphibian-green-111179/



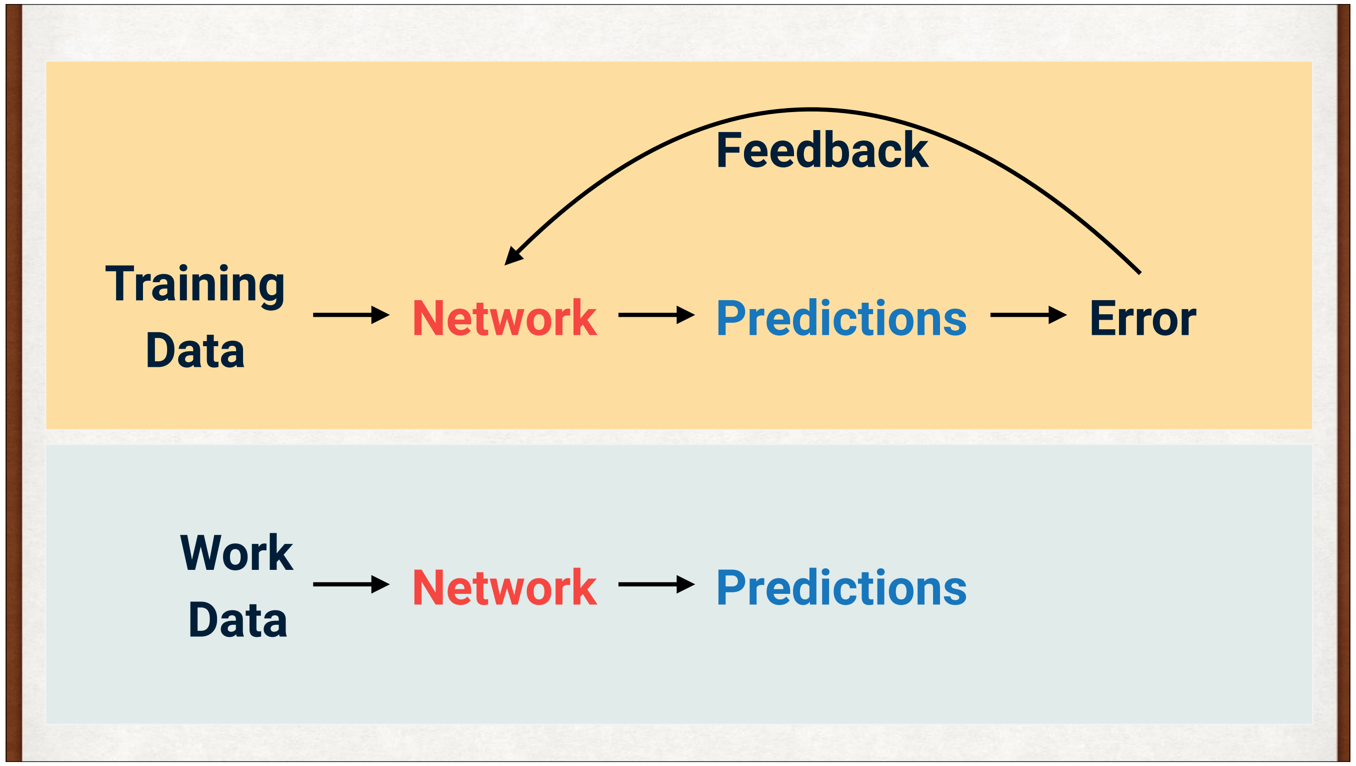
Hummingbird

Spoon

Corkscrew

Headphones

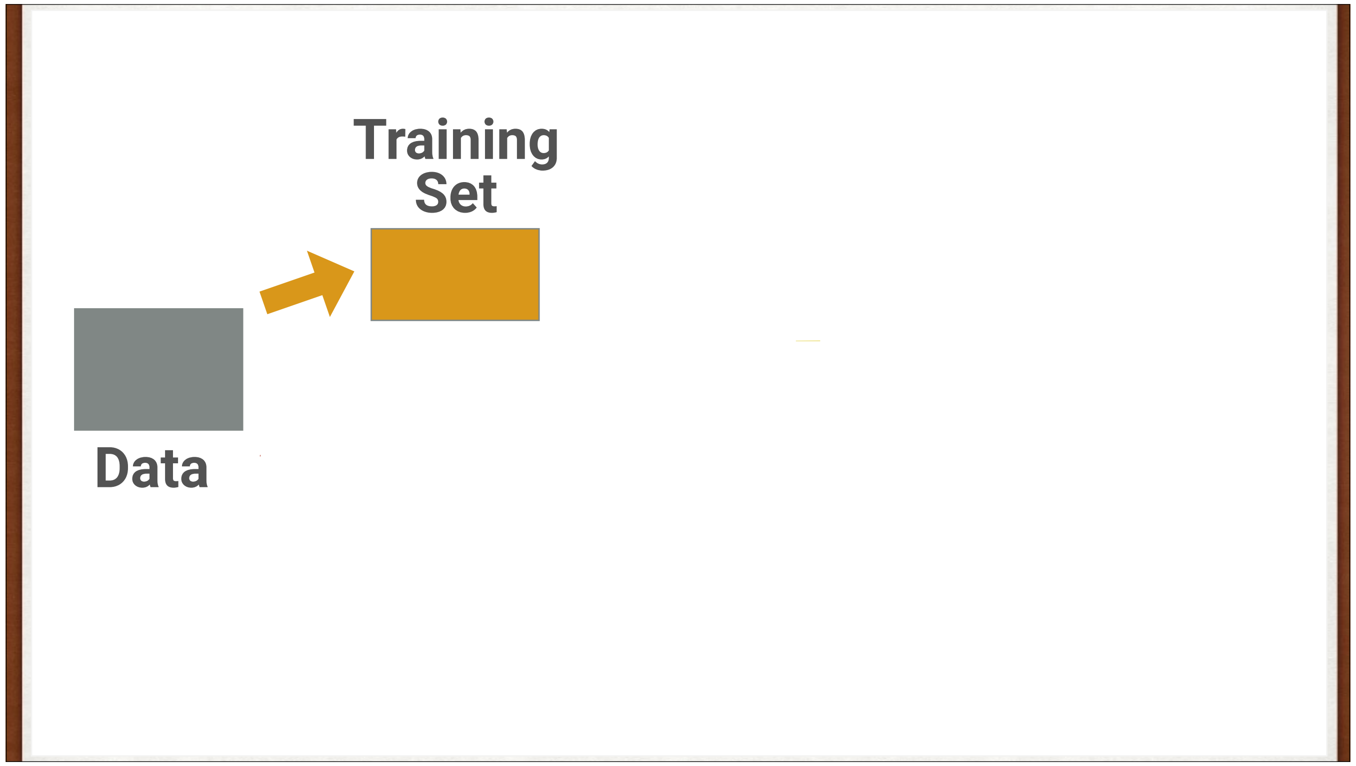
Some labels we’ve assigned to objects, so the system can learn which labels go with what objects.



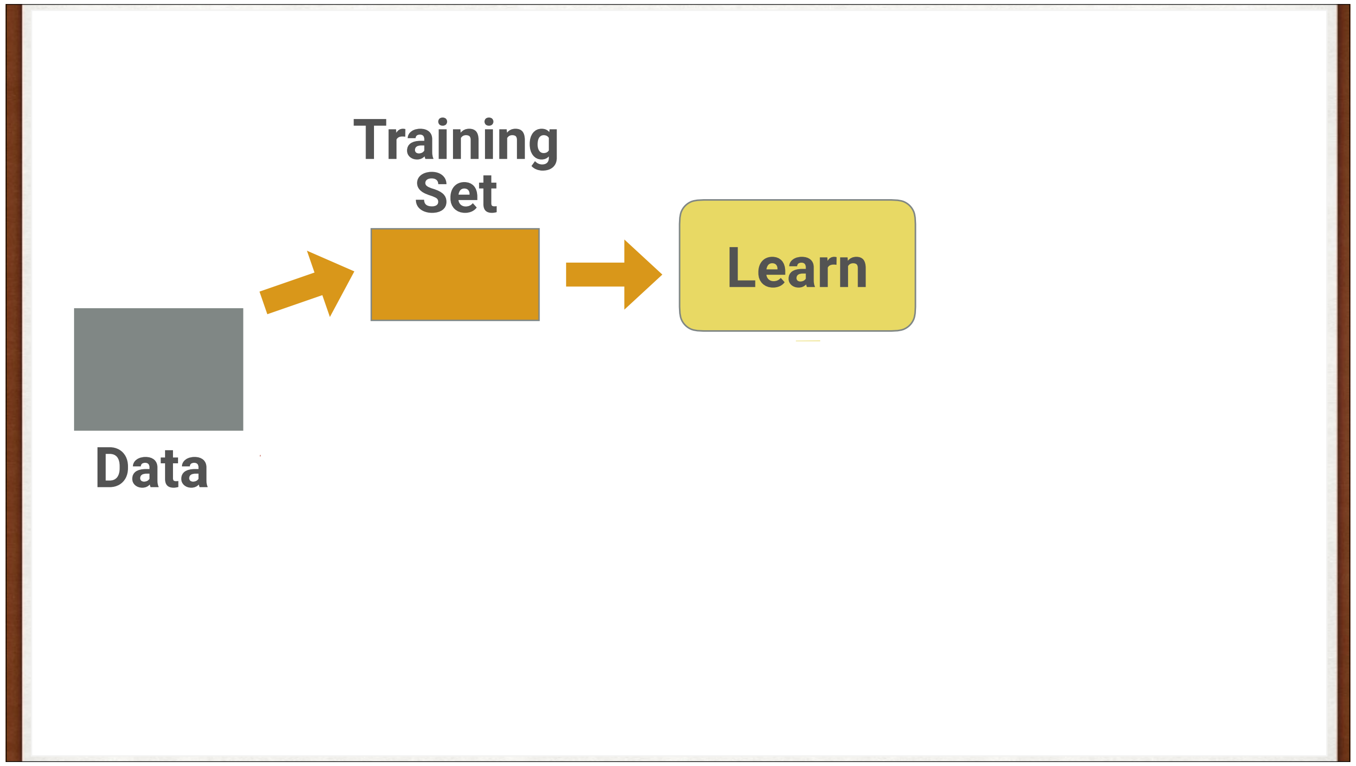
The process of training, step by step, and deployment

Data

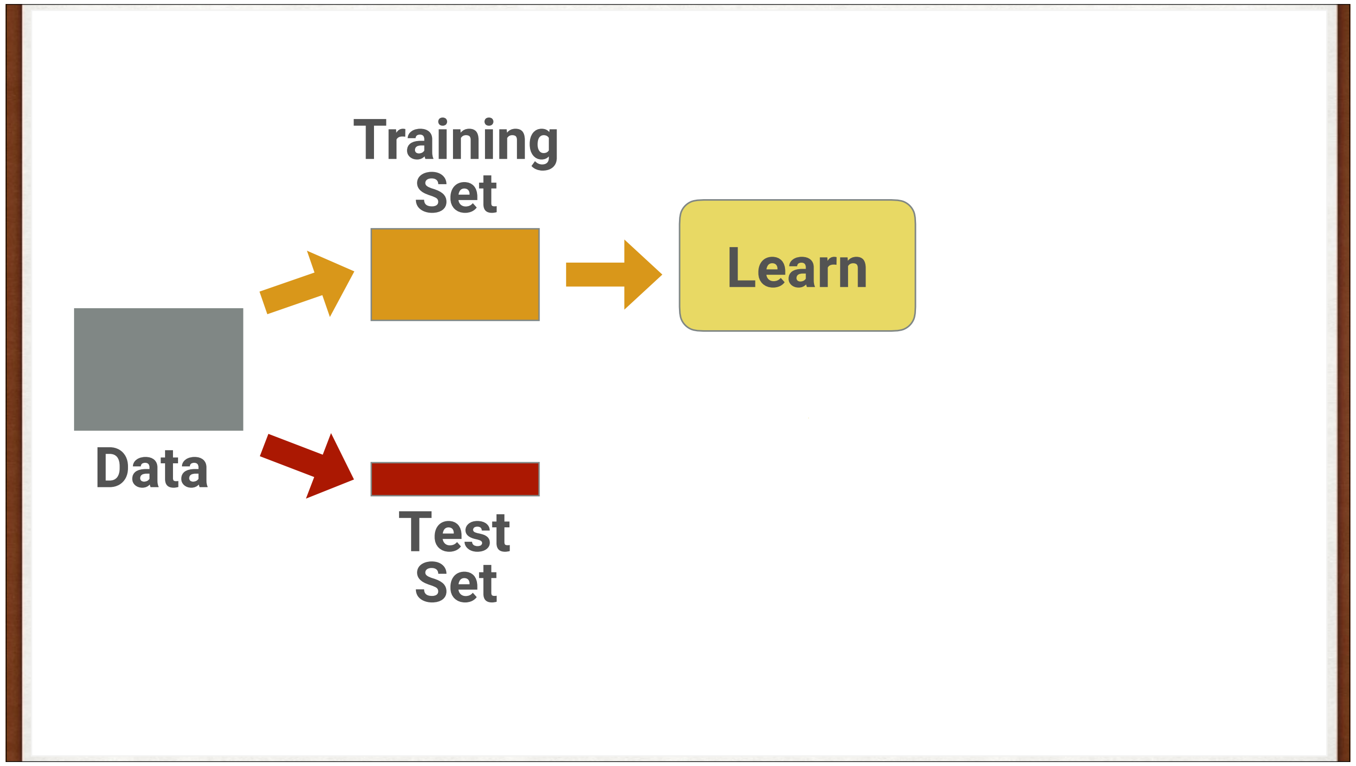
The basic supervised learning flow, using clunky Keynote graphics.



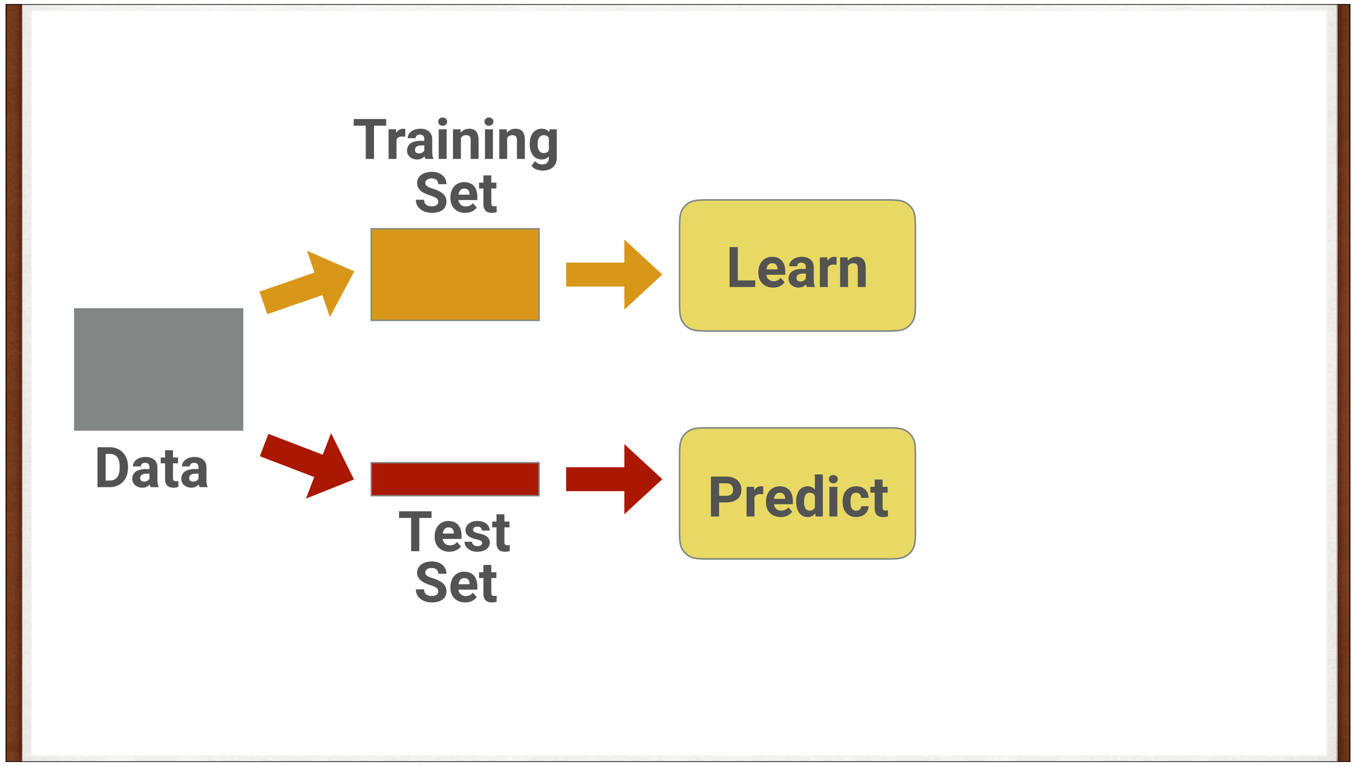
The basic supervised learning flow, using clunky Keynote graphics.



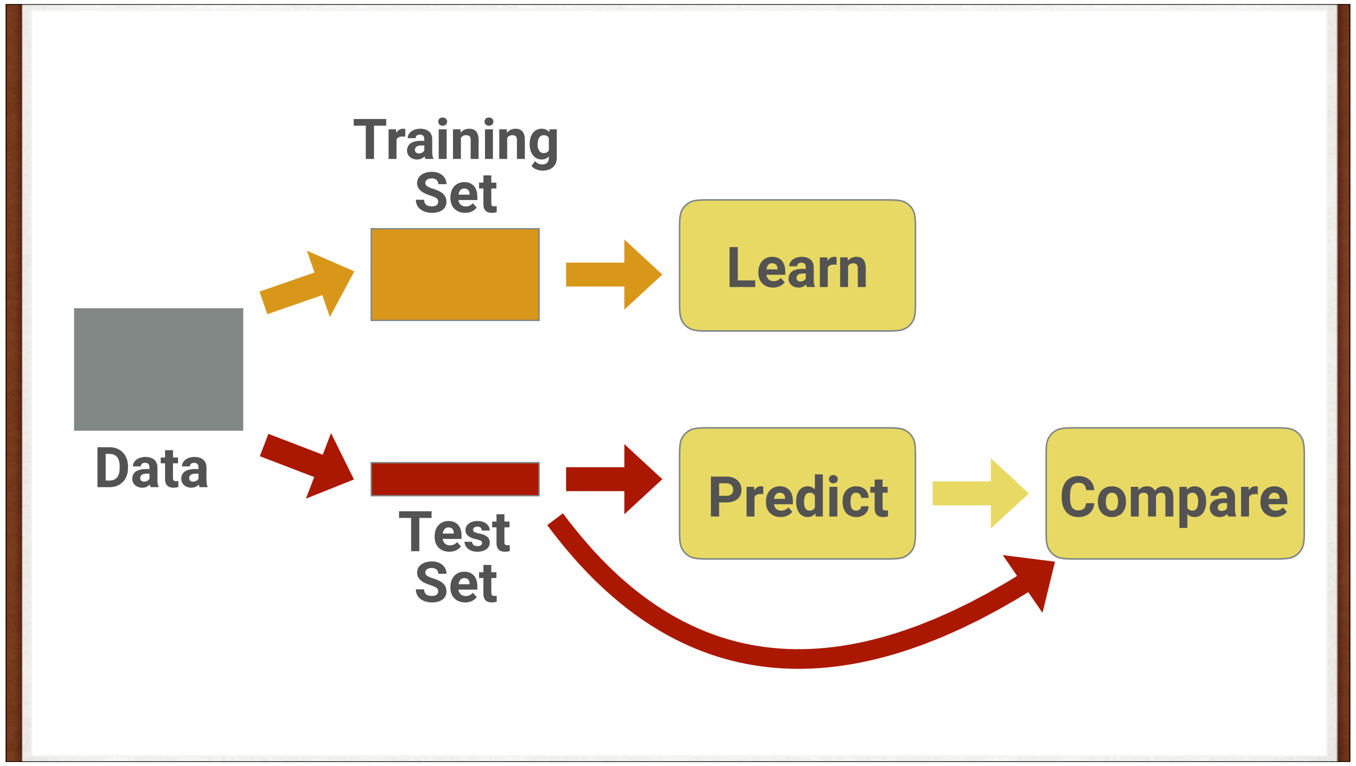
The basic supervised learning flow, using clunky Keynote graphics.



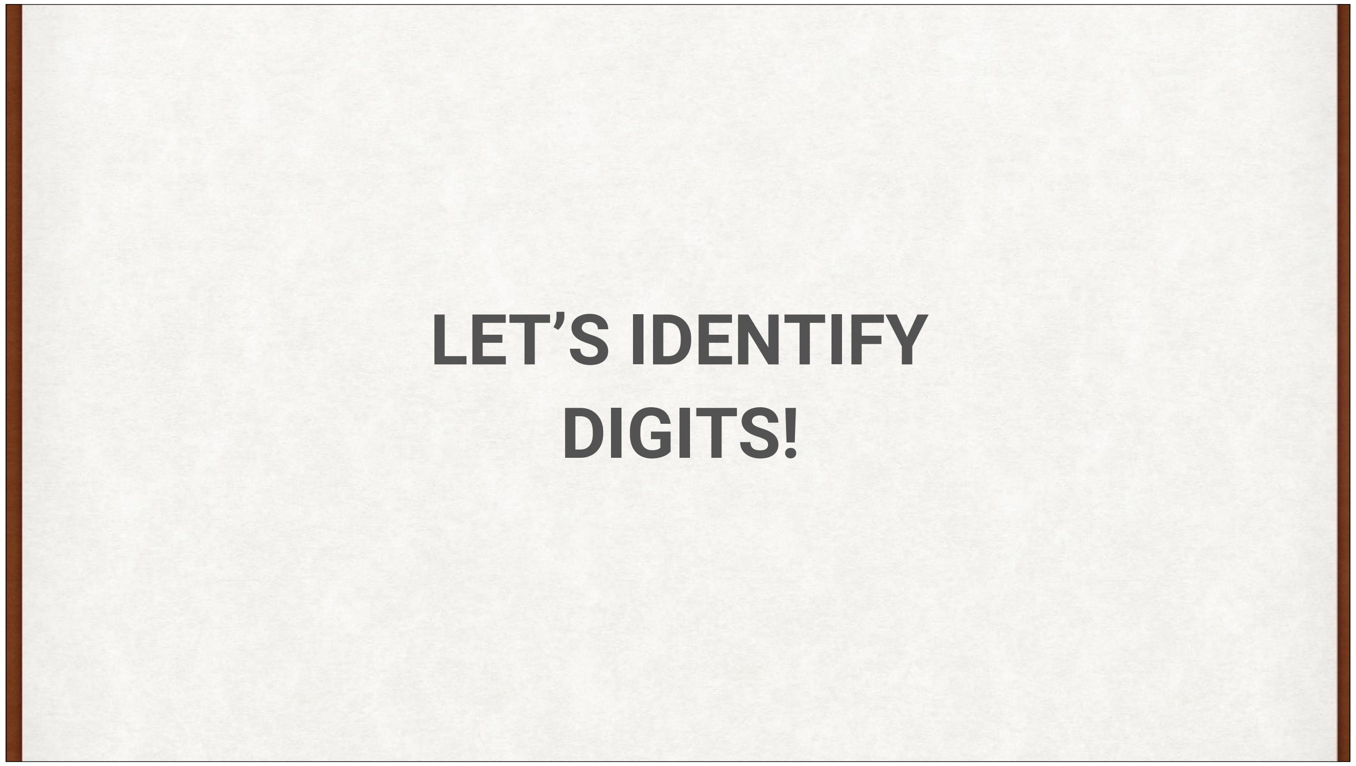
The basic supervised learning flow, using clunky Keynote graphics. The test set is used only once, at the very end. Think of that data as final exam questions. Once seen, they cannot be used again. If performance is not good enough, we need to start over with an untrained network.



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.

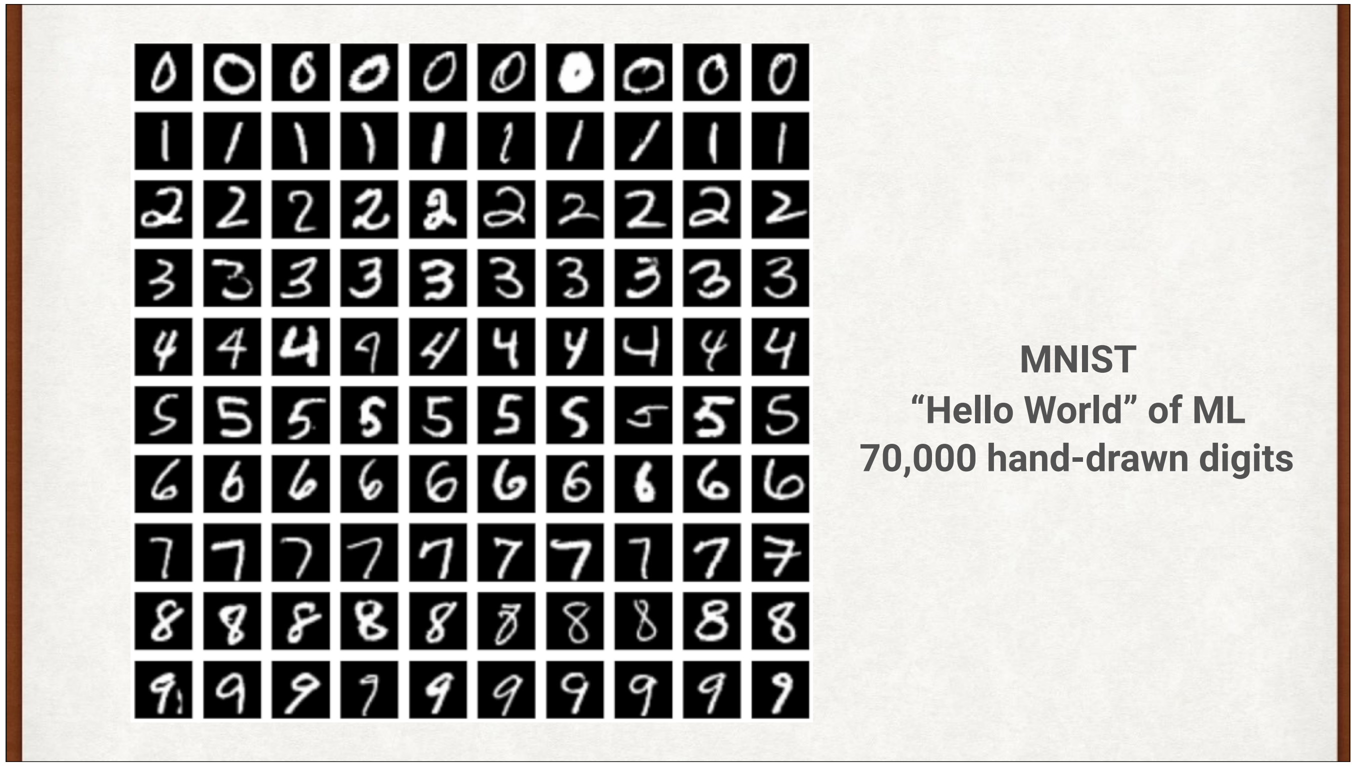


The basic supervised learning flow, using clunky Keynote graphics. The test set is used only once, at the very end. Think of that data as final exam questions. Once seen, they cannot be used again. If performance is not good enough, we need to start over with an untrained network.



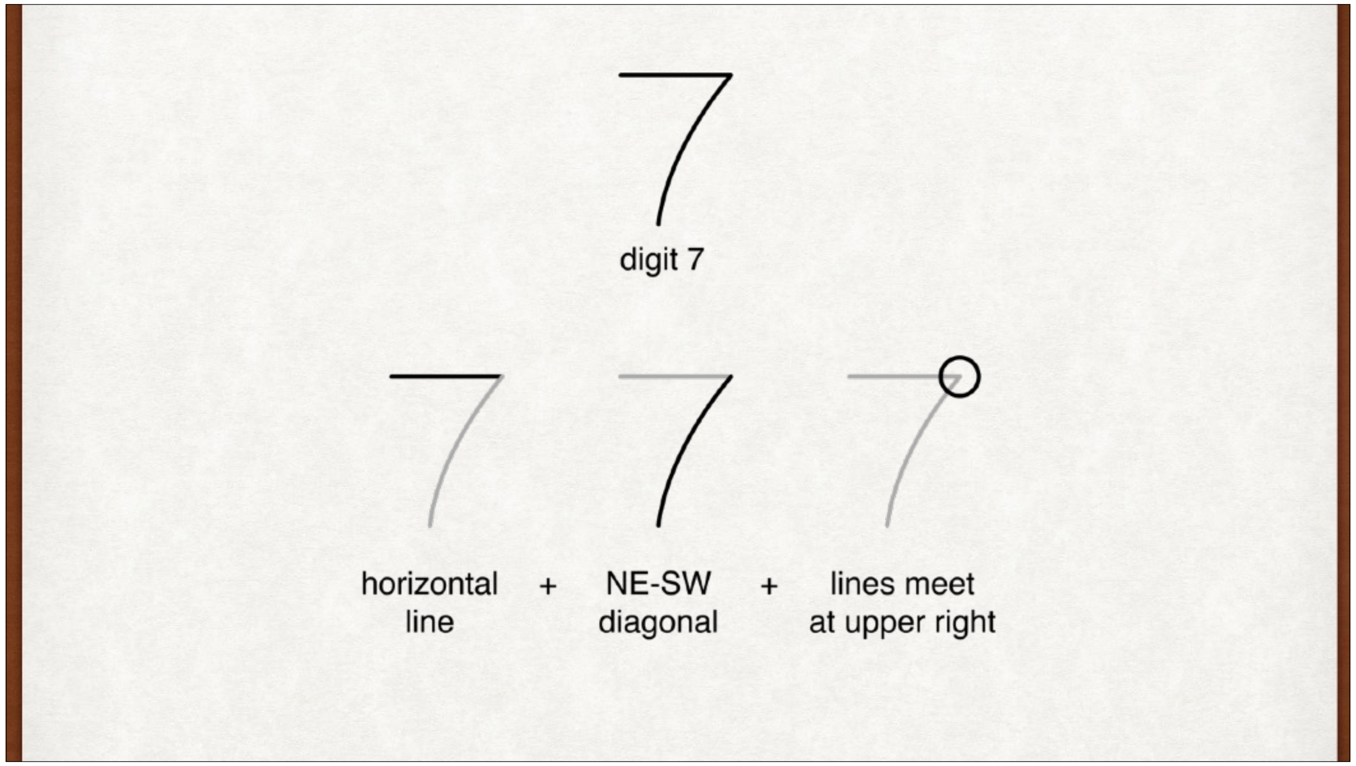
LET’S IDENTIFY DIGITS!

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



MNIST
“Hello World” of ML
70,000 hand-drawn digits

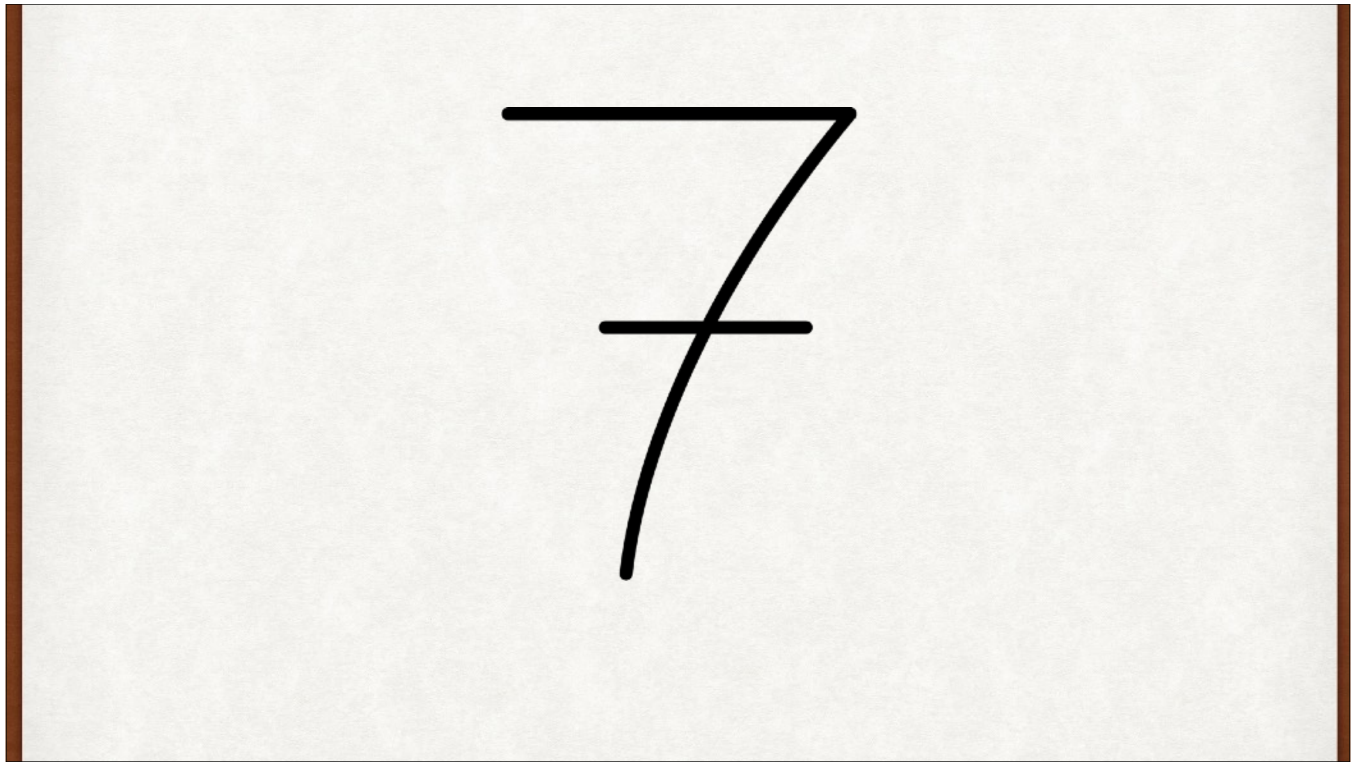
Everyone’s favorite classification problem! Called MNIST, it’s simple, clean, and just big and complicated enough to be interesting. 60k train, 10k test. US National Institute of Standards and Technology. 1/2 census bureau, 1/2 high schoolers.



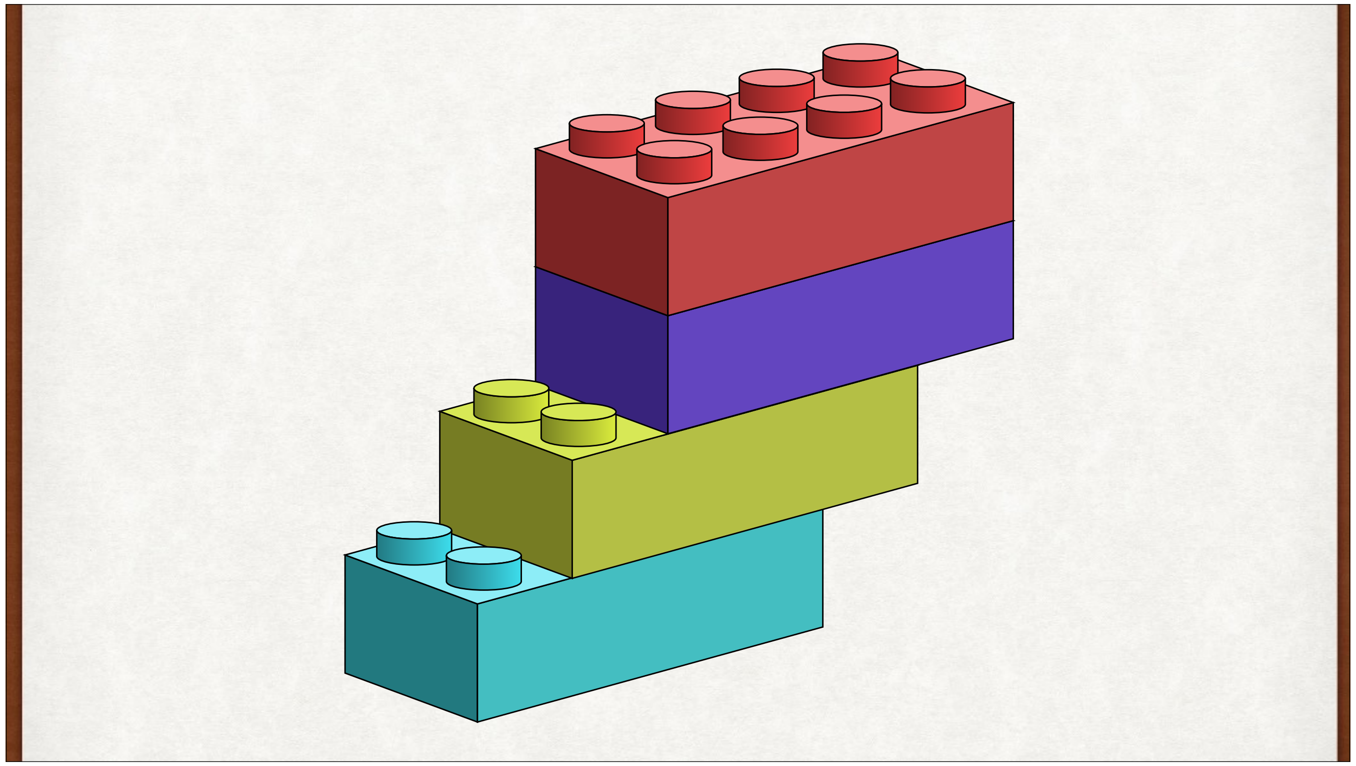
digit 7

horizontal line + NE-SW diagonal + lines meet at upper right

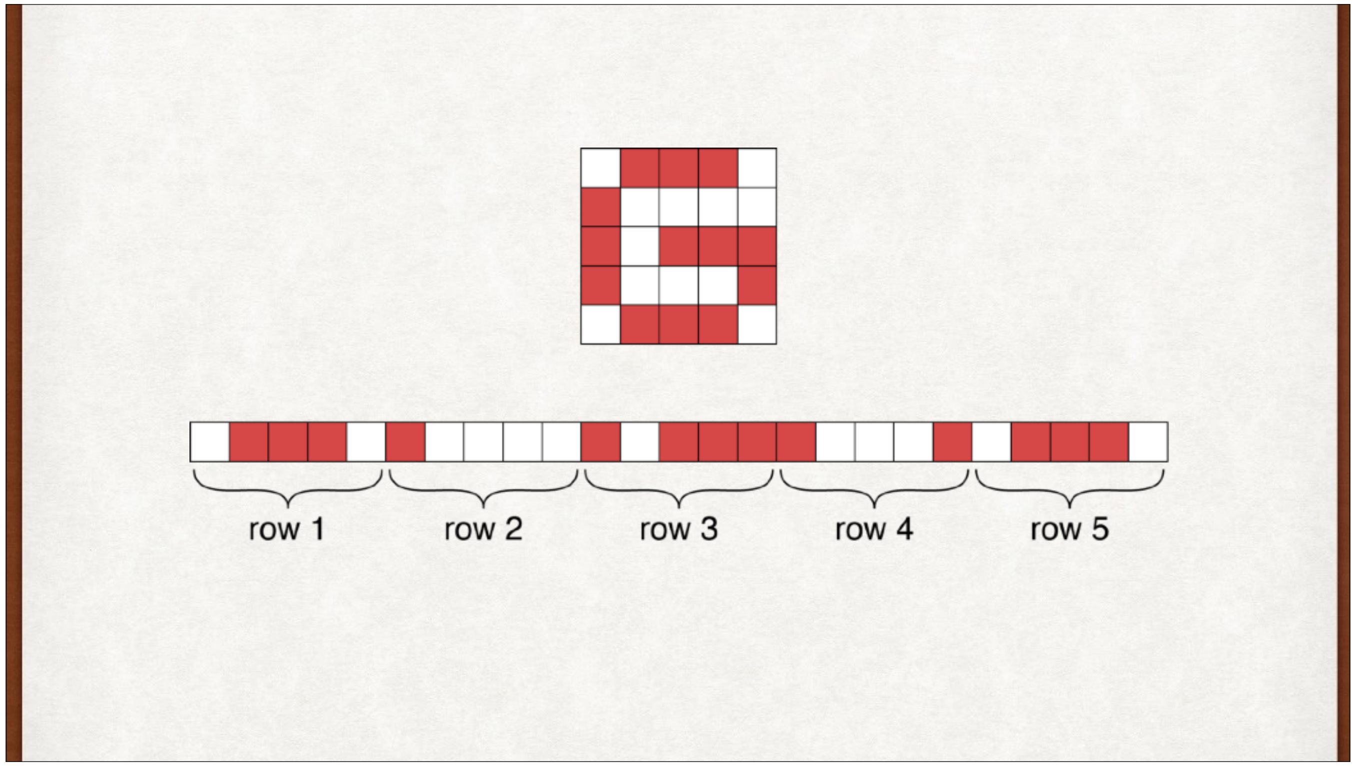
Expert system: craft rules that tell us which digit this is.



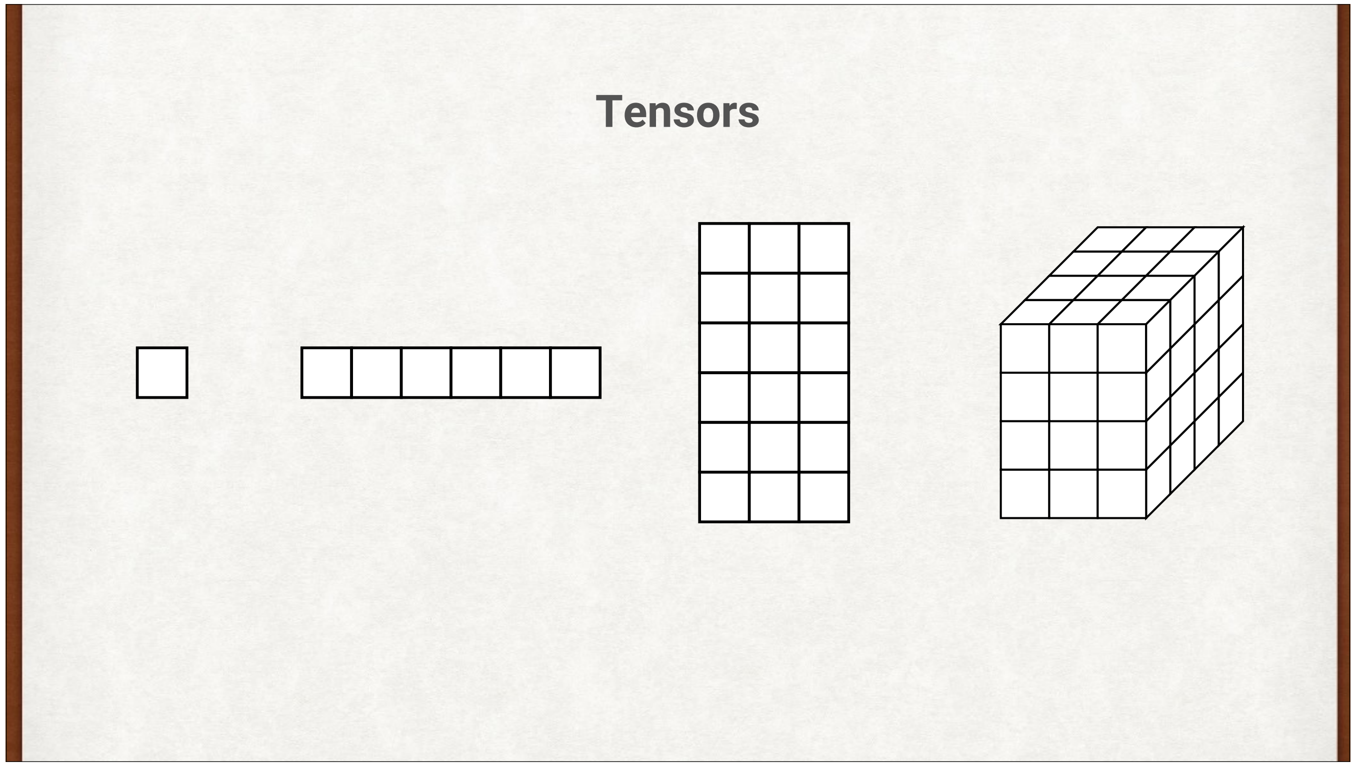
Uh oh, we forgot the line some people use. This is not going to scale up well for digits, much less reading X-rays or piloting an airplane. Instead of hand-crafting rules, let’s try DL.



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

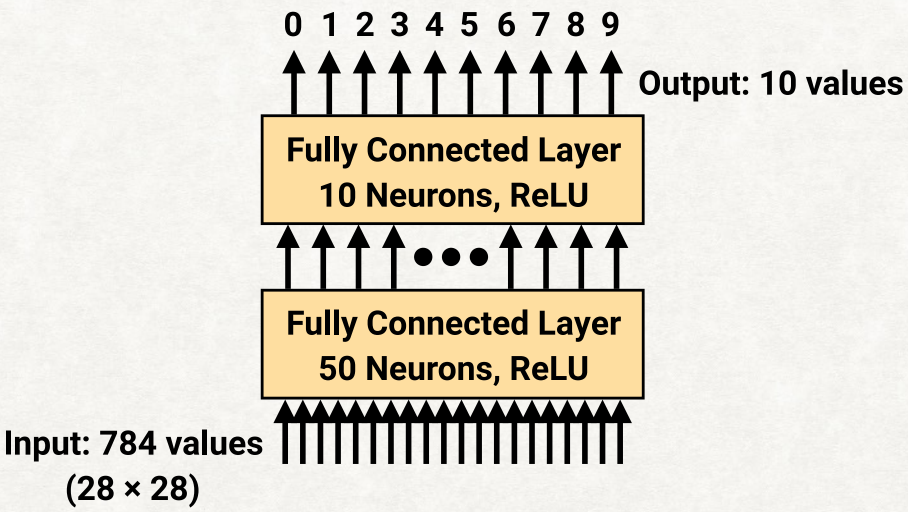


Input: 2D images as 1D arrays. Data shaping is important in DL.



Tensors

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



0 1 2 3 4 5 6 7 8 9

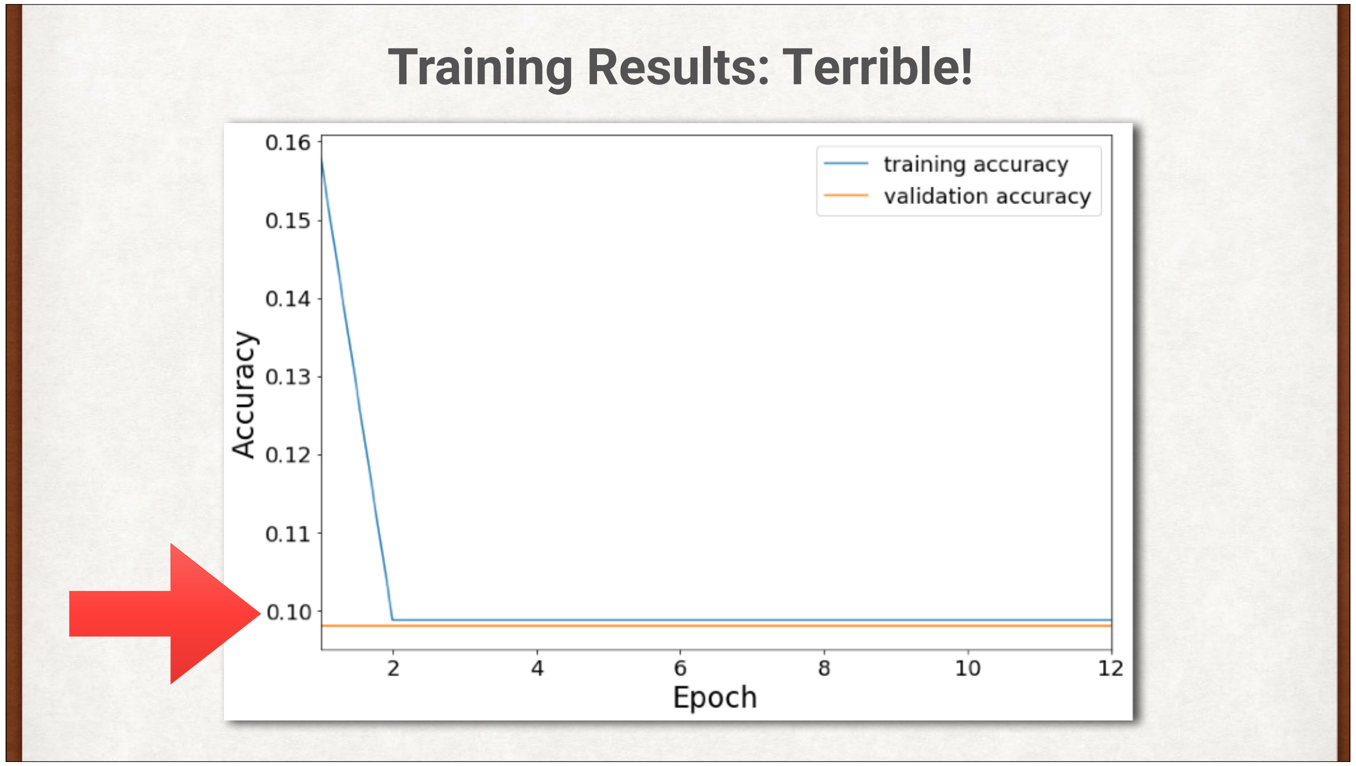
Output: 10 values

Fully Connected Layer
10 Neurons, ReLU

Fully Connected Layer
50 Neurons, ReLU

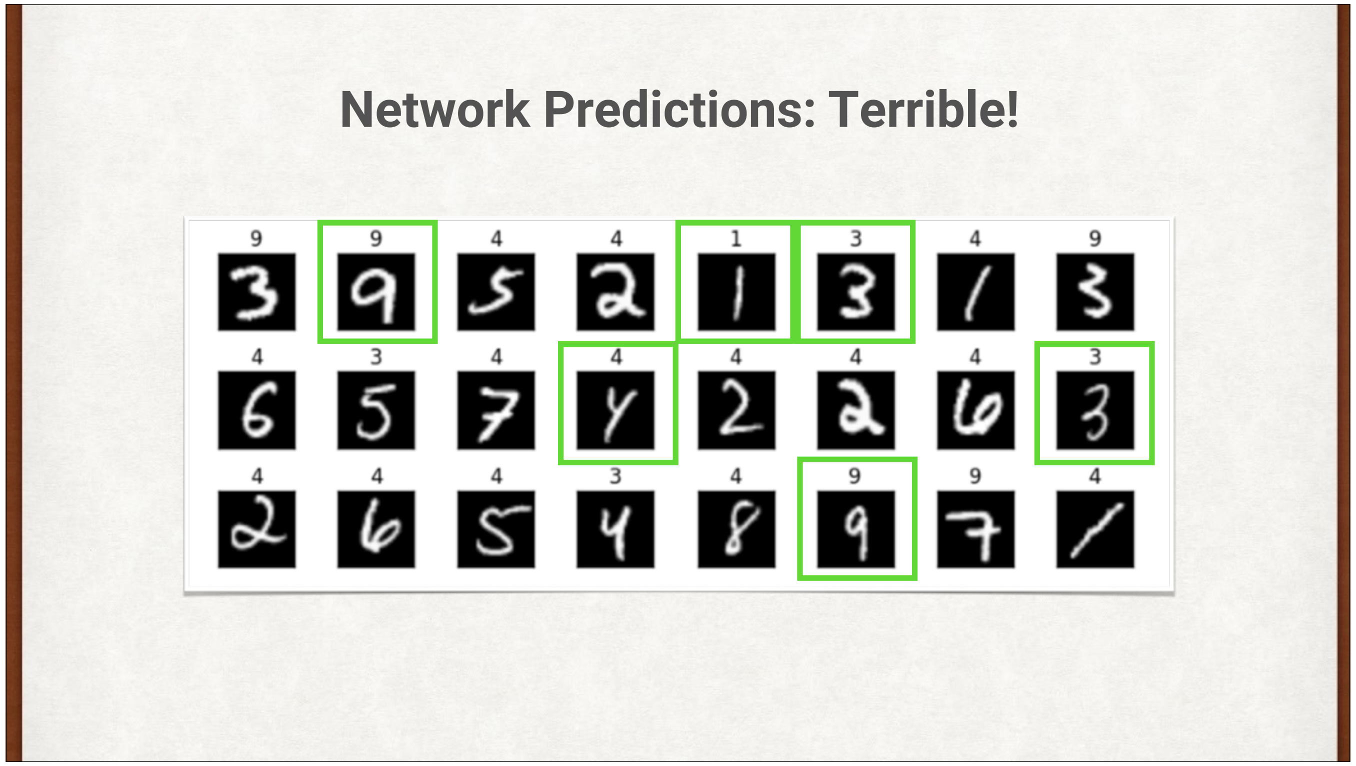
Input: 784 values
(28 × 28)

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



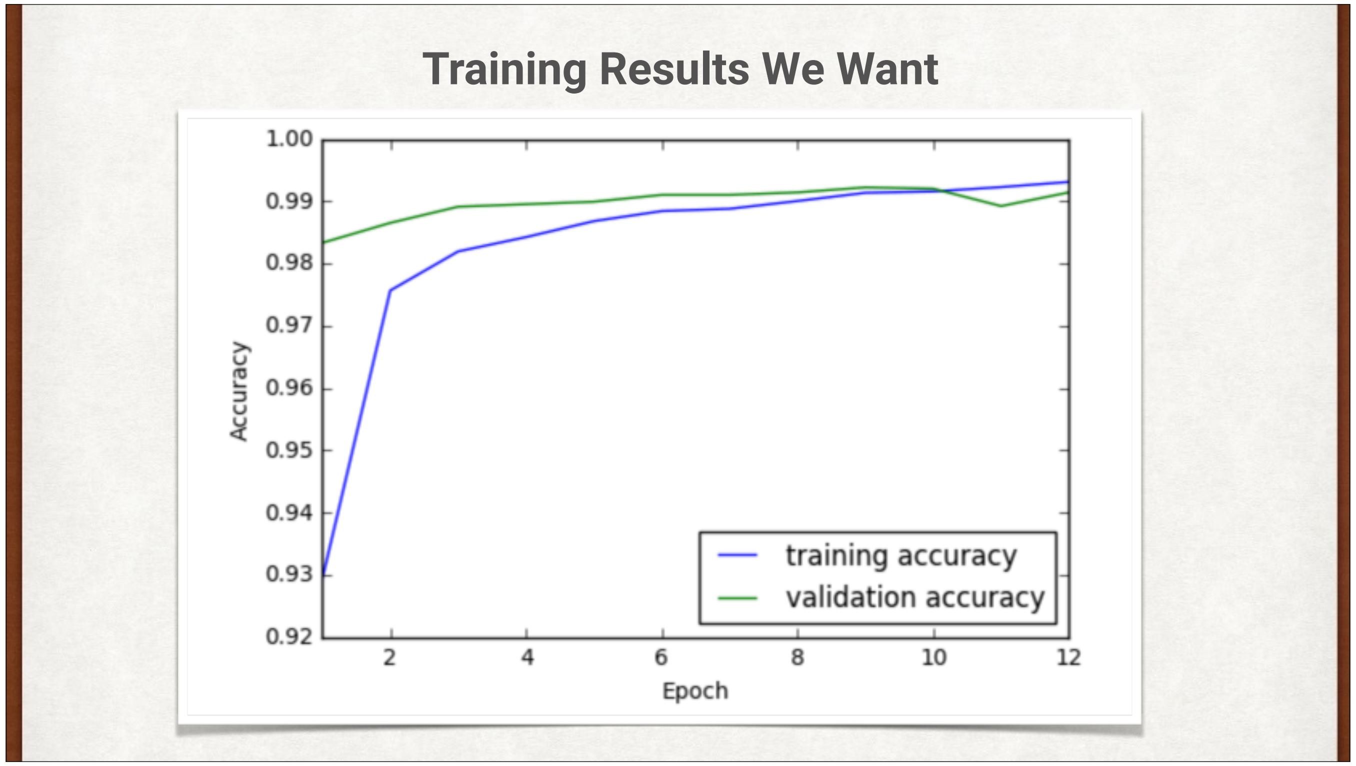
Training Results: Terrible!

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. Ignore the blue line for now.



Network Predictions: Terrible!

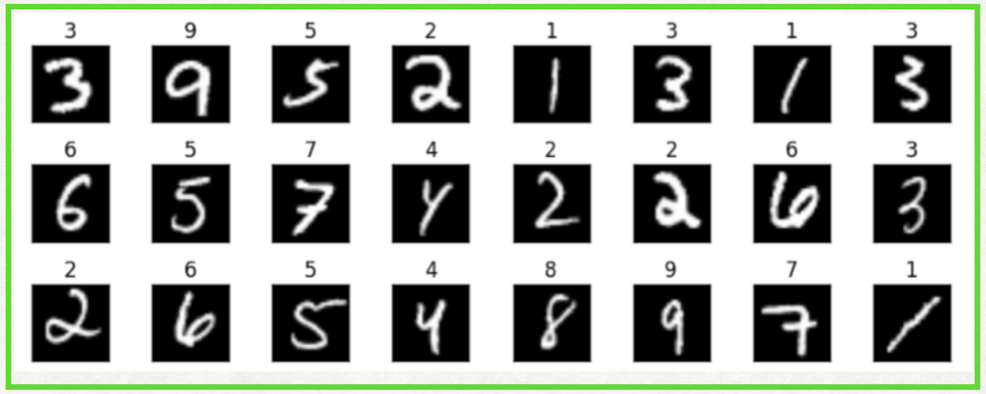
Only the green predictions are correct. Blech.



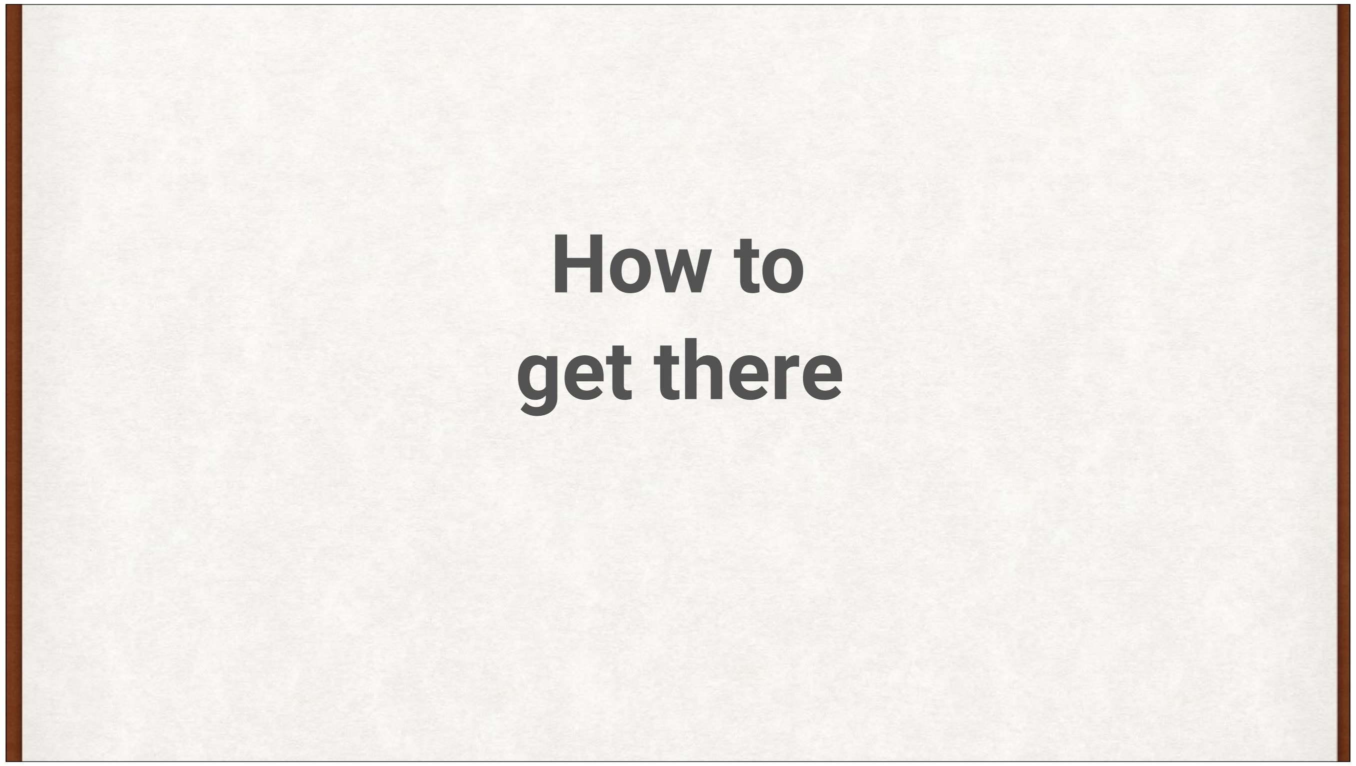
Training Results We Want

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

Network Predictions We Want



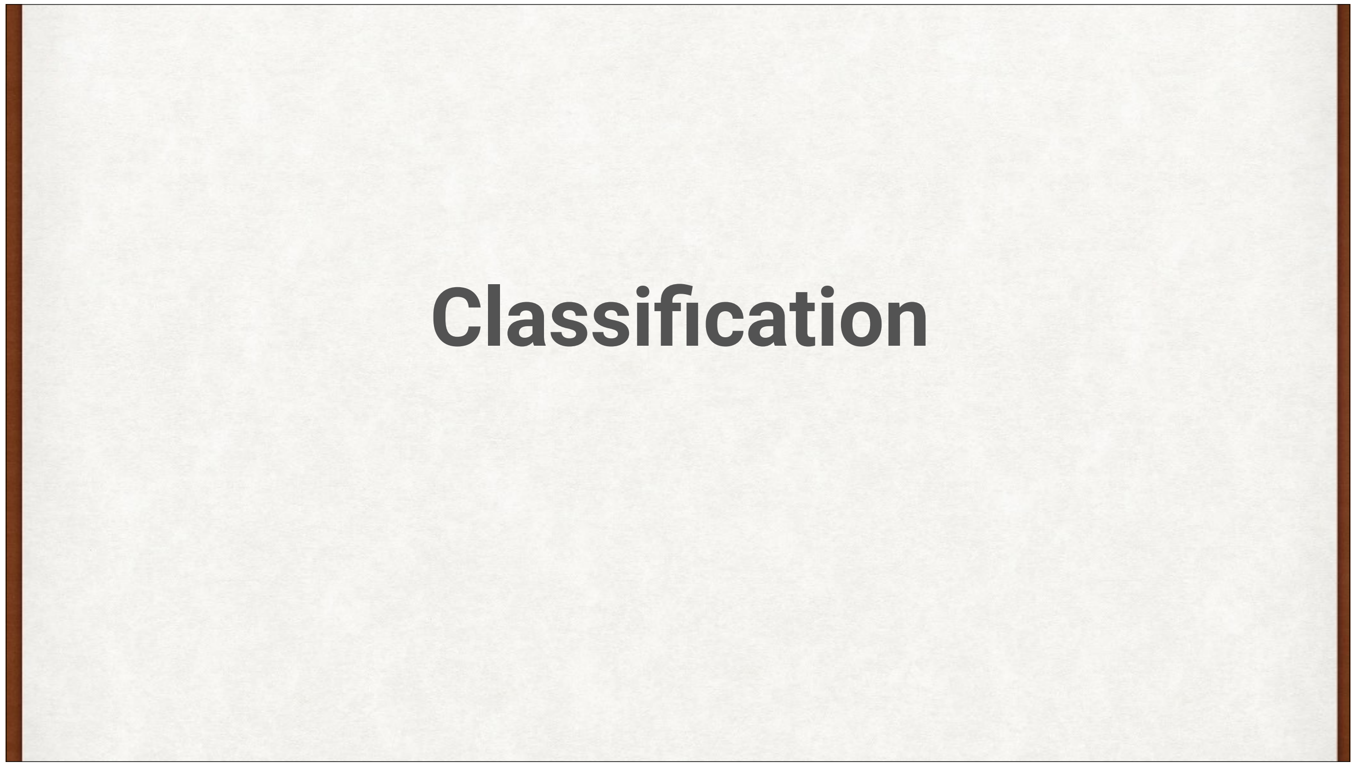
Yeah, this is what we want: correct labels (almost) every time.



How to get there

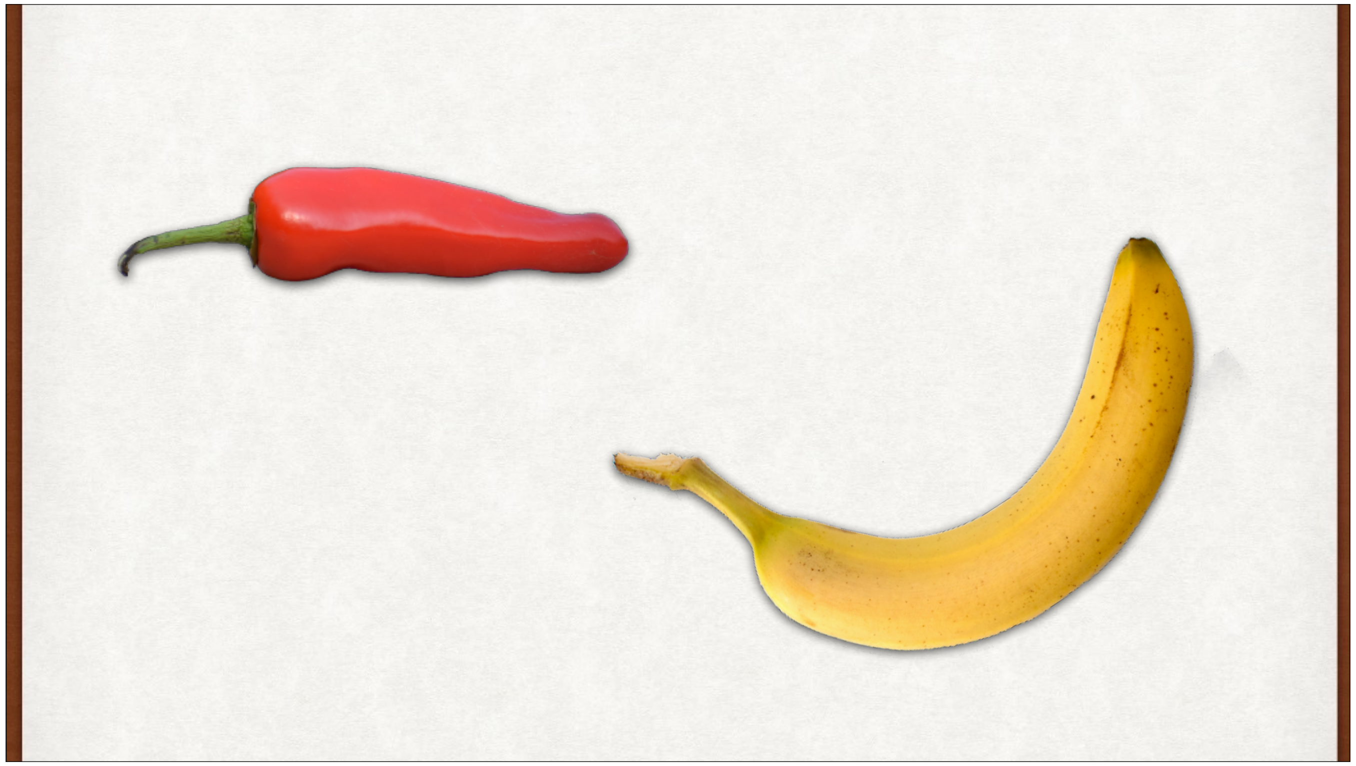
To get good results, we have to actually know something about what we’re doing! So let’s back up a bit.

Visual sherbert: Like the palette cleanser between courses in a big meal, a quick break to catch our breath and change topic.

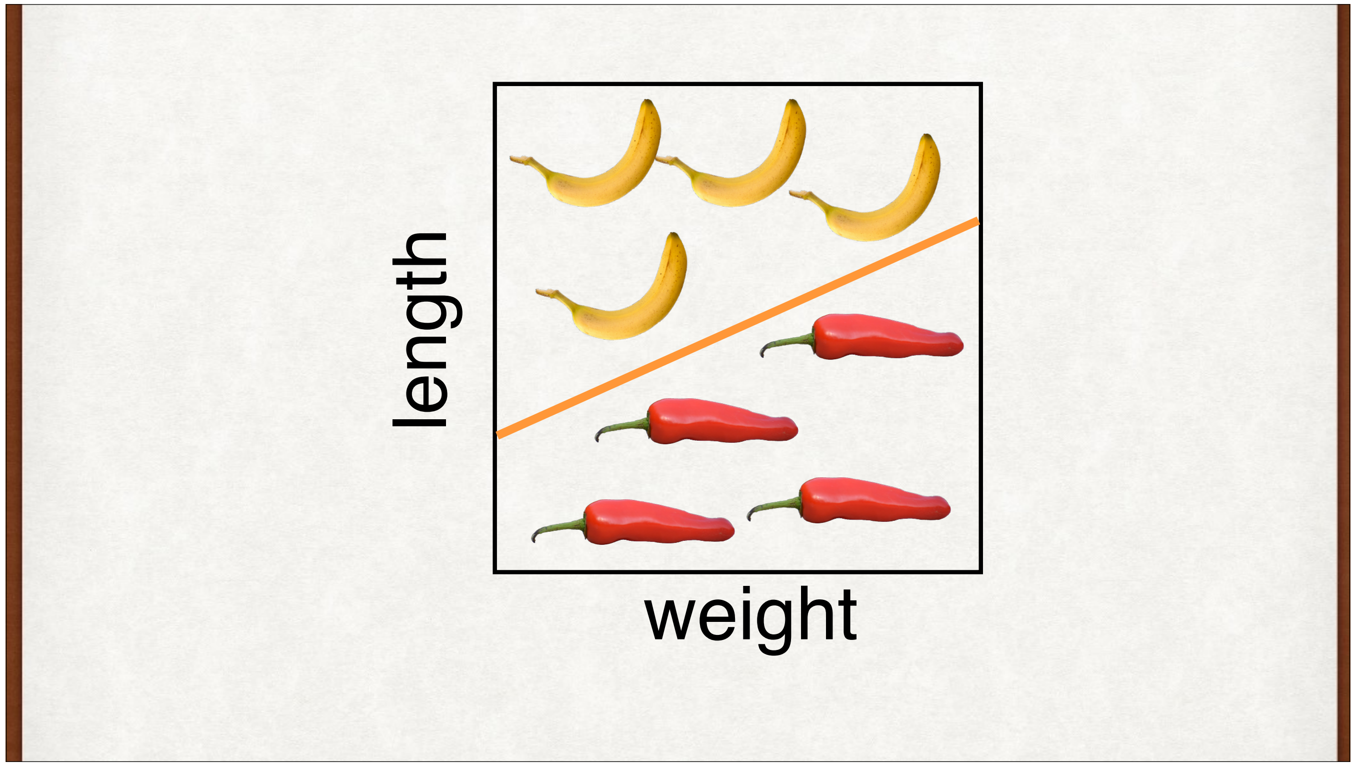


Classification

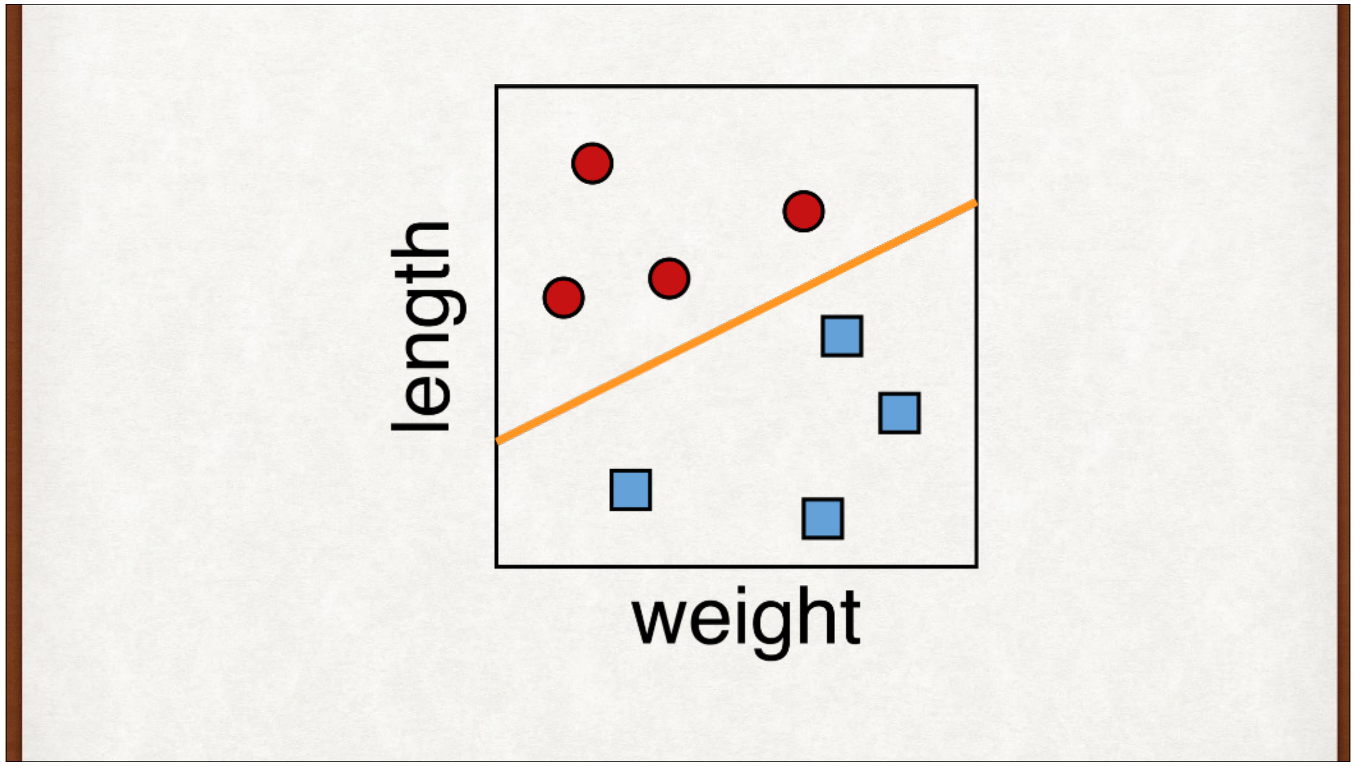
Digit labeling is a problem in “classification,” or assigning a category to each input.



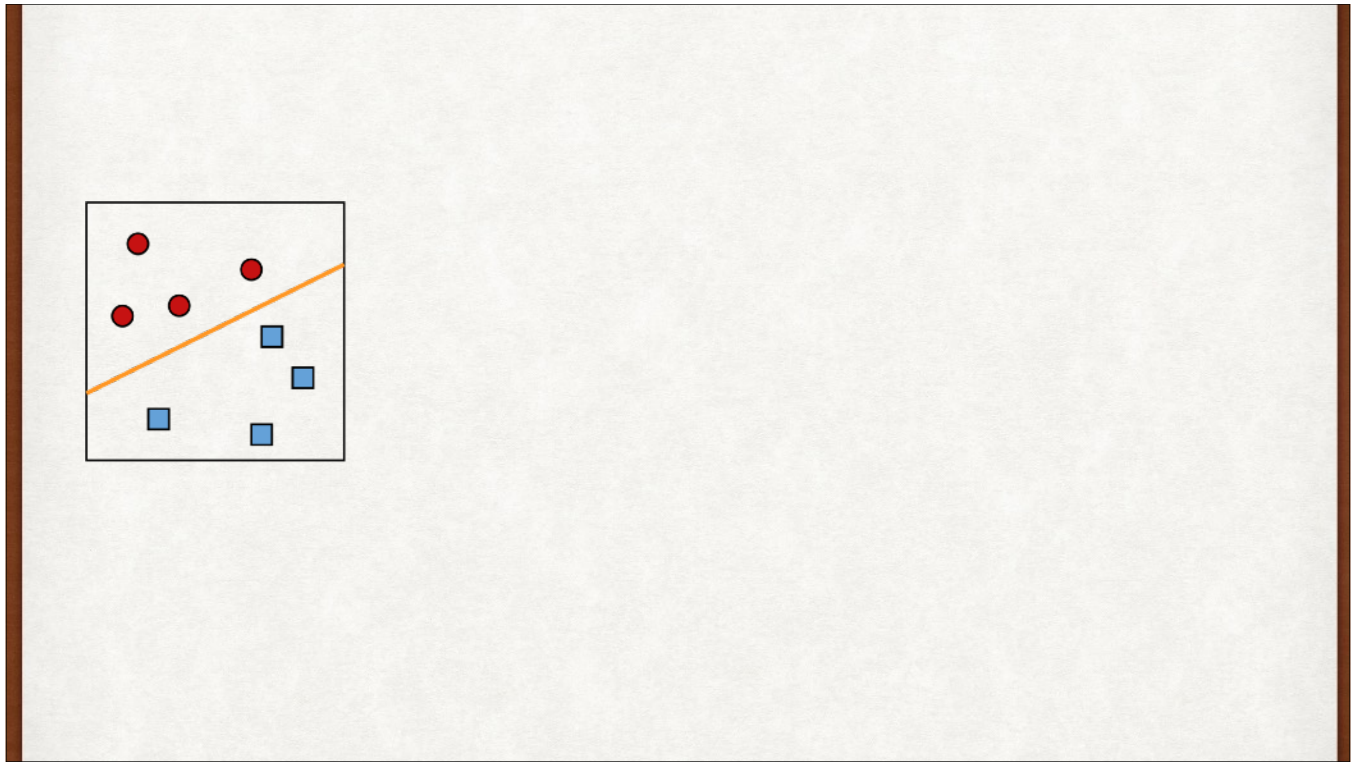
Let’s classify peppers and bananas
https://pixabay.com/en/banana-fruit-yellow-1504956/
https://pixabay.com/en/red-chili-food-pepper-cooking-3909271/



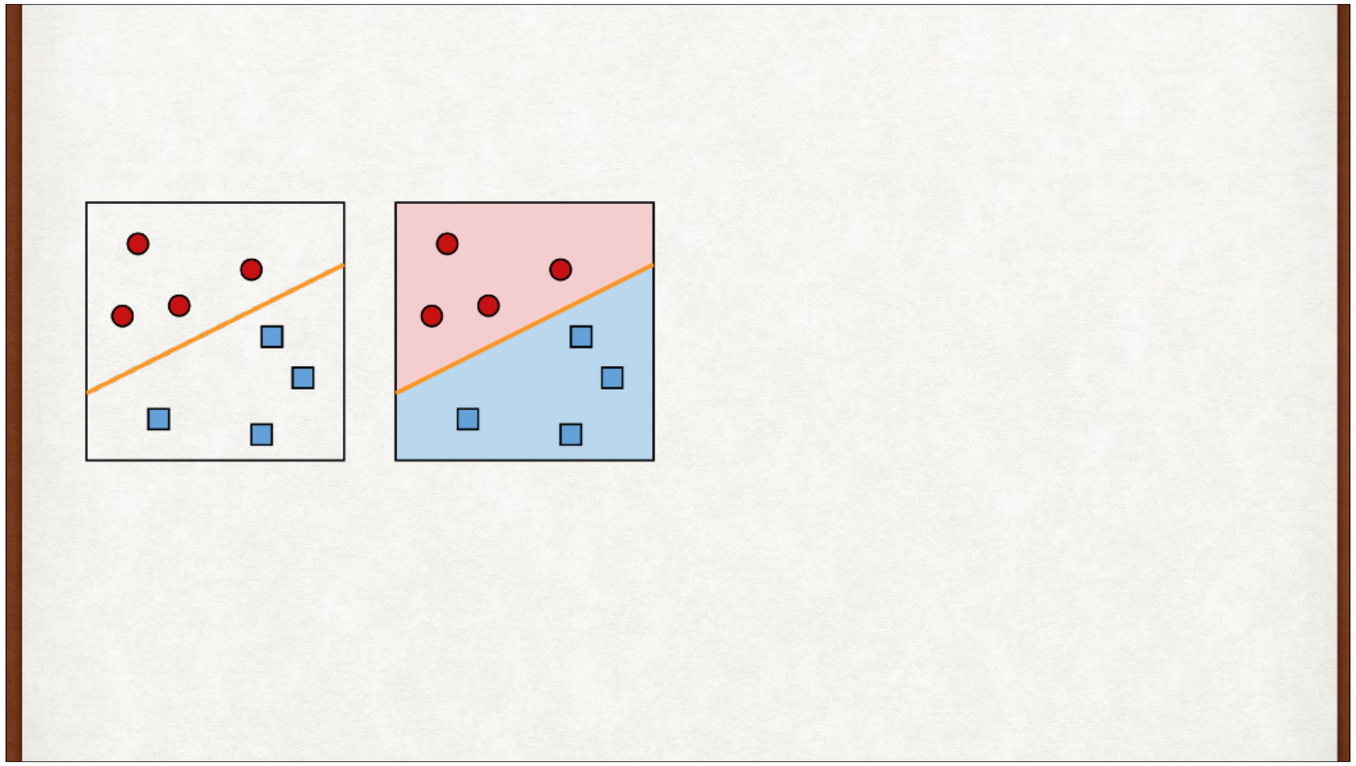
Suppose we're looking at bananas and peppers. For these examples, bananas are longer than the peppers as the weight of either one increases. The orange line splits the groups.



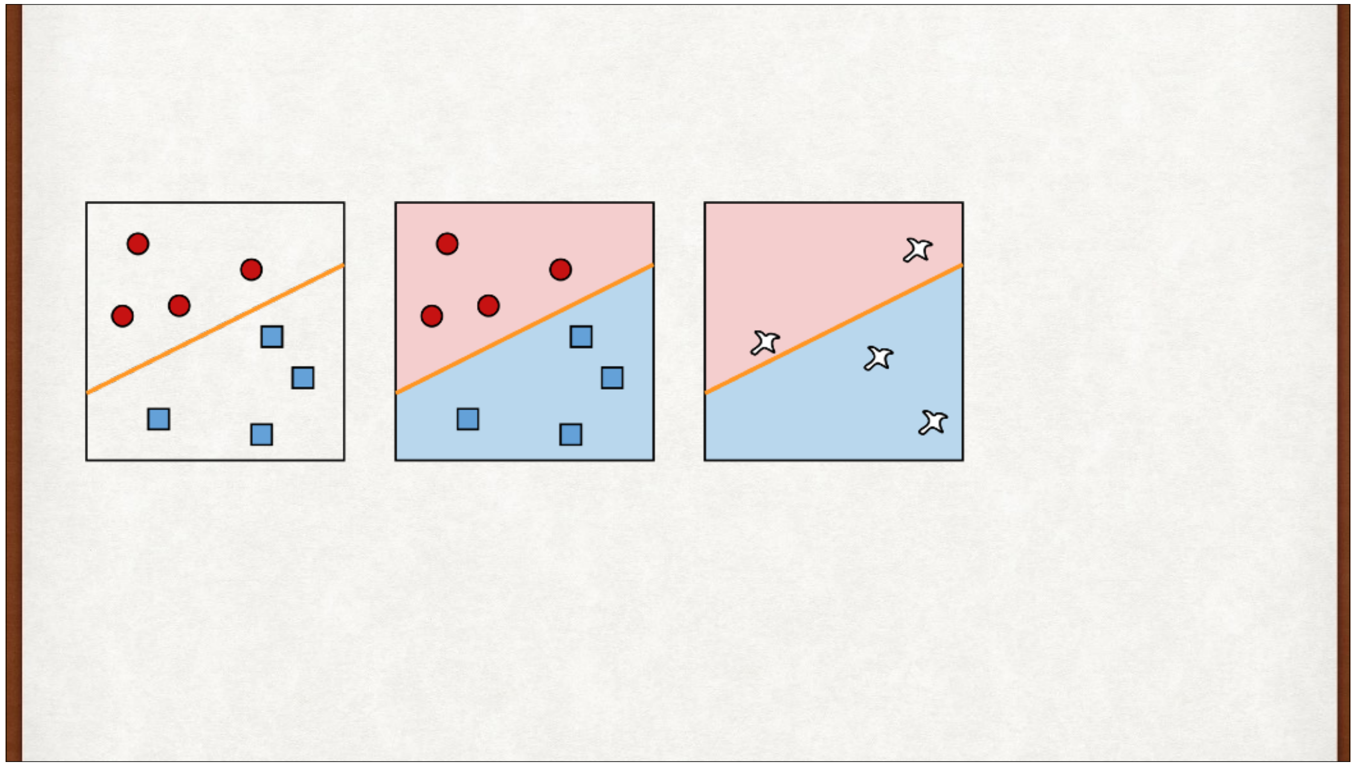
Using abstract shapes instead.



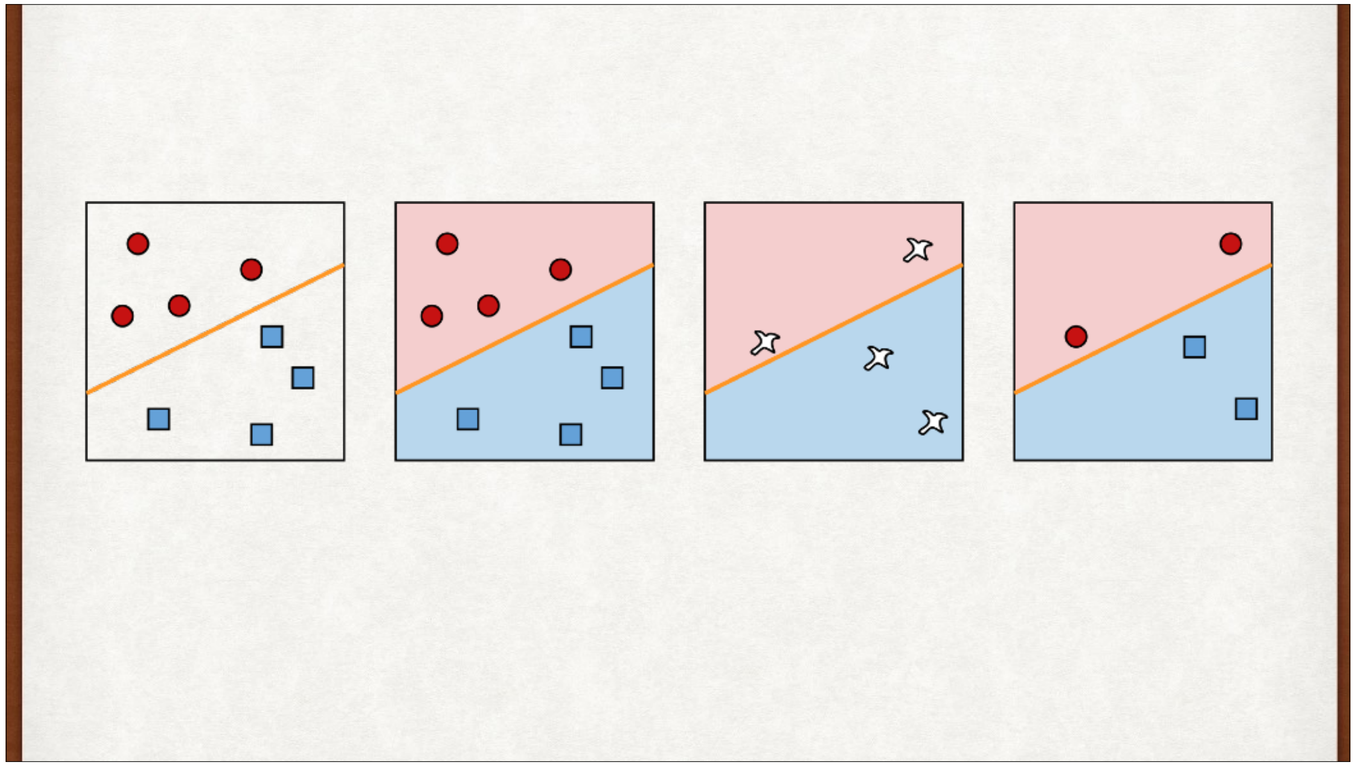
Now when new foods arrive (white splats), we can tell which species they are by seeing which side of the line they fall on.



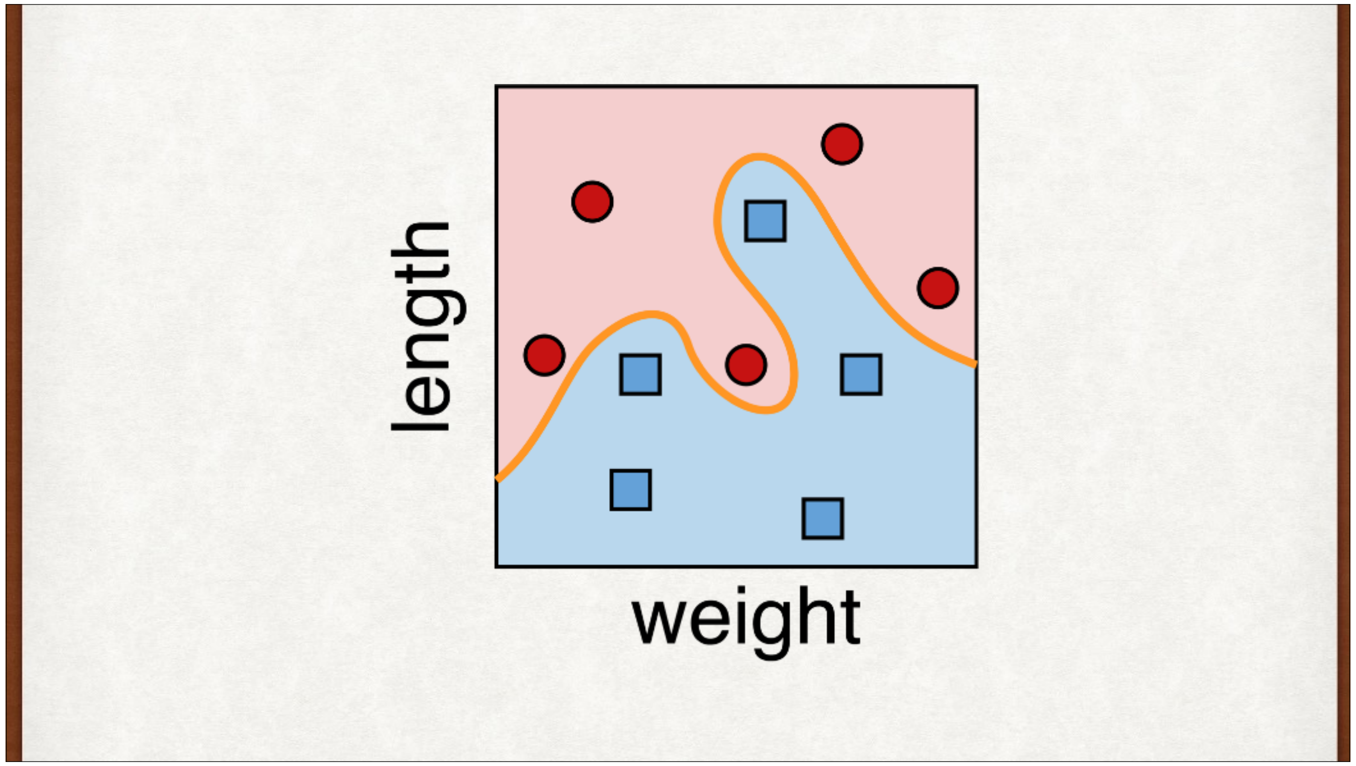
Now when new foods arrive (white splats), we can tell which species they are by seeing which side of the line they fall on.



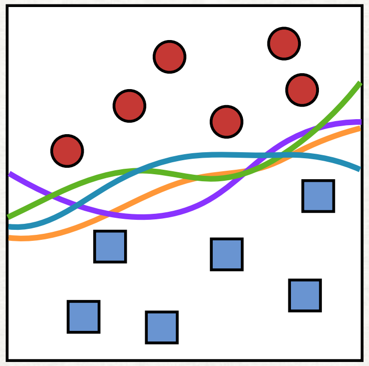
Now when new foods arrive (white splats), we can tell which species they are by seeing which side of the line they fall on.



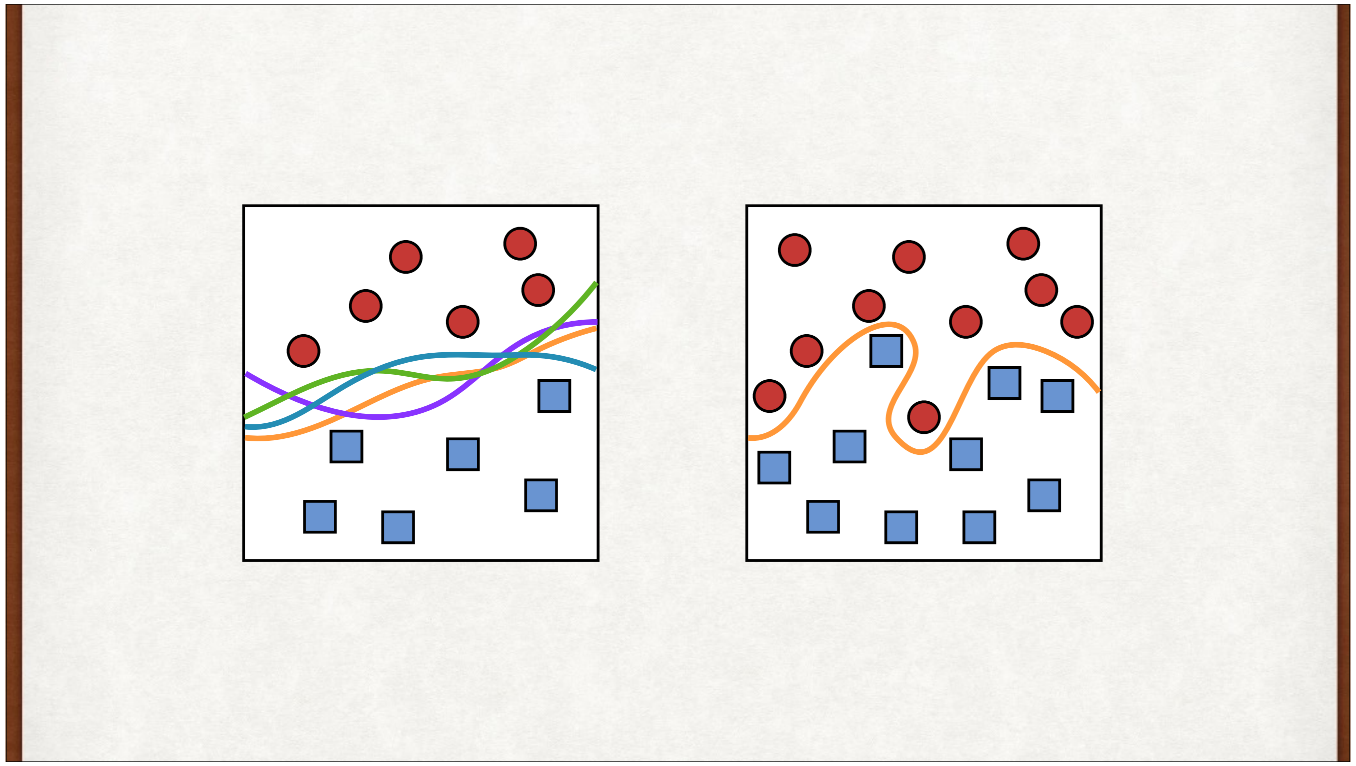
Now when new foods arrive (white splats), we can tell which species they are by seeing which side of the line they fall on. This data is **linearly separable**.



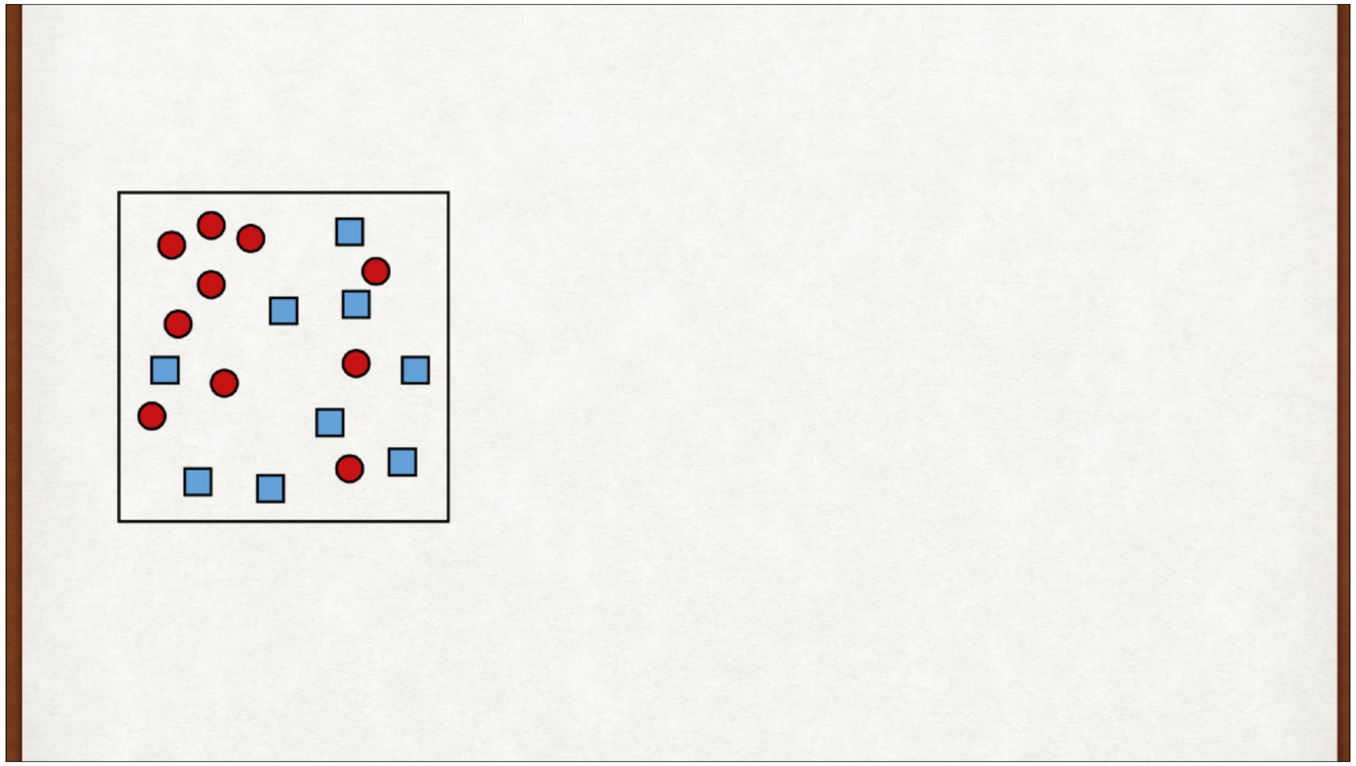
Here’s a couple of different foods. Now a straight line won’t do, but a curve can separate the two groups so we can predict which of the two classes each input belongs to.



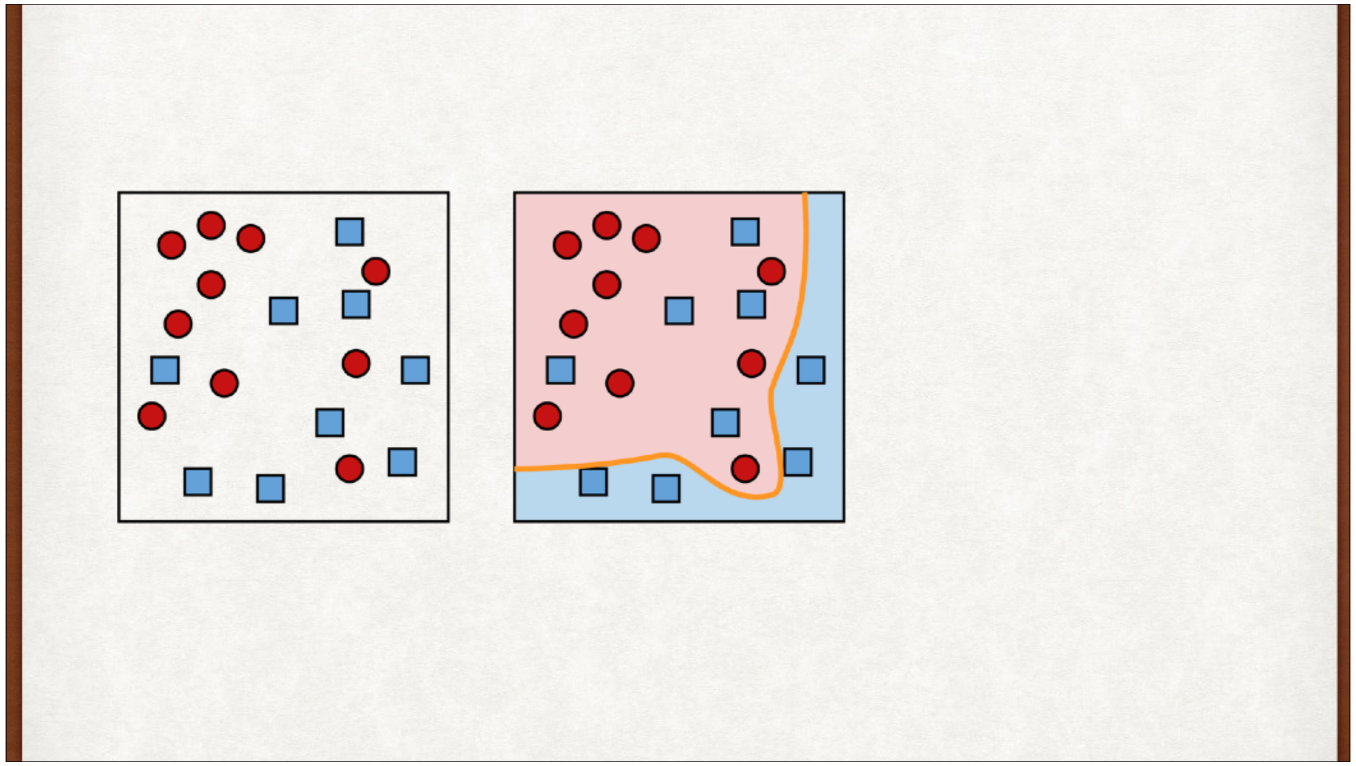
It’s not always clear which line, or curve, is best. On the left, they all split the two groups. On the right, after getting more data, and we can nail down the shape of the curve a little more.



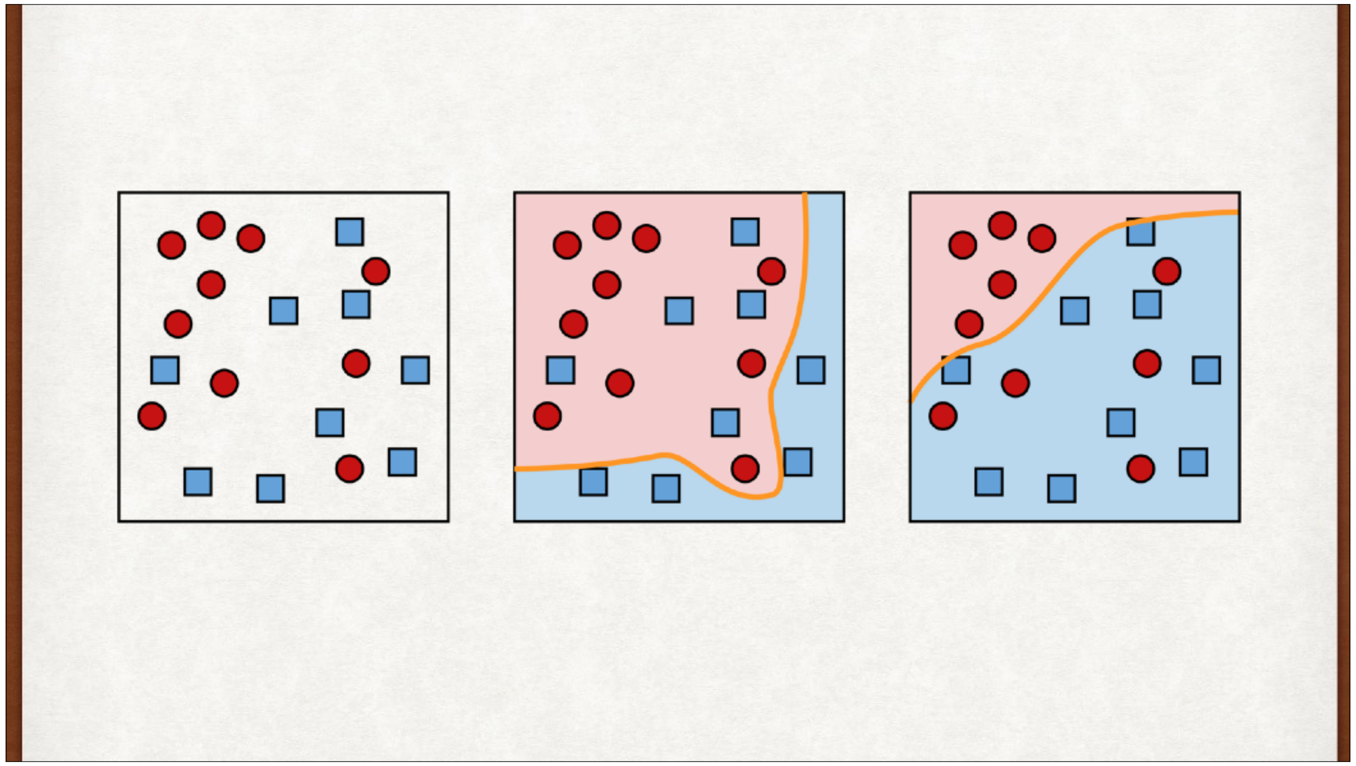
It’s not always clear which line, or curve, is best. On the left, they all split the two groups. On the right, after getting more data, and we can nail down the shape of the curve a little more.



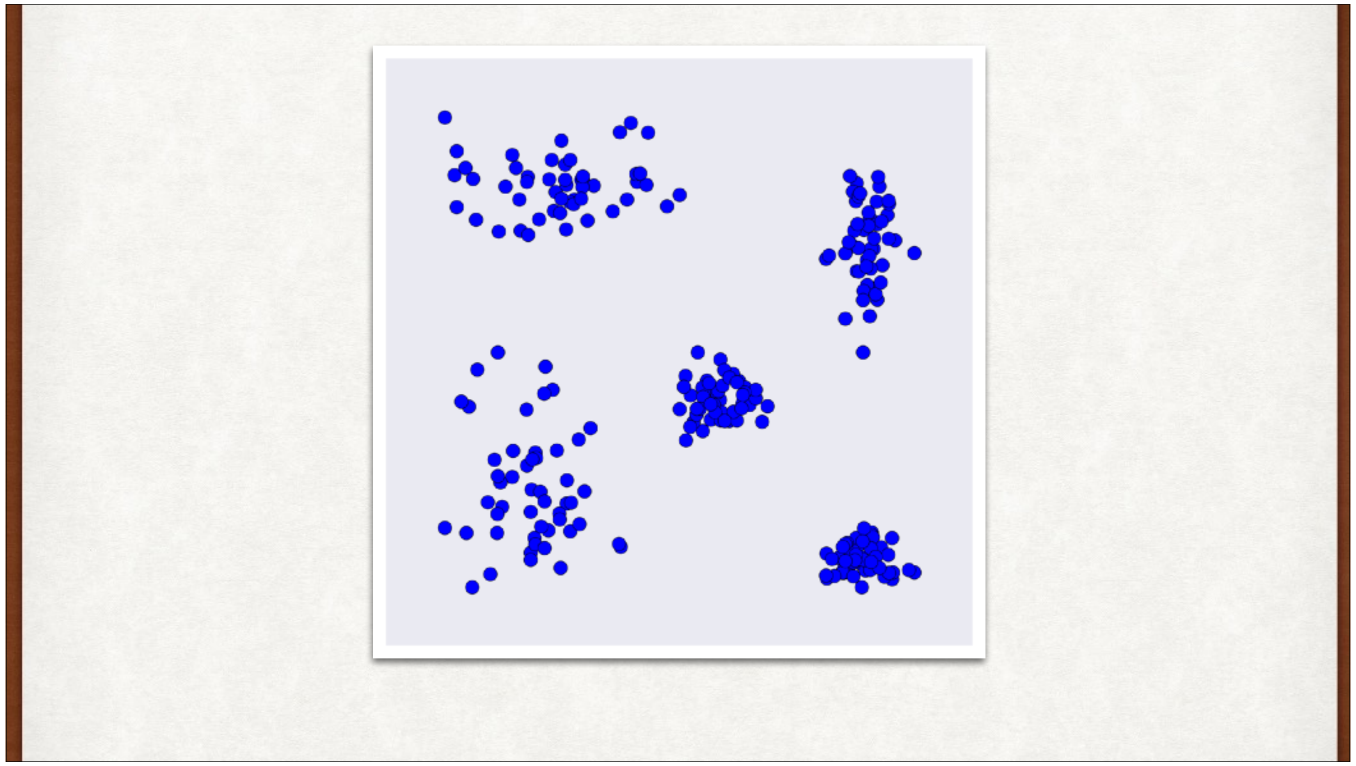
What’s better, a very wiggly but precise curve (middle), or a smooth and imprecise one (right)? Which is more likely to do better on future data? It depends on the data and what we want from our system. It’s a policy decision.



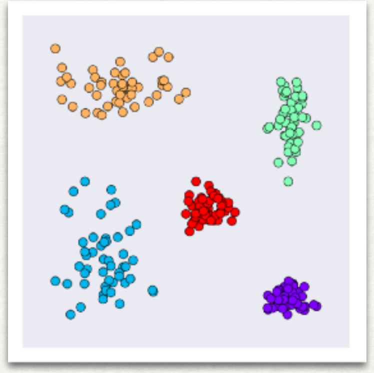
We want to make sure all red circles are correctly classified, even if we make mistakes with some blue ones.



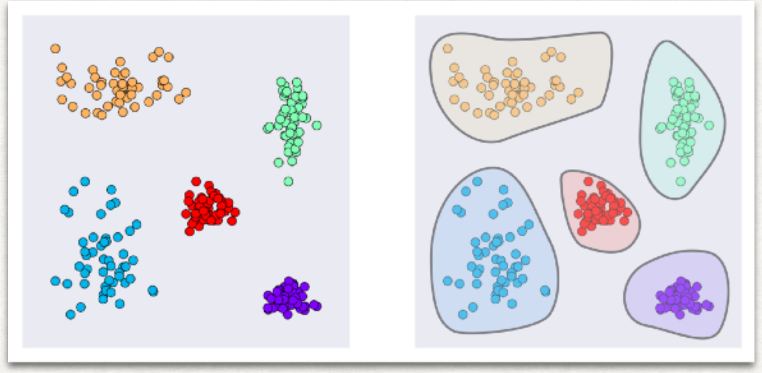
Far right, we want all blue squares correctly classified, even if it means some red dogs are wrong. It’s a policy decision.



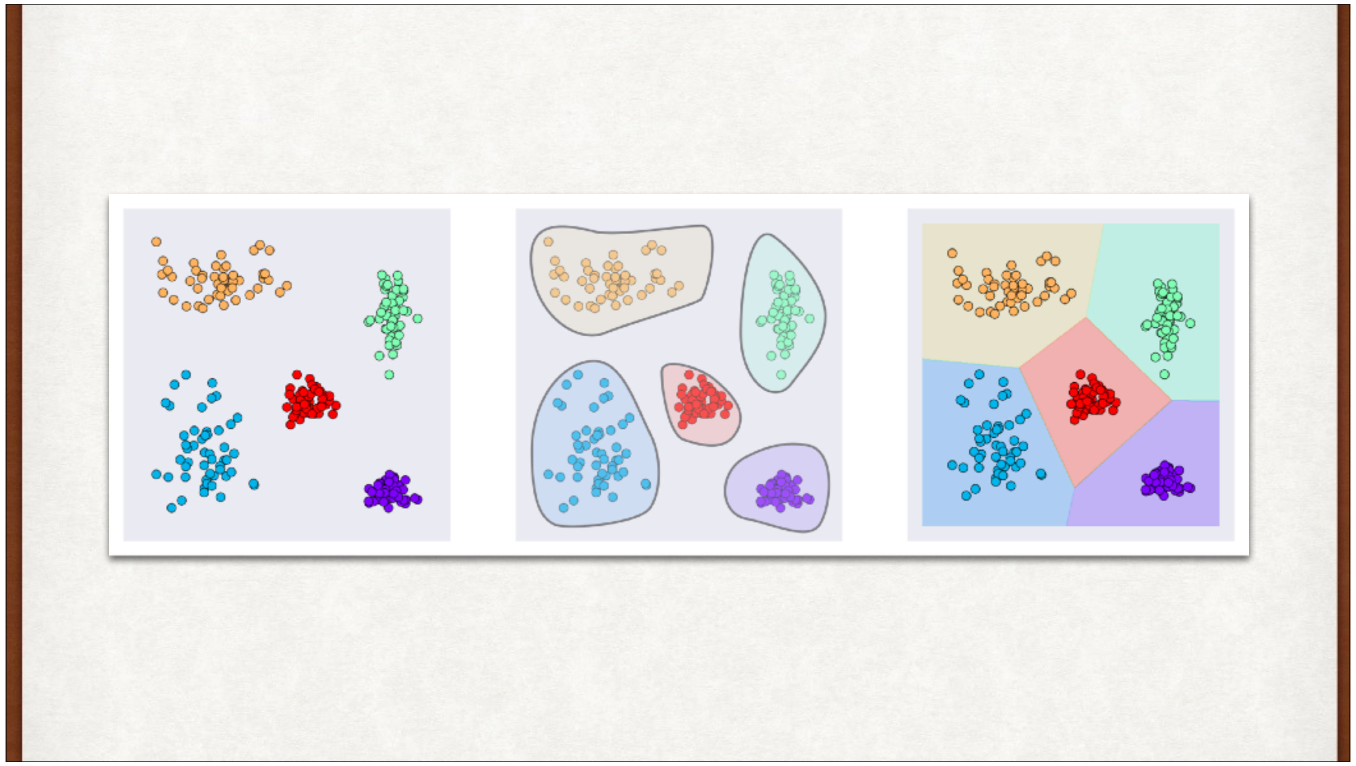
A different approach. Maybe we can just cluster things by finding natural boundaries.



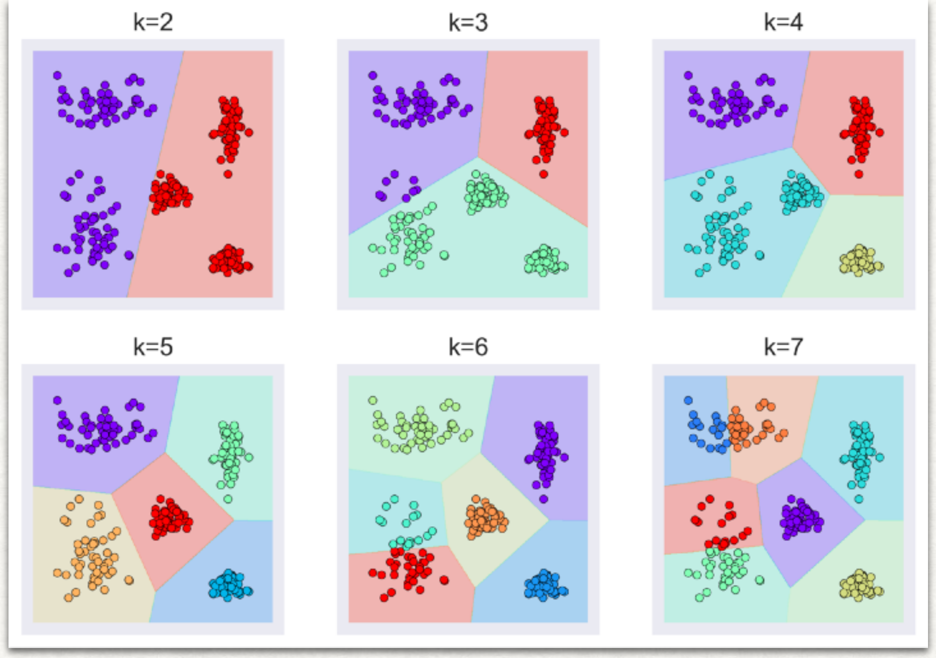
This algorithm, called k-means clustering, seems to be doing a good job. But we have to tell it how many clusters to use.



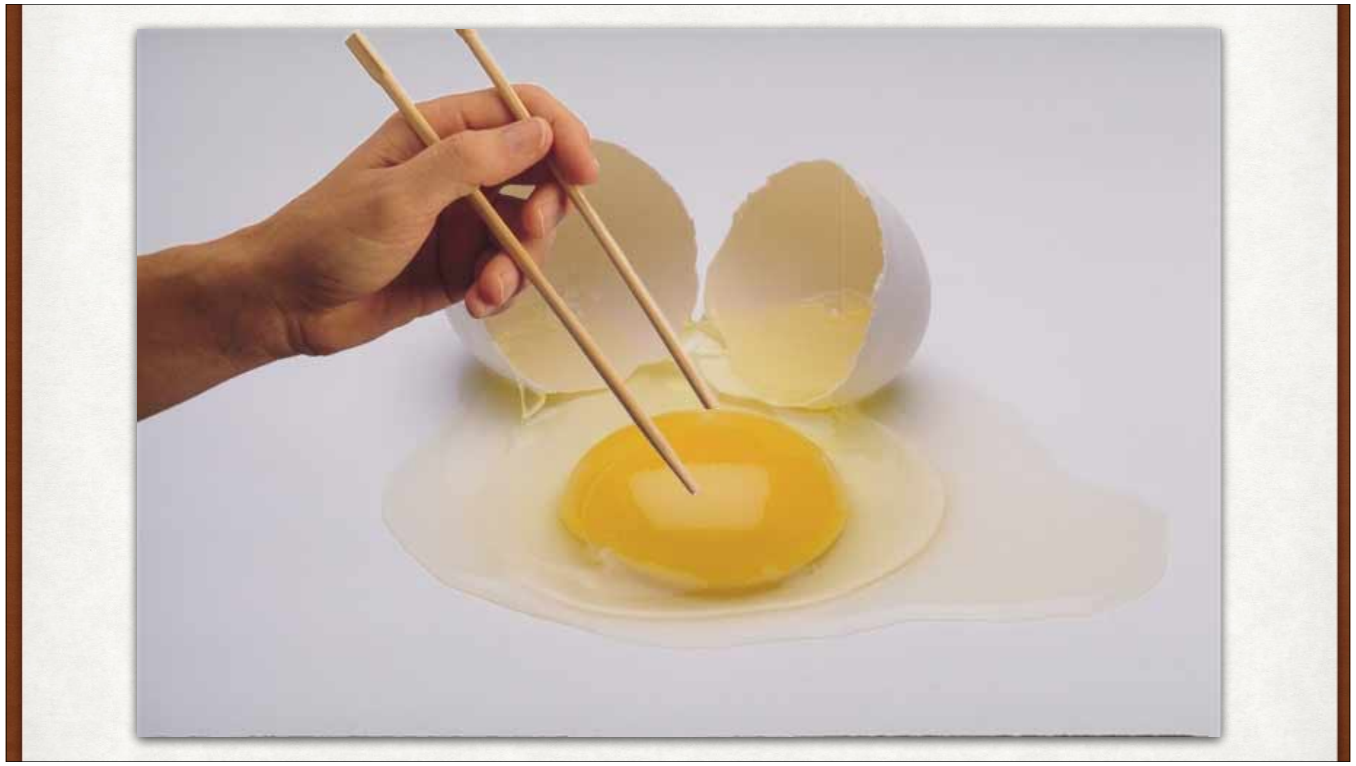
This algorithm, called k-means clustering, seems to be doing a good job. But we have to tell it how many clusters to use.



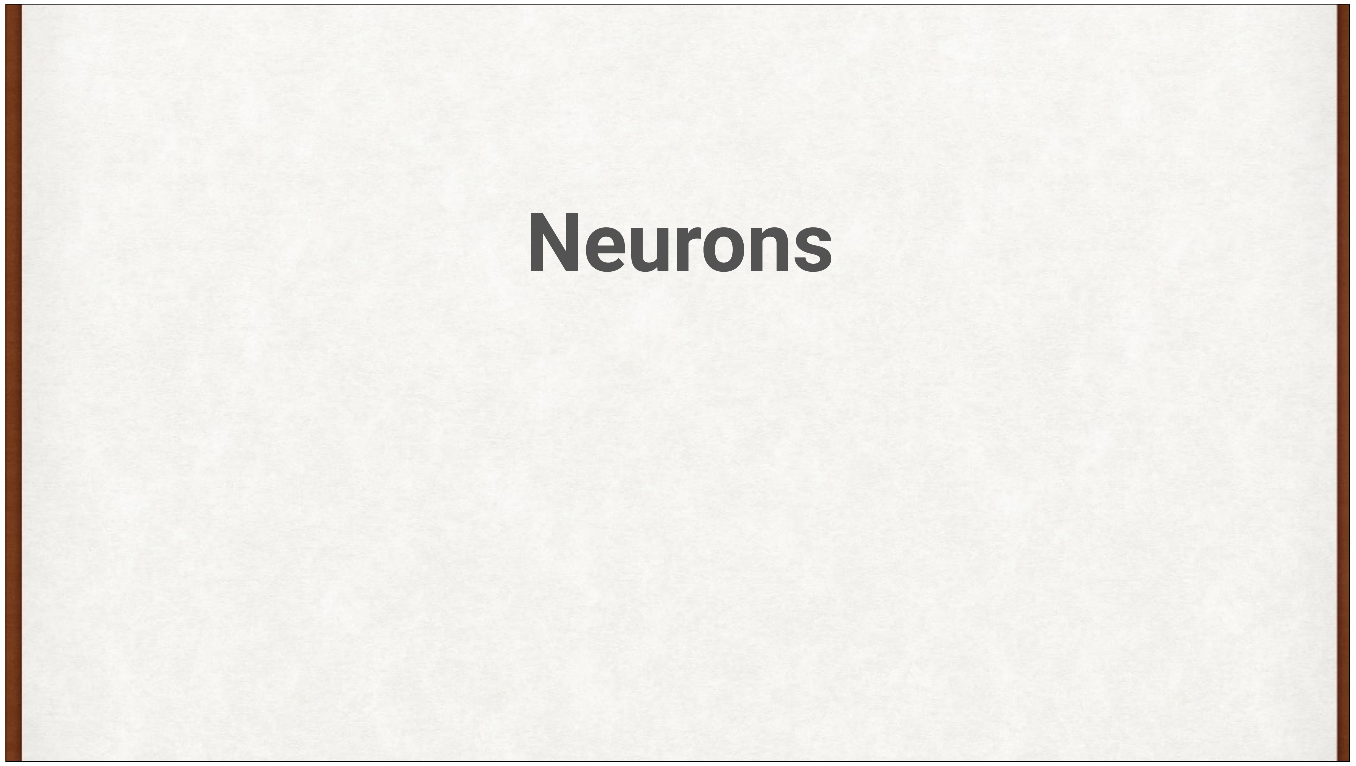
This algorithm, called k-means clustering, seems to be doing a good job. But we have to tell it how many clusters to use.



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

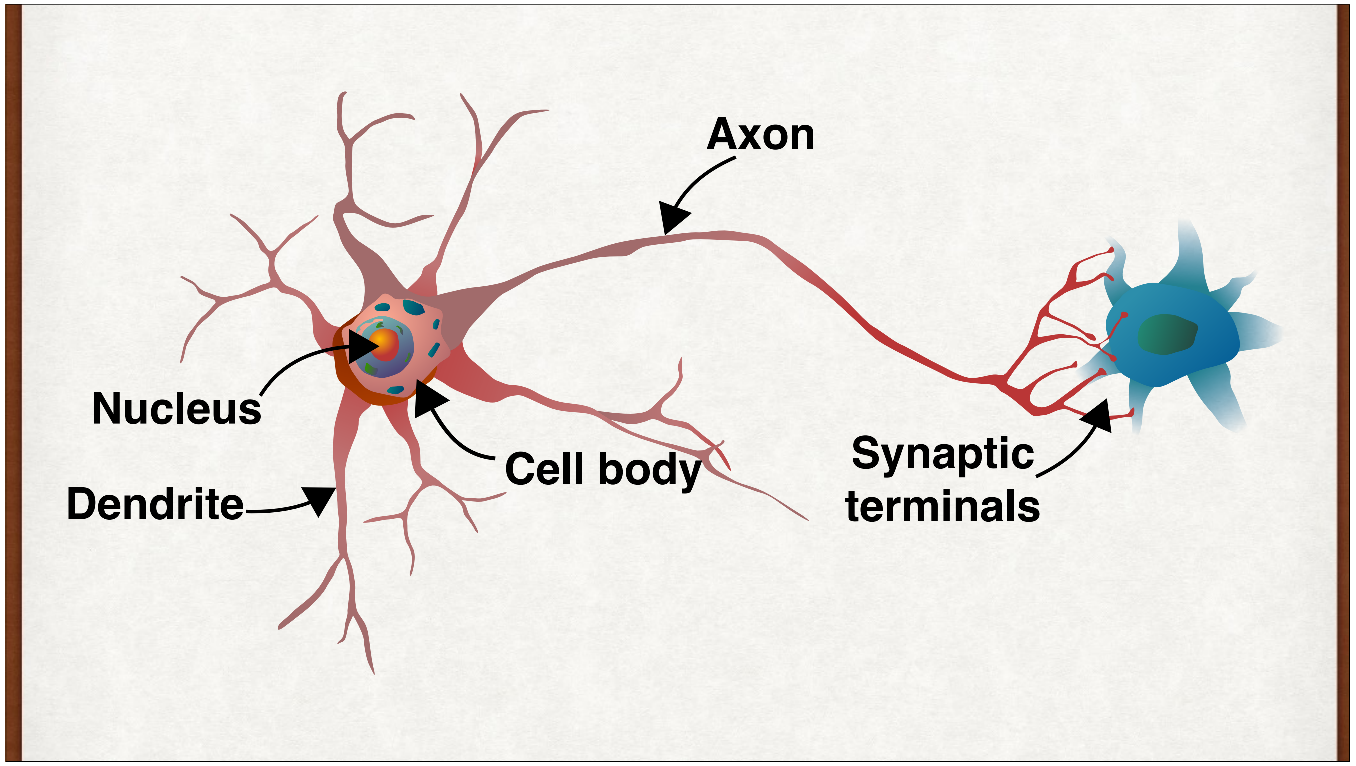


Take a breath. Now for something completely different.

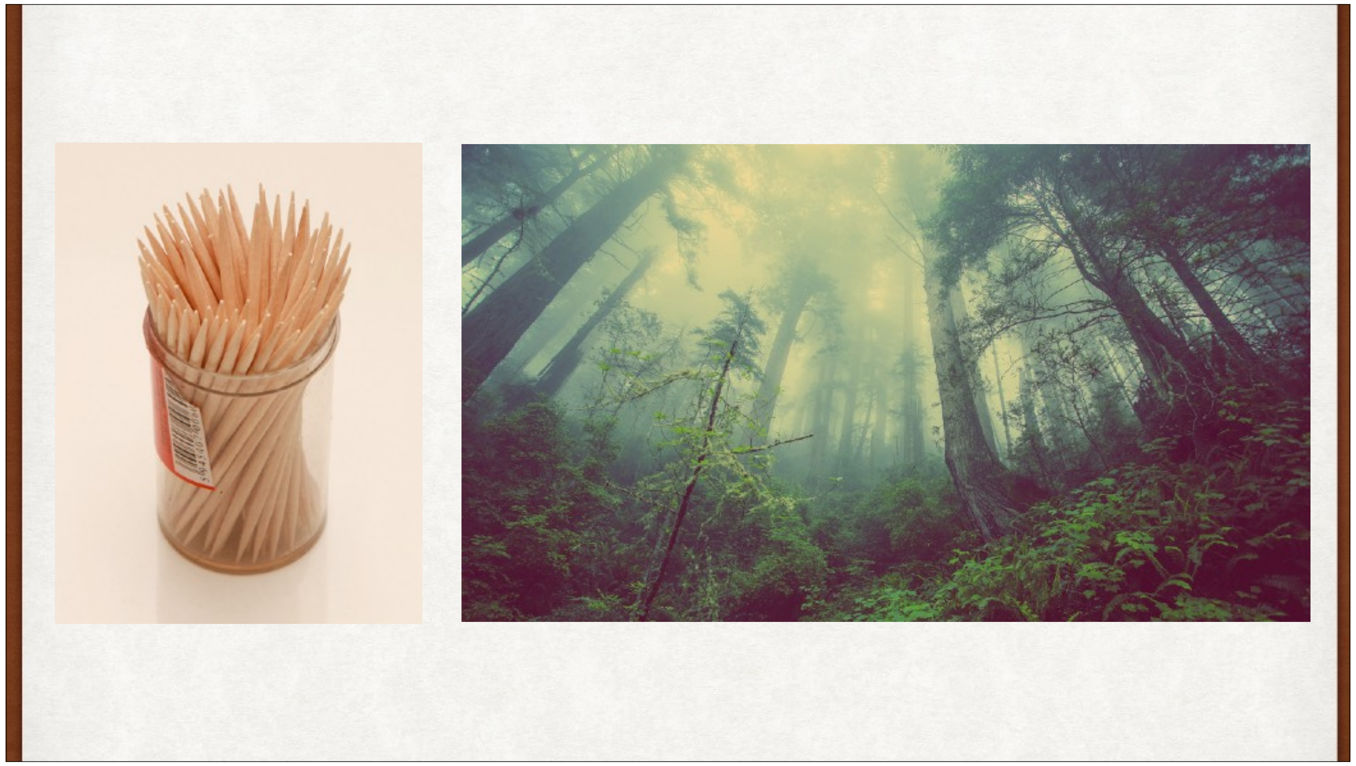


Neurons

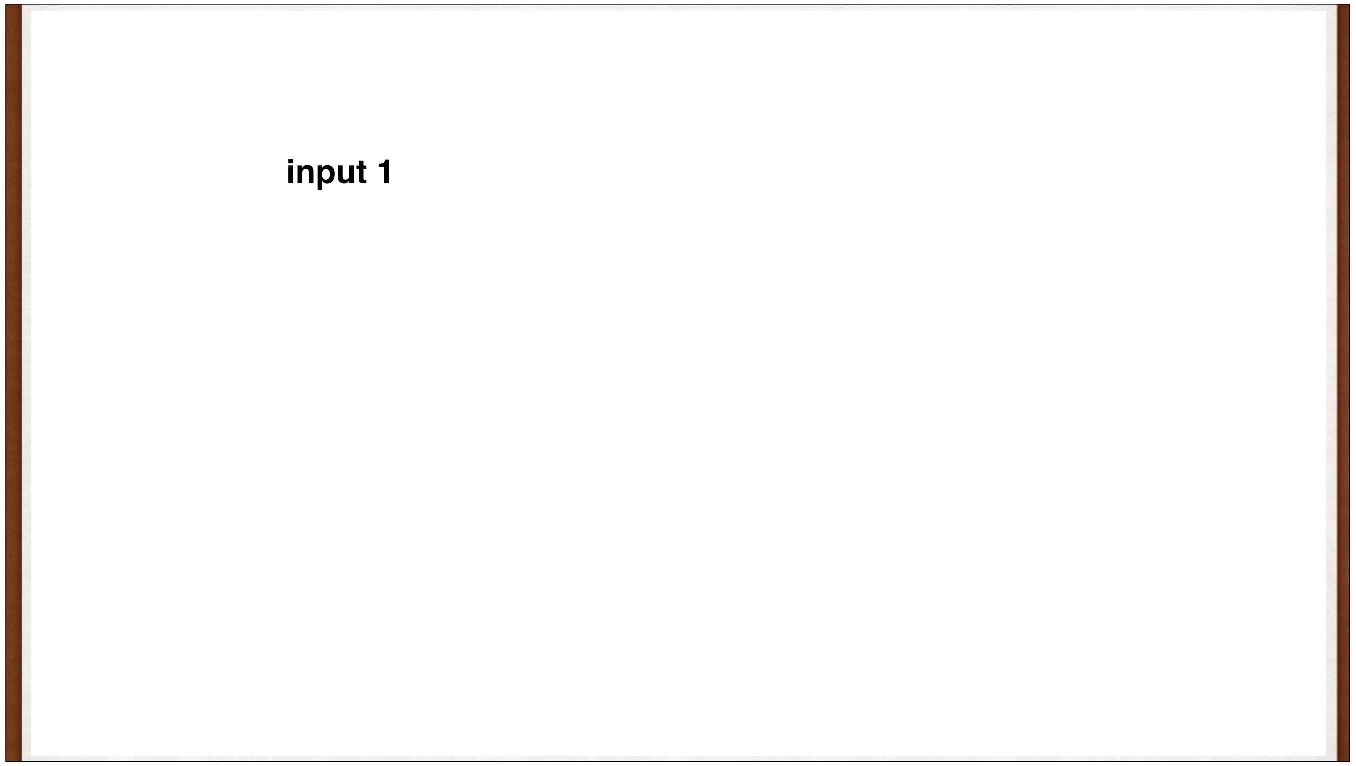
To find a good automatic classifier, we’ll start over. We’ll begin with artificial neurons.



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.

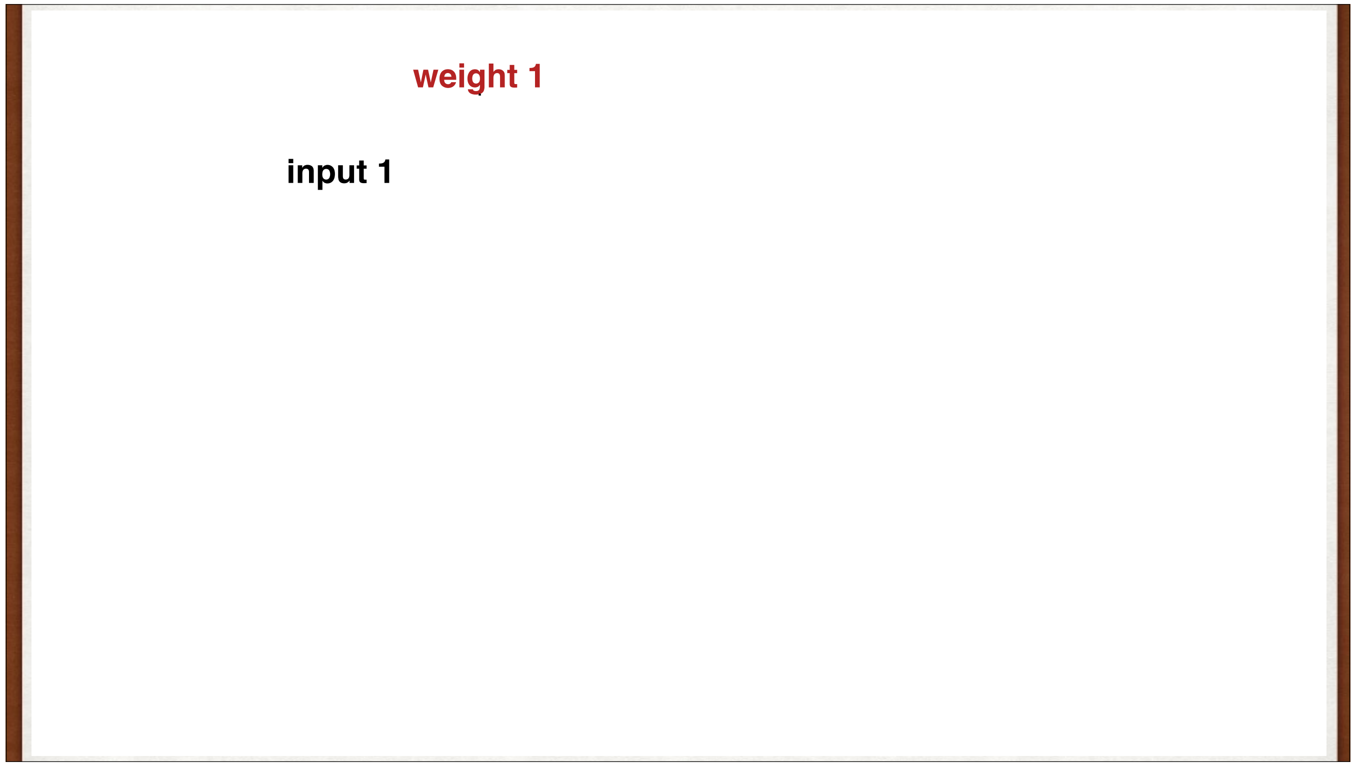


We’re going to create a “neuron" that is a model of a real neuron the way these toothpicks are a model of a forest.
https://pixabay.com/photos/peaks-set-sharp-sticks-toothpicks-88038/
https://pixabay.com/photos/forest-mist-nature-trees-mystic-931706/



input 1

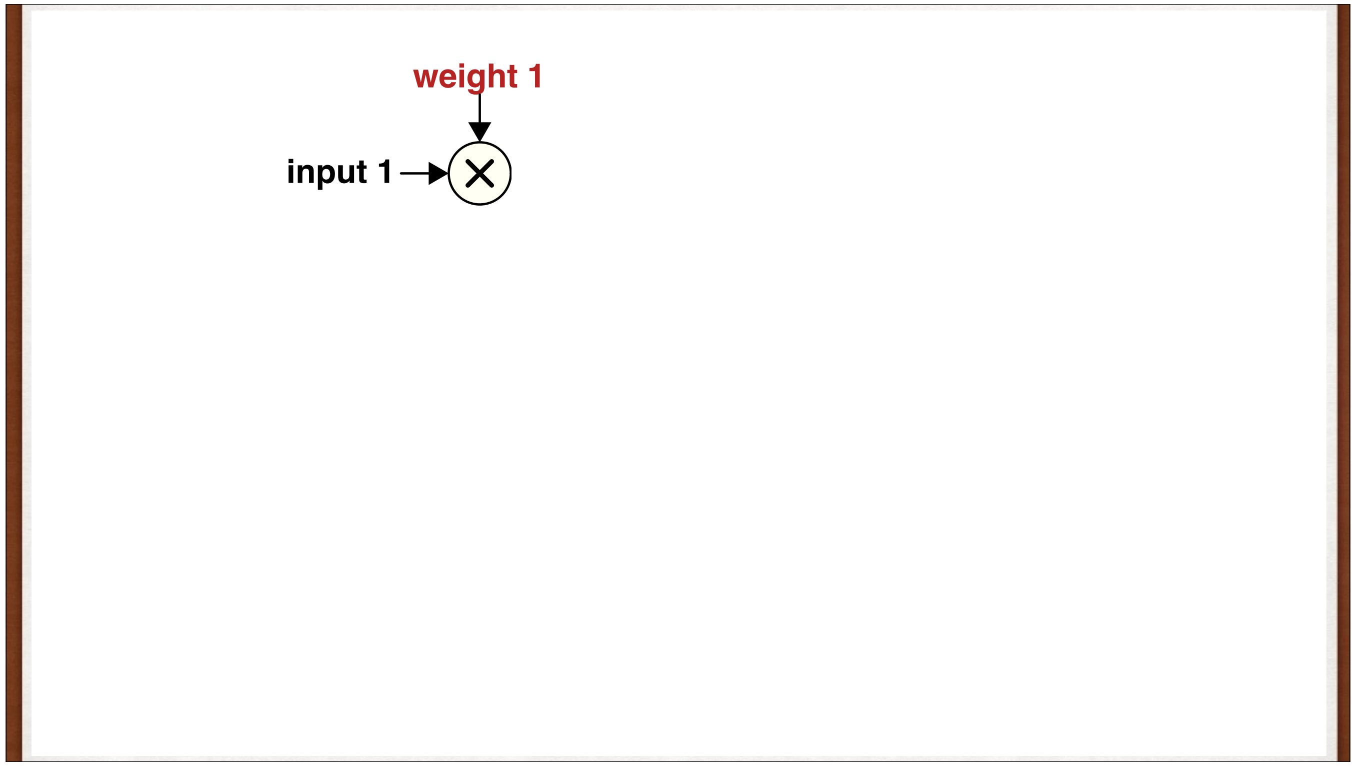
Our “artificial neuron”, or **perceptron**: inputs (numbers) come in. Each is “weighted” by another number. We add those all up, and threshold the result. The output is then -1 or 1 (some people use 0 and 1).



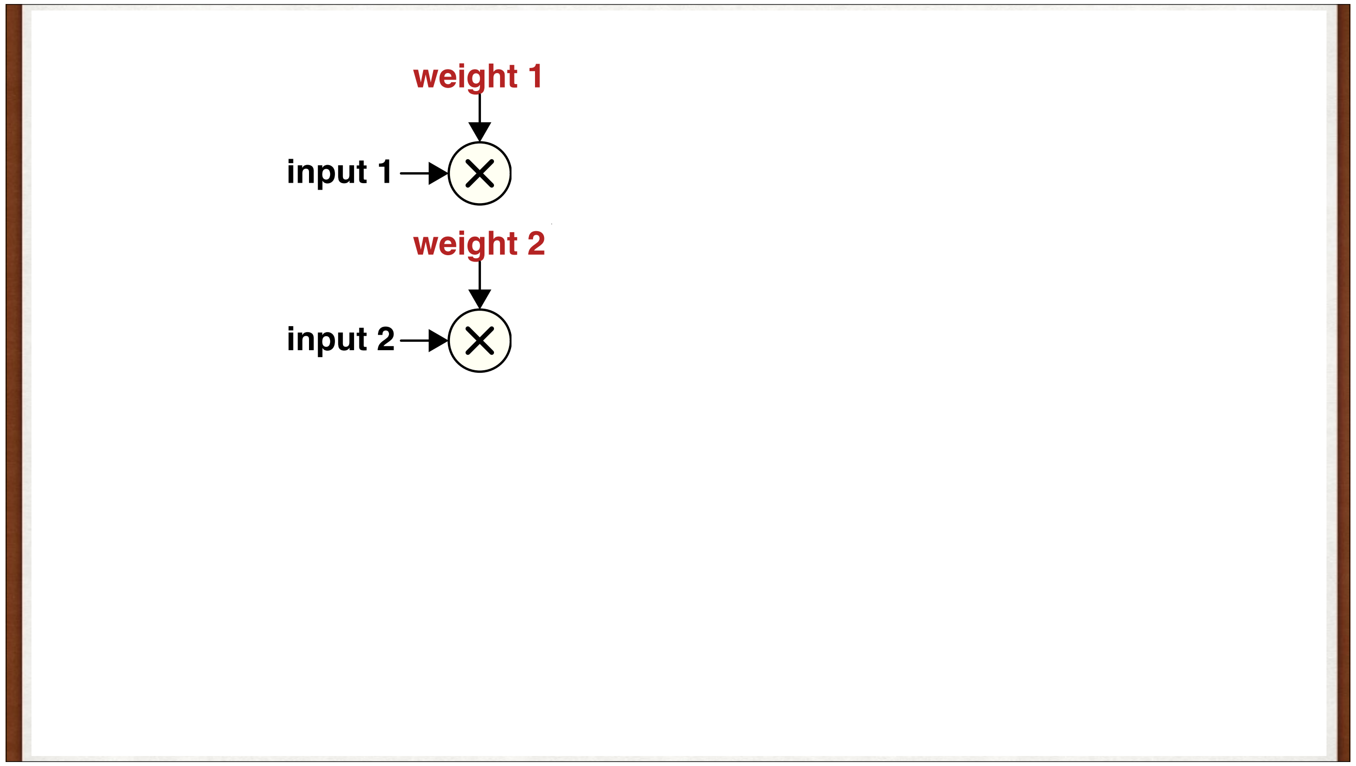
weight 1

input 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).



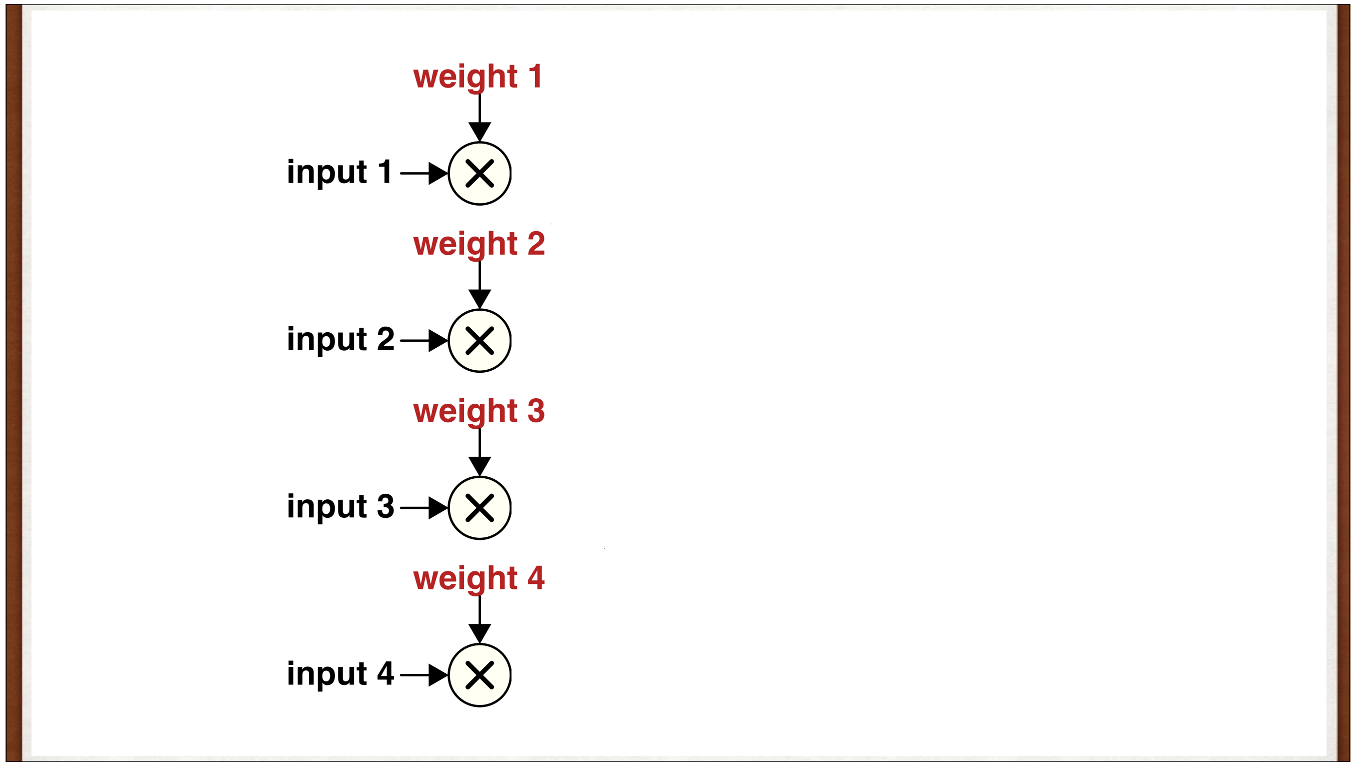
weight 1

input 1

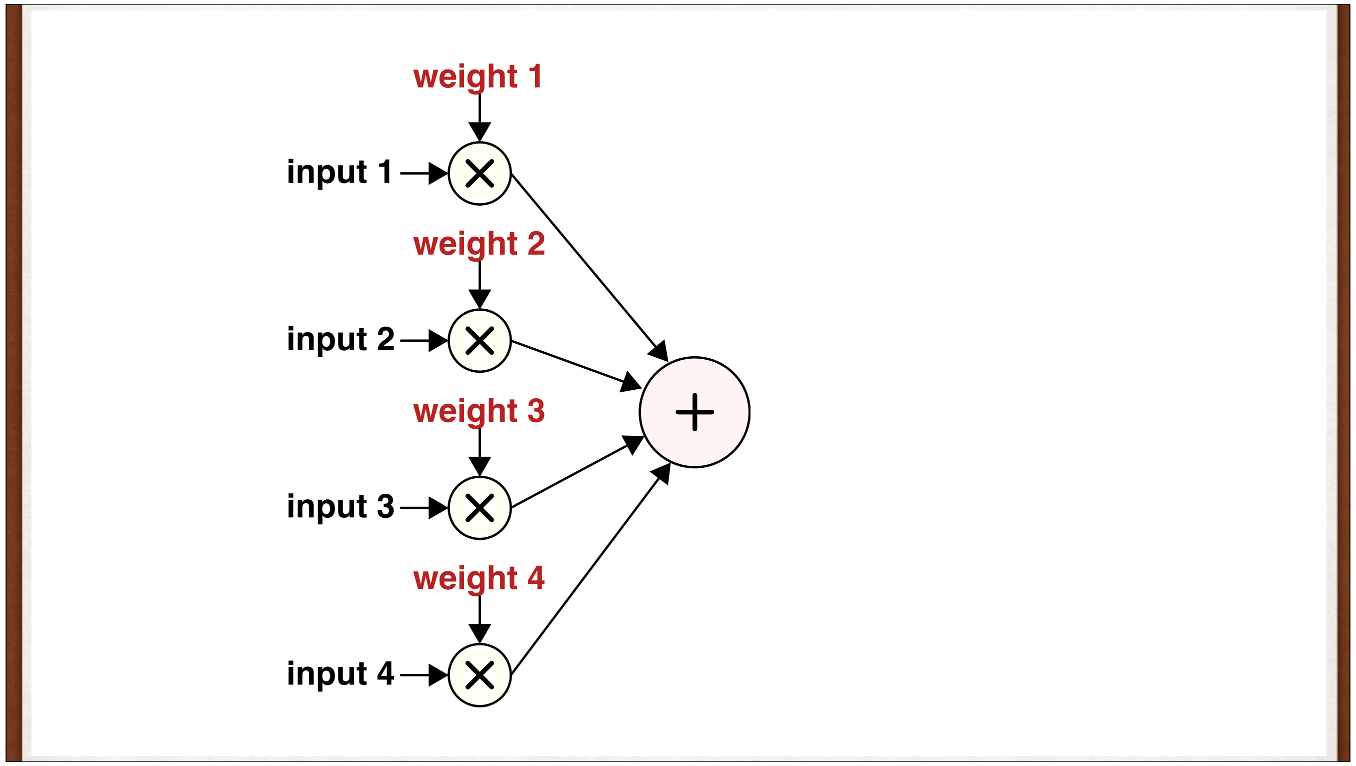
weight 2

input 2

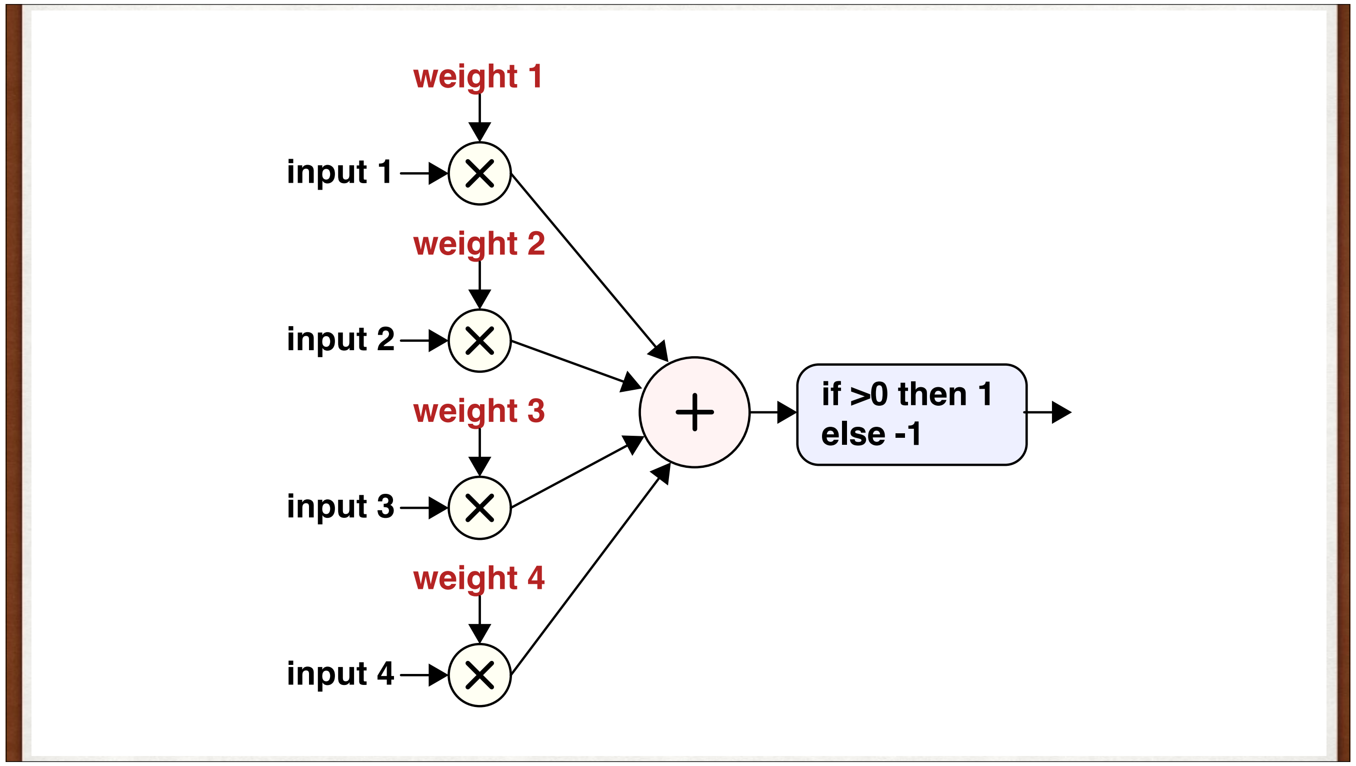
Our “artificial neuron”, or perceptron: inputs (numbers) come in. Each is “weighted” by another number. We add those all up, and threshold the result. The output is then -1 or 1 (some people use 0 and 1).



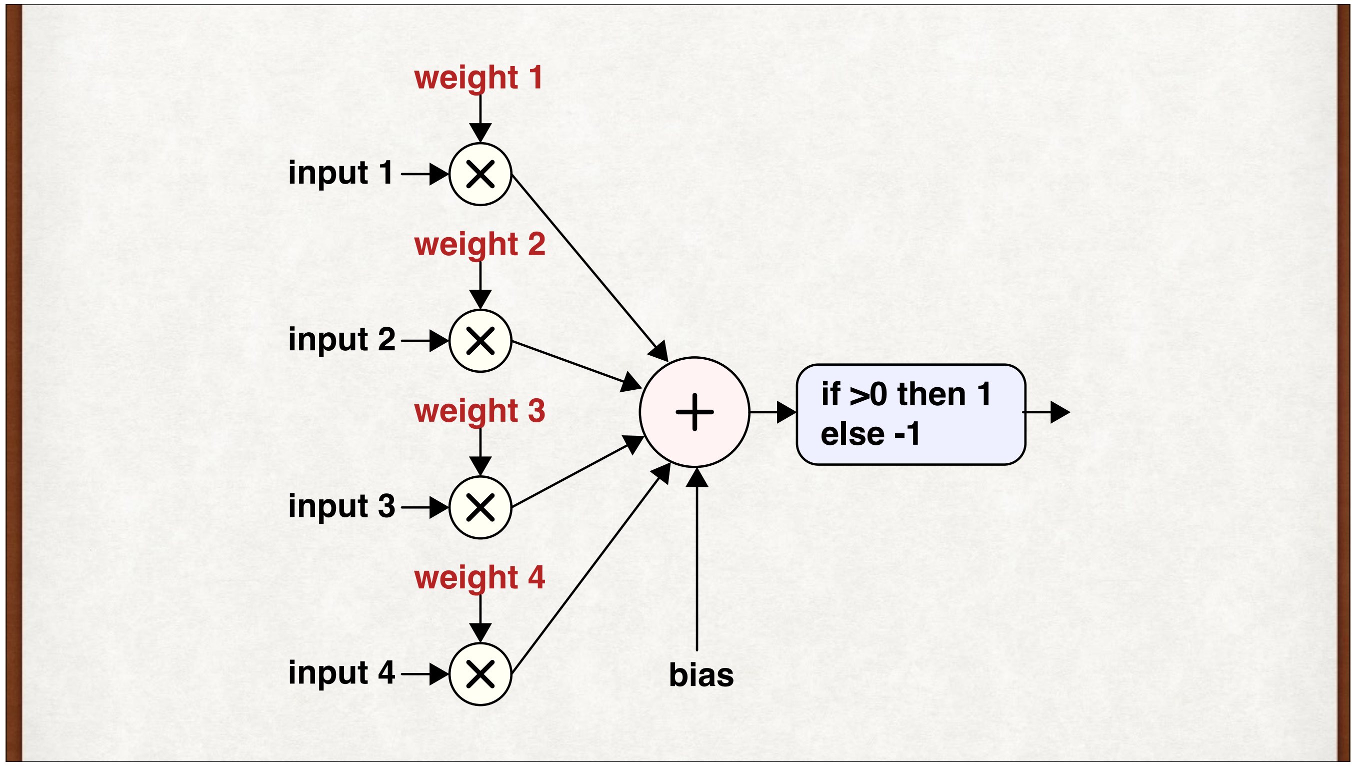
Our “artificial neuron”, or perceptron: inputs (numbers) come in. Each is “weighted” by another number. We add those all up, and threshold the result. The output is then -1 or 1 (some people use 0 and 1).



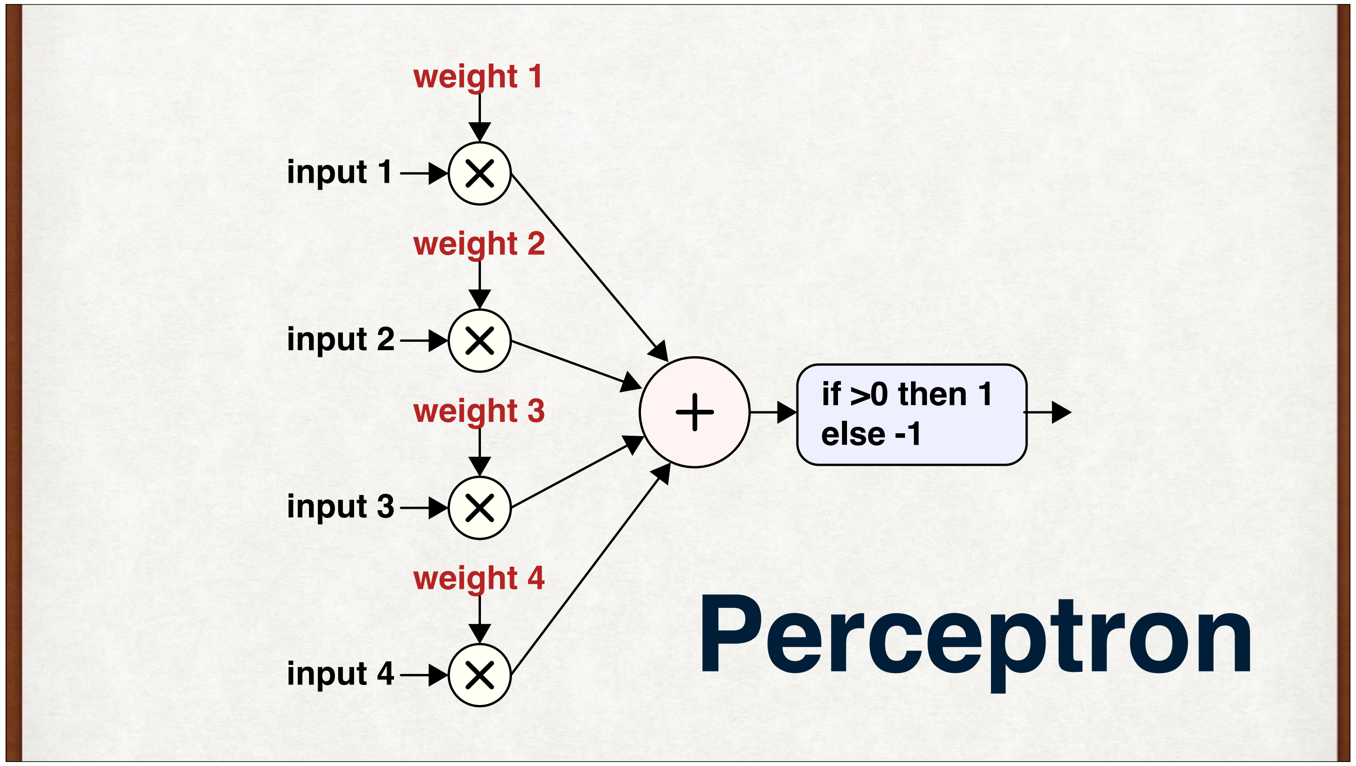
Our “artificial neuron”, or perceptron: inputs (numbers) come in. Each is “weighted” by another number. We add those all up, and threshold the result. The output is then -1 or 1 (some people use 0 and 1).



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).

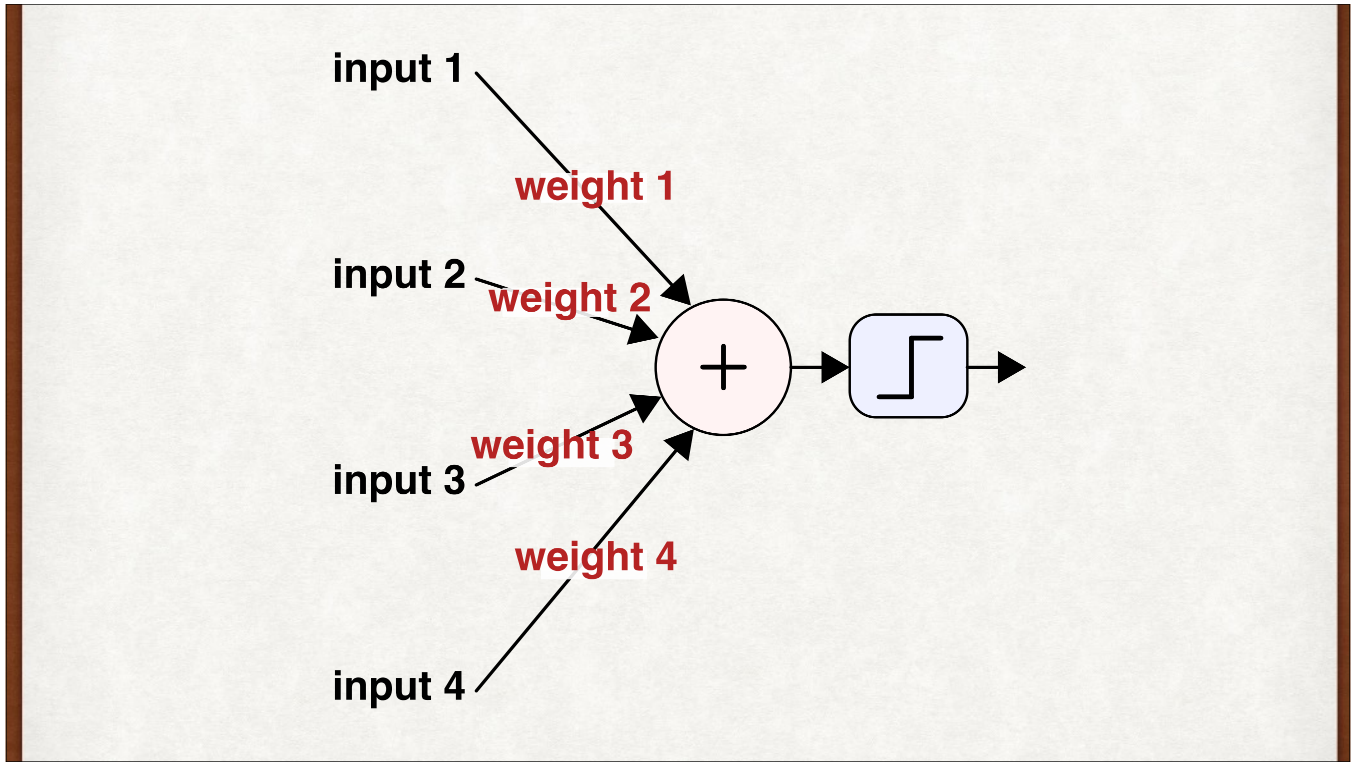


There’s a bias term. We can think of this as an input with a value of 1, and a weight that is the bias. So we can treat this term as just another input. This “bias trick” makes the code and math easier, so we’ll use it from now on. So the bias is always there, but we won’t draw it.

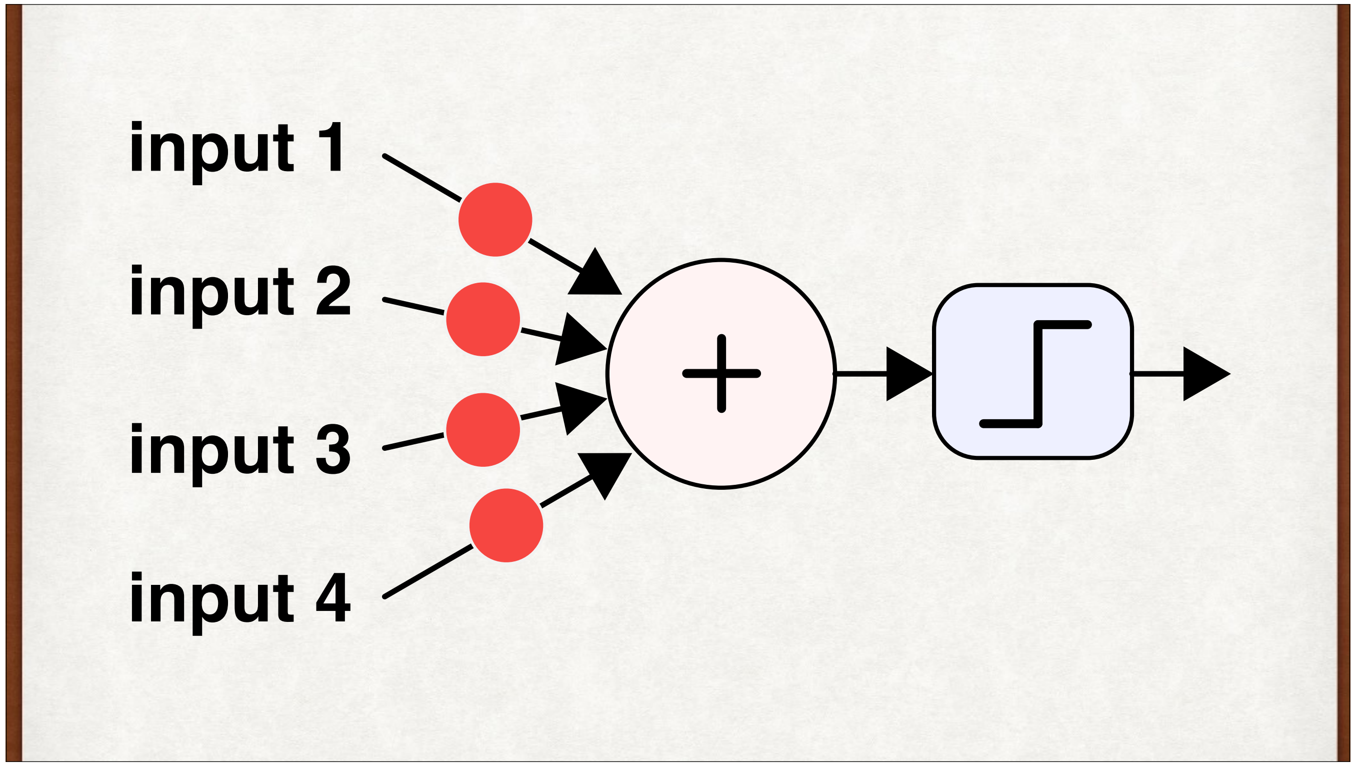


Perceptron

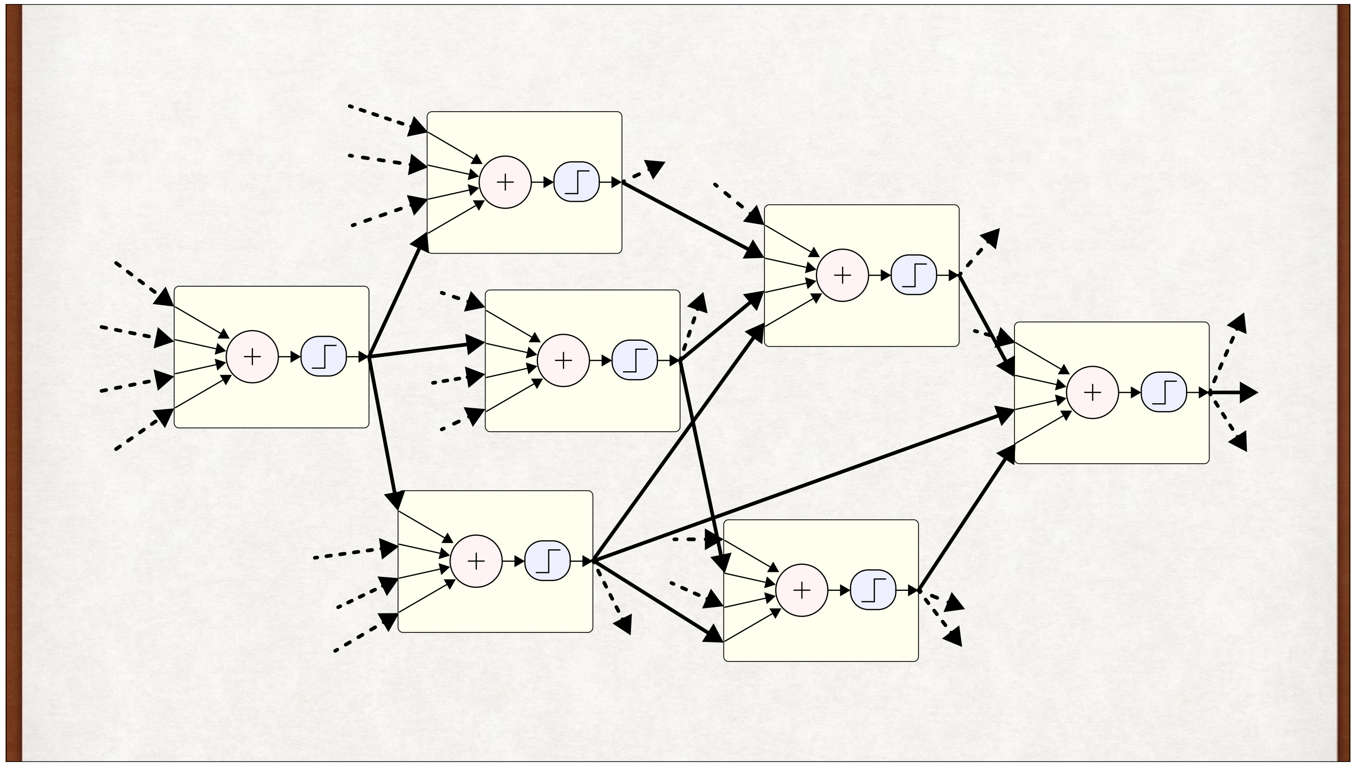
Our “artificial neuron”, or perceptron: inputs (numbers) come in. Each is “weighted” by another number. We add those all up, and threshold the result. The output is then -1 or 1 (some people use 0 and 1).



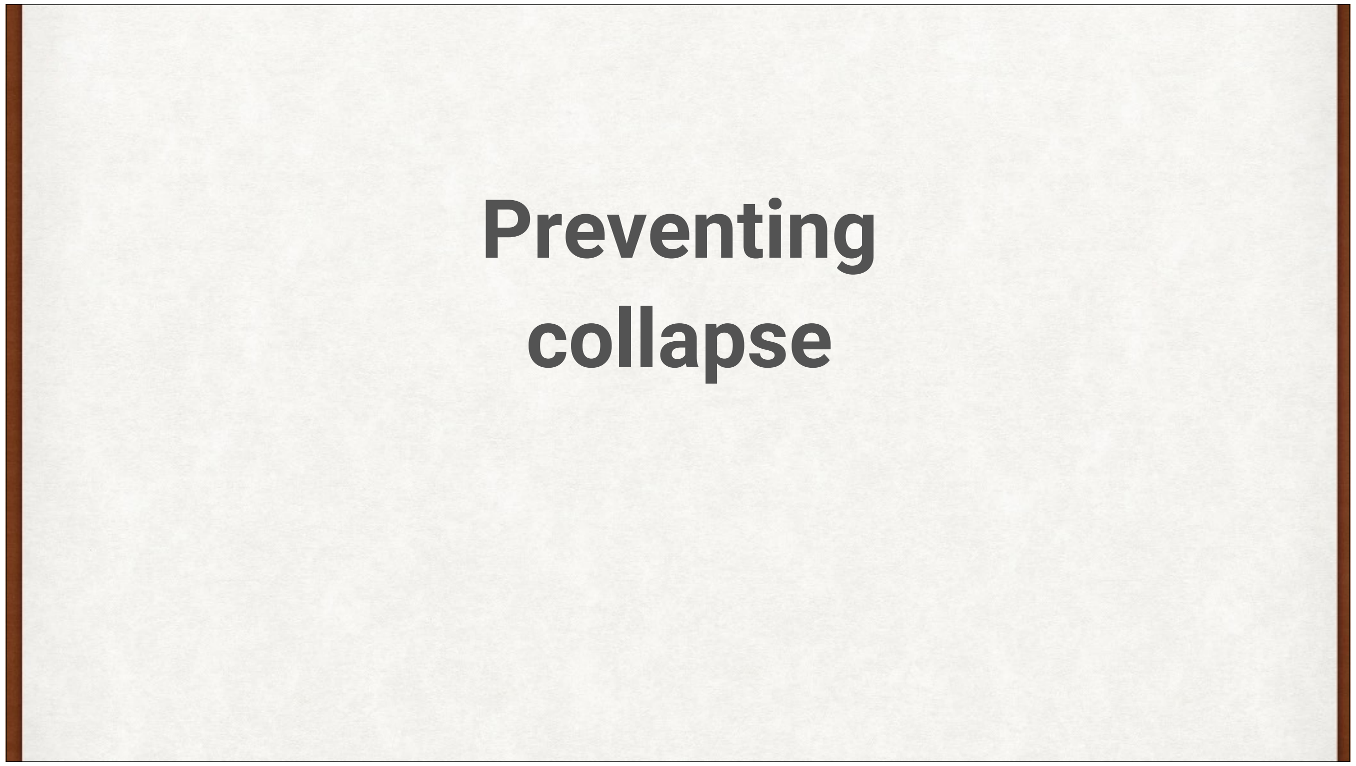
A simpler way to draw the last figure. The weights implicitly multiply the values on each “wire.”



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 “wire” bringing values into the neuron. Every input has a weight. In fact, the weights are the only things we have control over - the rest of the neuron (the summation and the threshold) are fixed.

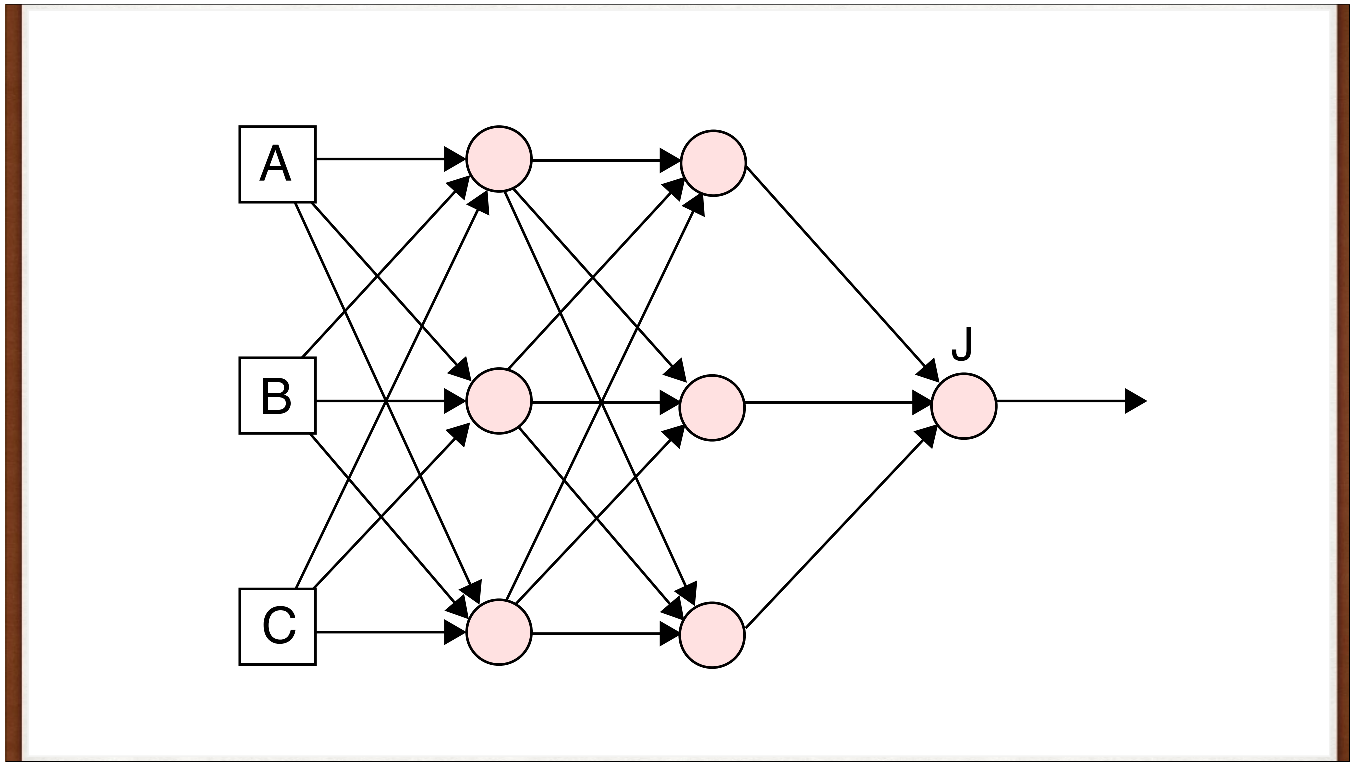


We can hook the output of each neuron into other neurons. A neural network!

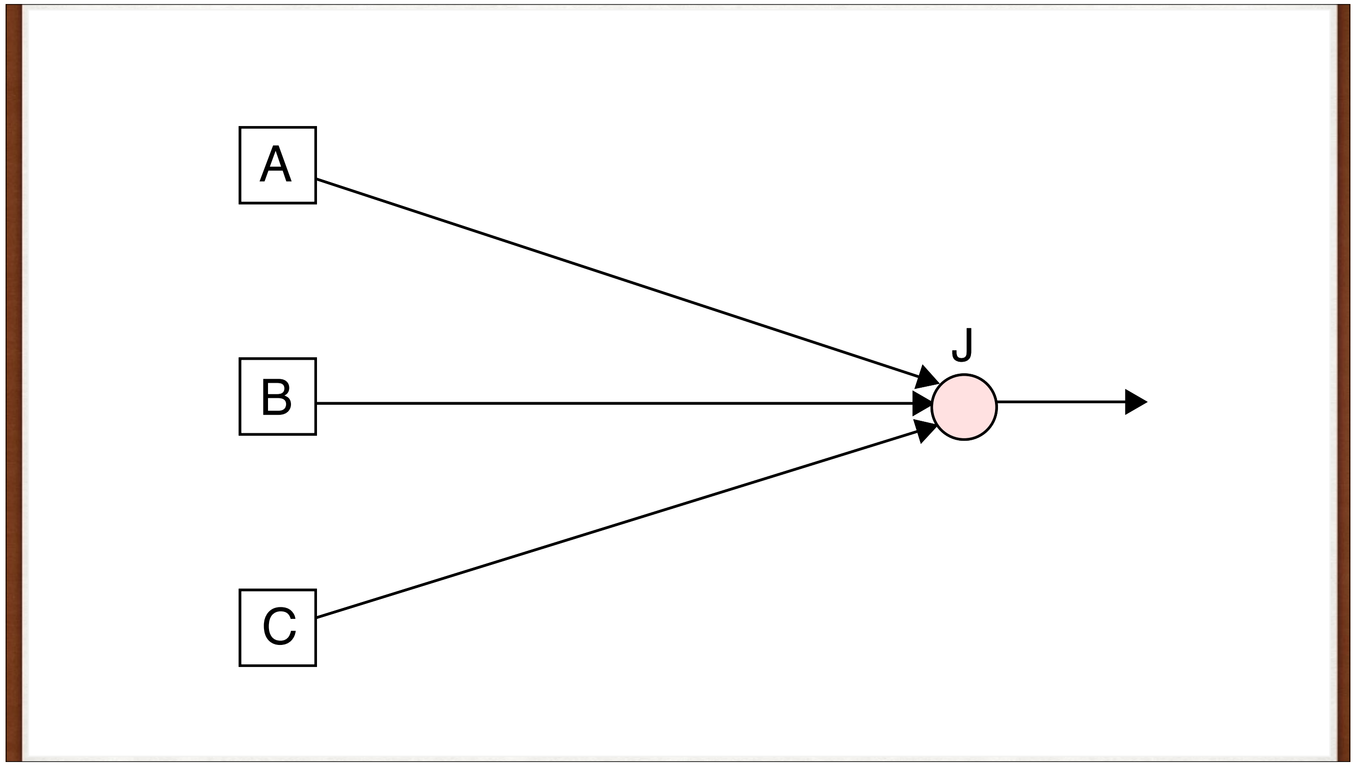


Preventing collapse

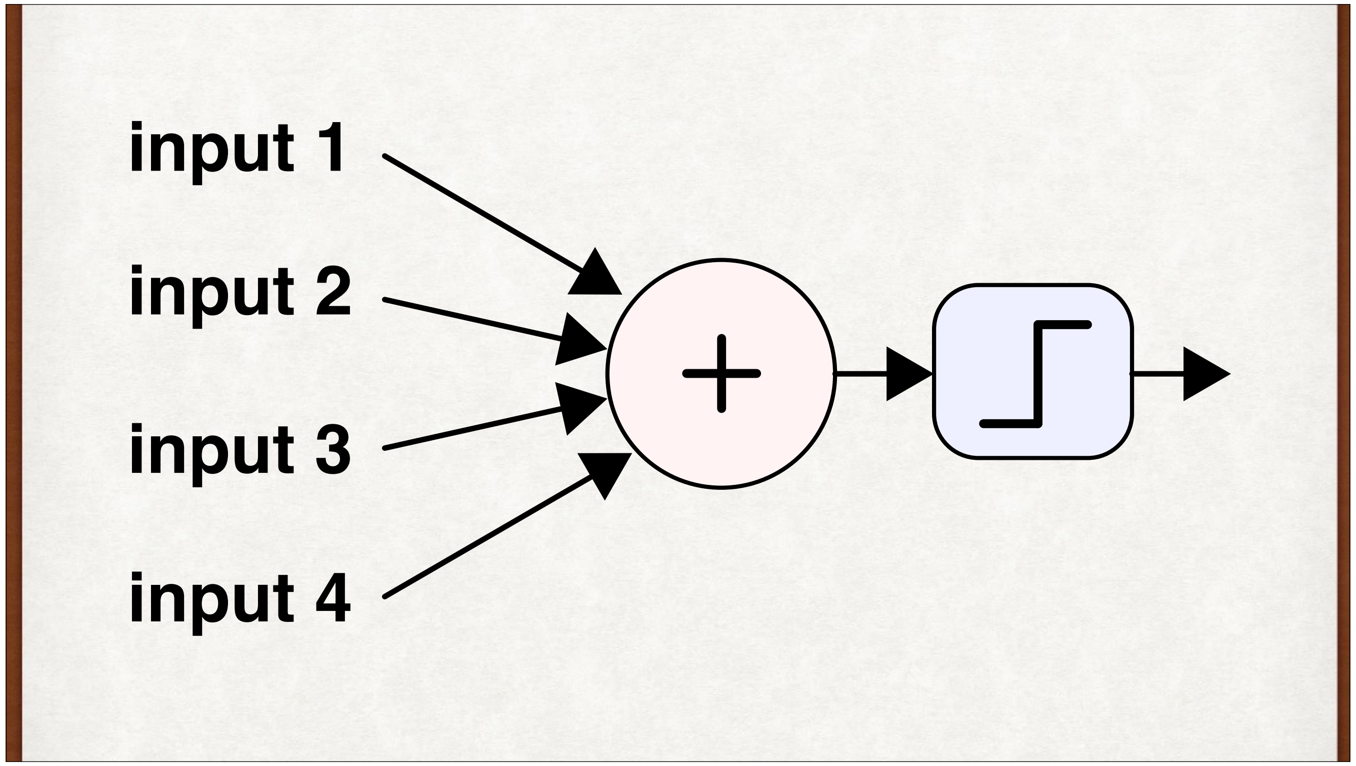
Networks *without* activation functions (the thresholds we’ve seen) can “collapse.” That’s bad, because it reduces them to just a single neuron. How does that happen?



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 (‘cause addition is commutative, 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.



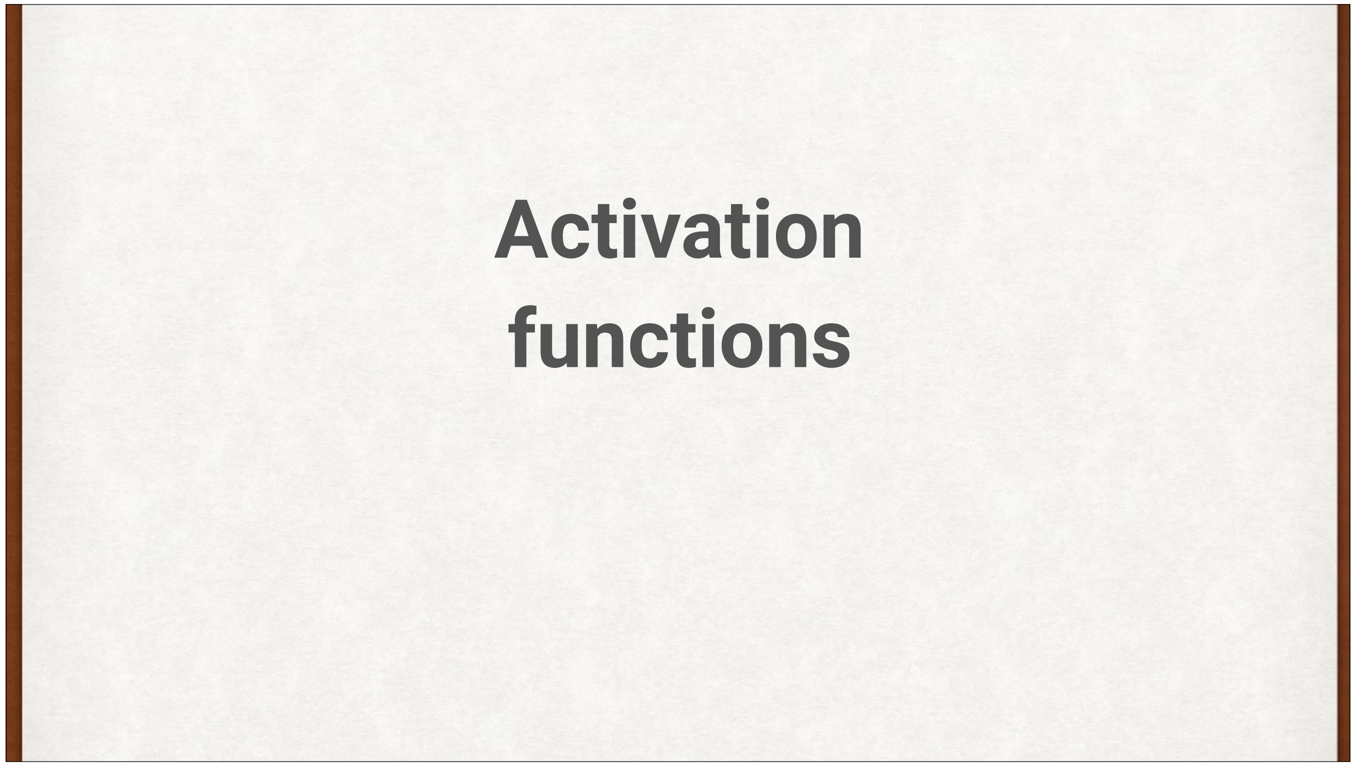
input 1

input 2

input 3

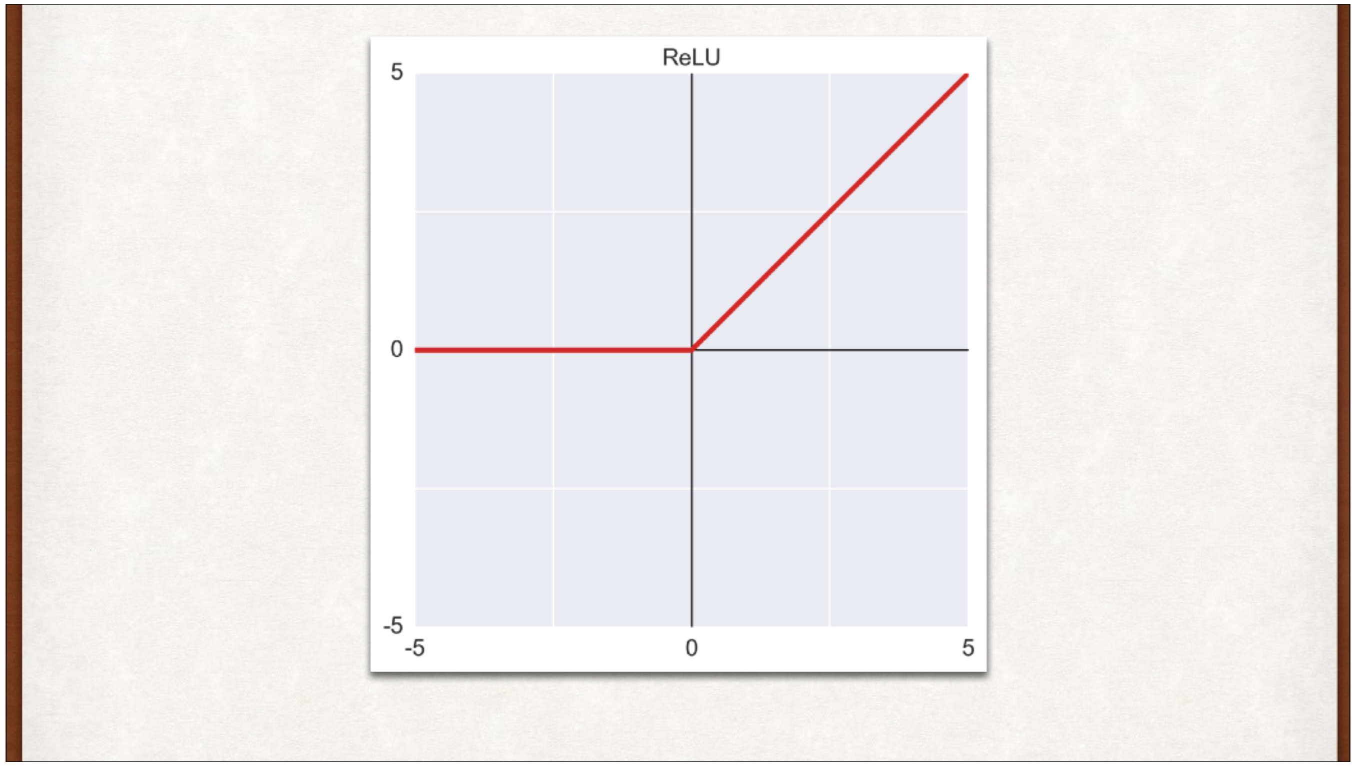
input 4

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.

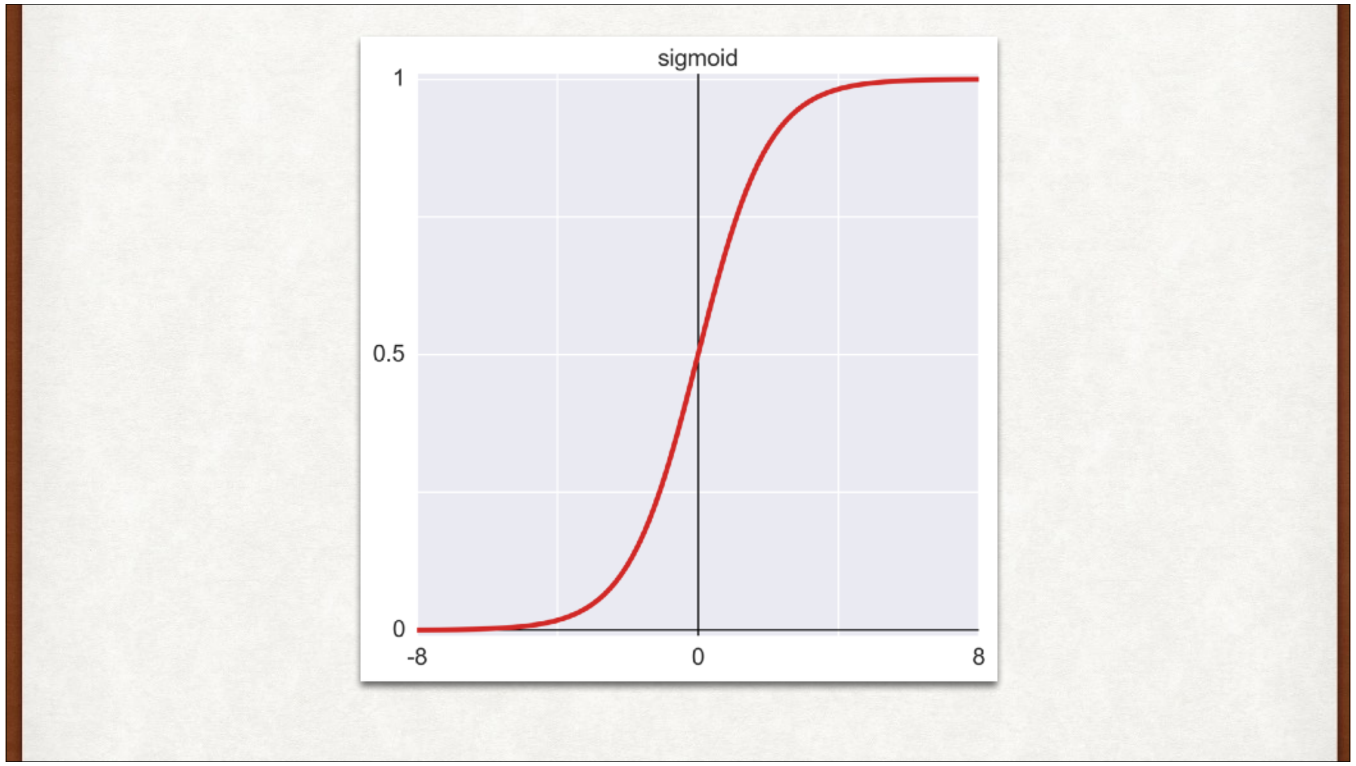


Activation functions

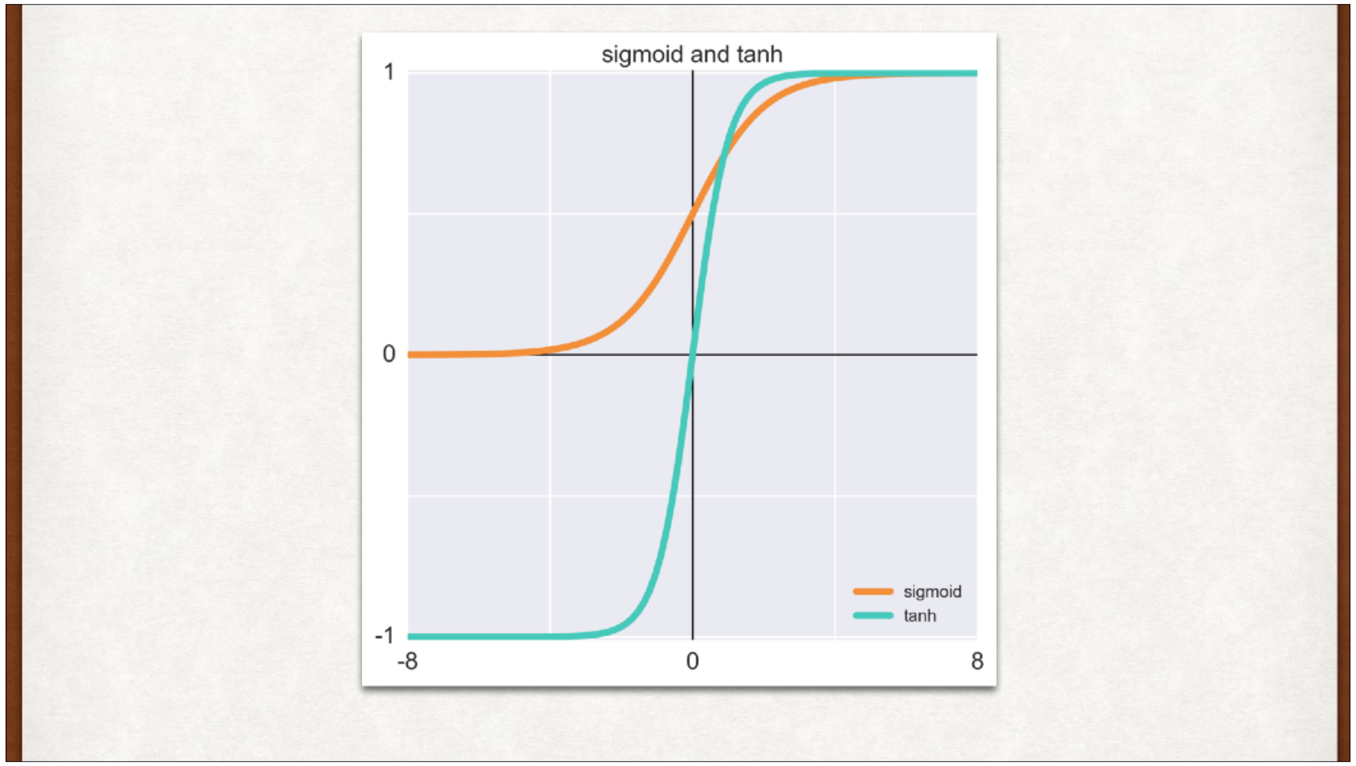
The threshold is also called an “activation function” because it produces the neuron’s output, the “activation.” Lots of non-linear activation functions have been proposed and experimented with.



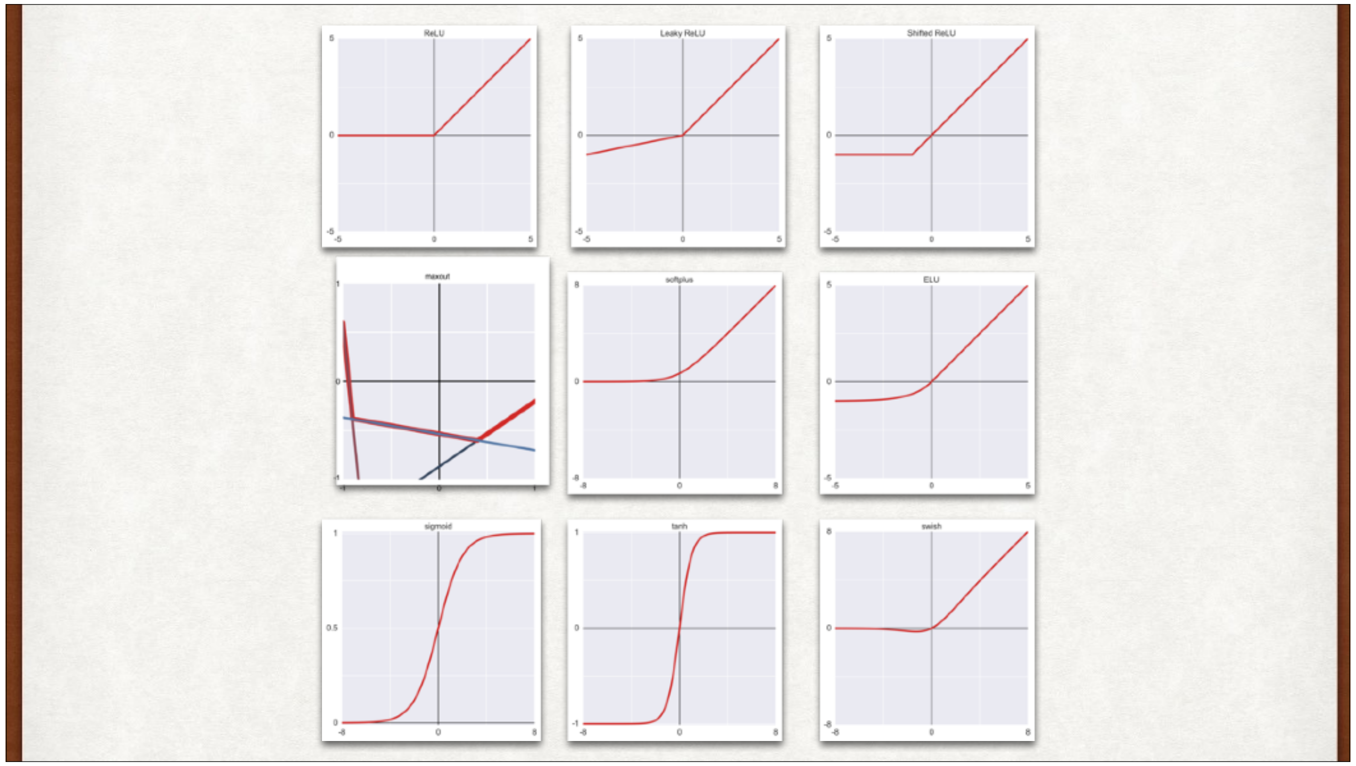
A Rectified Linear Unit (ReLU) is the most popular, and often the default choice in deep learning libraries.



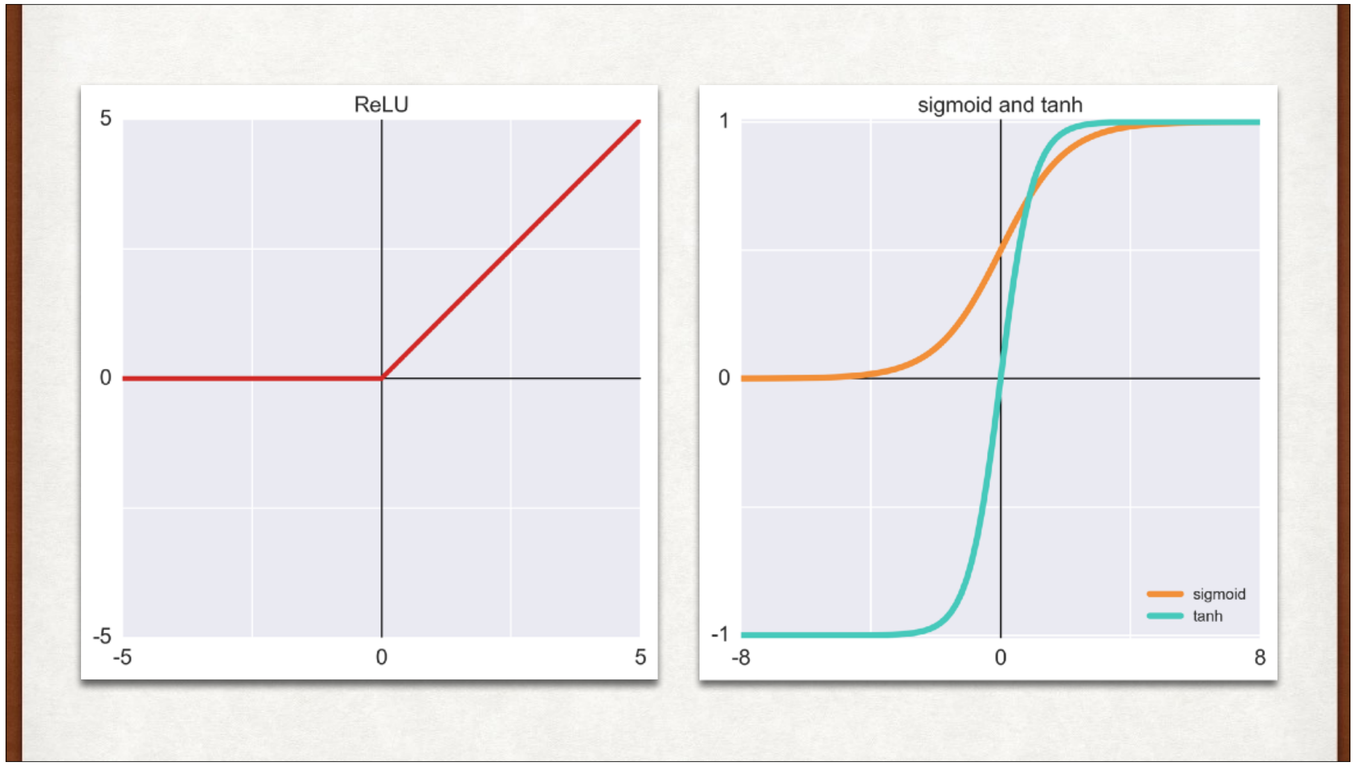
Sigmoid curve. It’s like a threshold, but smooth and continuous. We sometimes say it “squashes” the input, which can be any real number, to the range [0,1].



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



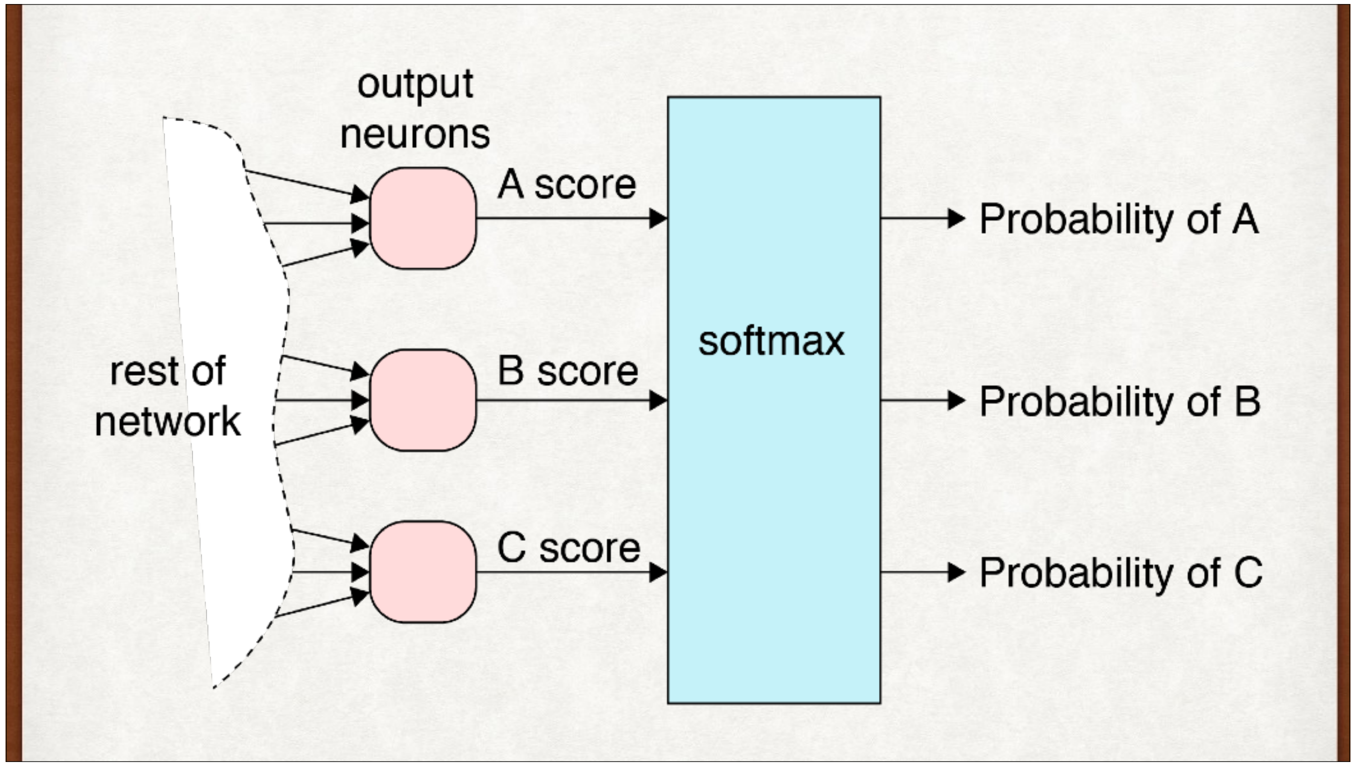
An activation function menagerie. And there are more!



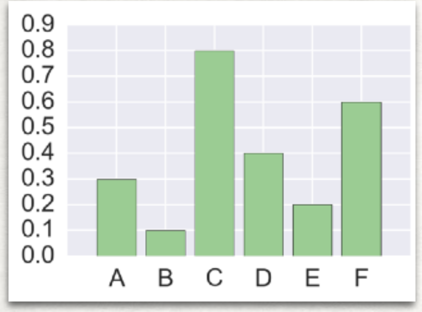
Picking the right one is partly theory, partly hunch and experience, and partly experiment. Usually all the neurons on any given layer share the same activation function. These are the most popular. The output layer of a classifier uses…

softmax

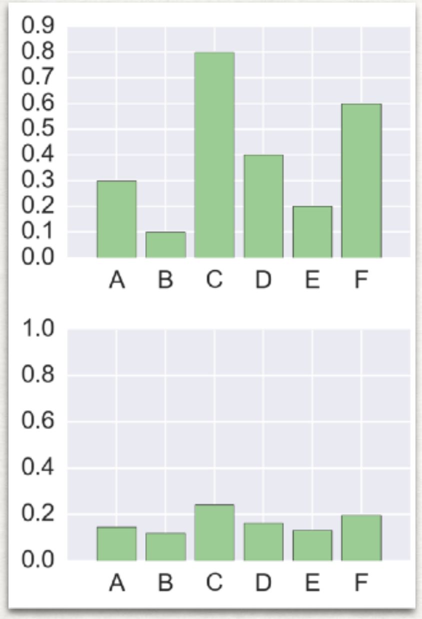
softmax. This is a little piece of mathematics. Not quite an activation function, because it takes as inputs *all* the outputs of the output layer neurons simultaneously.



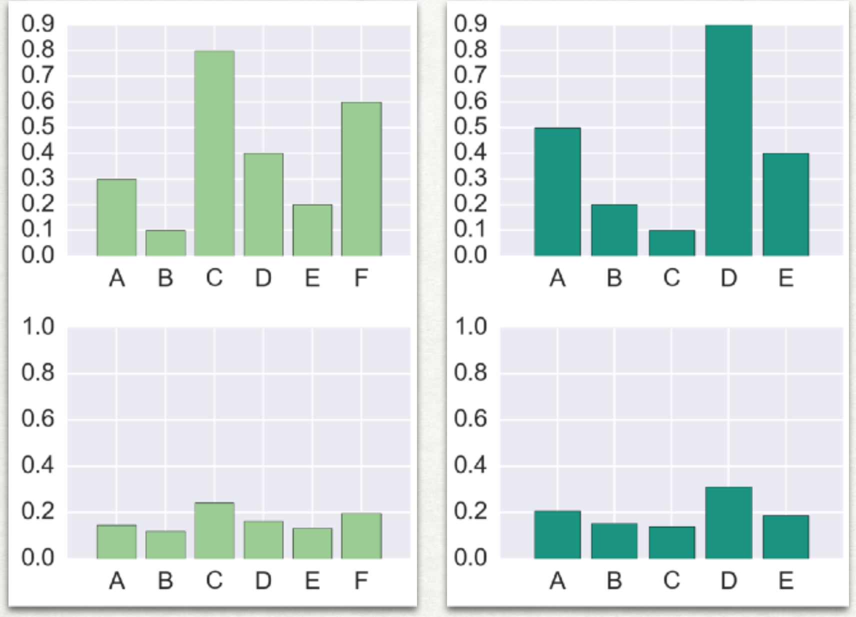
Unlike the functions we've seen so far, softmax takes in multiple inputs - usually all the outputs of the last layer of the network. It munches on them and produces a probability for each class.



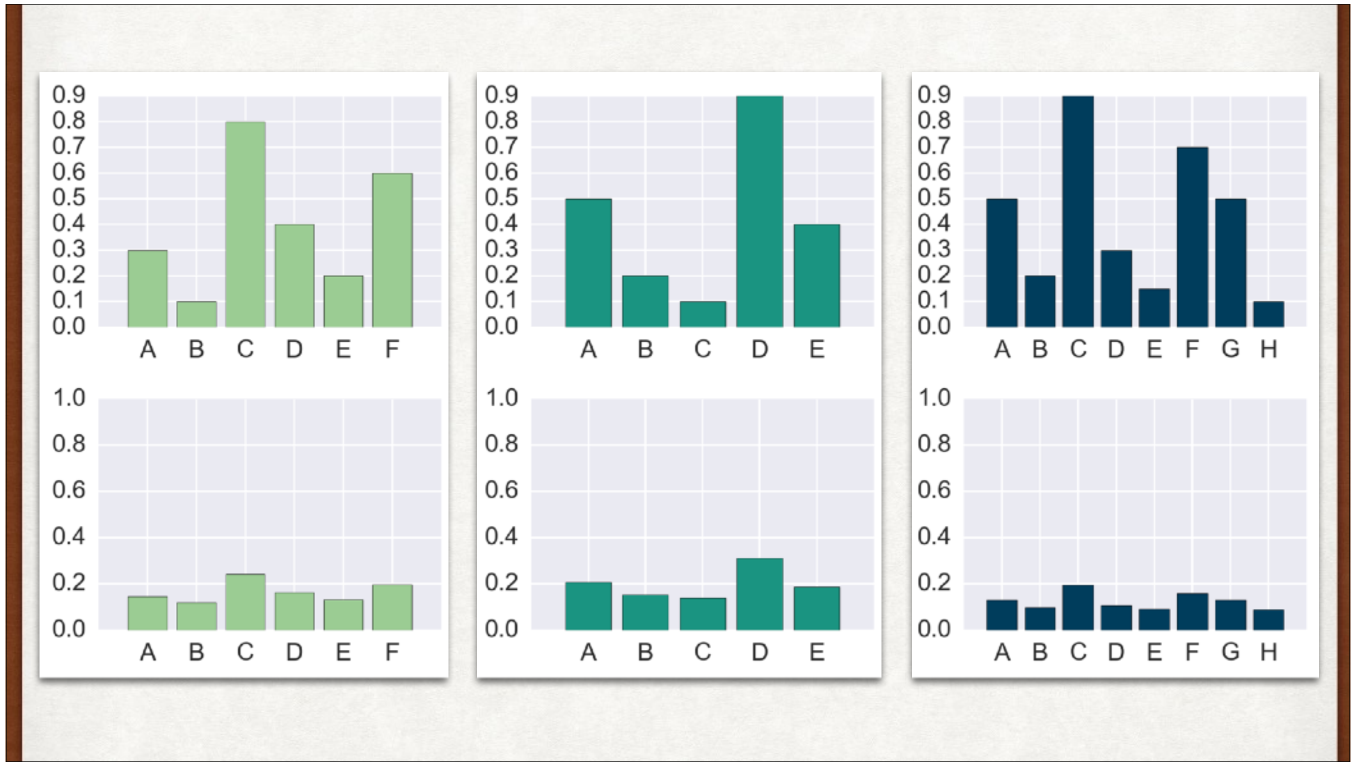
The results of softmax. Top, a set of inputs to softmax. Bottom, the output. The sorting stays the same from biggest to smallest, but each value is adjusted by a different amount. The outputs all add up to 1. Up top, we can only say that category C is more likely than, say, B, but we can’t say by how 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.



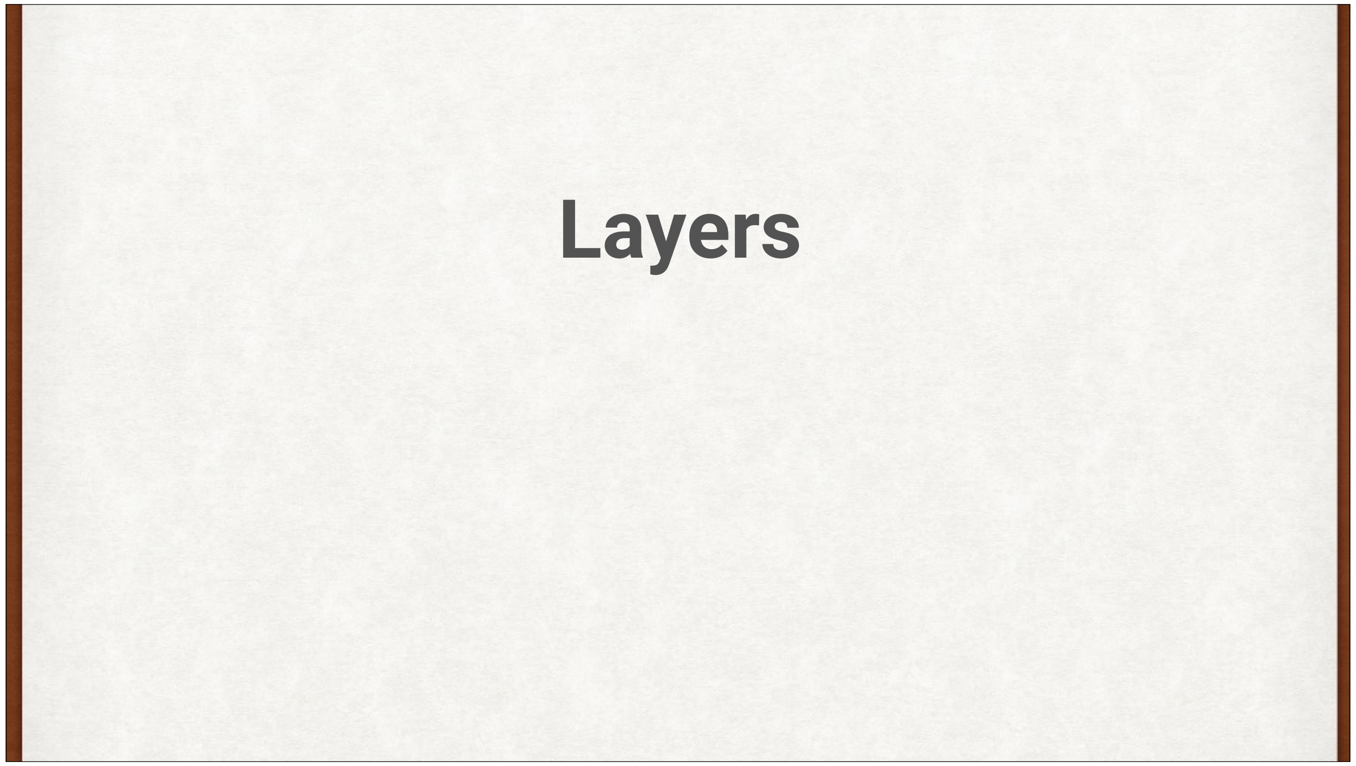
The results of softmax. Top, a set of inputs to softmax. Bottom, the output. The sorting stays the same from biggest to smallest, but each value is adjusted by a different amount. The outputs all add up to 1. Up top, we can only say that category C is more likely than, say, B, but we can't say by how 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.

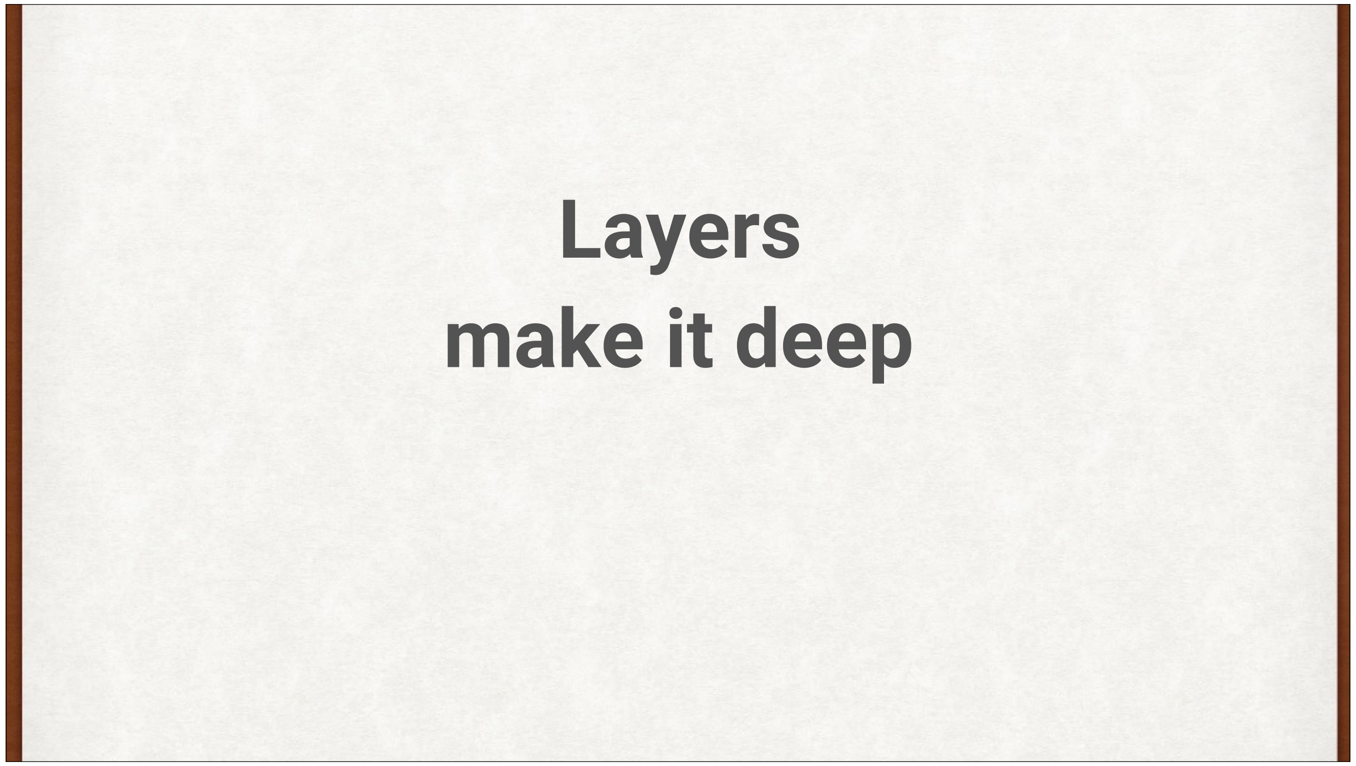


The results of softmax. Top, a set of inputs to softmax. Bottom, the output. The sorting stays the same from biggest to smallest, but each value is adjusted by a different amount. The outputs all add up to 1.



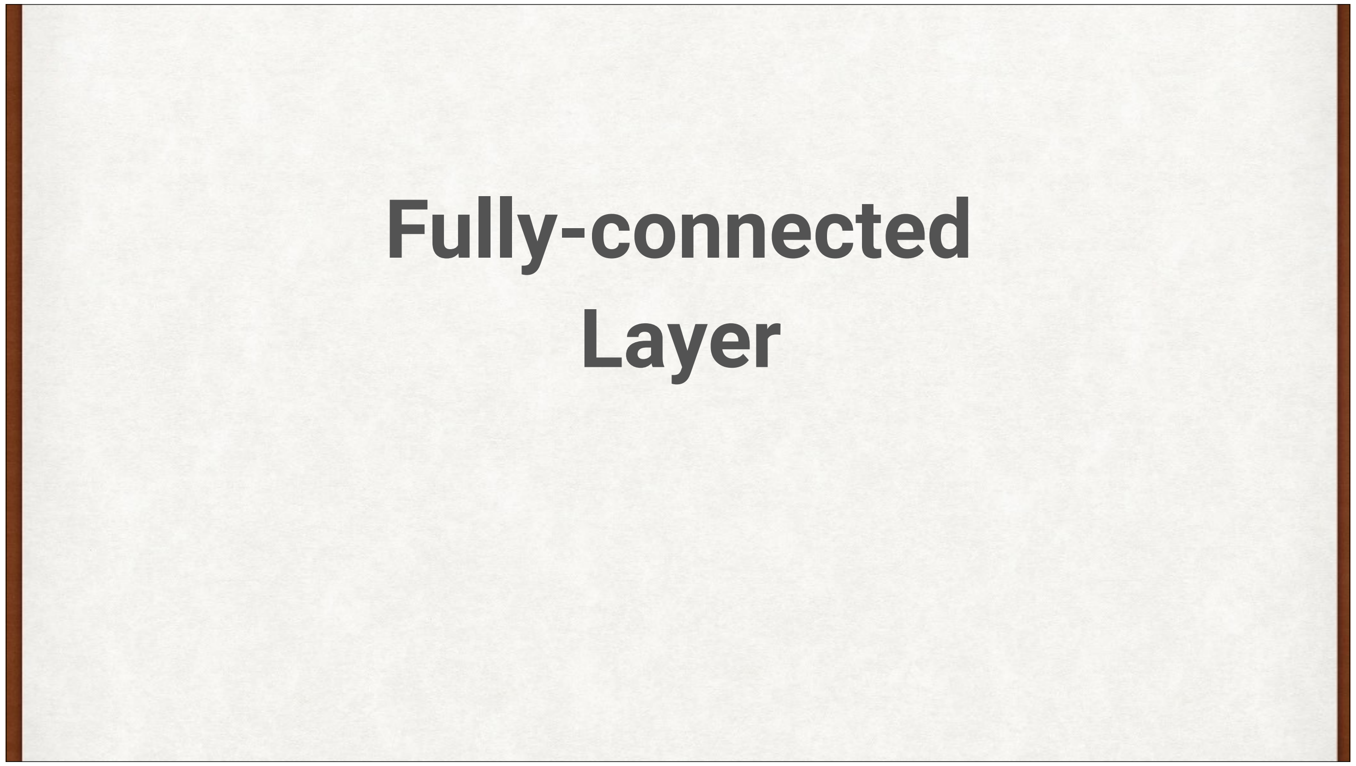
Layers

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



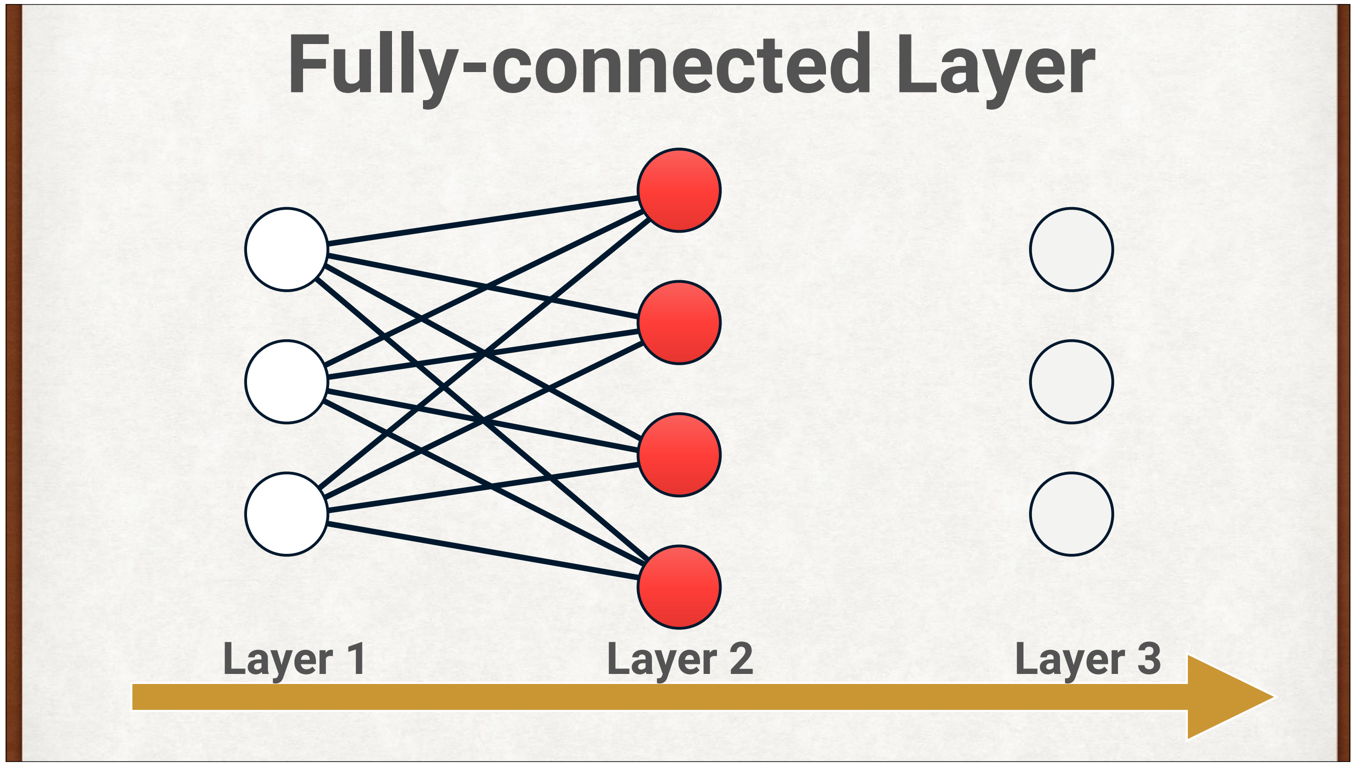
Layers
make it deep

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



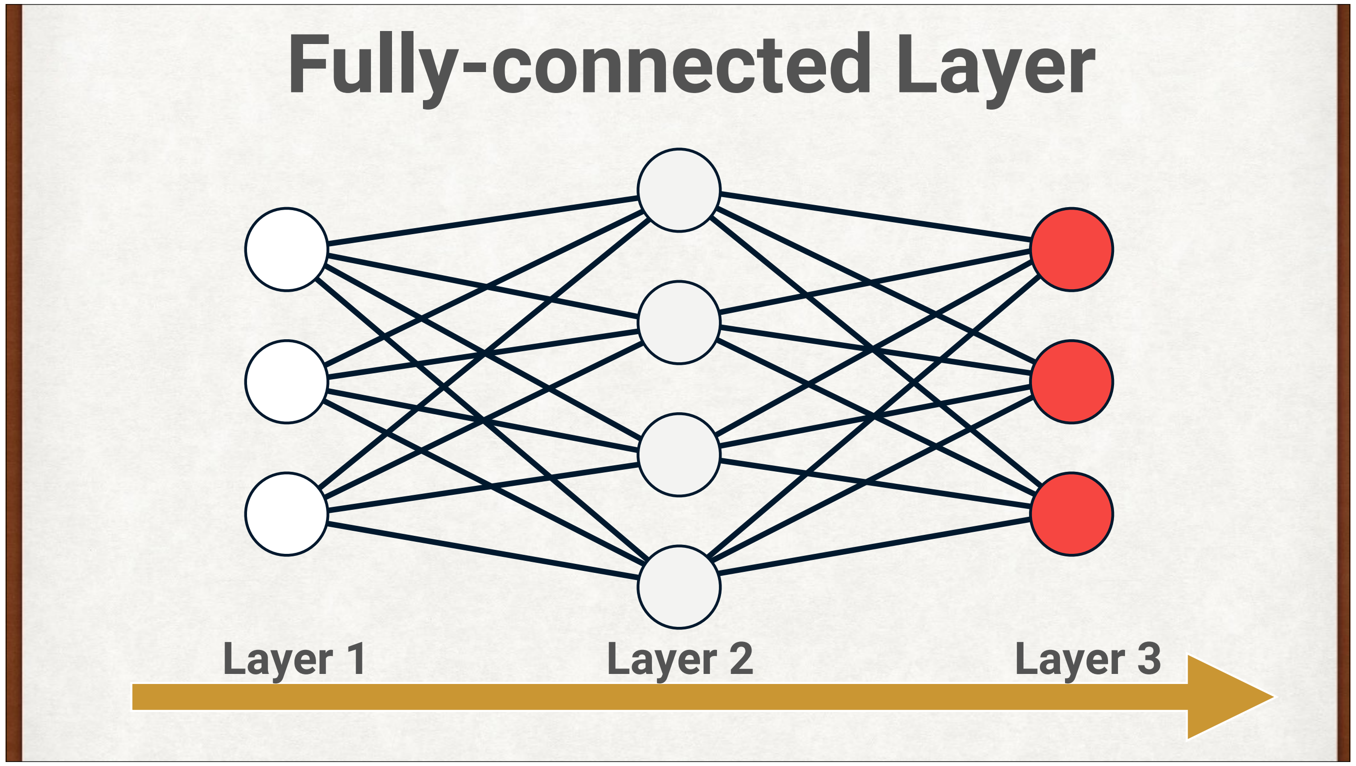
Fully-connected Layer

A fully-connected layer means each neuron on that layer receives input from every neuron on the previous layer.



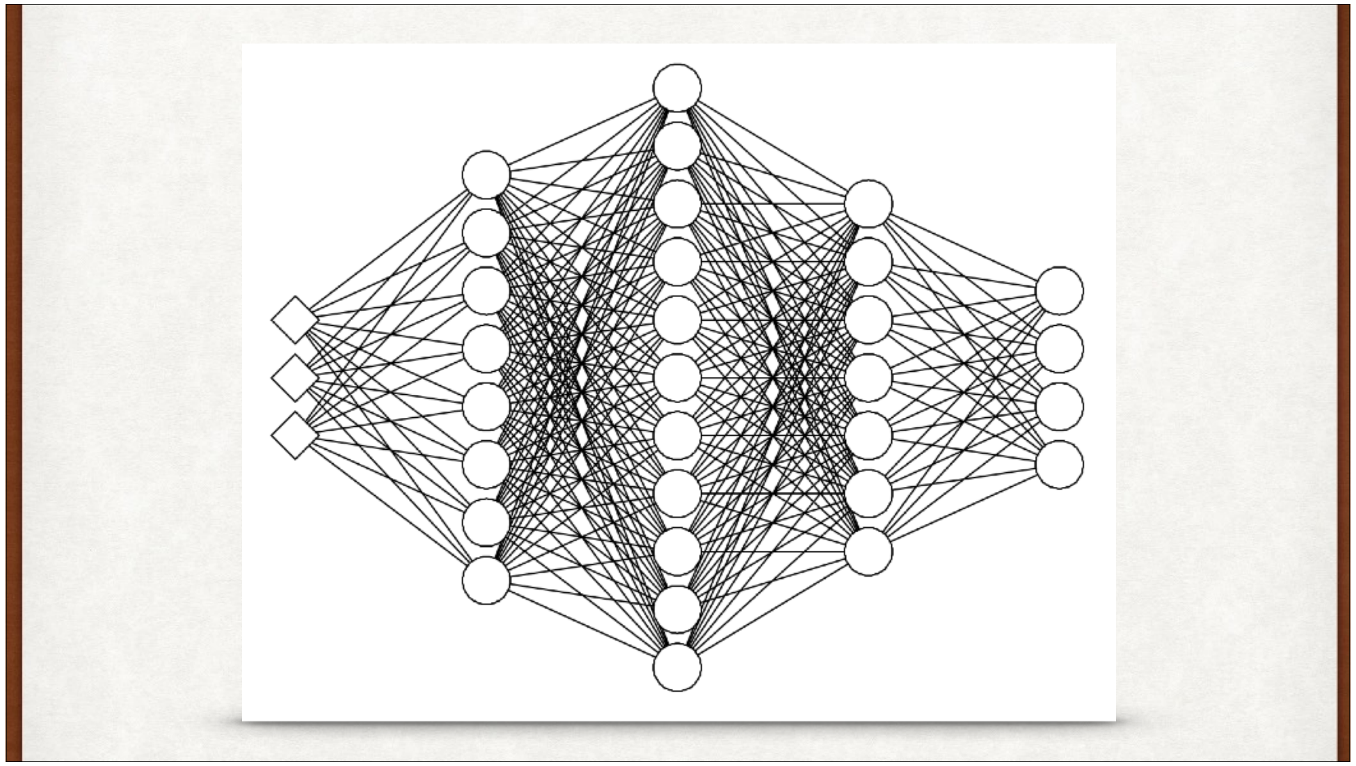
Fully-connected Layer

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

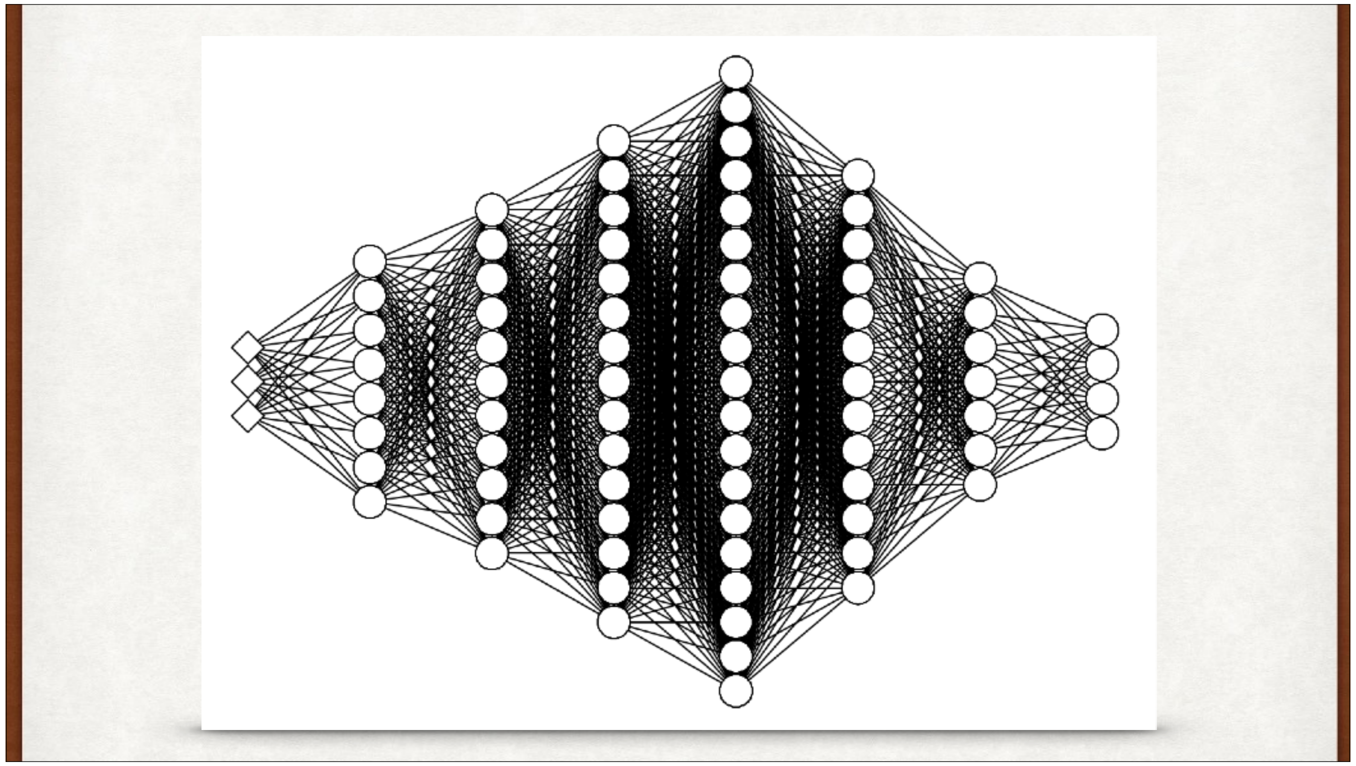


Fully-connected Layer

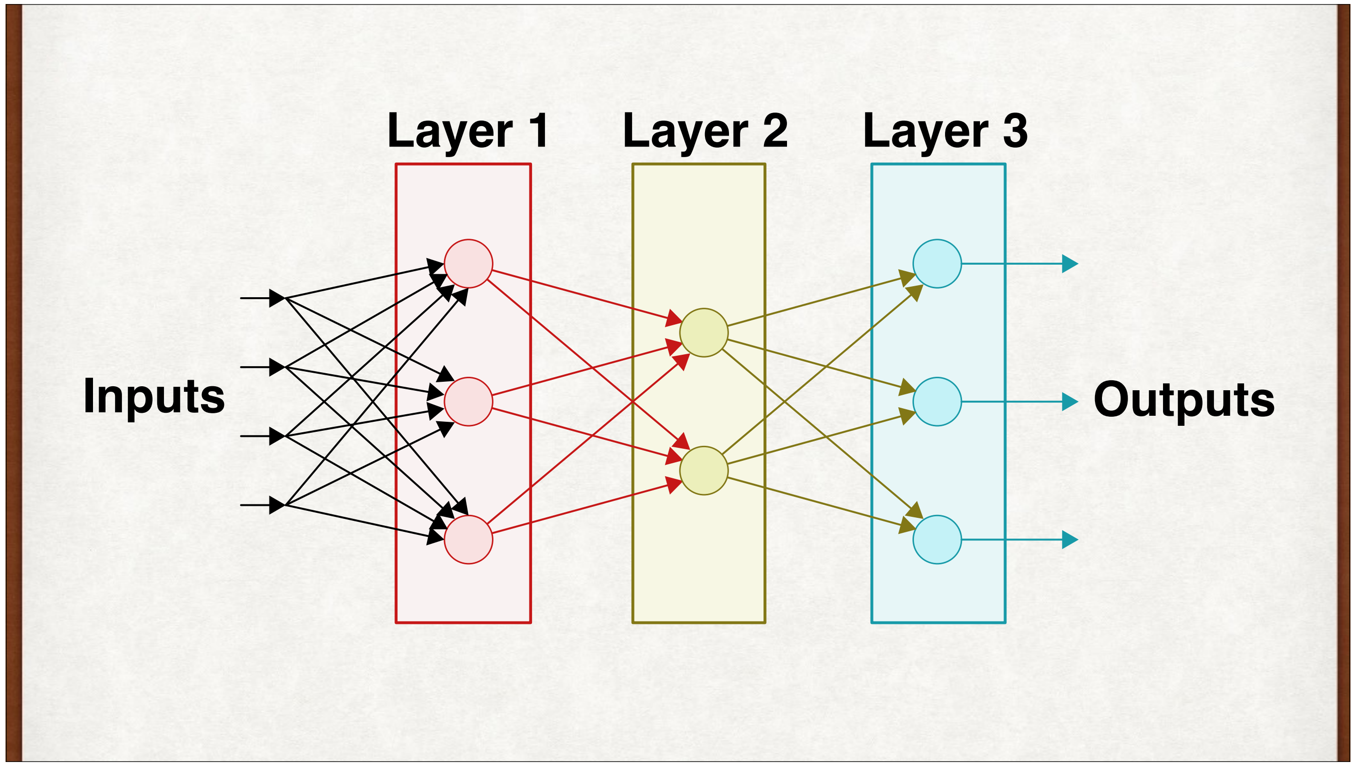
Here Layer 3 is fully-connected as well.



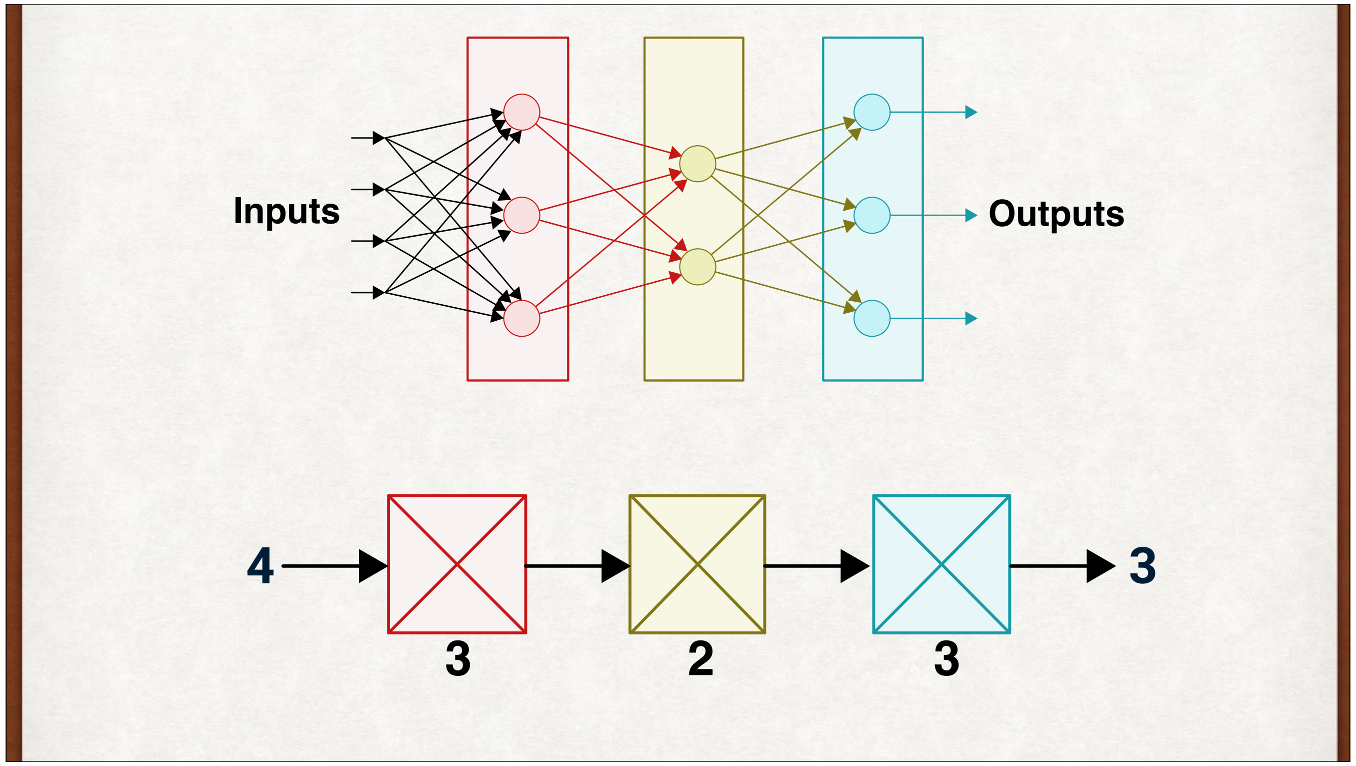
4 fully-connected layers. This kind of messy spider’s web of a diagram, though frequently seen, is just awful.



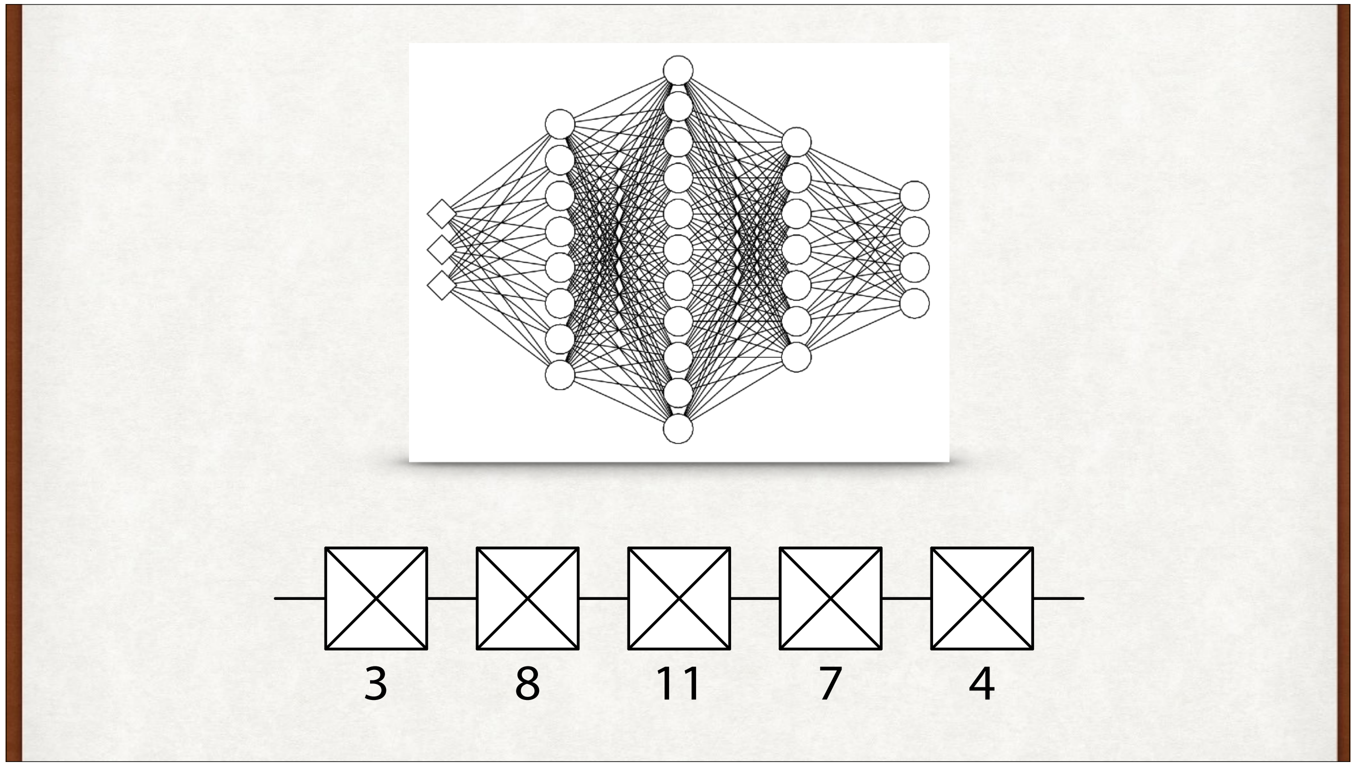
Sure, it’s kind of pretty, but this kind of diagram just doesn’t scale up well. We need something better.



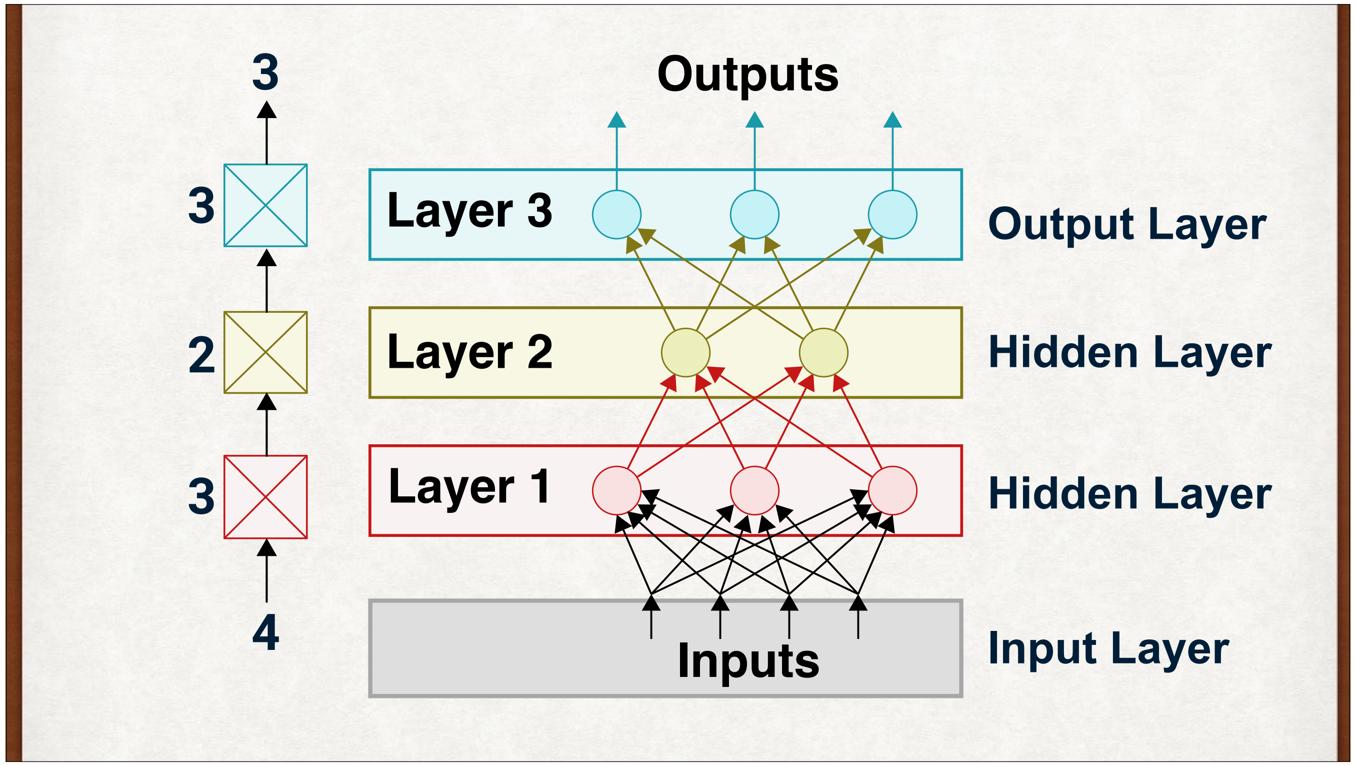
A simple network.



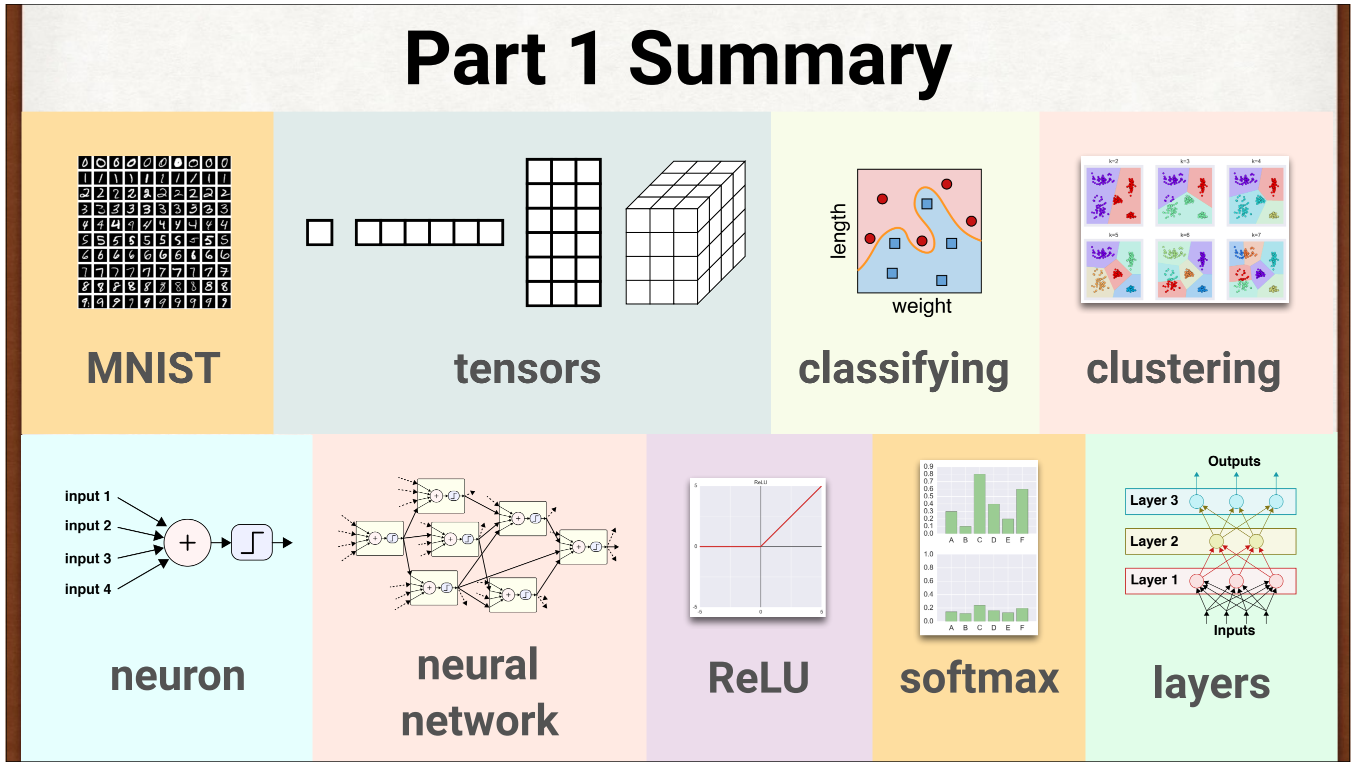
These are my icons for fully-connected layers. This is much easier to draw and interpret at a glance. The number of neurons is drawn beneath each icon.



The spider’s web and the icons.



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.



Part 1 Summary

MNIST

tensors

classifying

clustering

neuron

neural network

ReLU

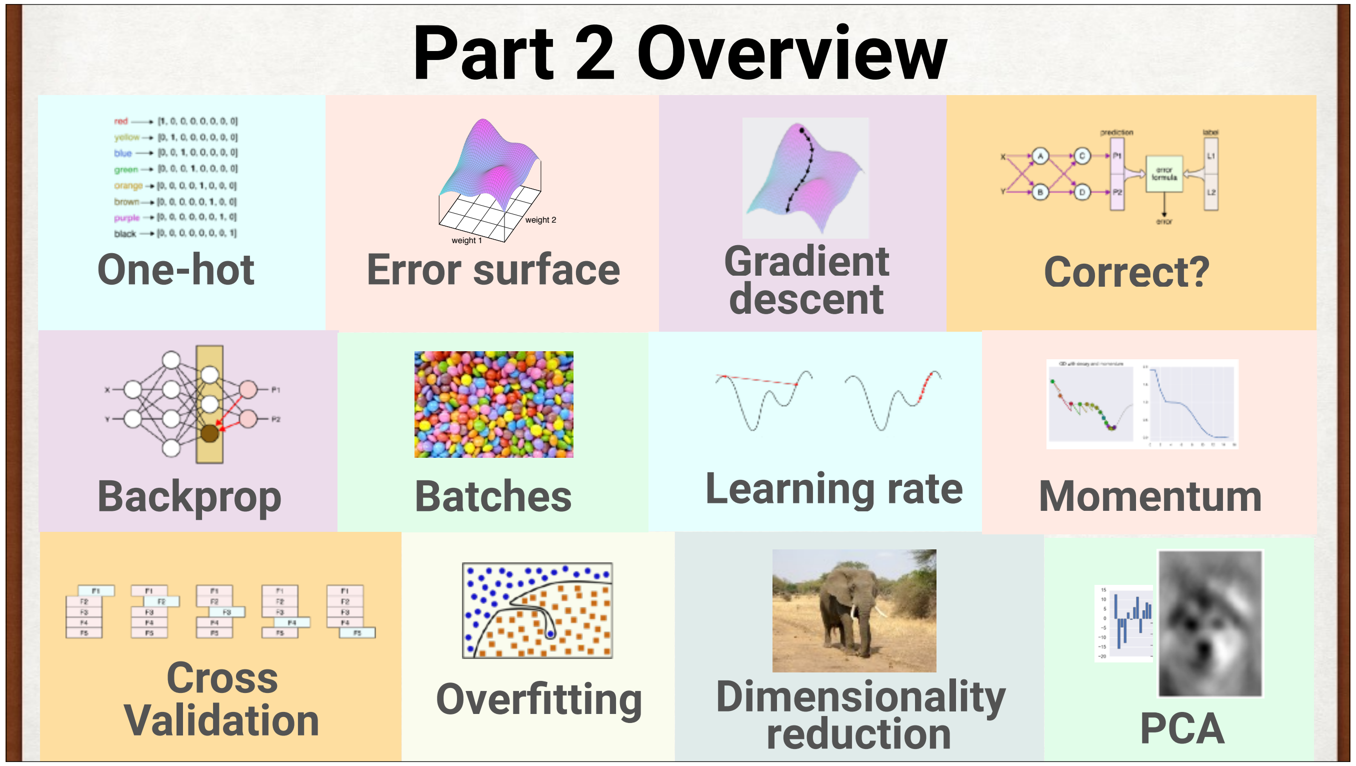
softmax

layers

A review

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

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



Part 2 Overview

One-hot

Error surface

Gradient descent

Correct?

Backprop

Batches

Learning rate

Momentum

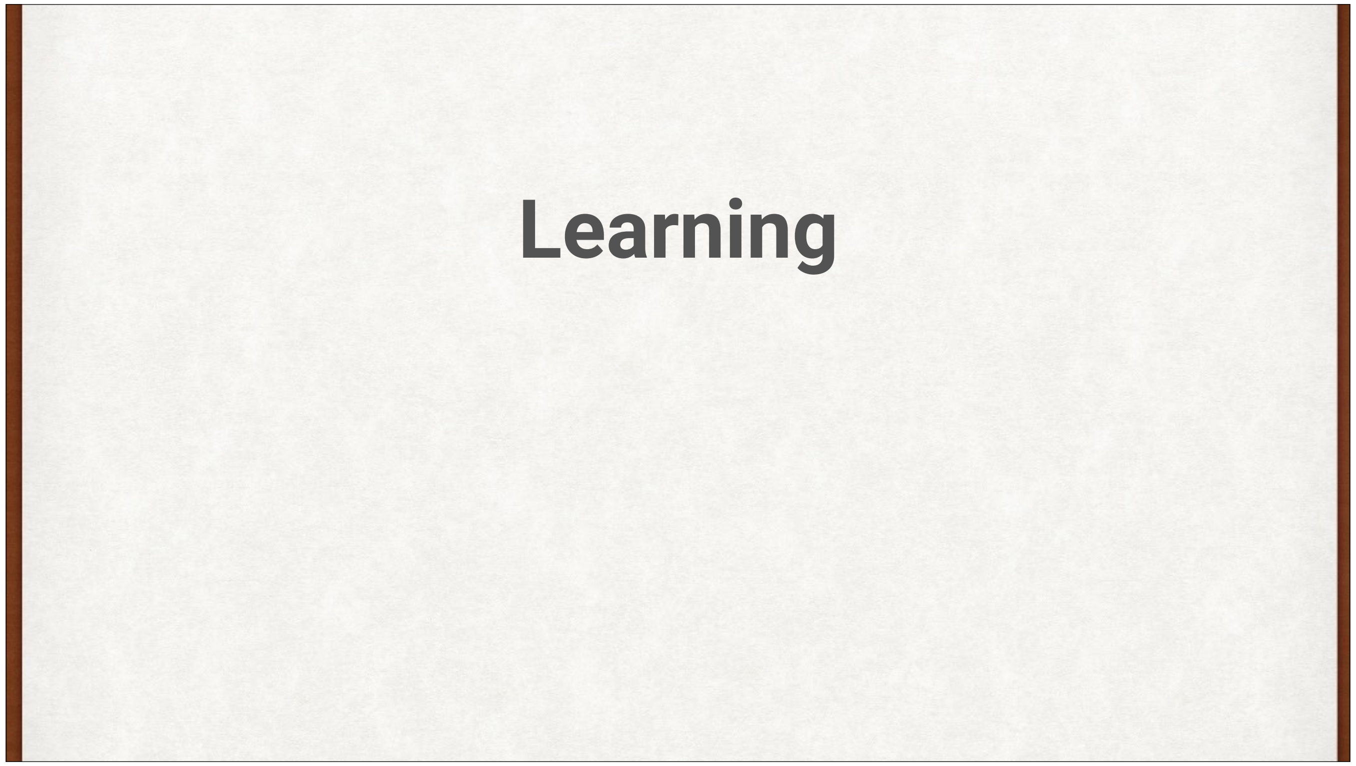
Cross Validation

Overfitting

Dimensionality reduction

PCA

The next set of topics

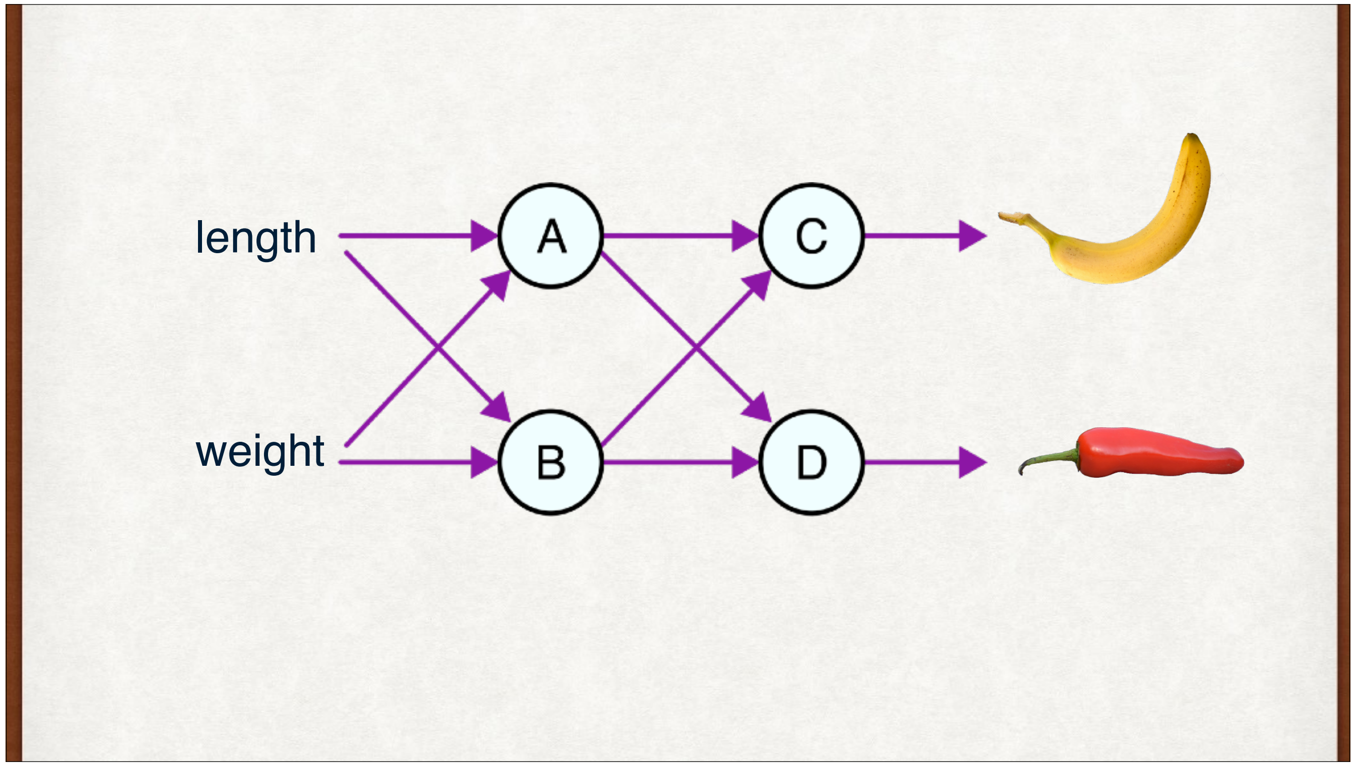


Learning

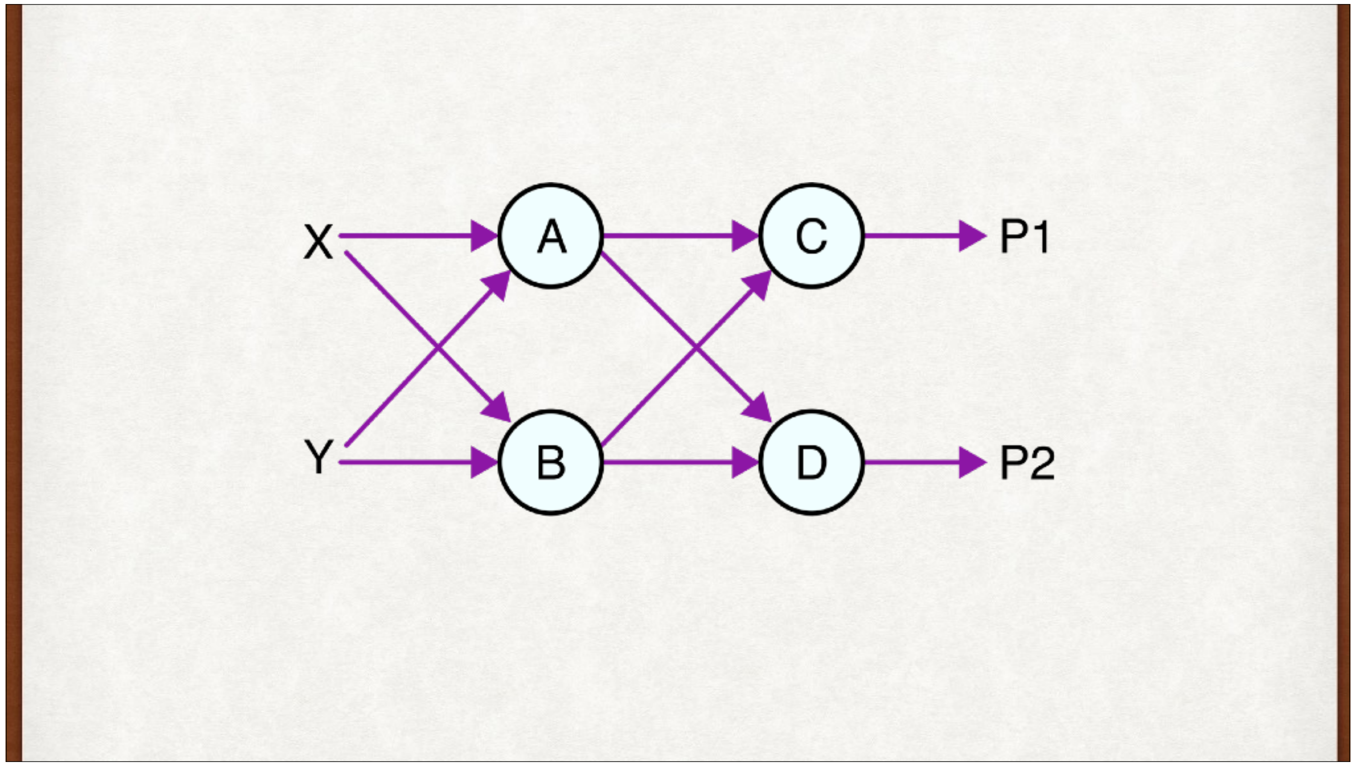
To learn from mistakes, we first have to know when we’ve made one.

Learning
by fixing mistakes

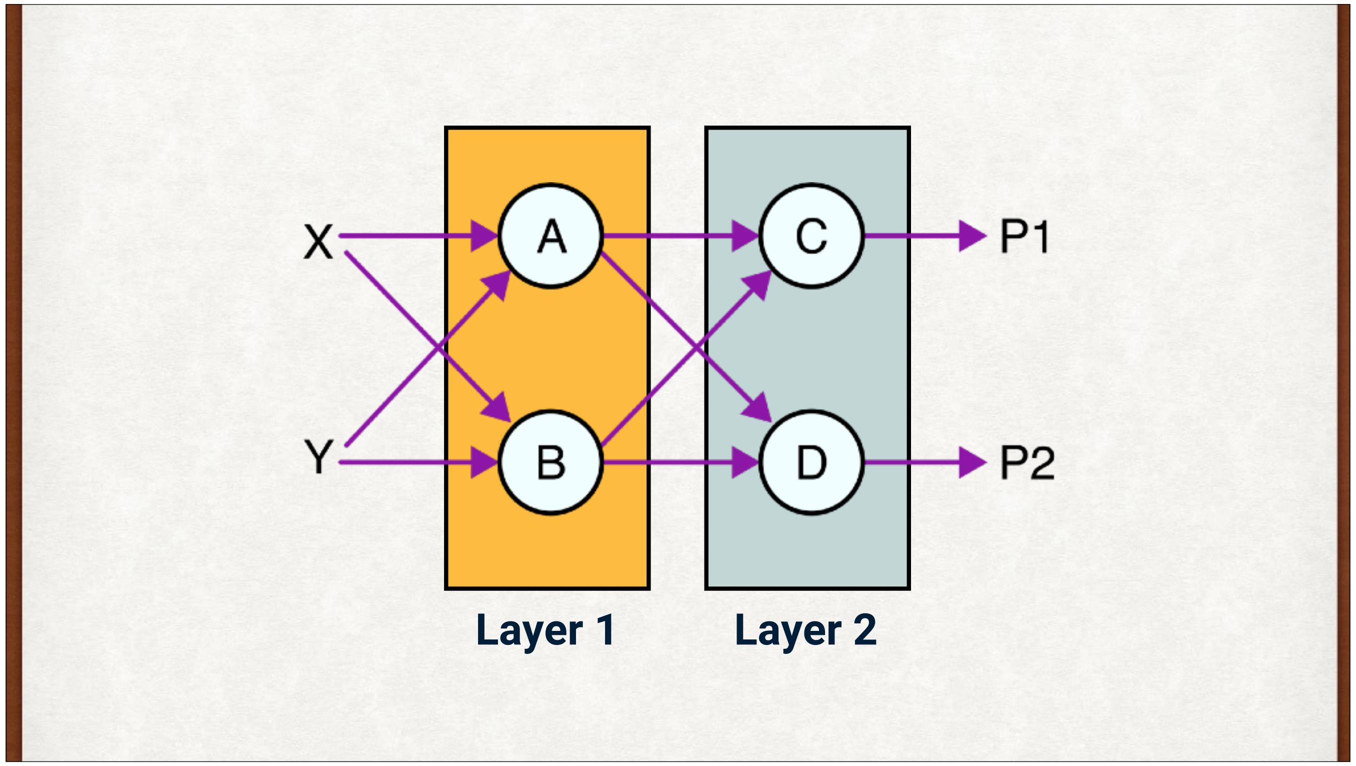
We learn when we make an error.



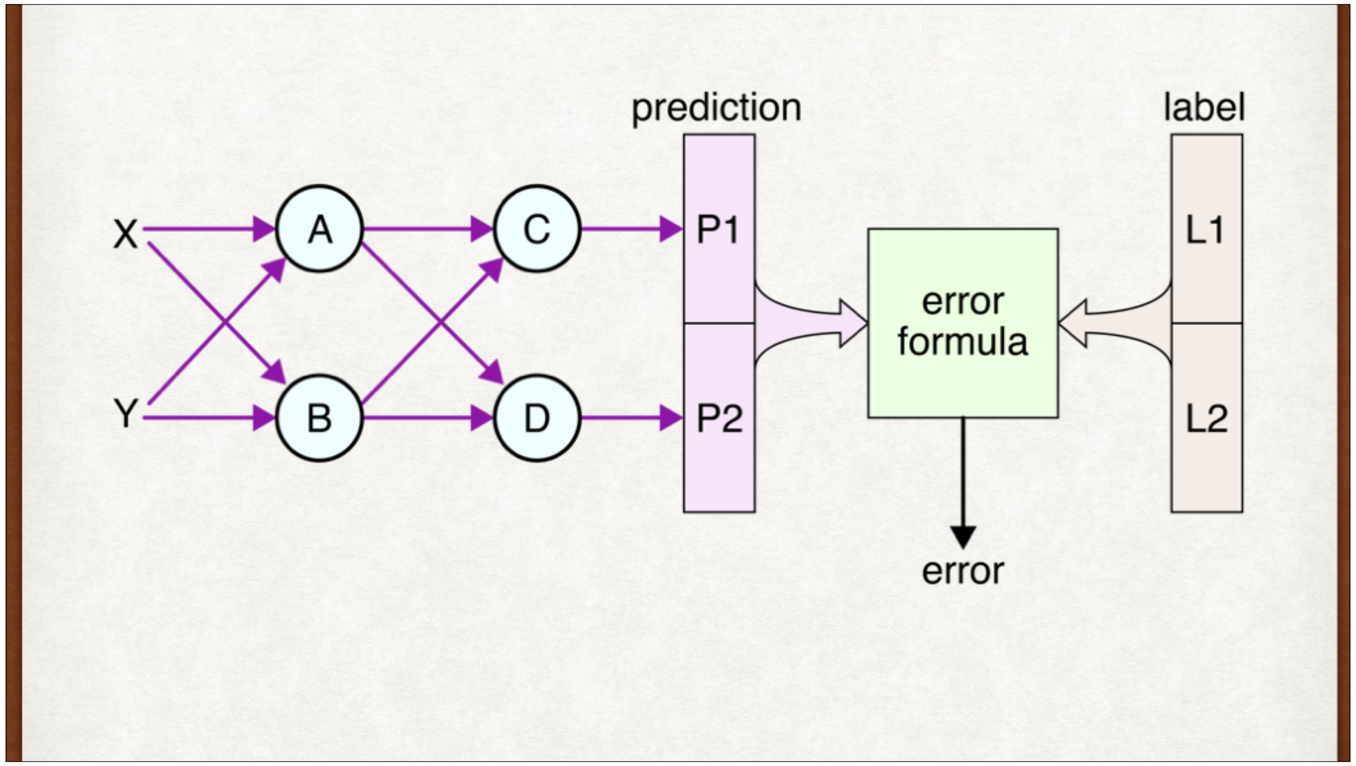
A tiny neural network. Let’s imagine there’s a softmax at the end.



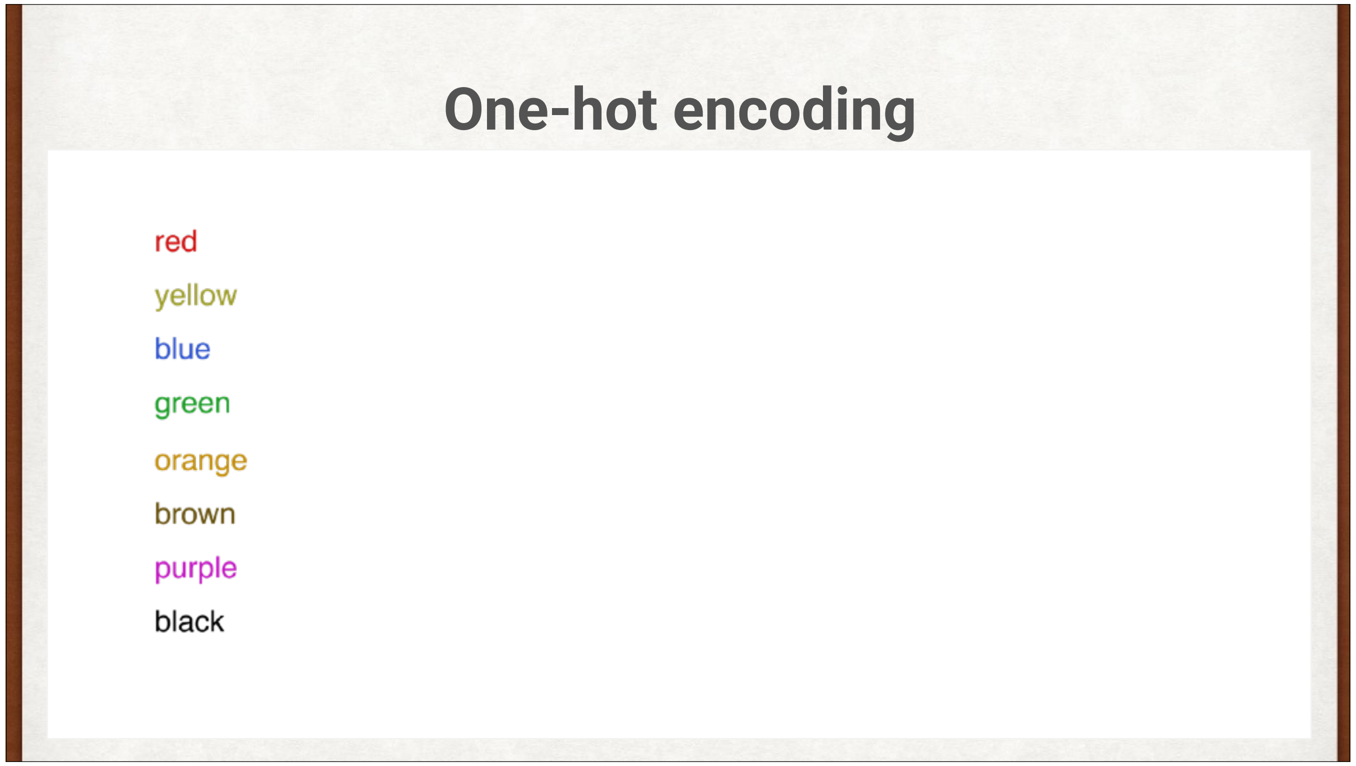
Let’s make it abstract and more general.



It’s a tiny neural network. A Multi-Layer Perceptron, or MLP.



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. Comparing a list of predictions with a desired list of predictions (the label) is part of information theory - it’s called calculating the cross-entropy.



One-hot encoding

red

yellow

blue

green

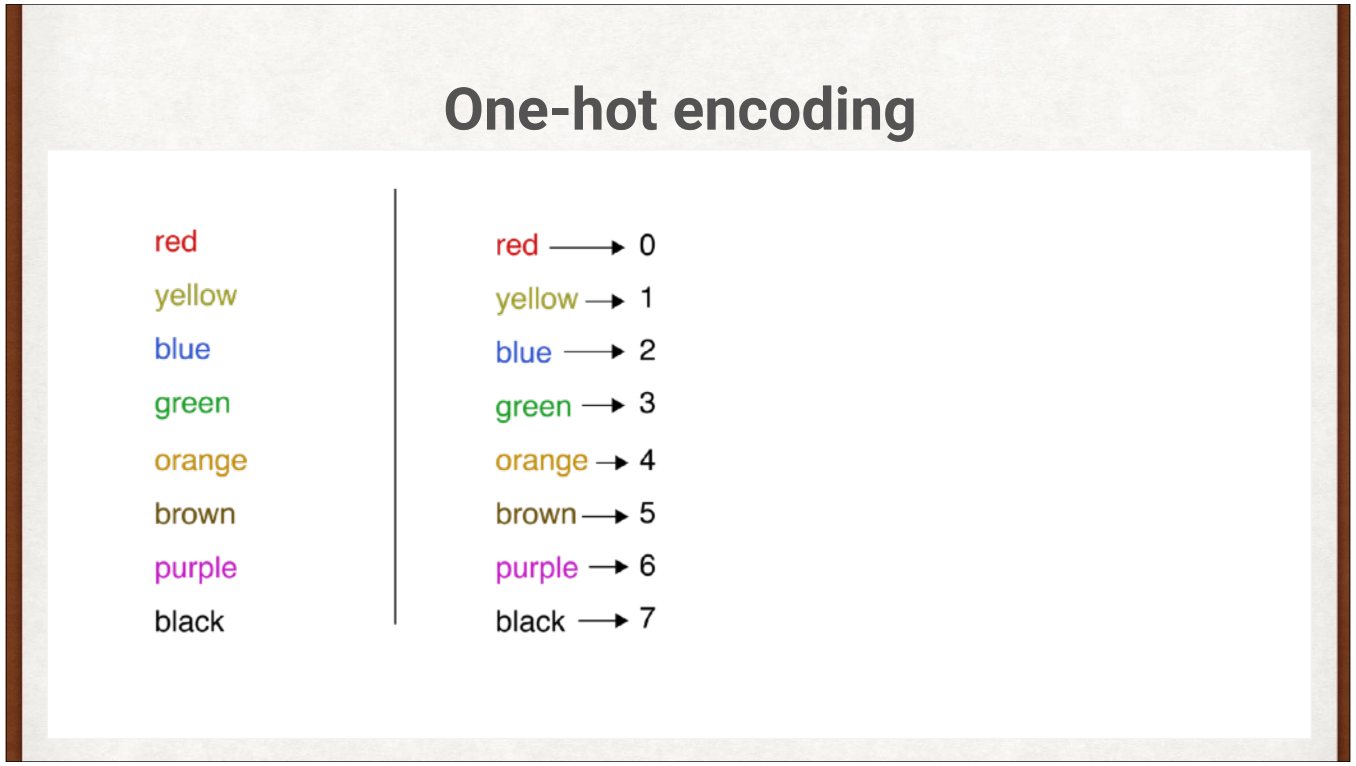
orange

brown

purple

black

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.

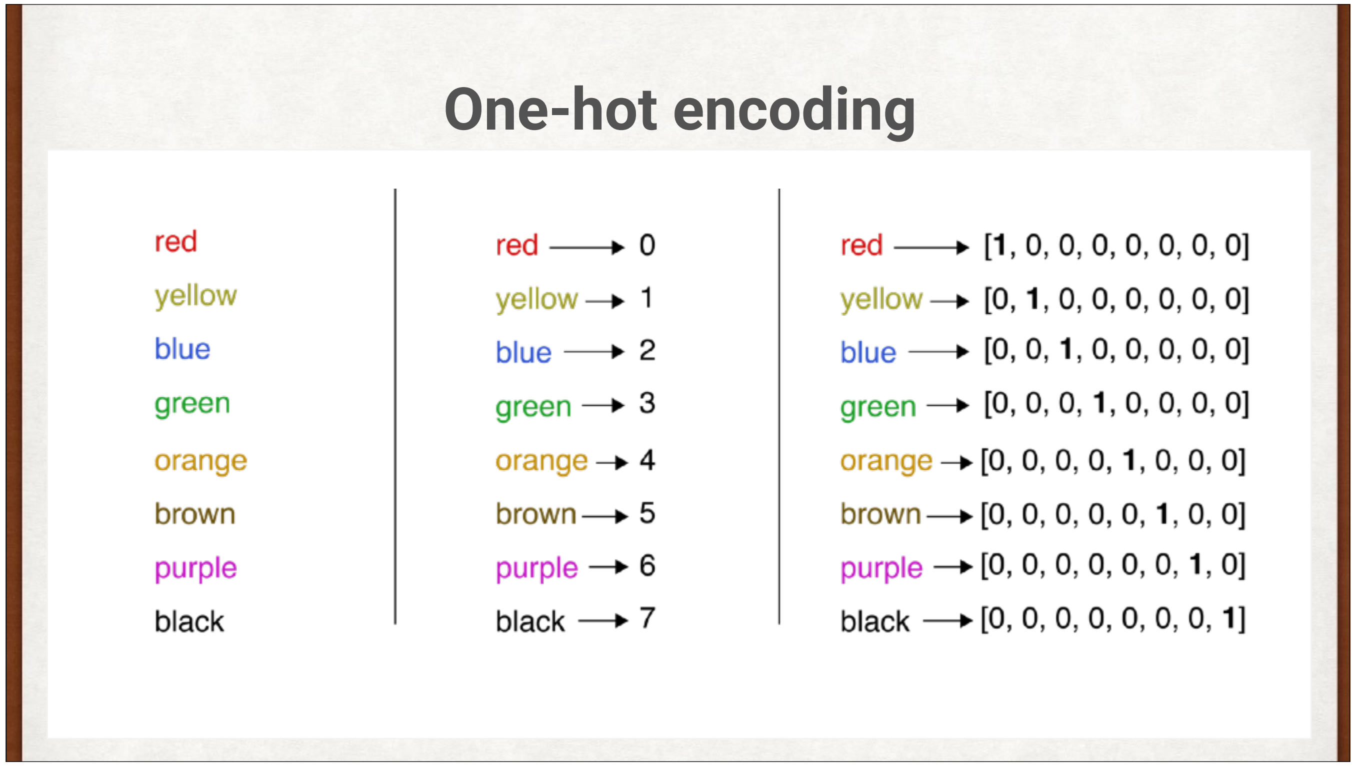


One-hot encoding

red
yellow
blue
green
orange
brown
purple
black

red → 0
yellow → 1
blue → 2
green → 3
orange → 4
brown → 5
purple → 6
black → 7

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.



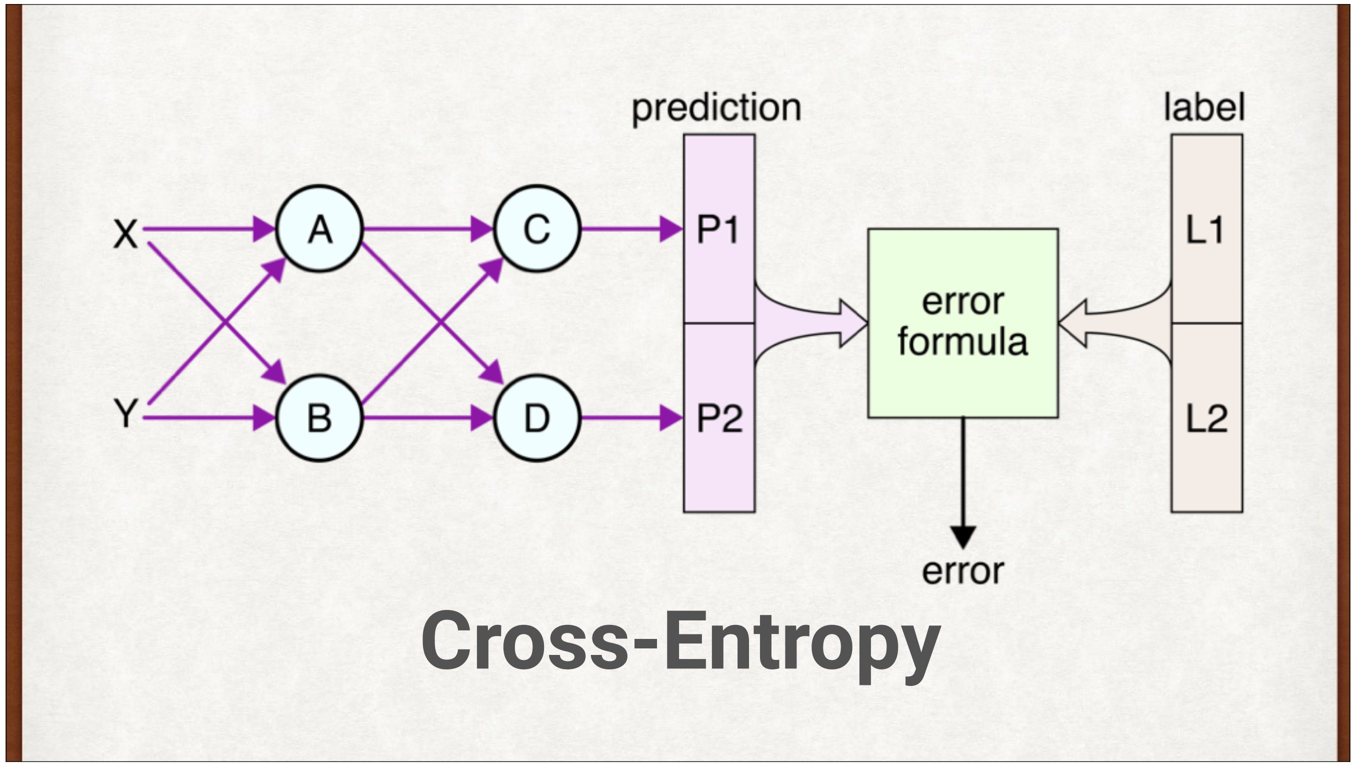
One-hot encoding

red
yellow
blue
green
orange
brown
purple
black

red → 0
yellow → 1
blue → 2
green → 3
orange → 4
brown → 5
purple → 6
black → 7

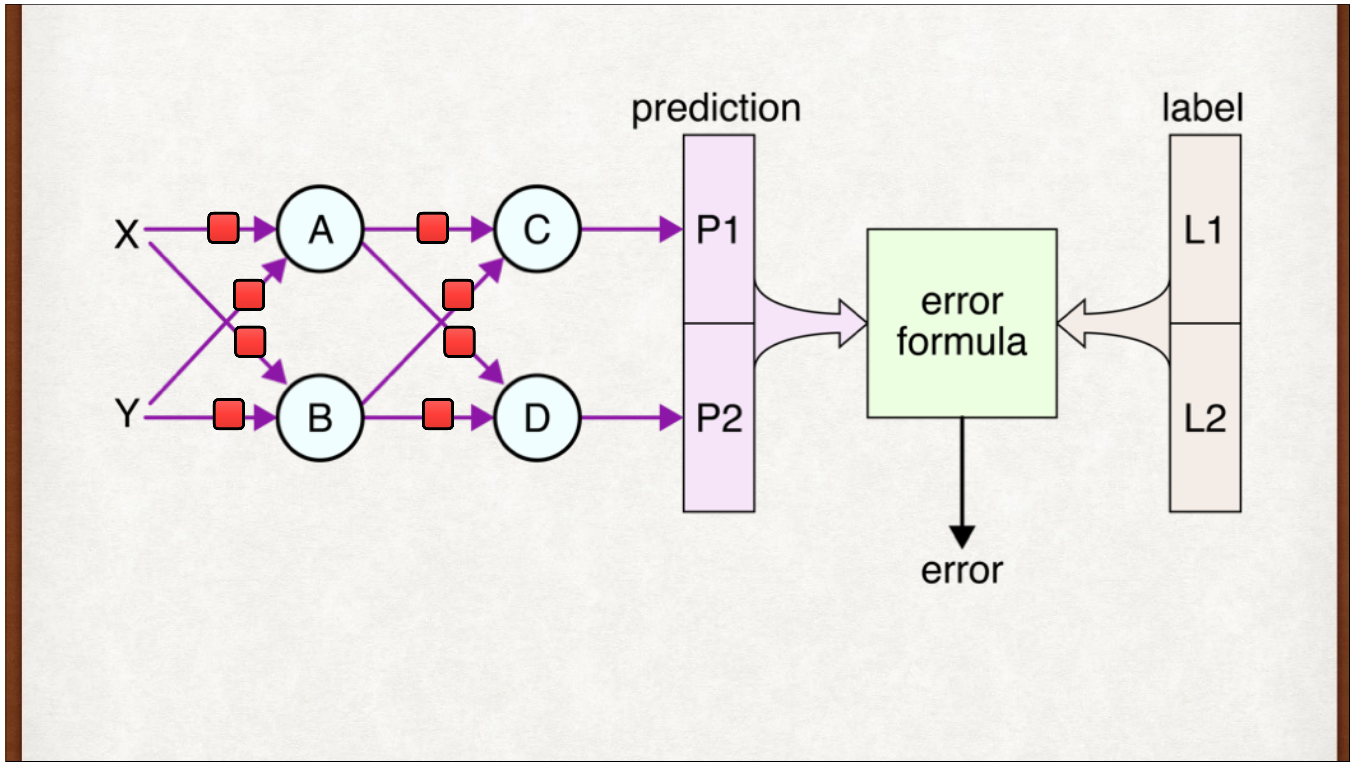
red → [**1**, 0, 0, 0, 0, 0, 0, 0]
yellow → [0, **1**, 0, 0, 0, 0, 0, 0]
blue → [0, 0, **1**, 0, 0, 0, 0, 0]
green → [0, 0, 0, **1**, 0, 0, 0, 0]
orange → [0, 0, 0, 0, **1**, 0, 0, 0]
brown → [0, 0, 0, 0, 0, **1**, 0, 0]
purple → [0, 0, 0, 0, 0, 0, **1**, 0]
black → [0, 0, 0, 0, 0, 0, 0, **1**]

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.

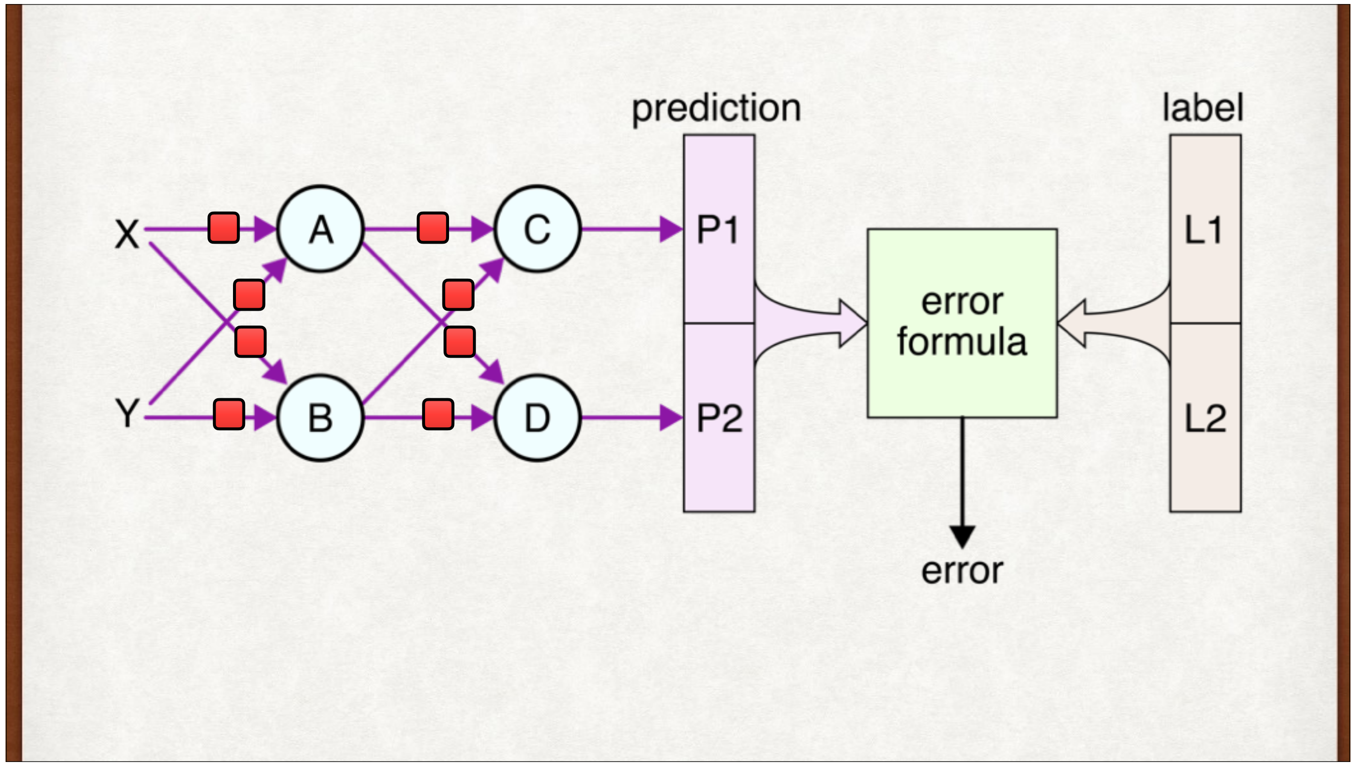


Cross-Entropy

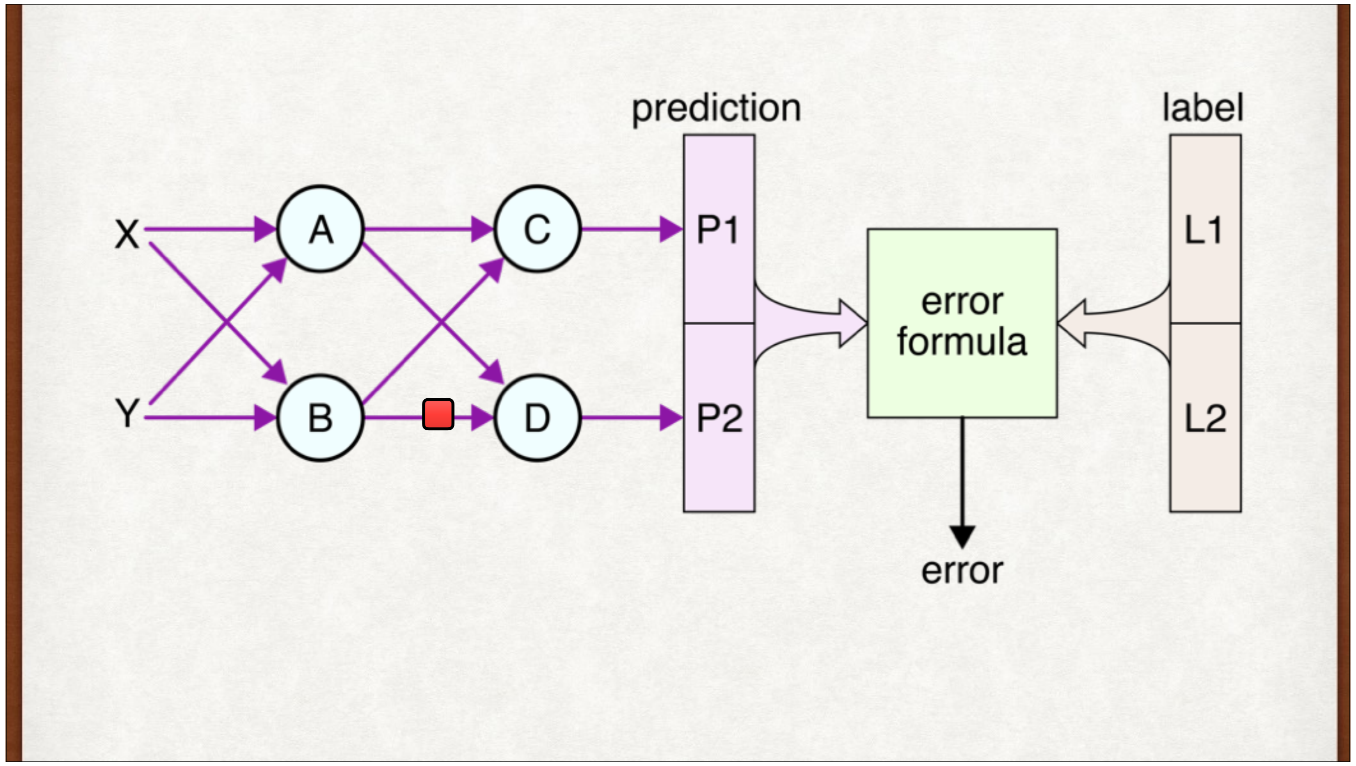
The output of the net is the “prediction”, here for two classes. Let’s suppose that we have softmax applied to the outputs of C and D. We compare the predicted probabilities to the correct answer in the “label” to find our error. Comparing a list of predictions with a desired list of predictions (the label) is part of information theory - it’s called calculating the cross-entropy.



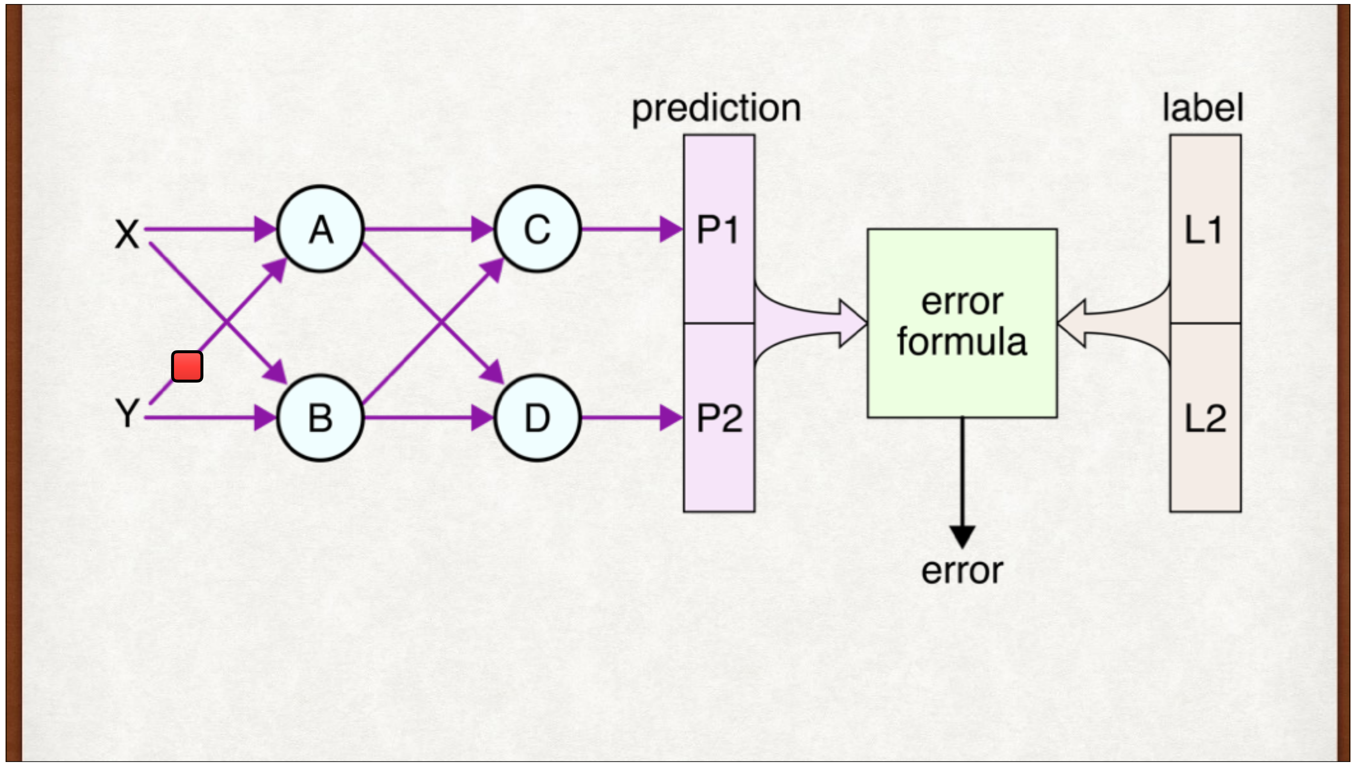
If the prediction and label don’t match, the network made an error. How do we use this error curve/surface stuff to change the weights in the network?



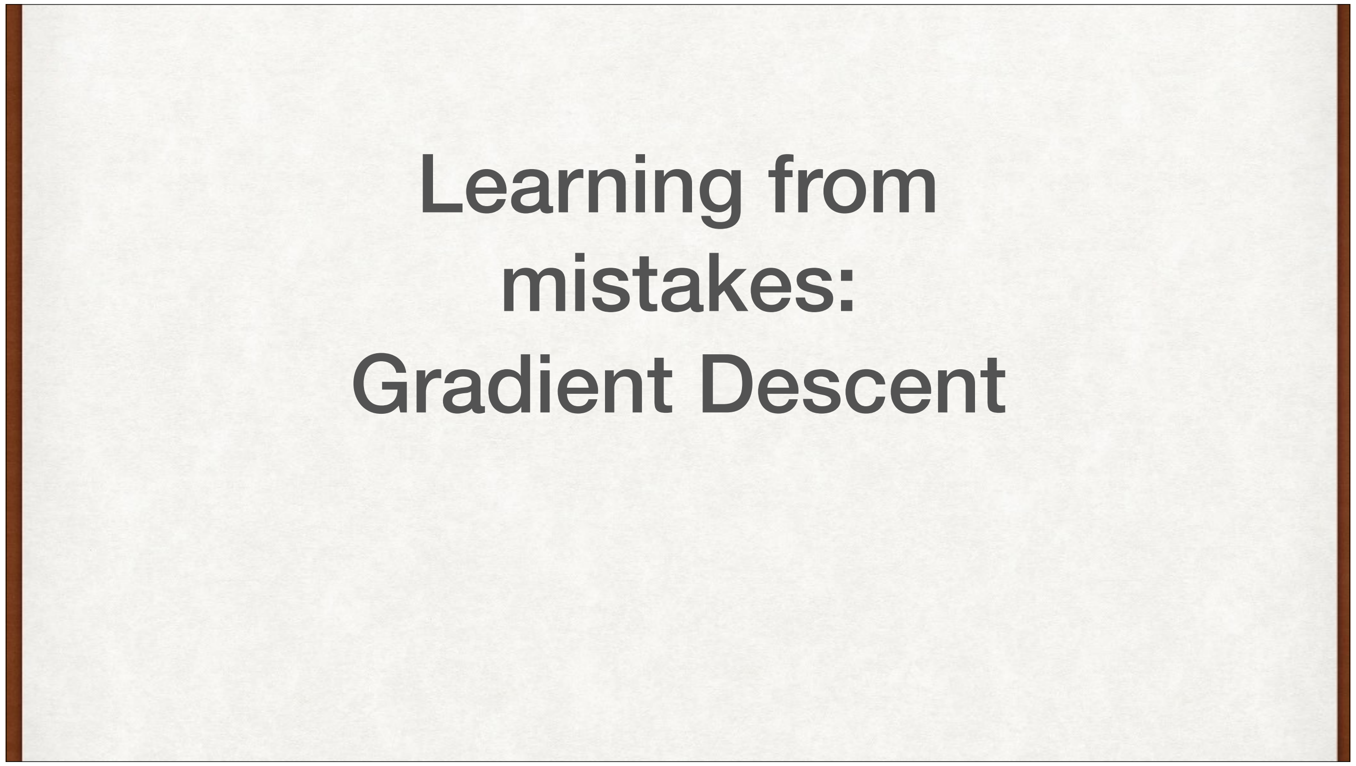
All we can do to improve the network is to change the values of the weights.



Pick this one weight. Now keep everything fixed, including X and Y, but change the weight, and see what happens to the error.

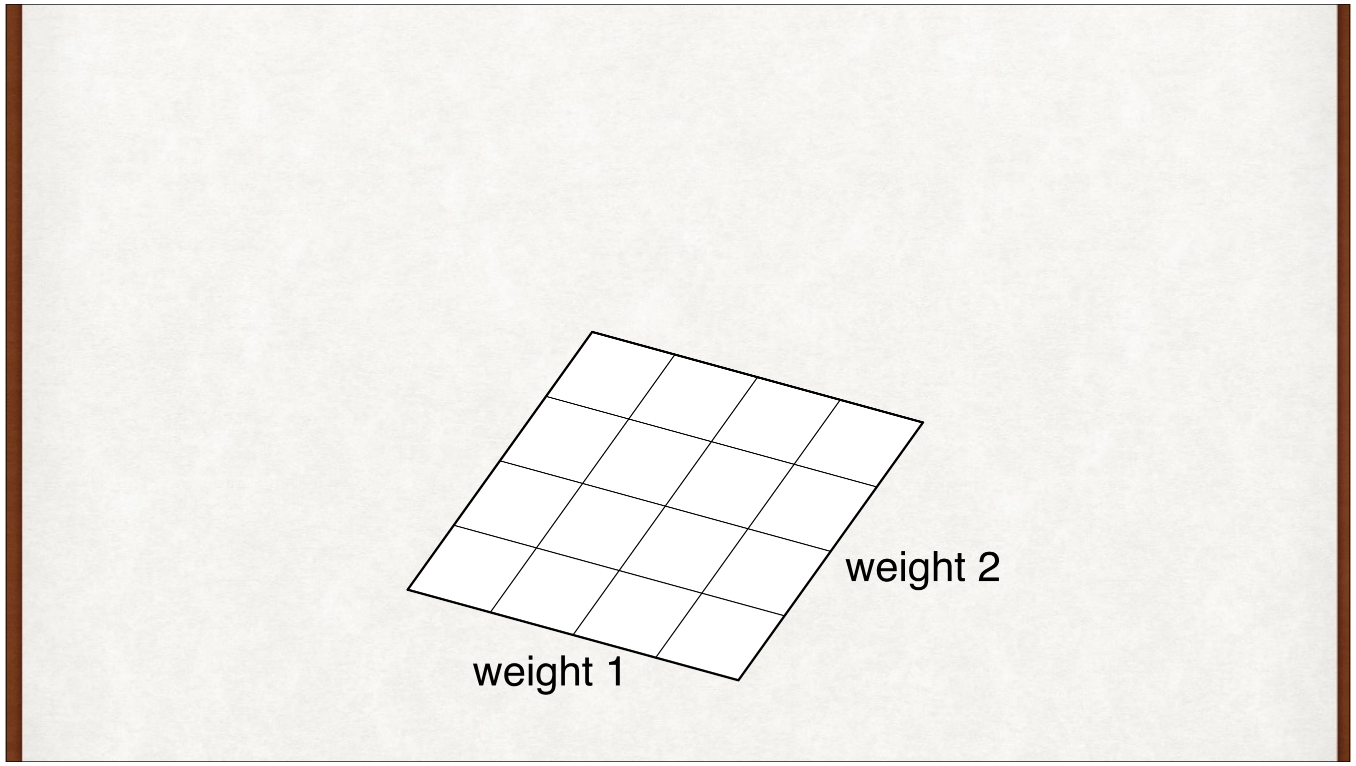


Repeat for this weight.

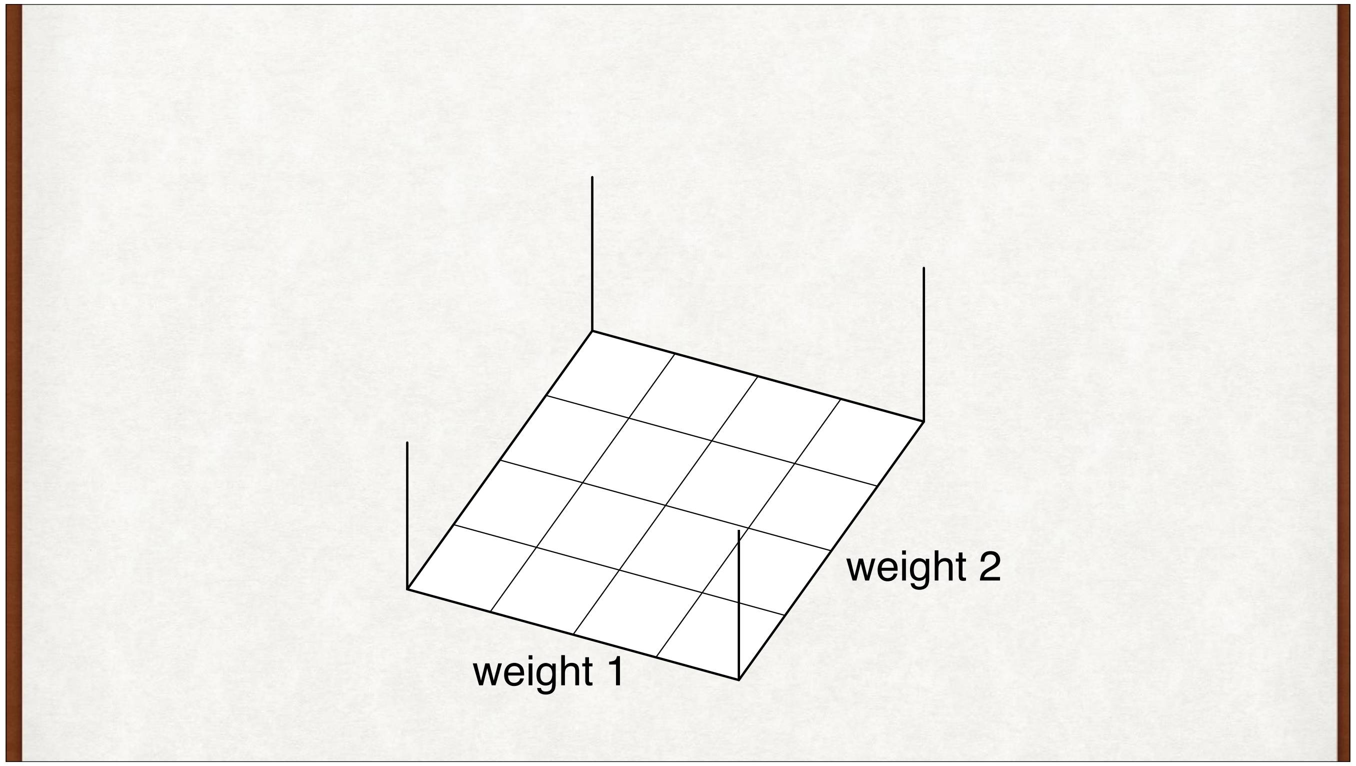


Learning from mistakes: Gradient Descent

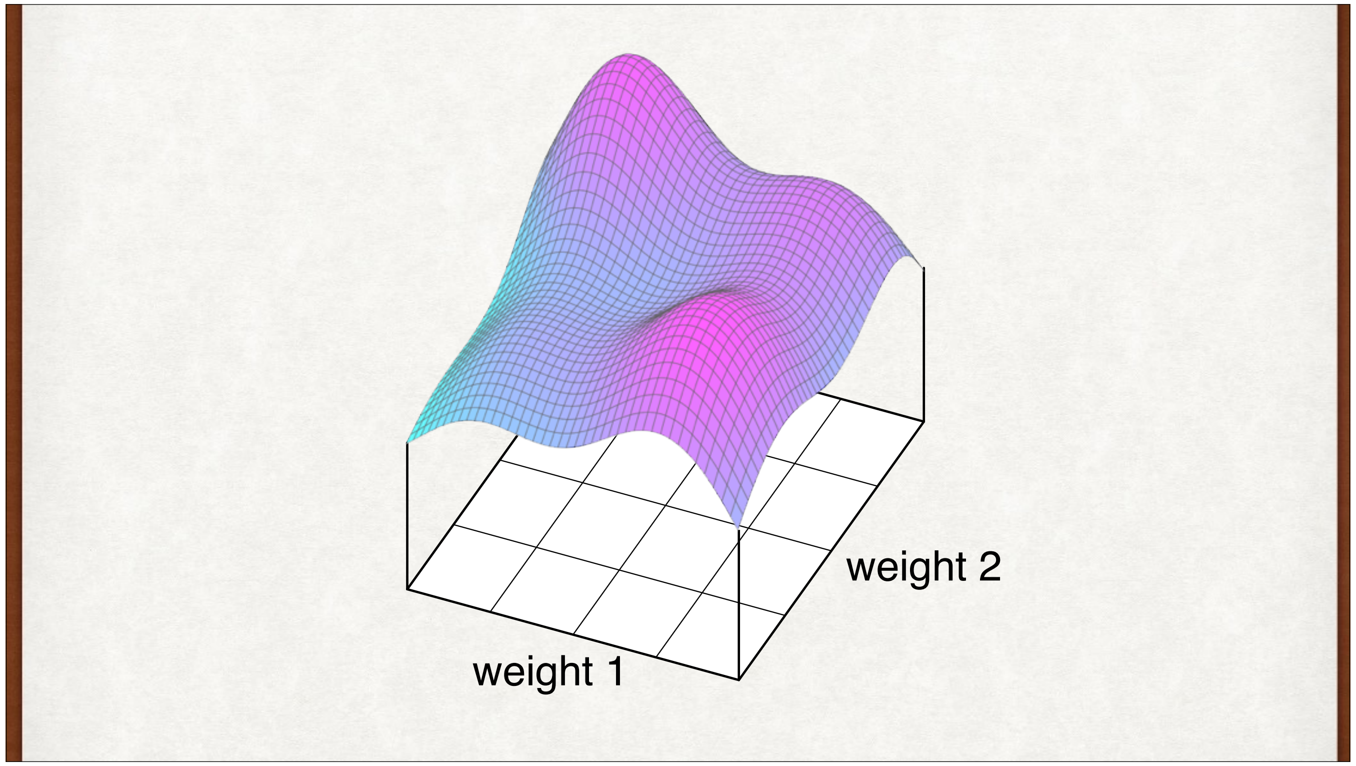
Learning from misteaks is a good idea. Learning from misstakes. From mistakes. We’ll look at the error as a curve, where we plug in the weights in our network, and get back an error. We’ll want to adjust the weights to make the error go down. The standard, and super useful, way to look at this is to 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).



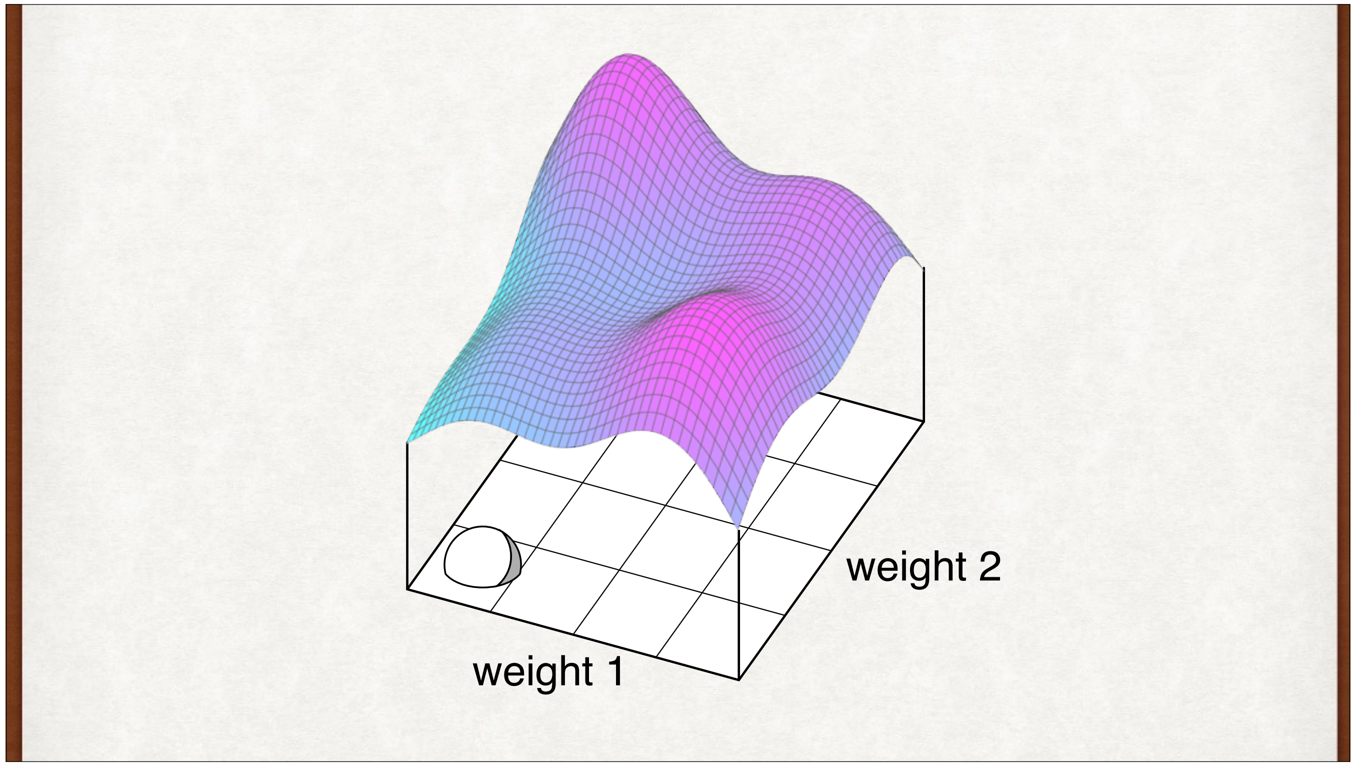
For each value of the two weights in a given neuron, we can find the error for that input, holding all other values fixed.



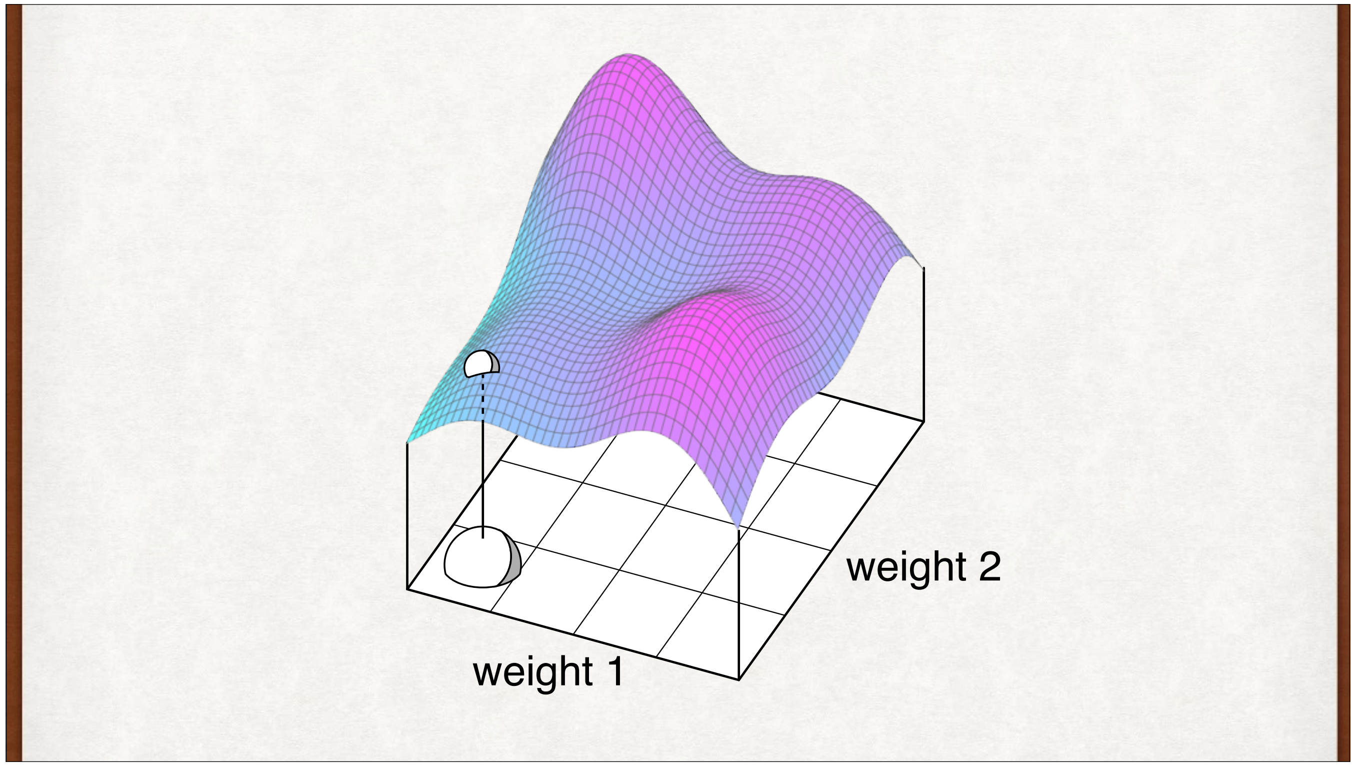
For each value of the two weights in a given neuron, we can find the error for that input, holding all other values fixed.



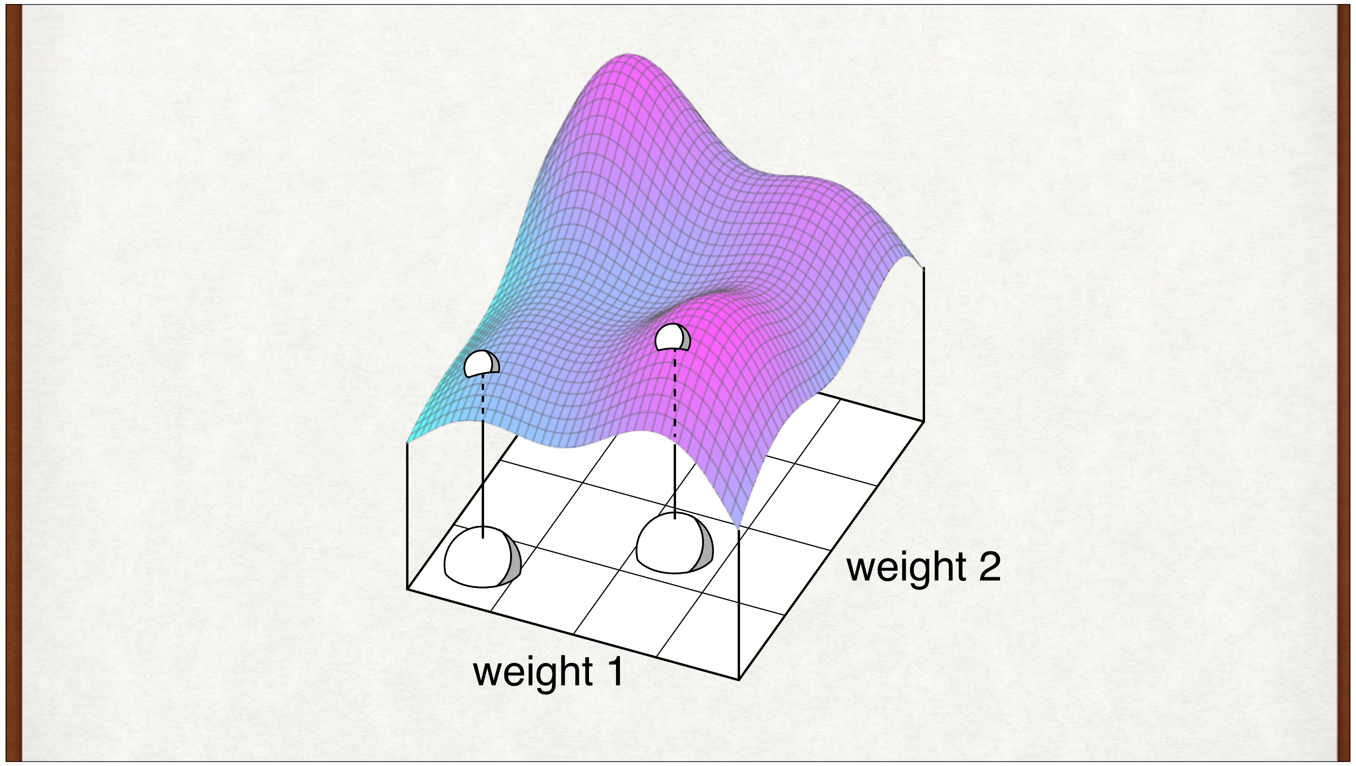
We can plot the amount of error we get for every pair of these 2 weights, creating an “error surface.”



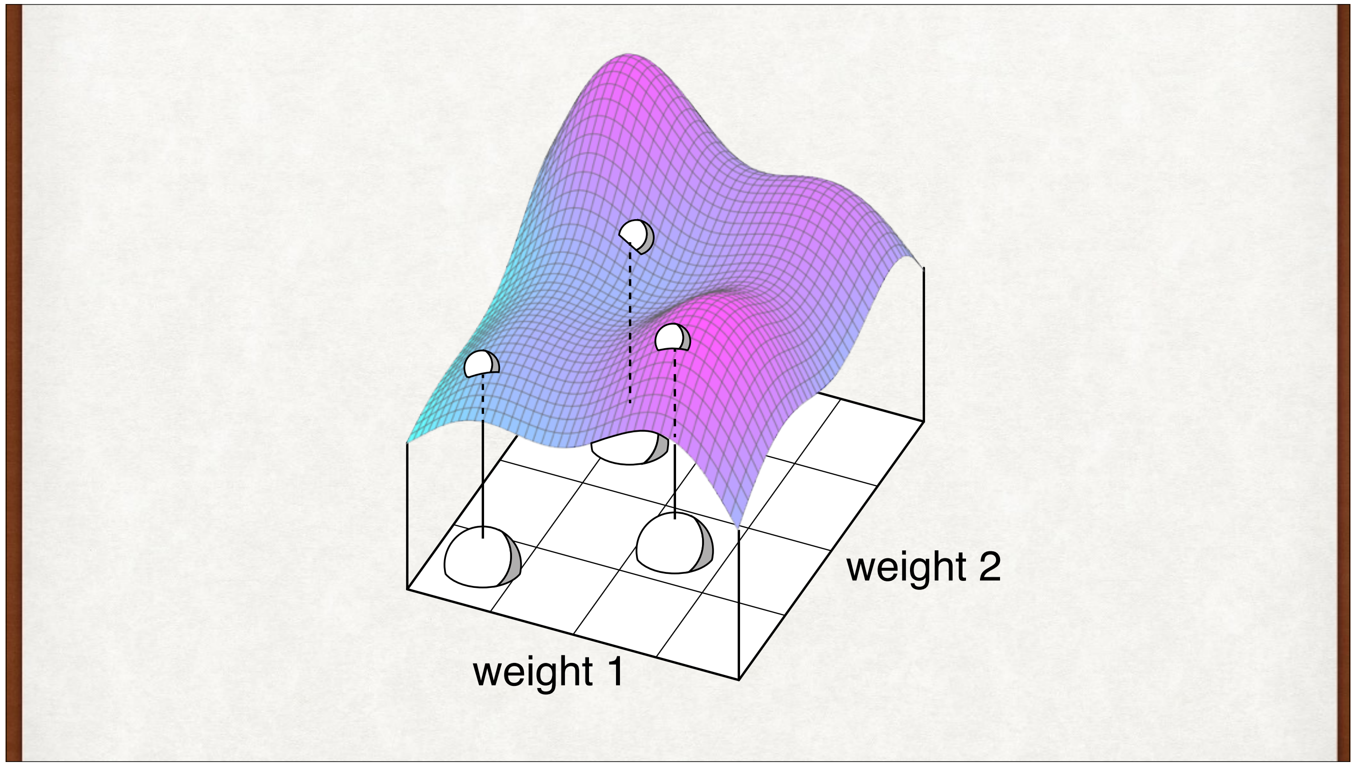
For each value of the two weights in a given neuron, we can find the error for that input, holding all other values fixed.



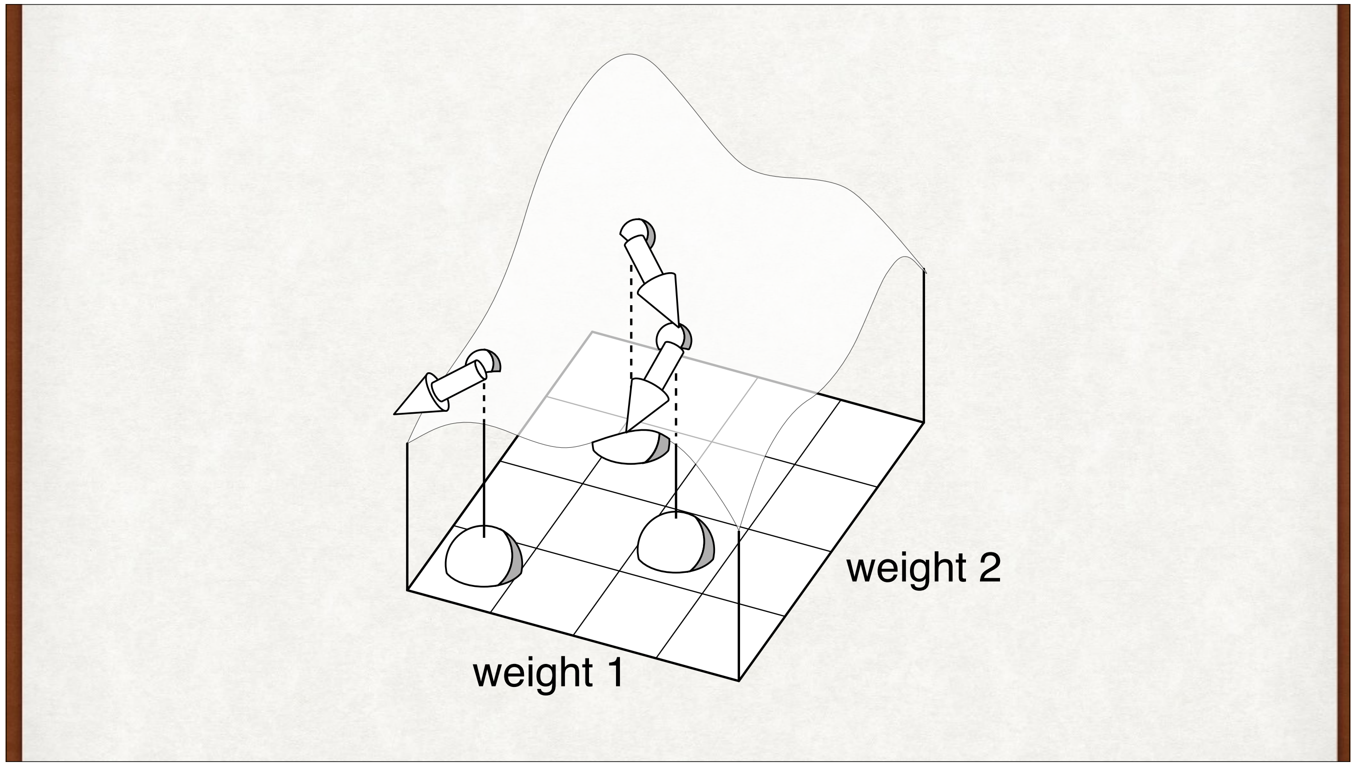
For each value of the two weights in a given neuron, we can find the error for that input, holding all other values fixed.



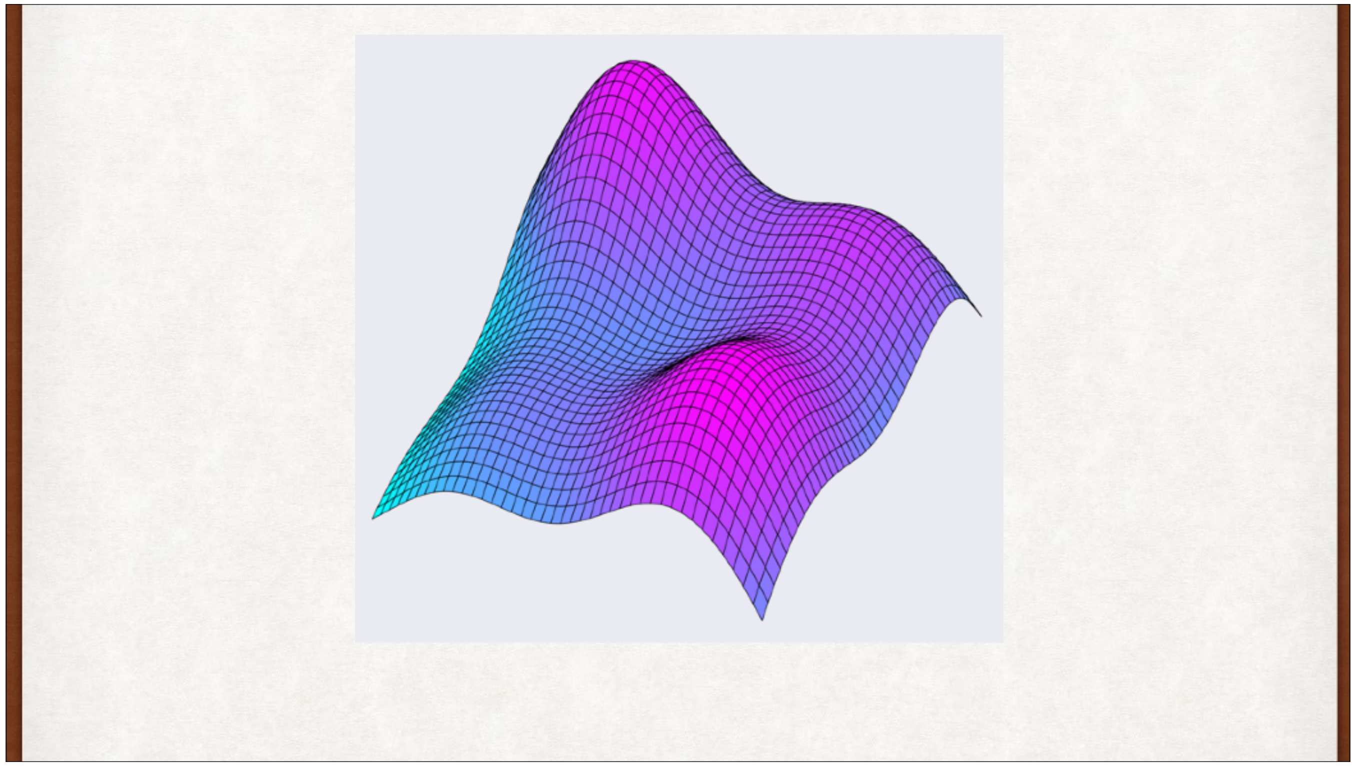
For each value of the two weights in a given neuron, we can find the error for that input, holding all other values fixed.



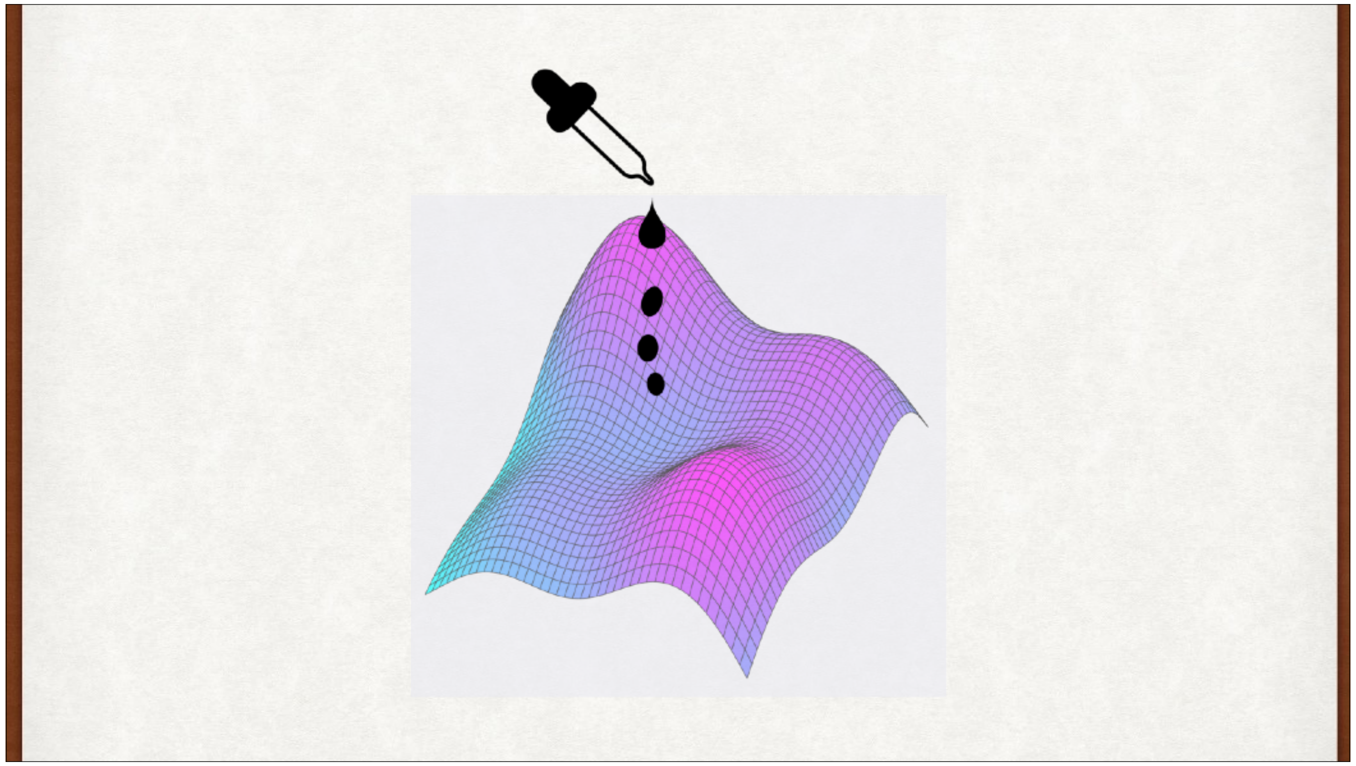
For each value of the two weights in a given neuron, we can find the error for that input, holding all other values fixed.



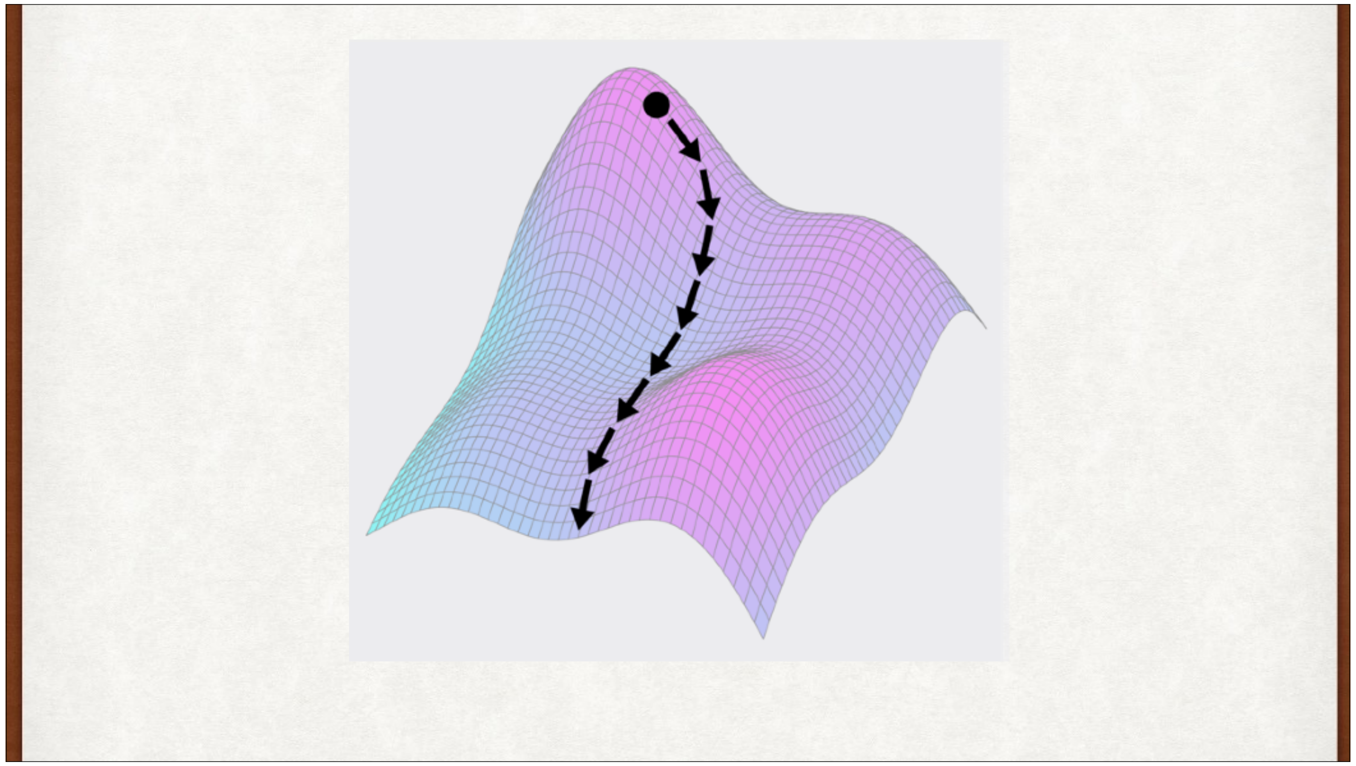
We don’t have the surface - we just have the heights at specific points.



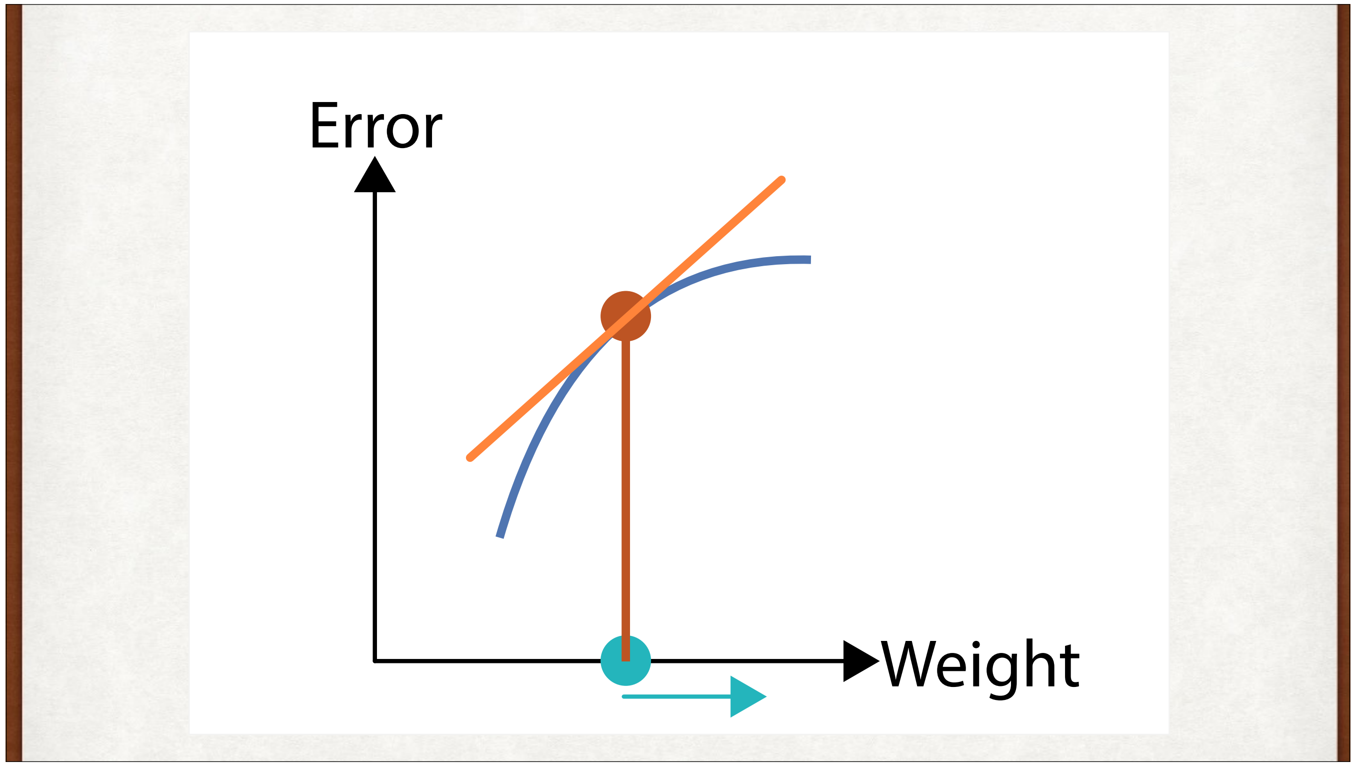
Back to the error height field by itself, **as if** we knew it.



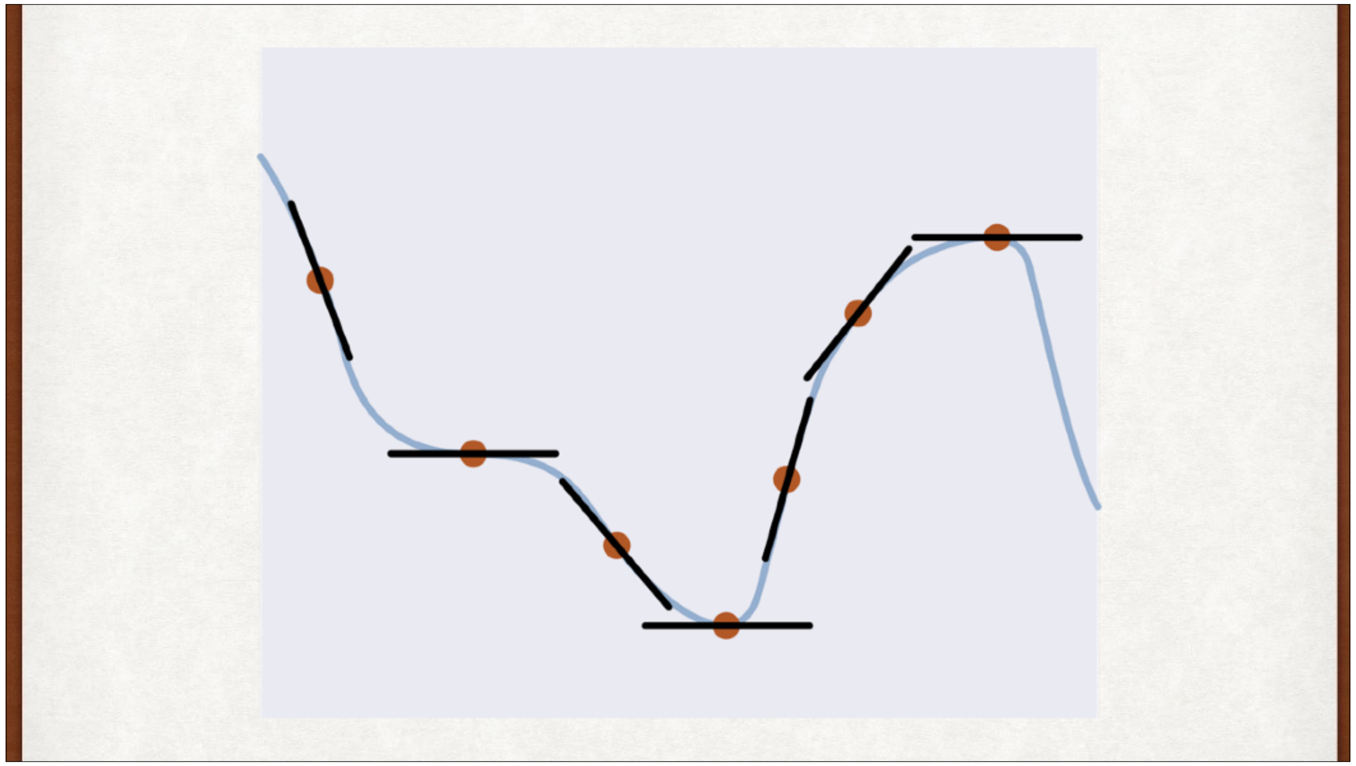
A natural example of finding the gradient is watching the path of water dropped on the surface.



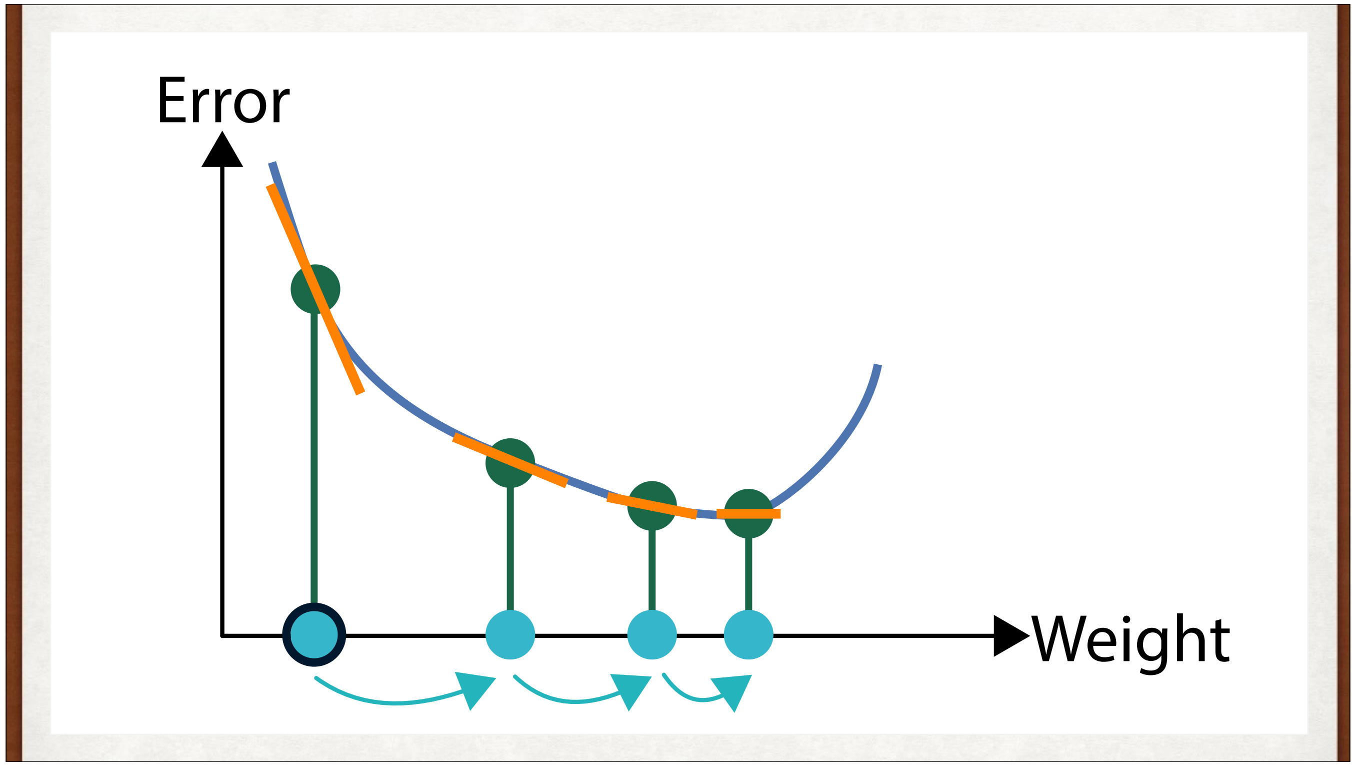
The water will follow the fastest route downhill. That’s the same as following the negative gradient (the gradient points to the direction of steepest ascent, so it’s opposite, or negative, is in the direction of steepest descent).



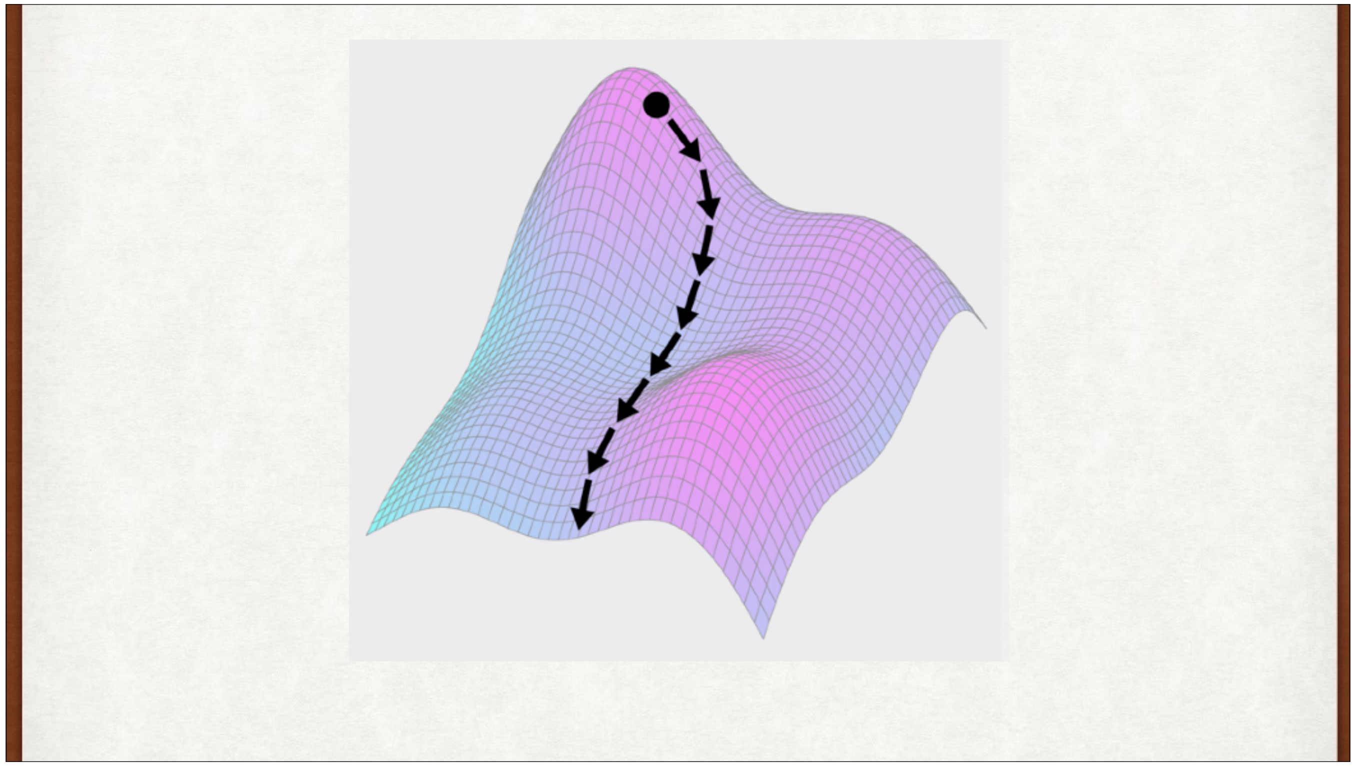
The derivative of a 2D curve can be thought of as the slope (orange) at a point on the curve.



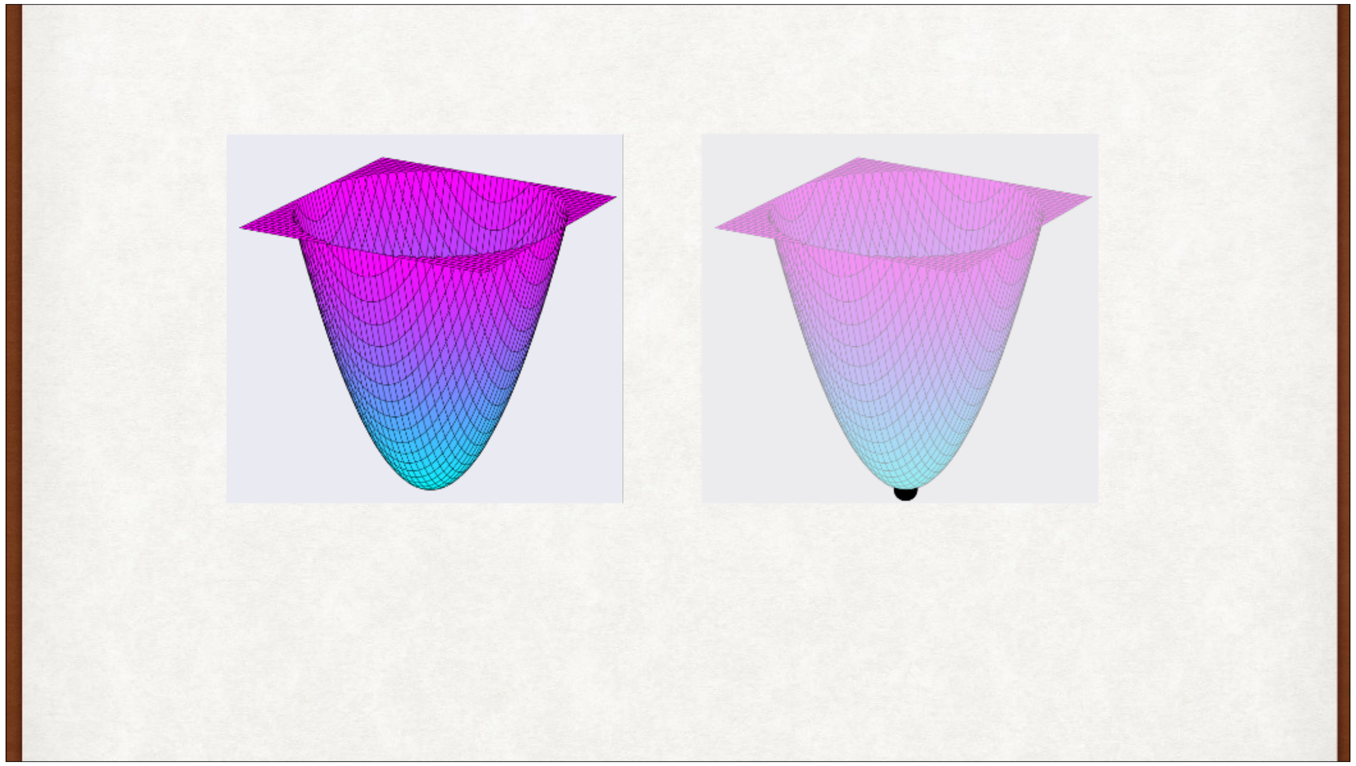
A bunch of points on a curve with their derivatives. We usually draw the length of the line to correspond to the magnitude of the derivative, but here they're all the same so they're easier to see.



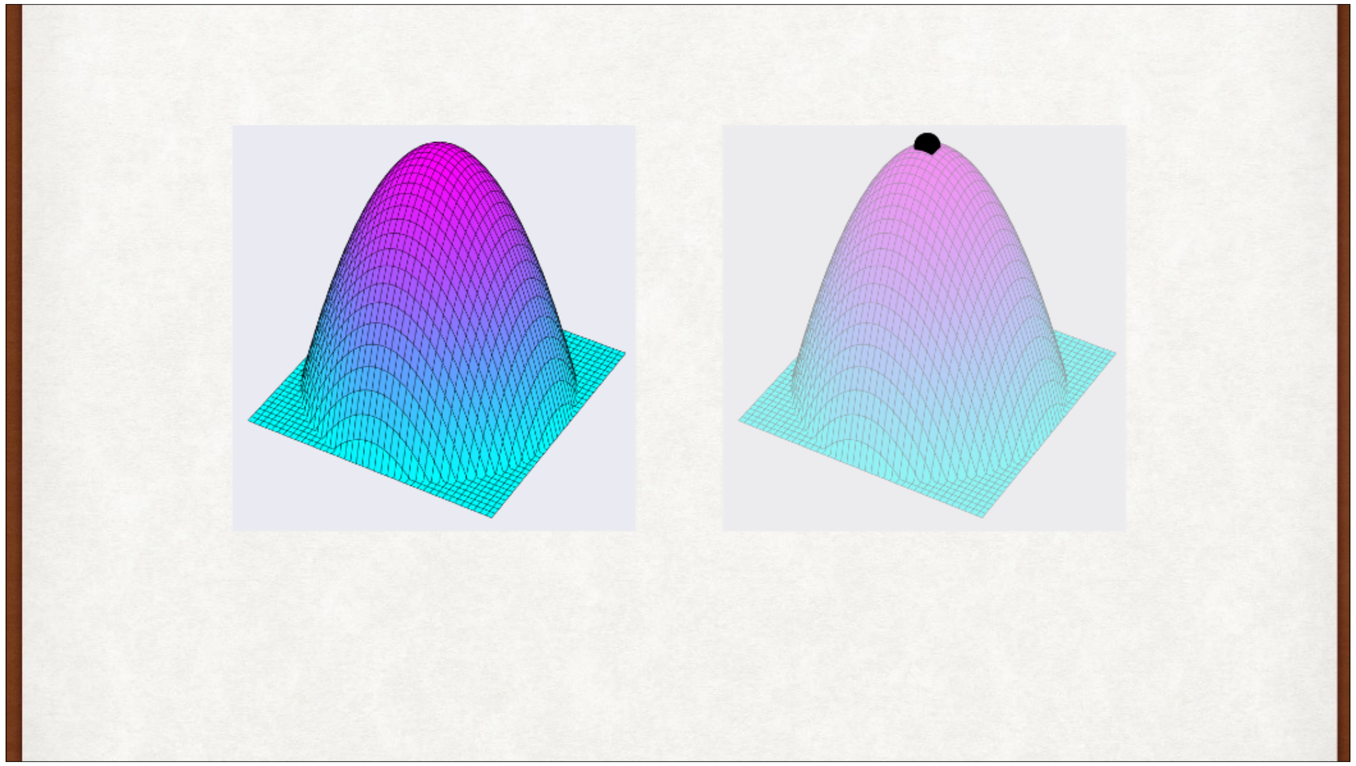
We start with the weight with a black circle around it. We find the slope of the error there. Because the slope decreases to the right, we move right, and continue this process until the slope is flat.



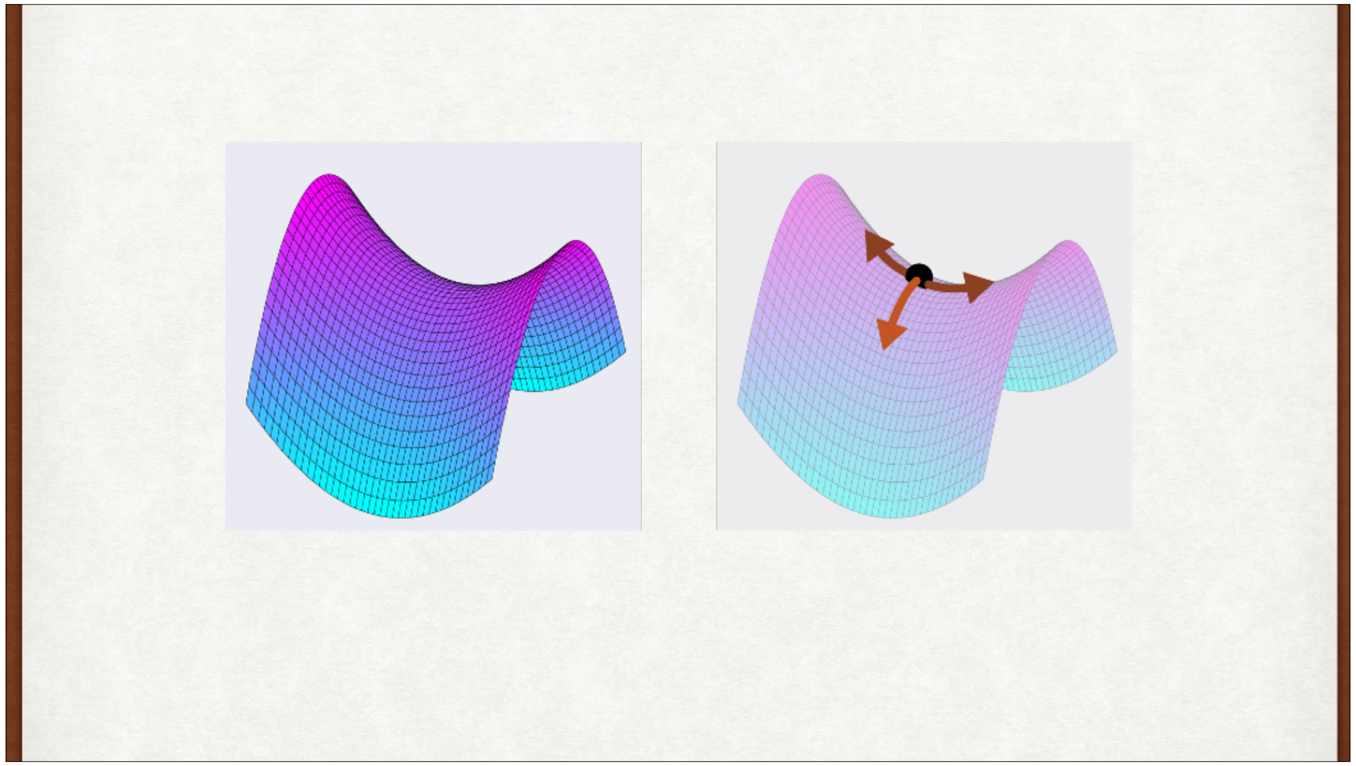
The water will follow the fastest route downhill. That’s the same as following the negative gradient (the gradient points to the direction of steepest ascent, so it’s opposite, or negative, is in the direction of steepest descent).



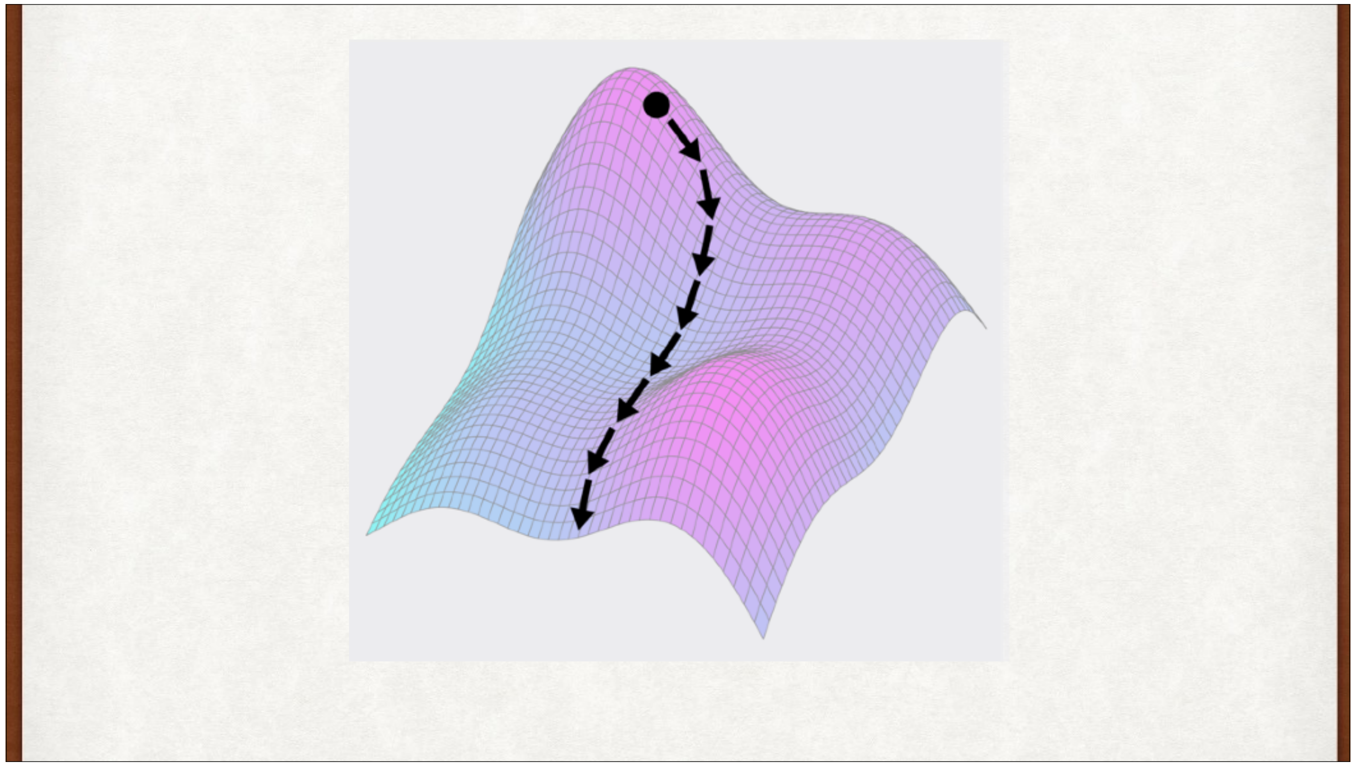
There’s no gradient at the bottom of a bowl. In that immediate neighborhood, the surface is flat.



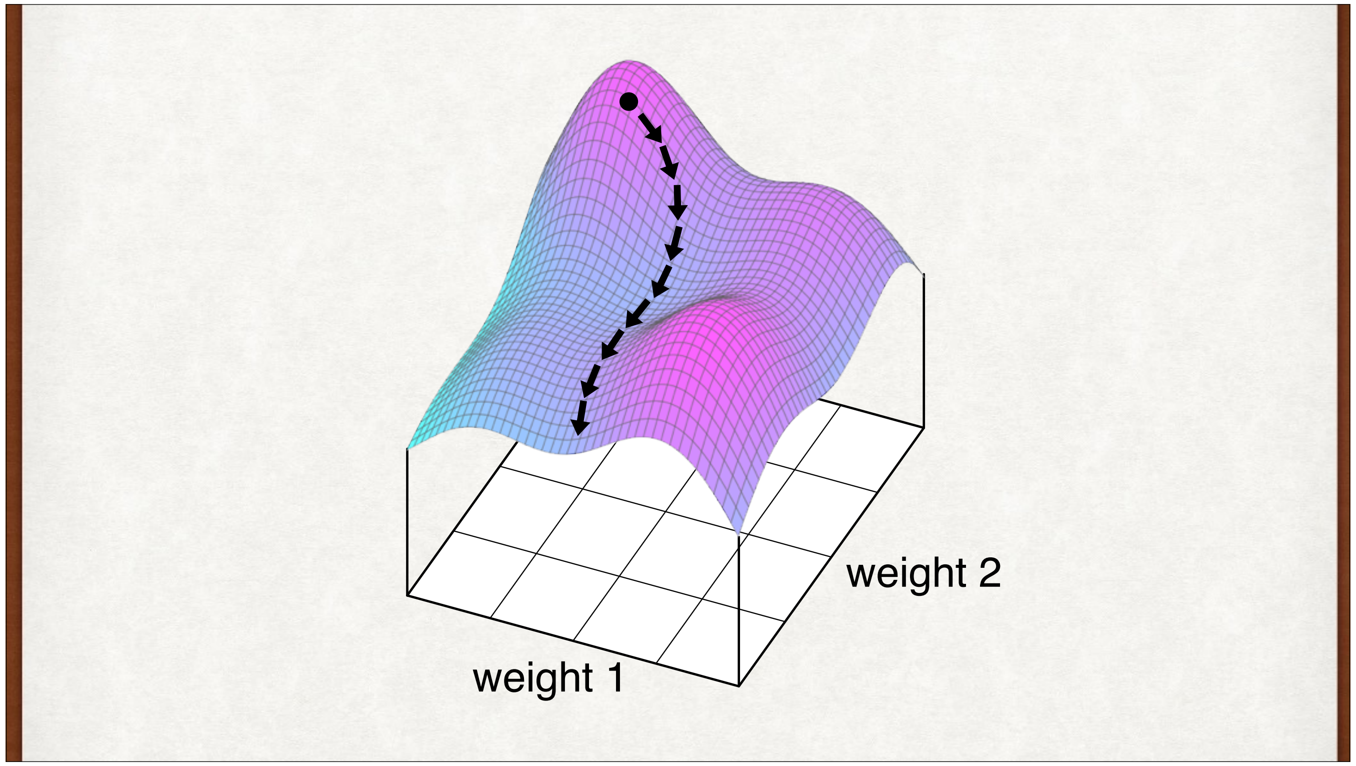
Same thing at the top of a mountain.



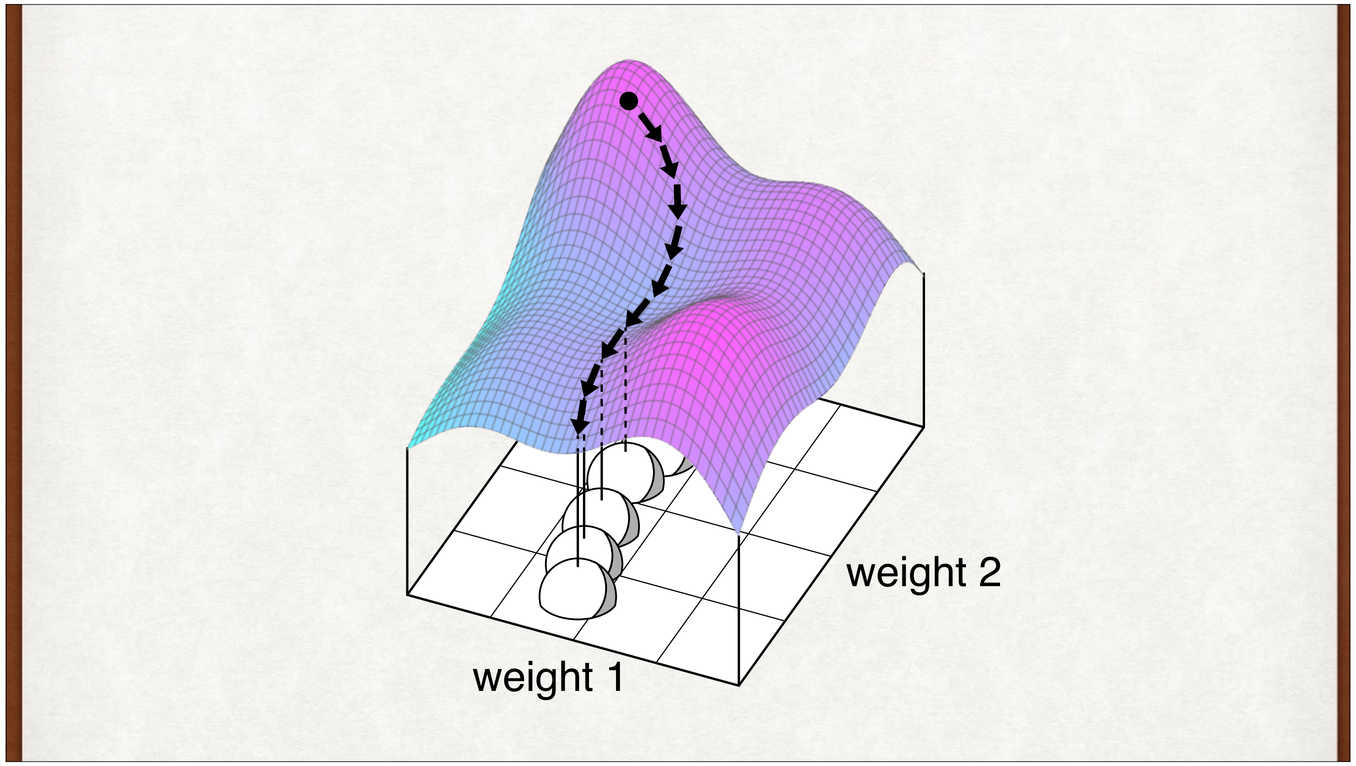
Saddles show up in 3D. They have a zero gradient right in the middle, but positive on one side and negative on the other.



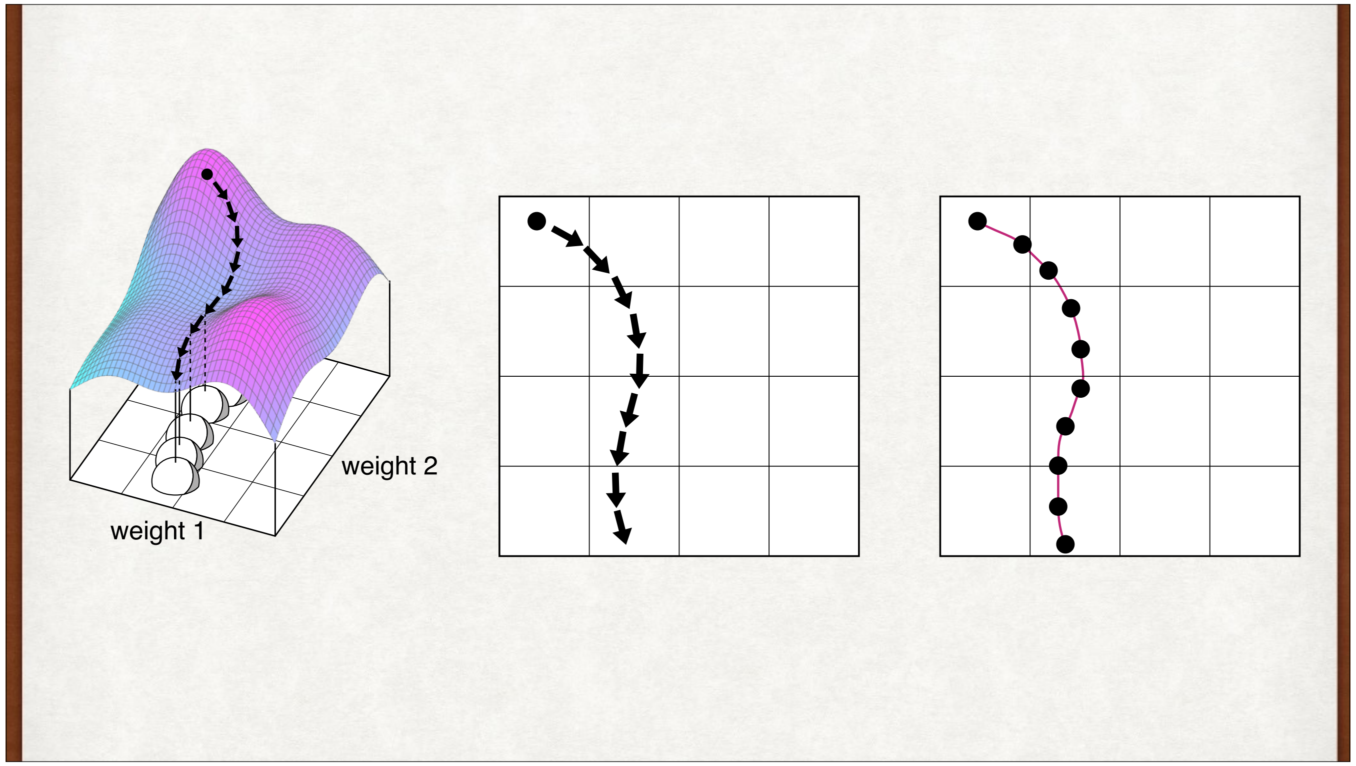
Here’s the key idea again. Follow the gradient in reverse, and you head downhill. If the height of this sheet is f(x,y), then we can find the values of x and y so that f(x,y) is at its smallest (at least in this region) by following this path. If the height of the sheet is the error in our network for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



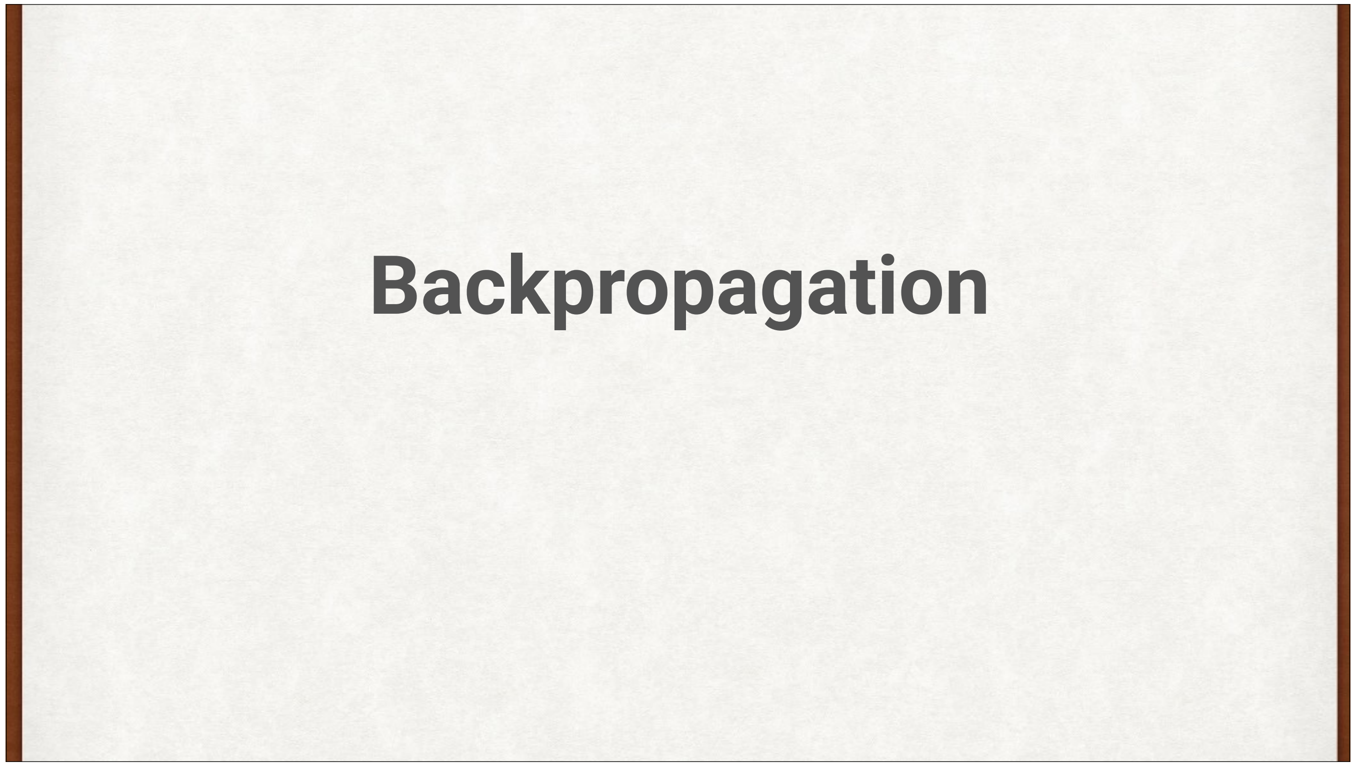
Revisiting the error, showing the path of a drop of water.



The path of the water corresponds to changes in the weights.

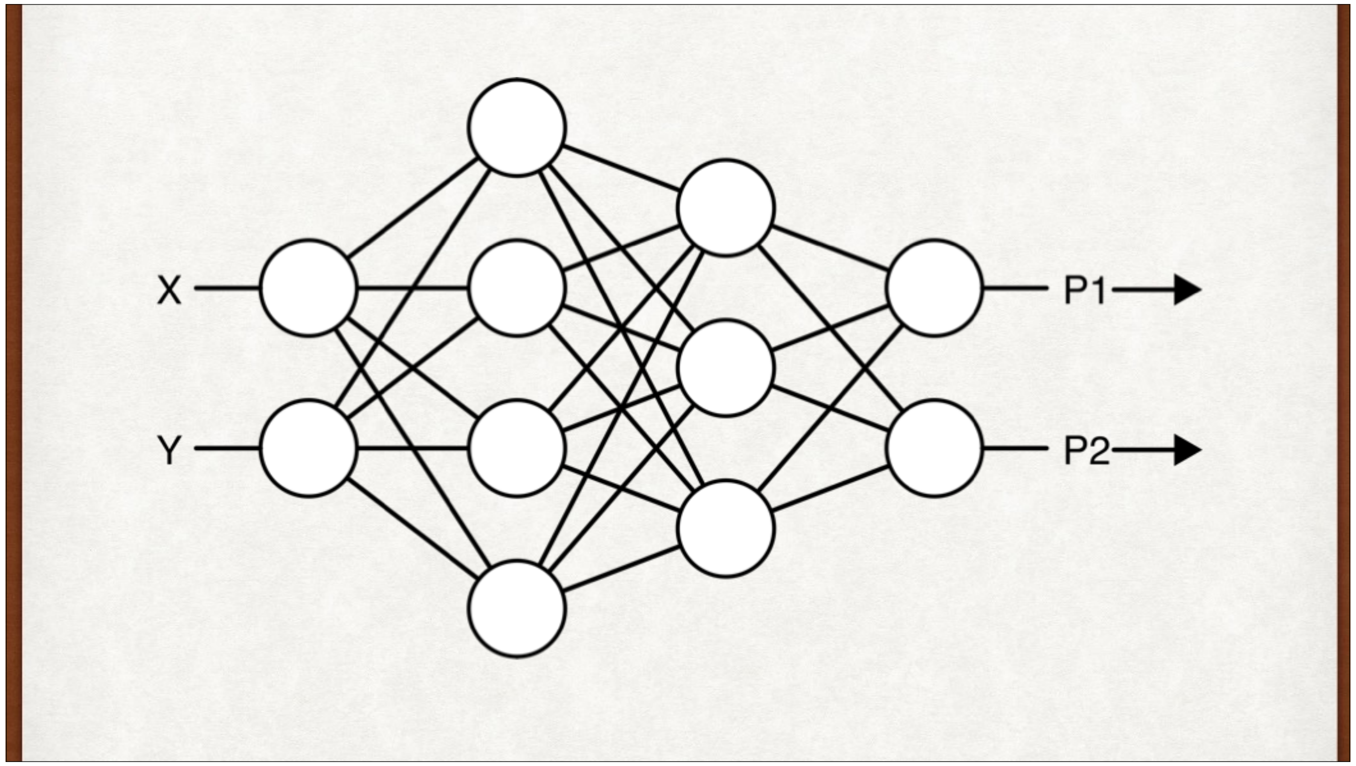


The path of the weights. The weights are what we can control, so those are the values we care about.

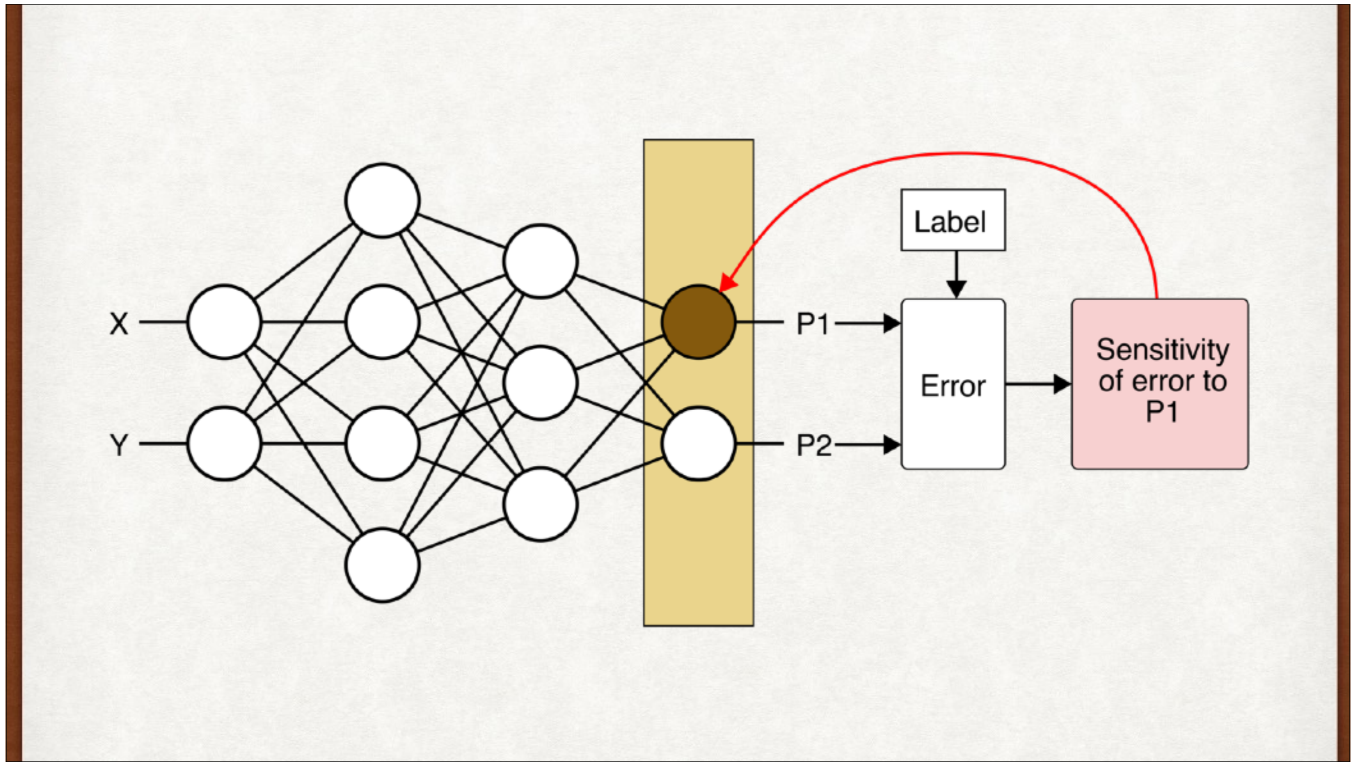


Backpropagation

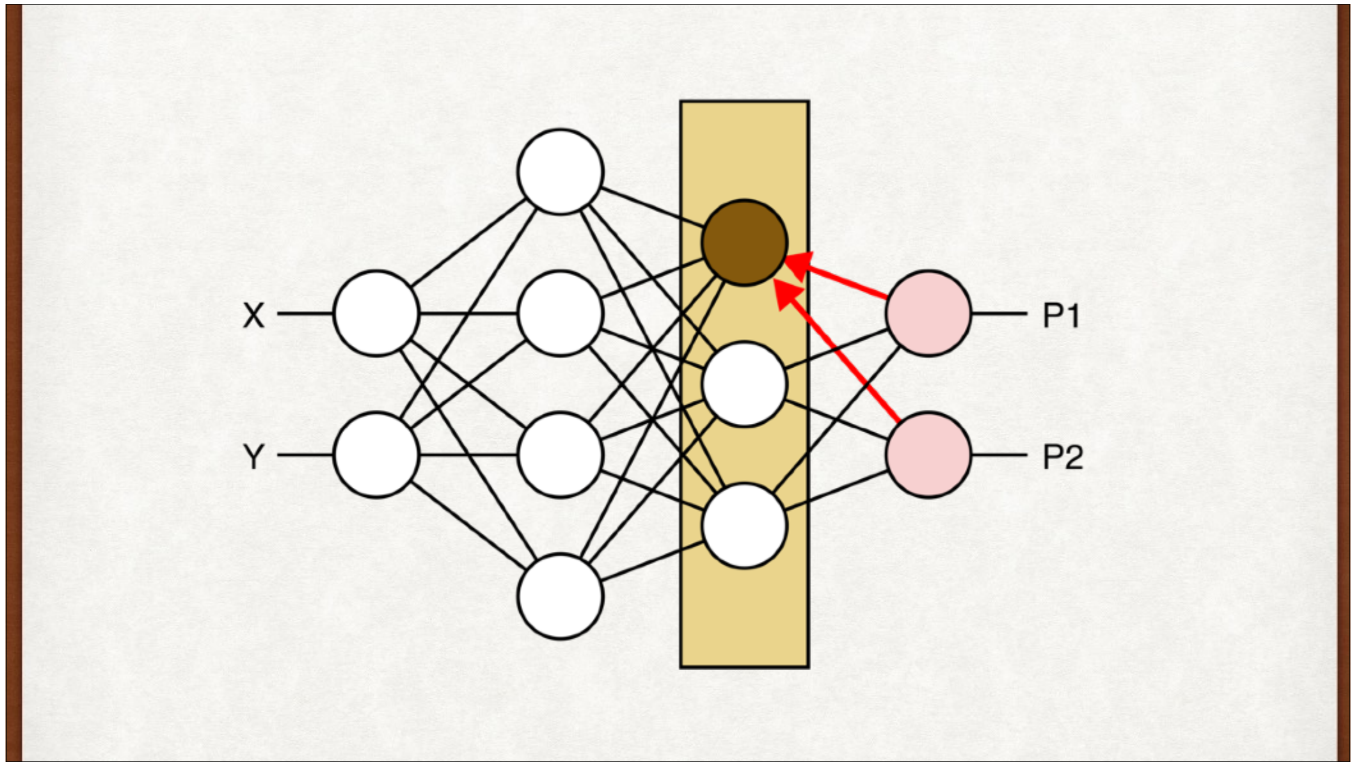
We fix errors in two steps. First, we find the gradients, which tells us whether each weight in the network should be increased or decreased. Then we use those gradients to update the weight values. For the first step, we use this algorithm: backpropagation. It’s simple but subtle. It’s well worth knowing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



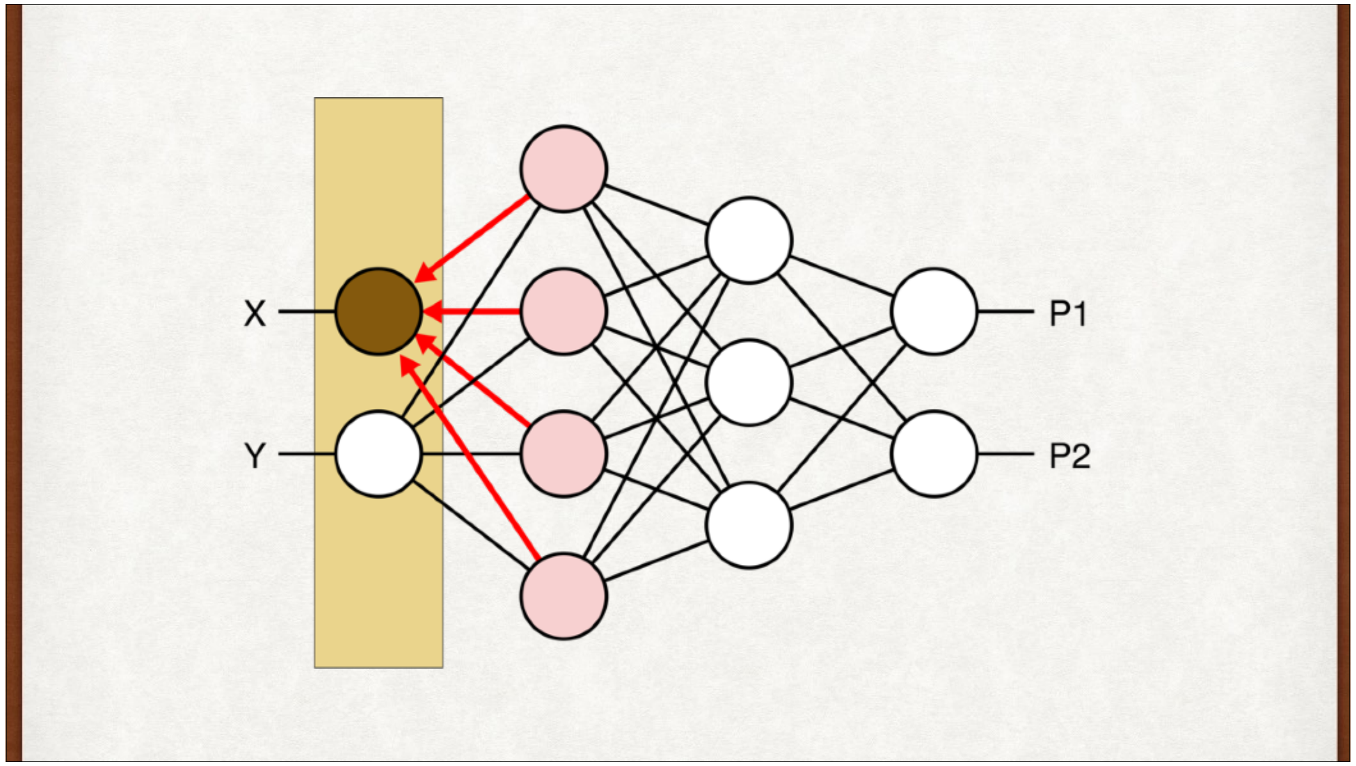
Suppose this is our 4-layer network.



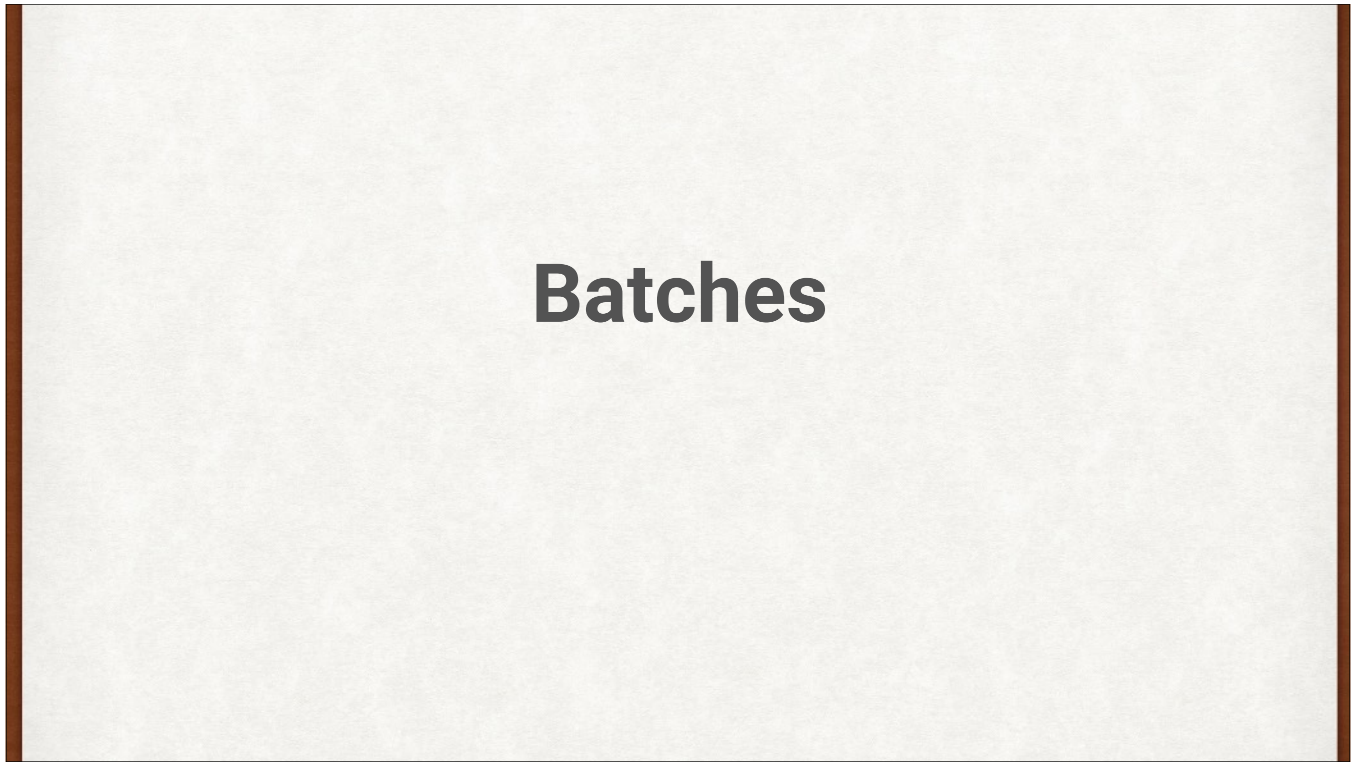
We find how the error changes as P1 changes. That is, the derivative, or gradient, of the error, with respect to P1. We can determine how to change the weights in the brown neuron so that the error (that is, the mismatch between the output and the label) is reduced.



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.



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!



Batches

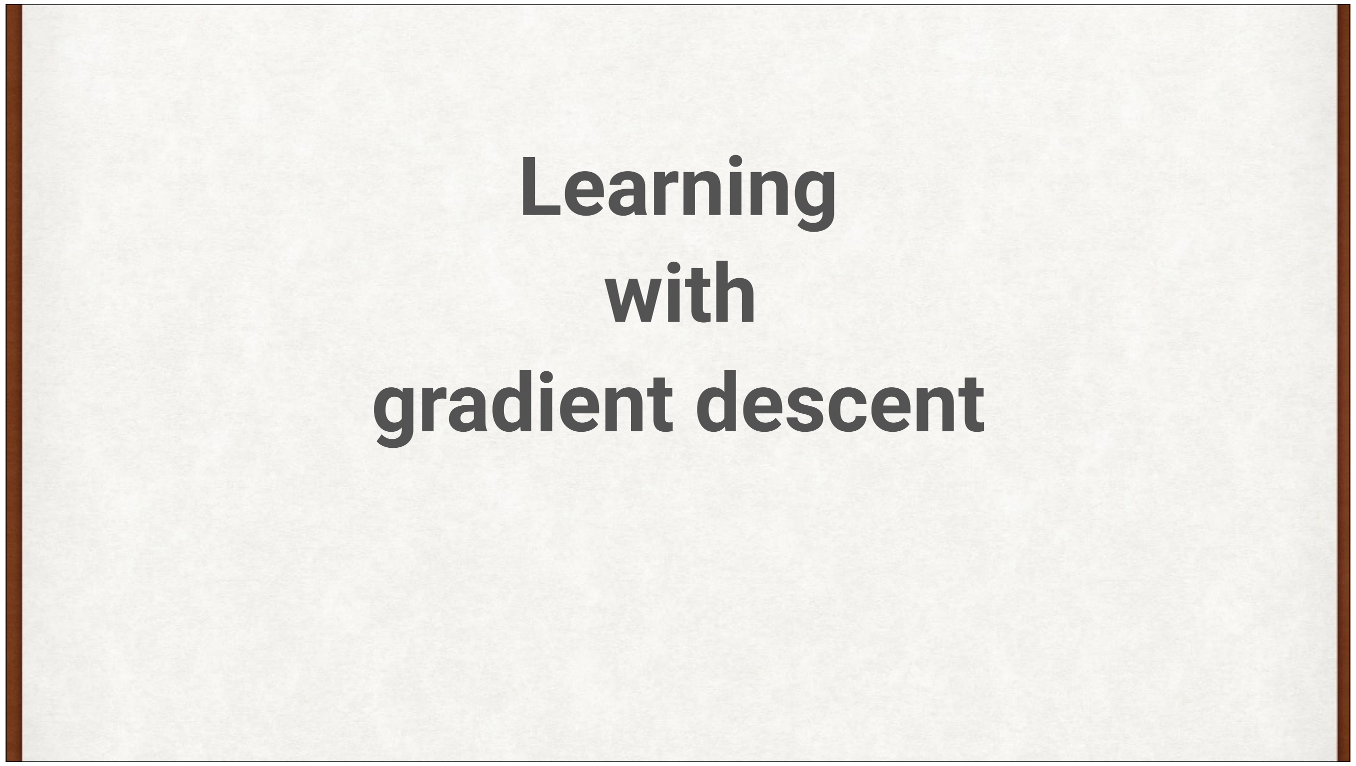
We usually don’t do backprop and gradient descent after every training piece. We usually use batches - often a bunch of samples that fit on the GPU in one go, such as 16 or 32. We run them all through, collect the errors, then run backprop. These batches are also called mini-batches.

We can update one by one after each sample…
https://pixabay.com/en/smarties-confectionery-sugar-candy-50838/

Or we can update after doing a batch of samples

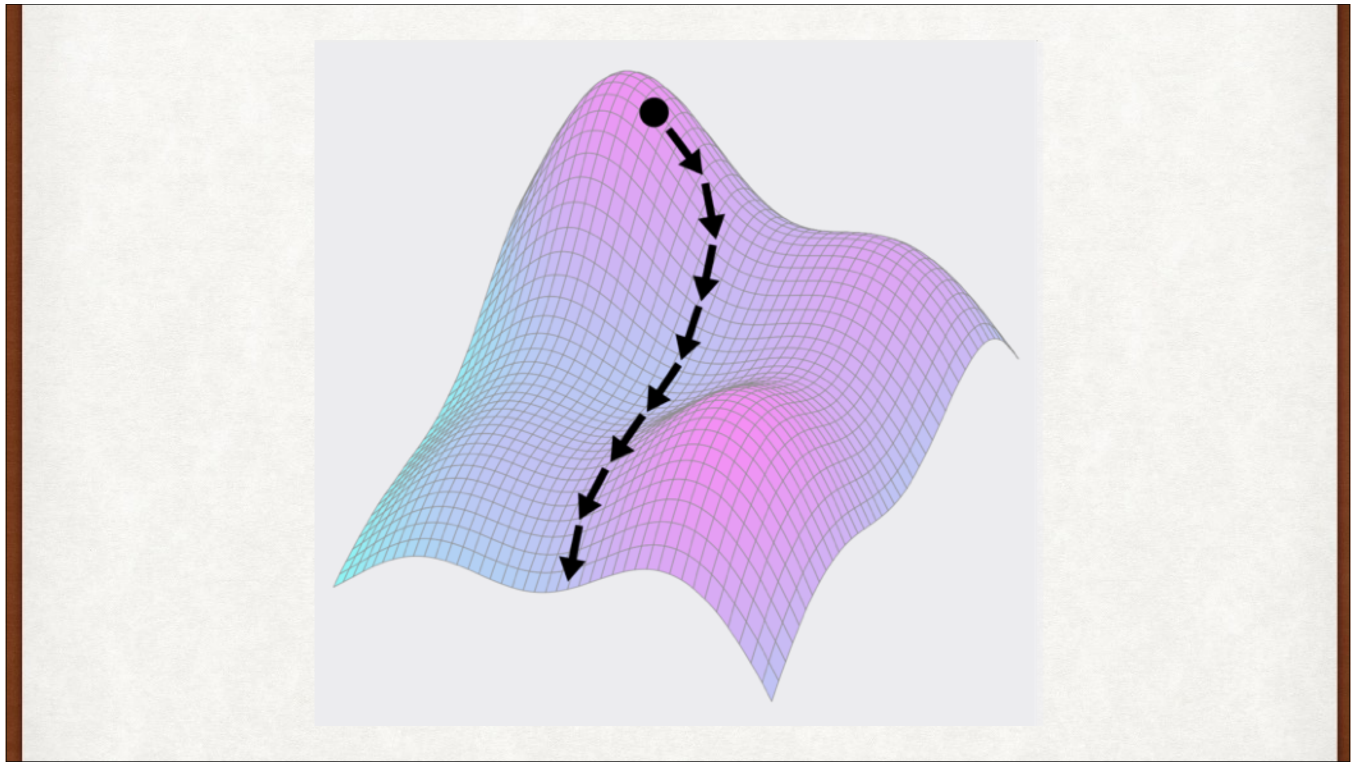
https://pixabay.com/en/candy-dragee-food-sweet-eat-3757566/

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



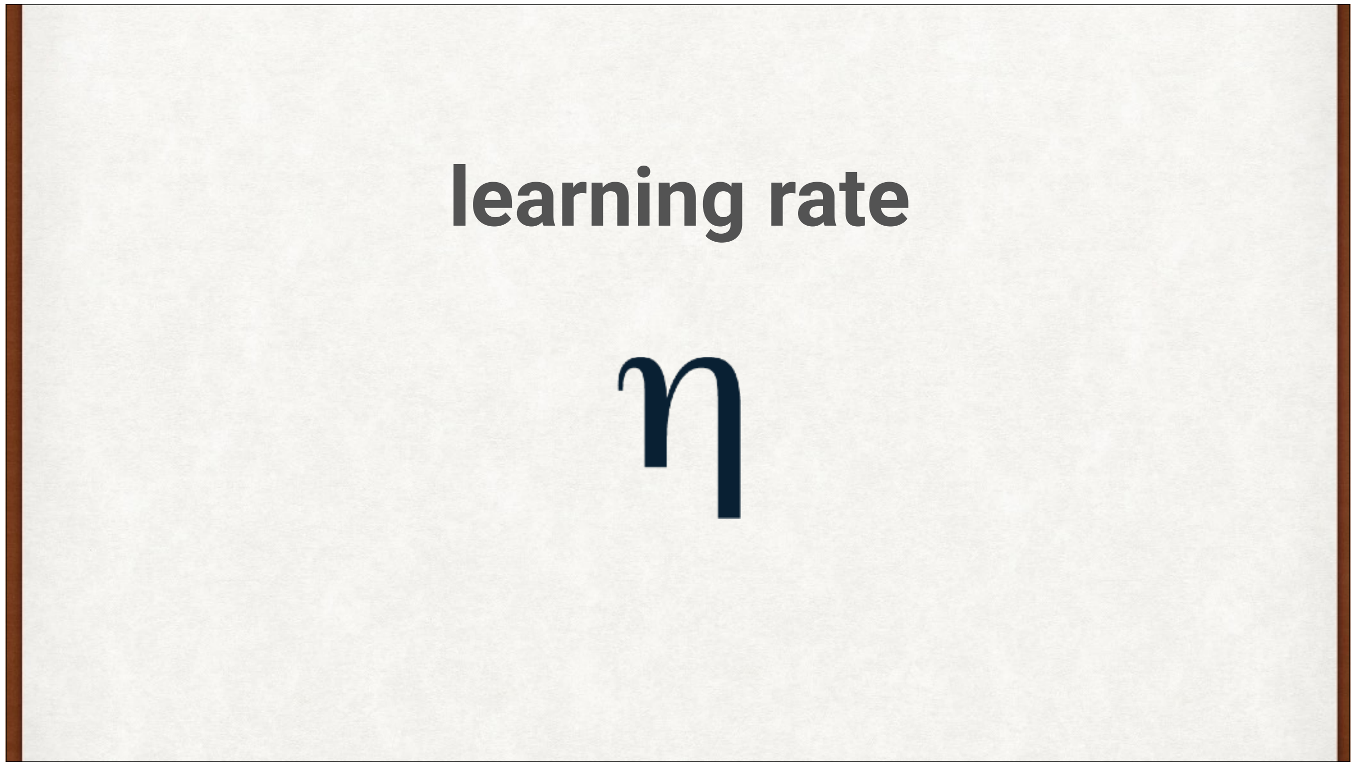
Learning with gradient descent

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.

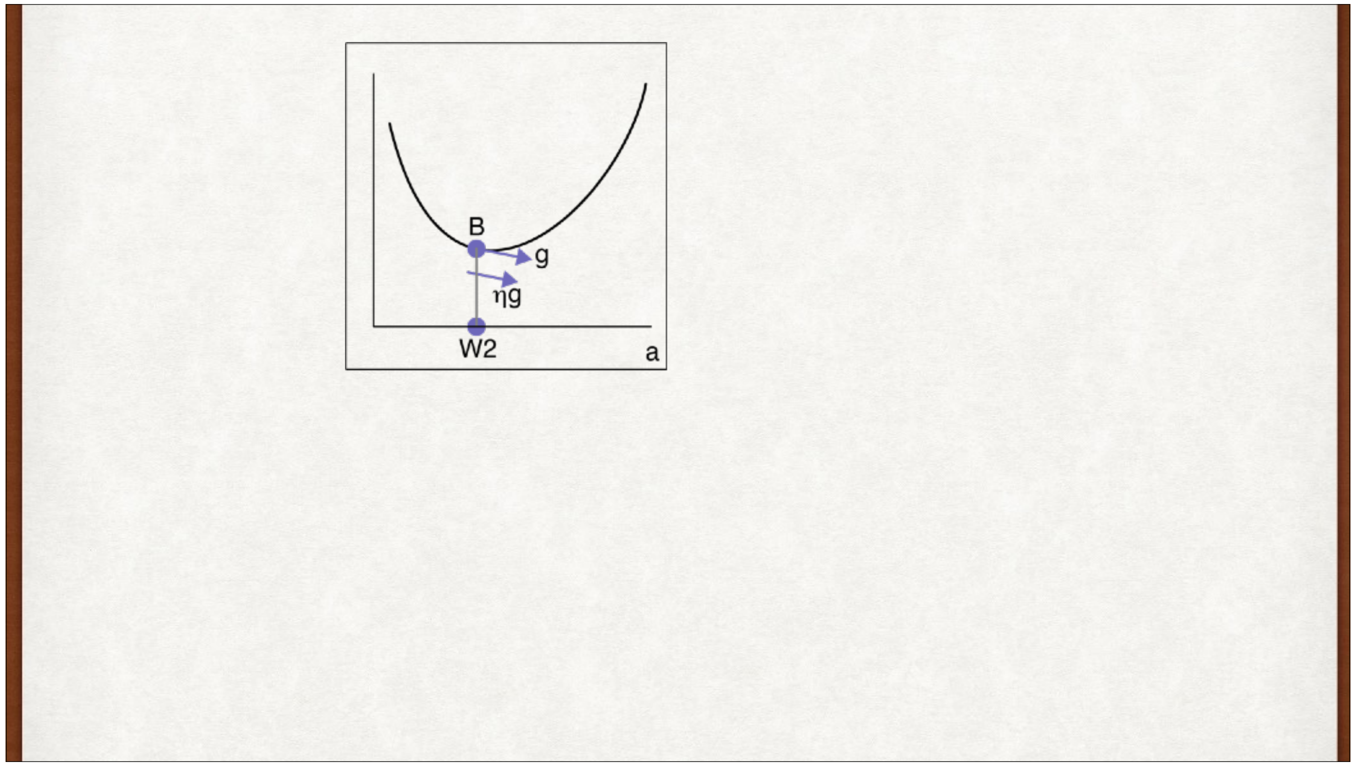
Big steps, then little steps as we get closer
https://pixabay.com/en/beach-sunrise-sea-vacations-sun-2902747/



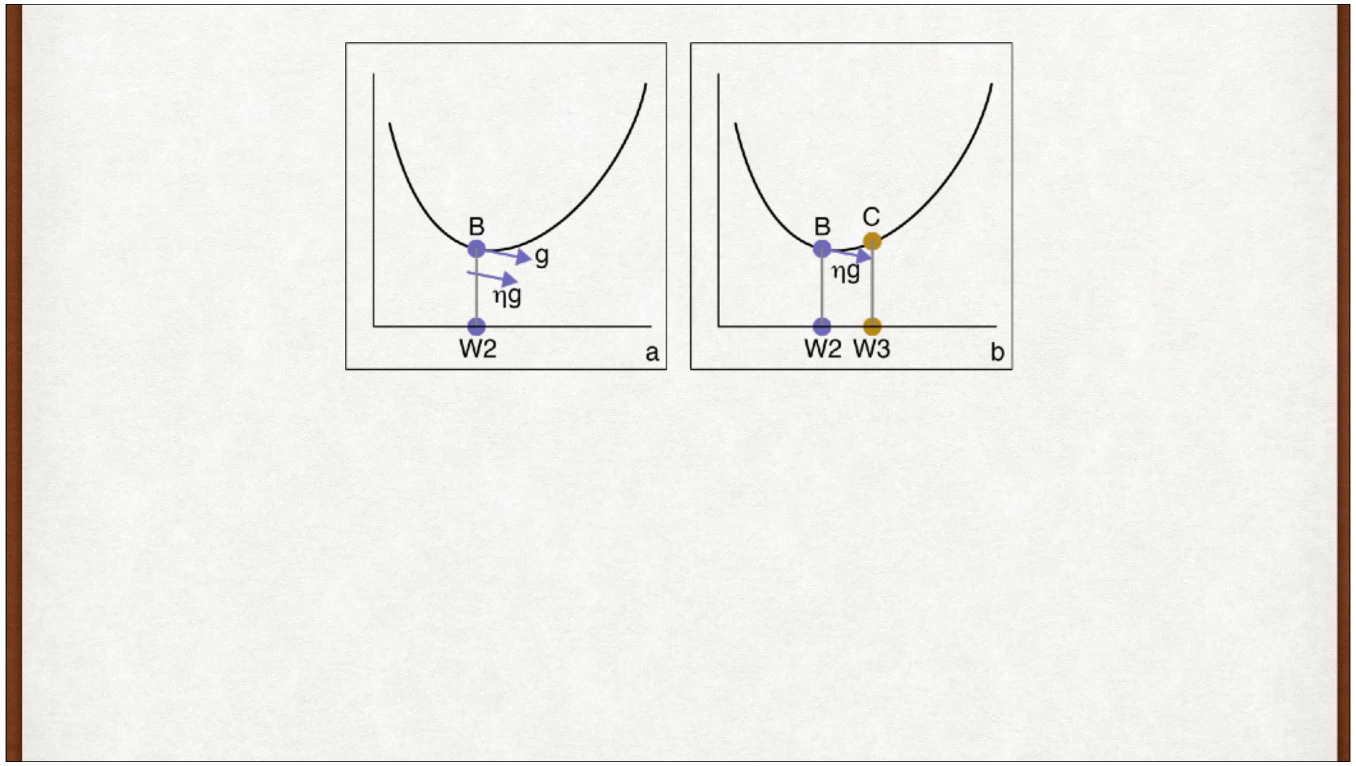
learning rate

η

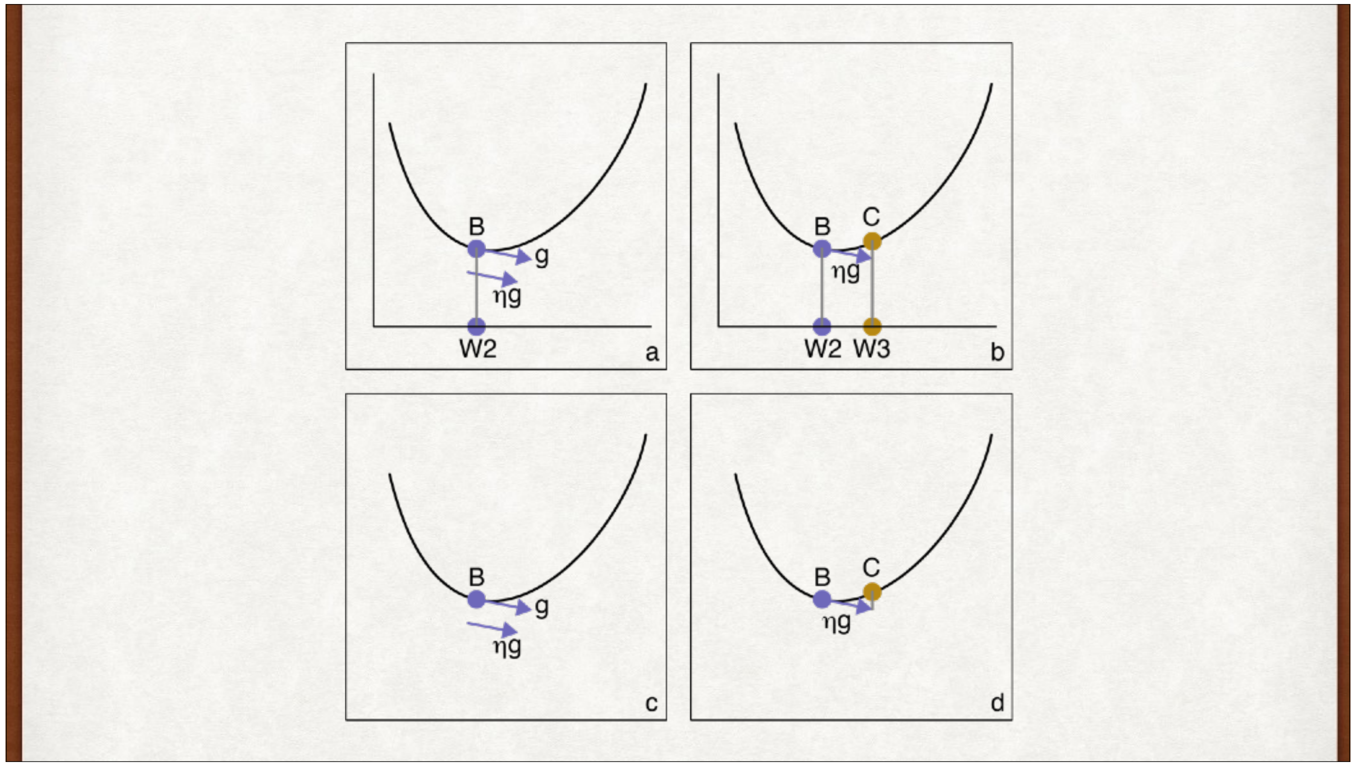
The amount by which we change each weight is governed by a real number called the learning rate, often represented by the Greek letter eta. This is a real number, usually around 0.2 or less. It’s a parameter to the network, but one we provide, rather than one that’s learned. Such parameters are called **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.



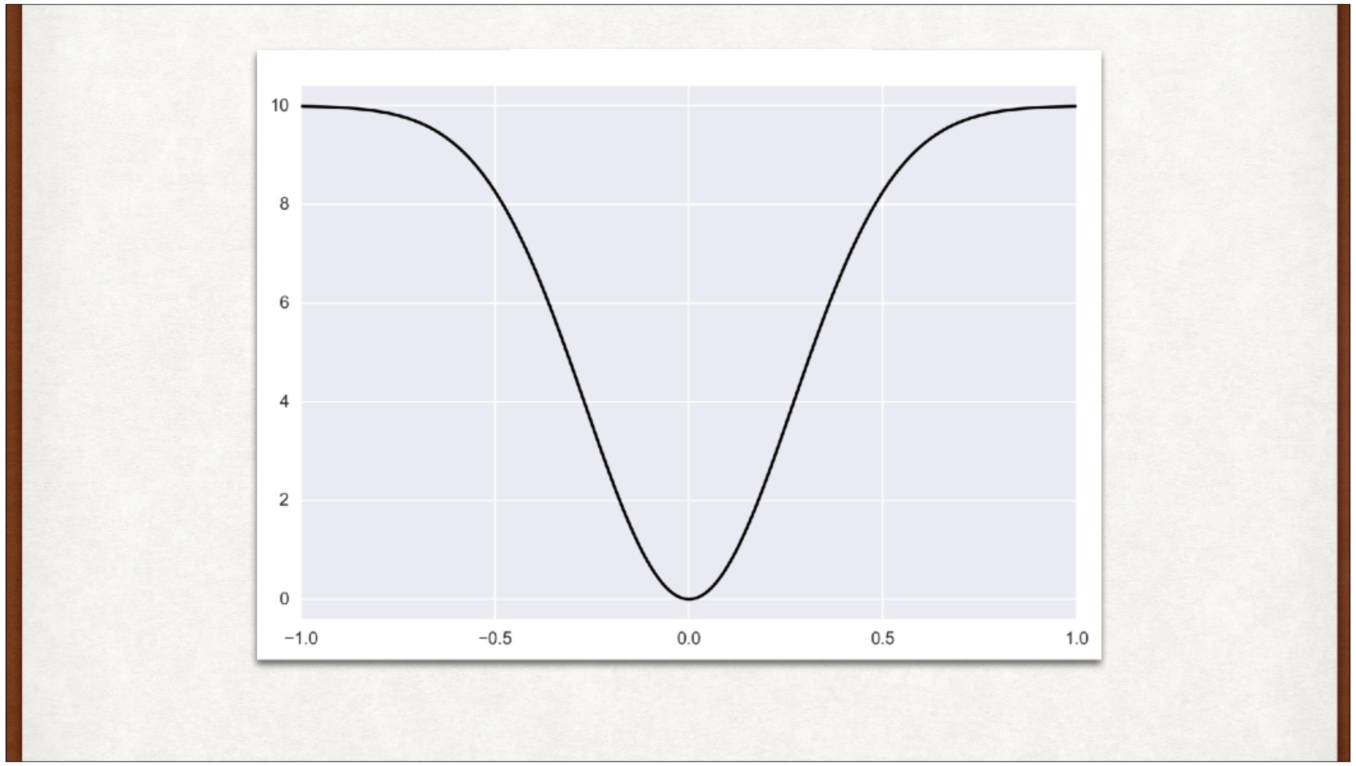
Here’s the idea for improving weight W2. Its error value is point B on the error curve, with gradient g. Adding a scaled amount of g takes us to point C, and corresponding weight W3. We moved too far in this example, so our error actually went up a little.



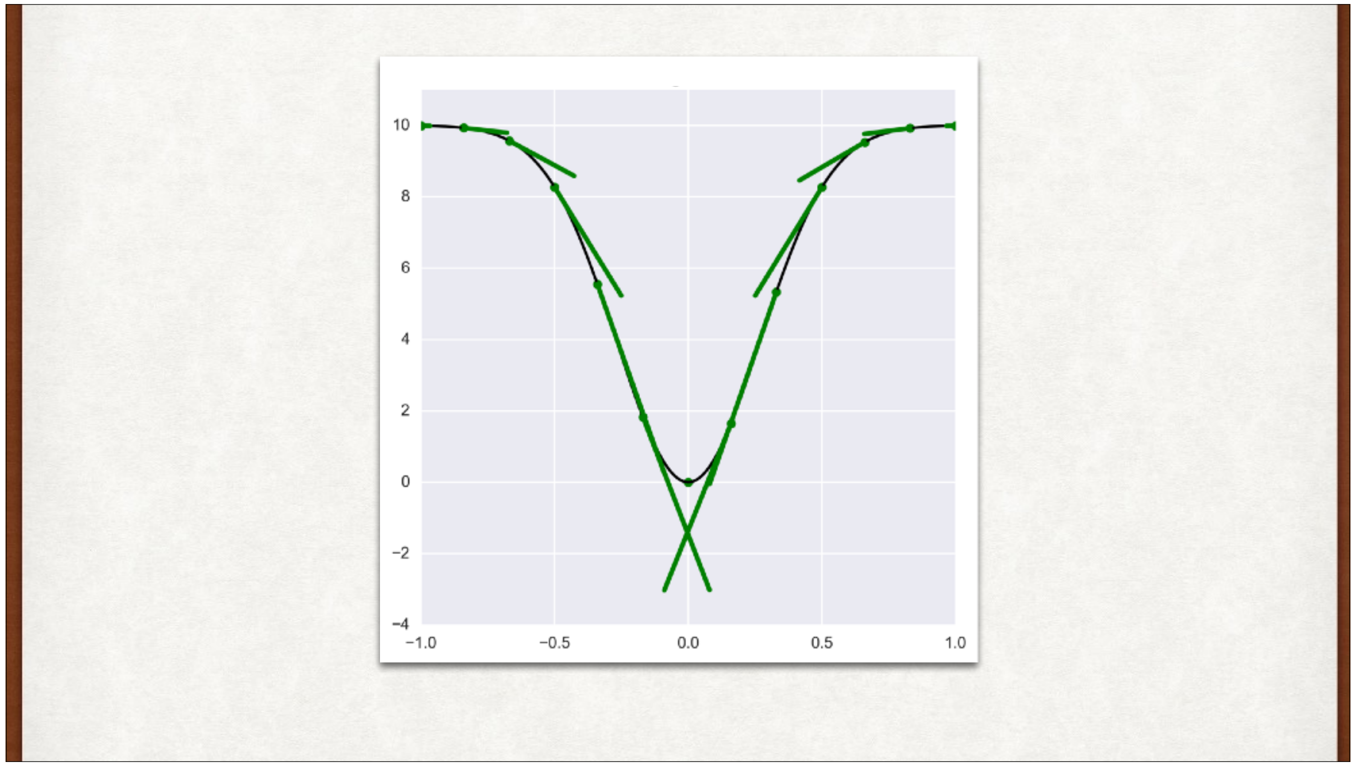
Here’s the idea for improving weight W2. Its error value is point B on the error curve, with gradient g. Adding a scaled amount of g takes us to point C, and corresponding weight W3. We moved too far in this example, so our error actually went up a little.



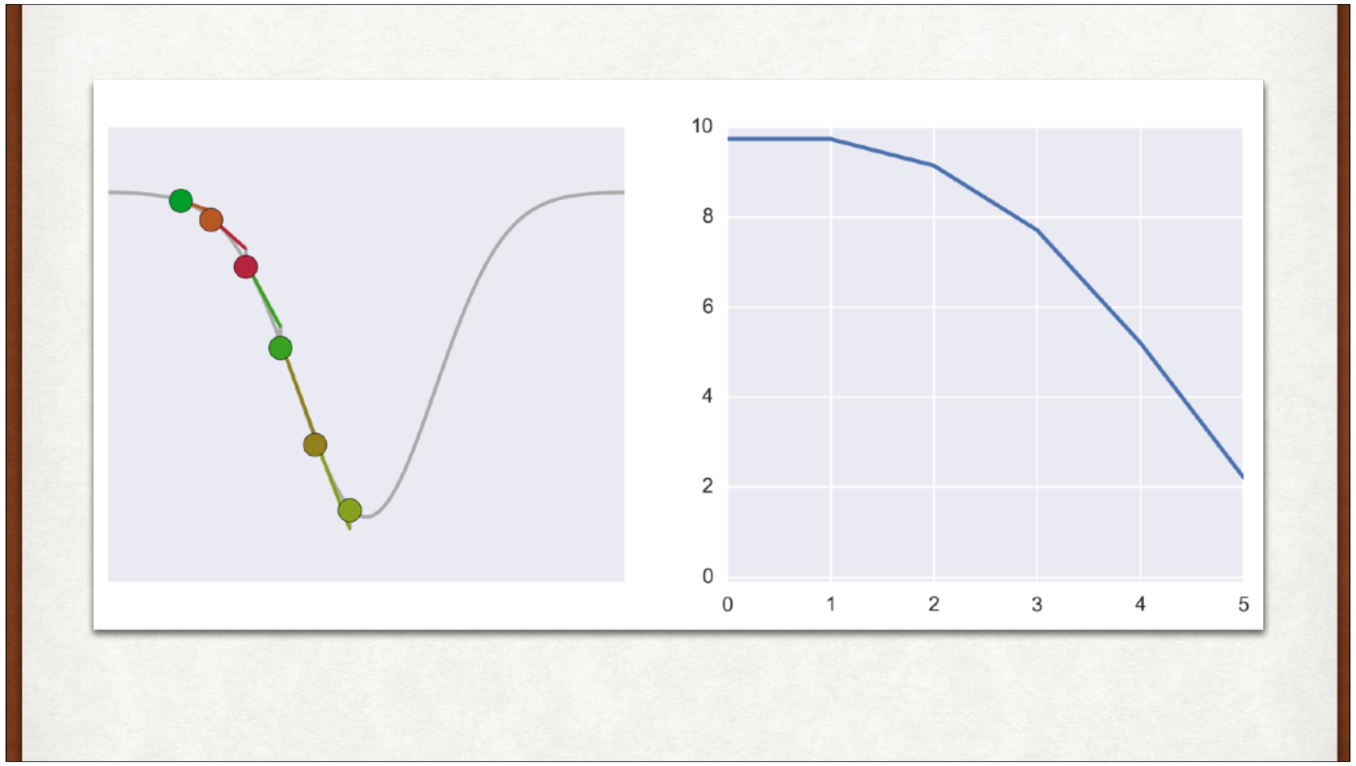
As shown at the bottom, usually we leave out the explicit axes and weights, understanding them to be there, so we can focus on what’s happening with the error curve.



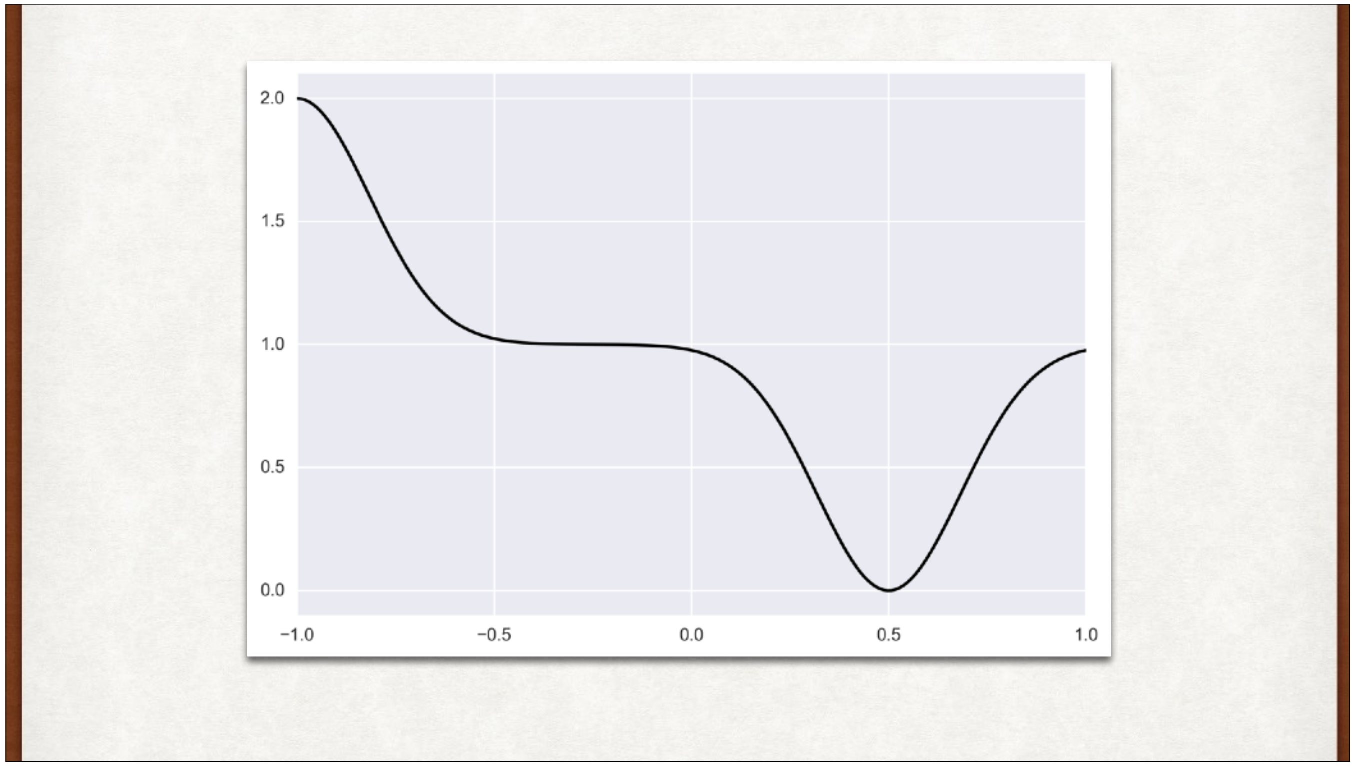
Suppose this is our error curve. We want to get to the bottom.



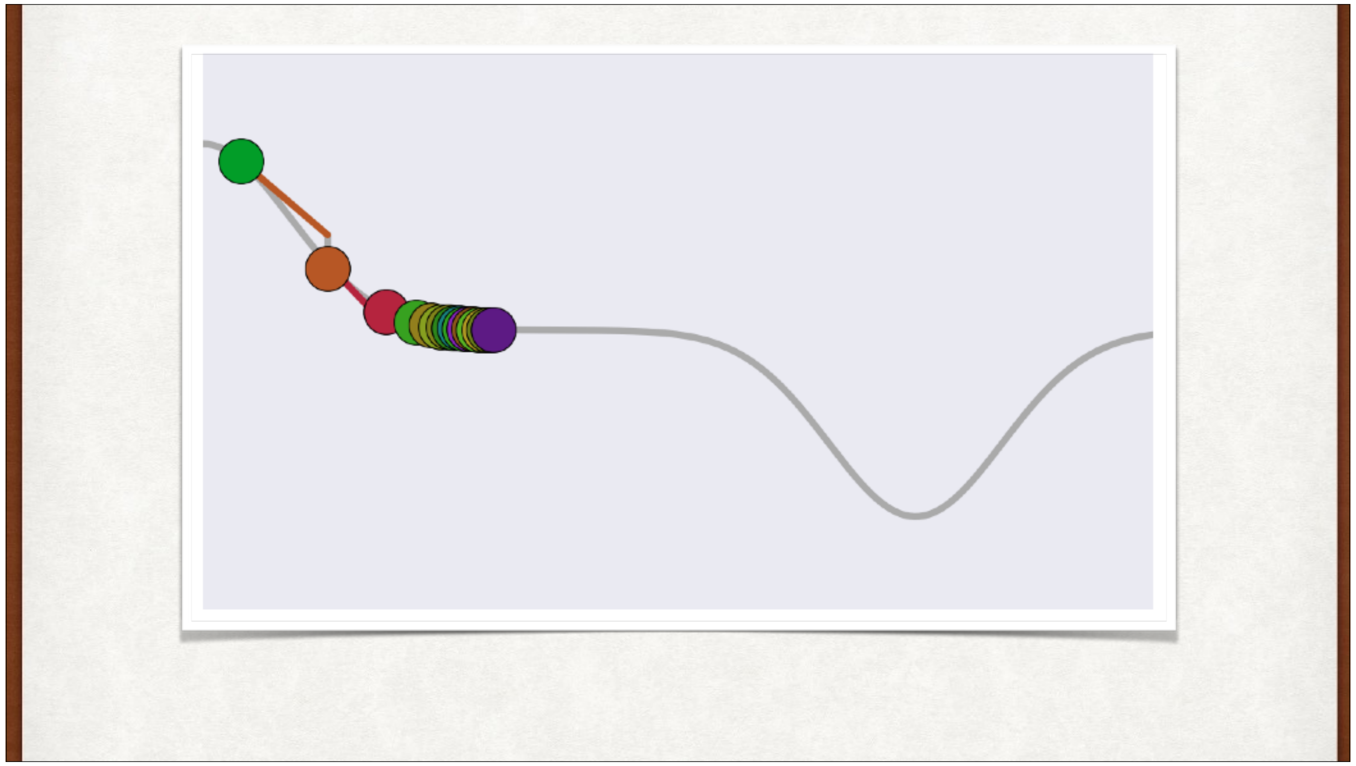
Every point has a gradient, except the bottom of the bowl, where the derivative (or gradient) is zero.



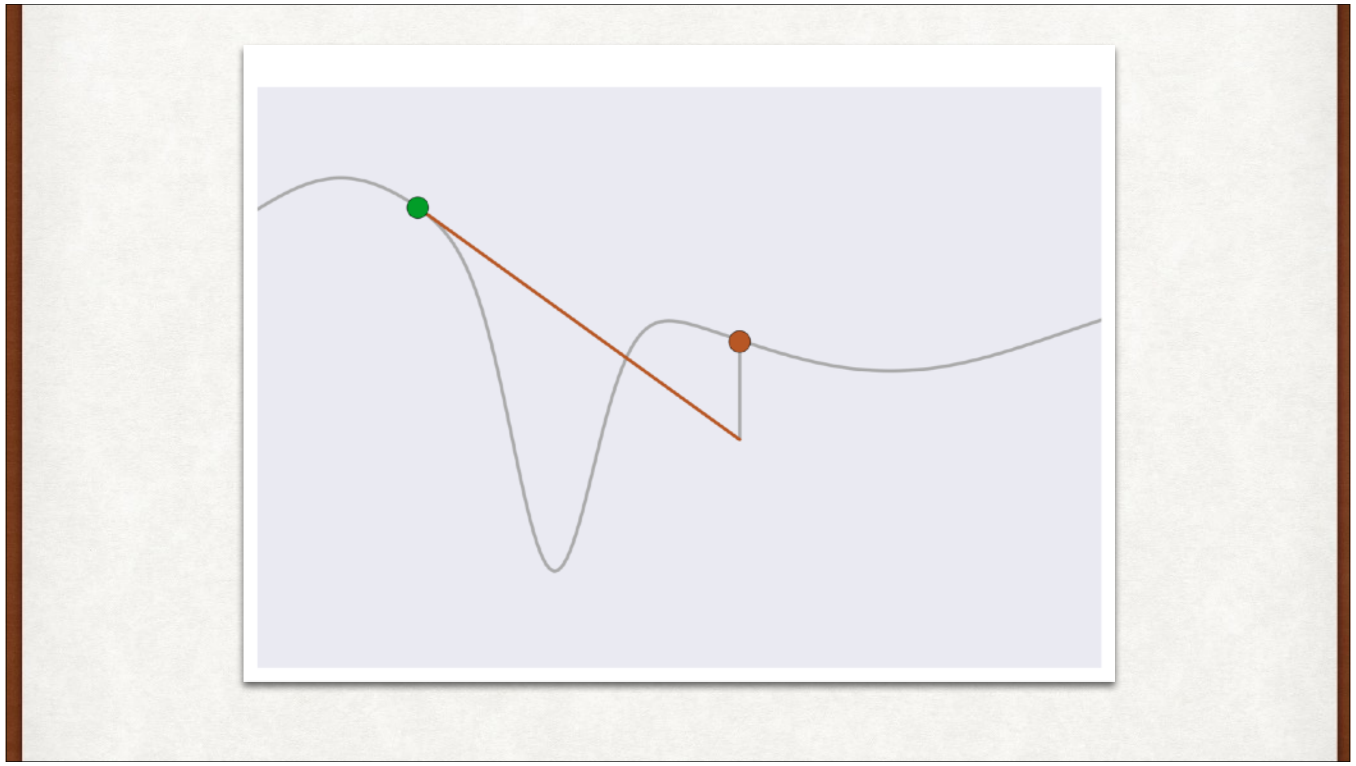
Starting at the far left, the left image shows successive values as we move the value of the weight to follow the gradient downhill. On the right is the error after each step.



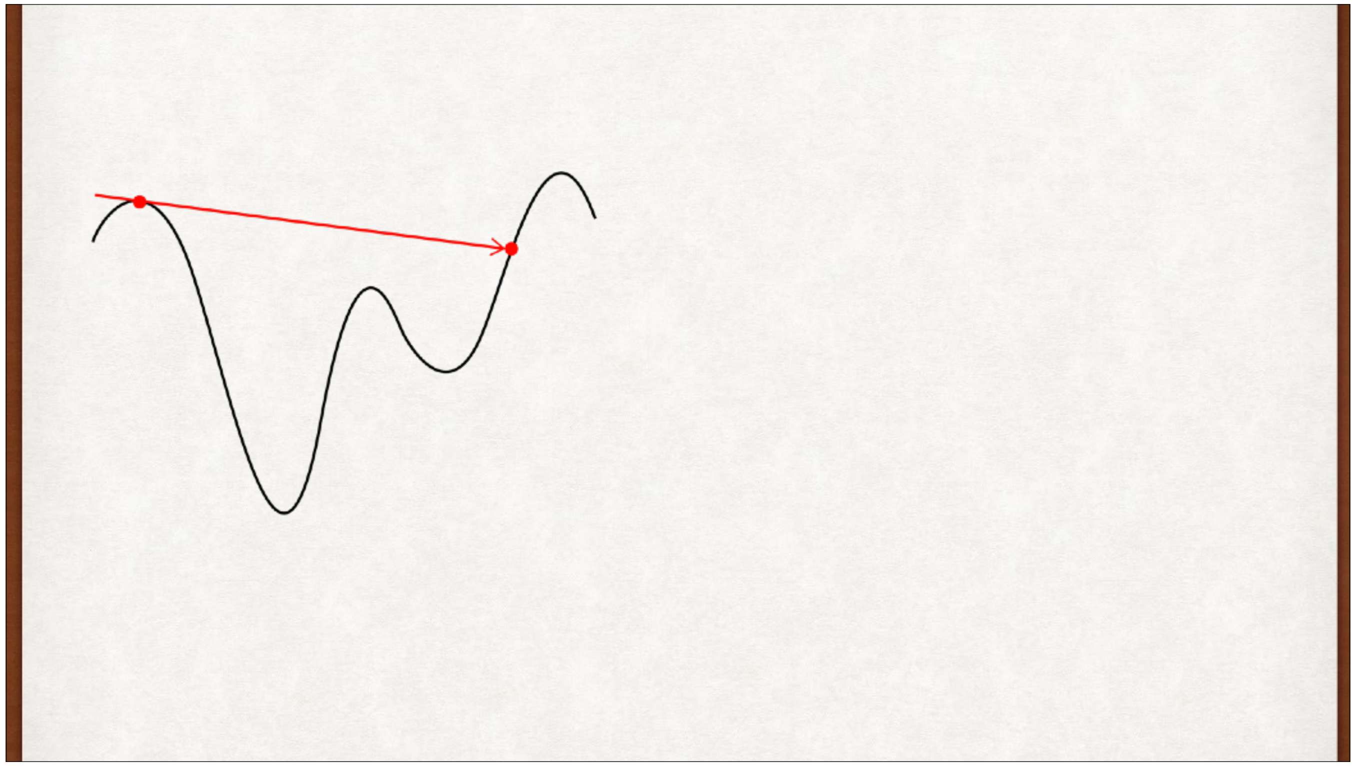
A more interesting error curve.



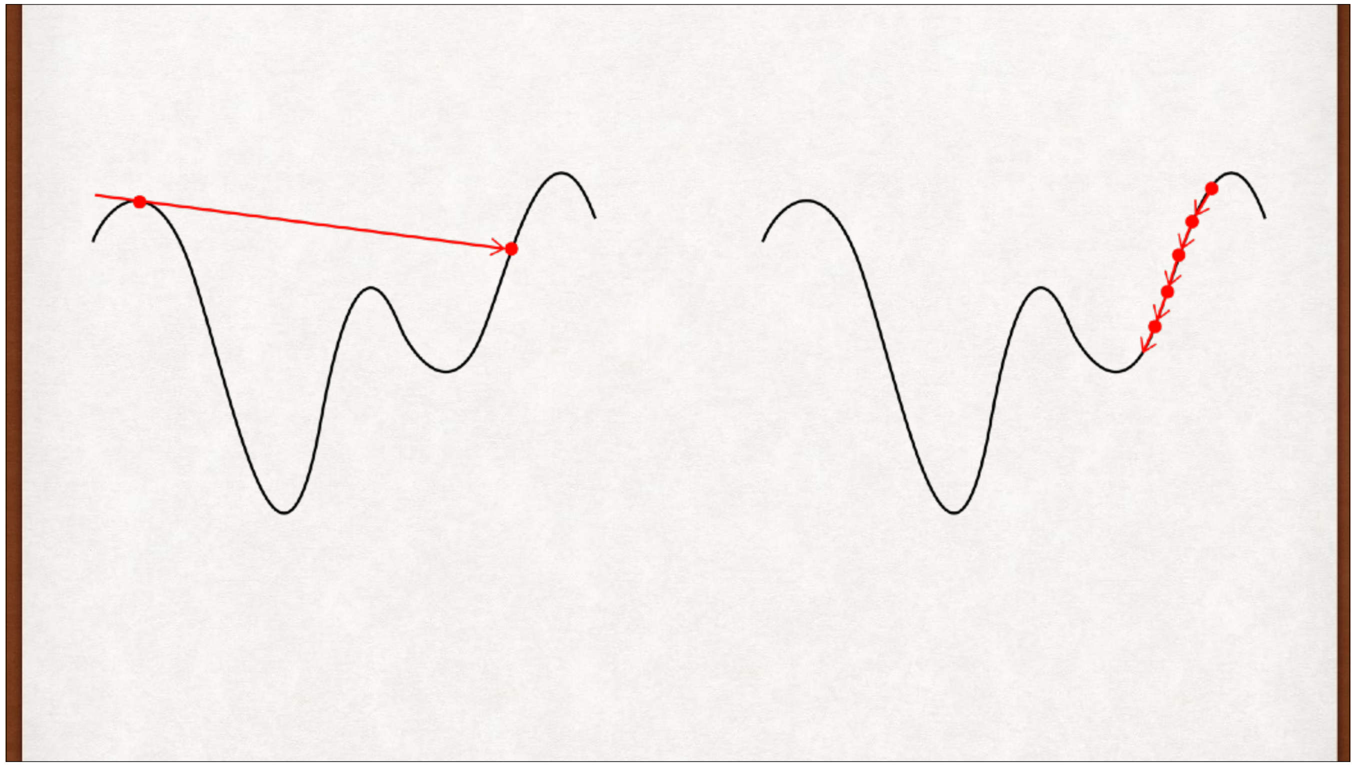
Starting with the green dot at the far left, and… uh-oh. The gradient goes to zero on the flat region, and we get slow down…. until we’re stuck. Zero gradient, zero progress. No more learning!



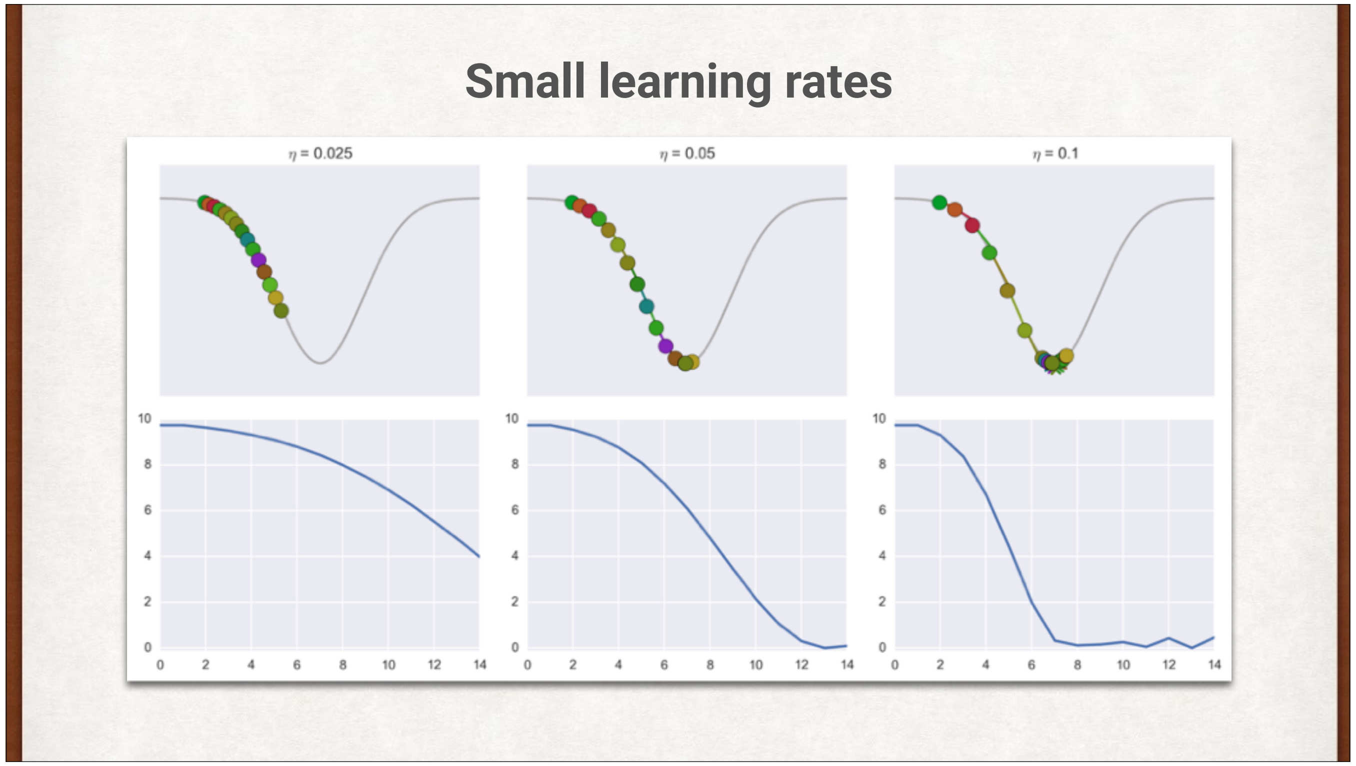
We could take bigger steps, but that’s not always productive. Here we pop right over the minimum, and further steps will move us to the right.



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.

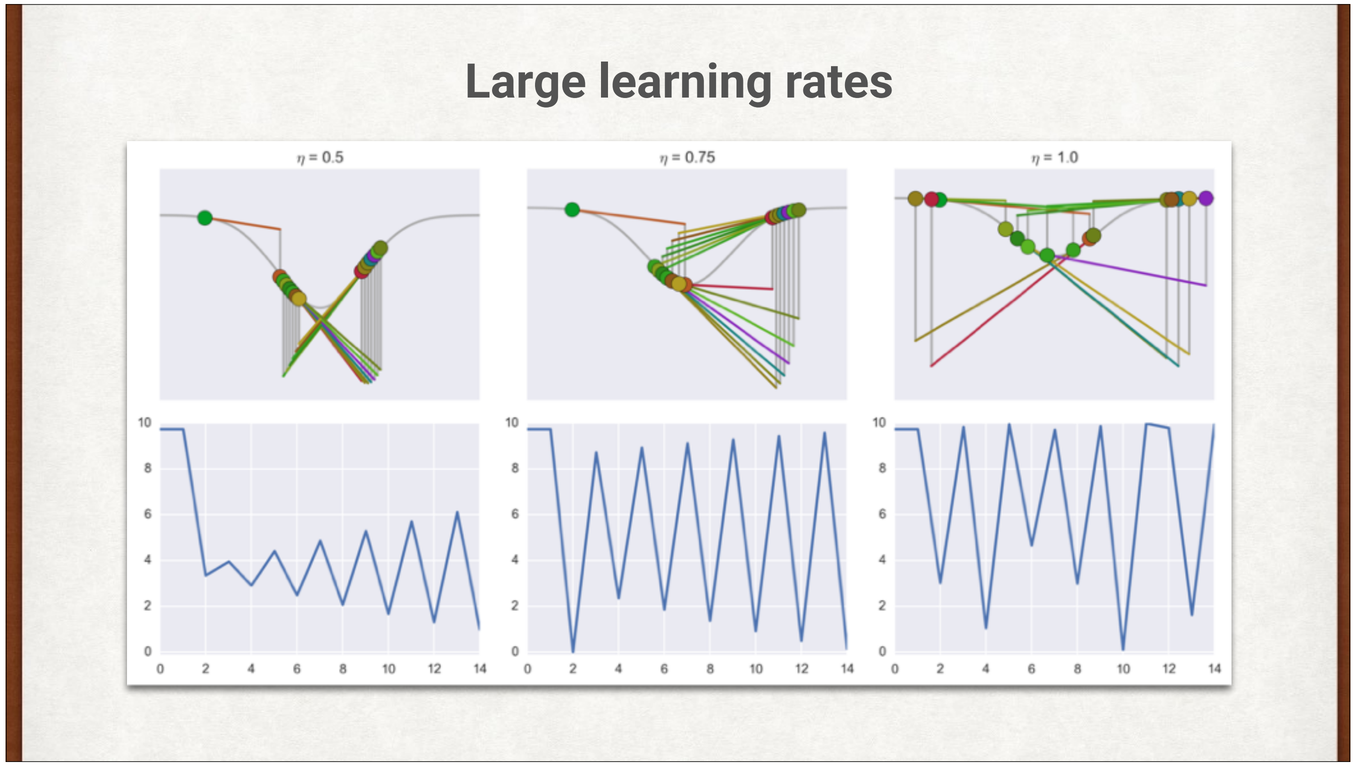


Steps too big and we overshoot our minimum. Steps too small and we take forever to move downhill, and can get stuck in a local minimum when a larger minimum is nearby. Both are problems.



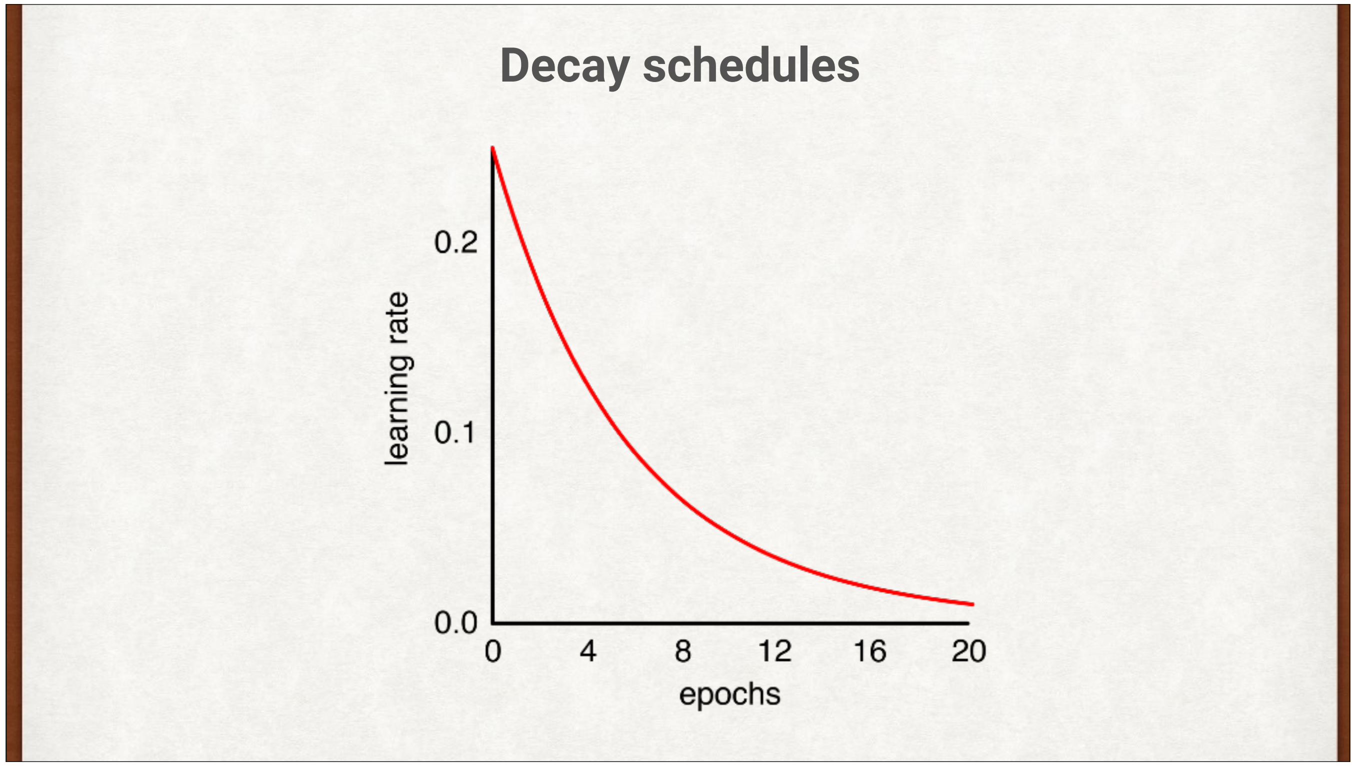
Small learning rates

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.



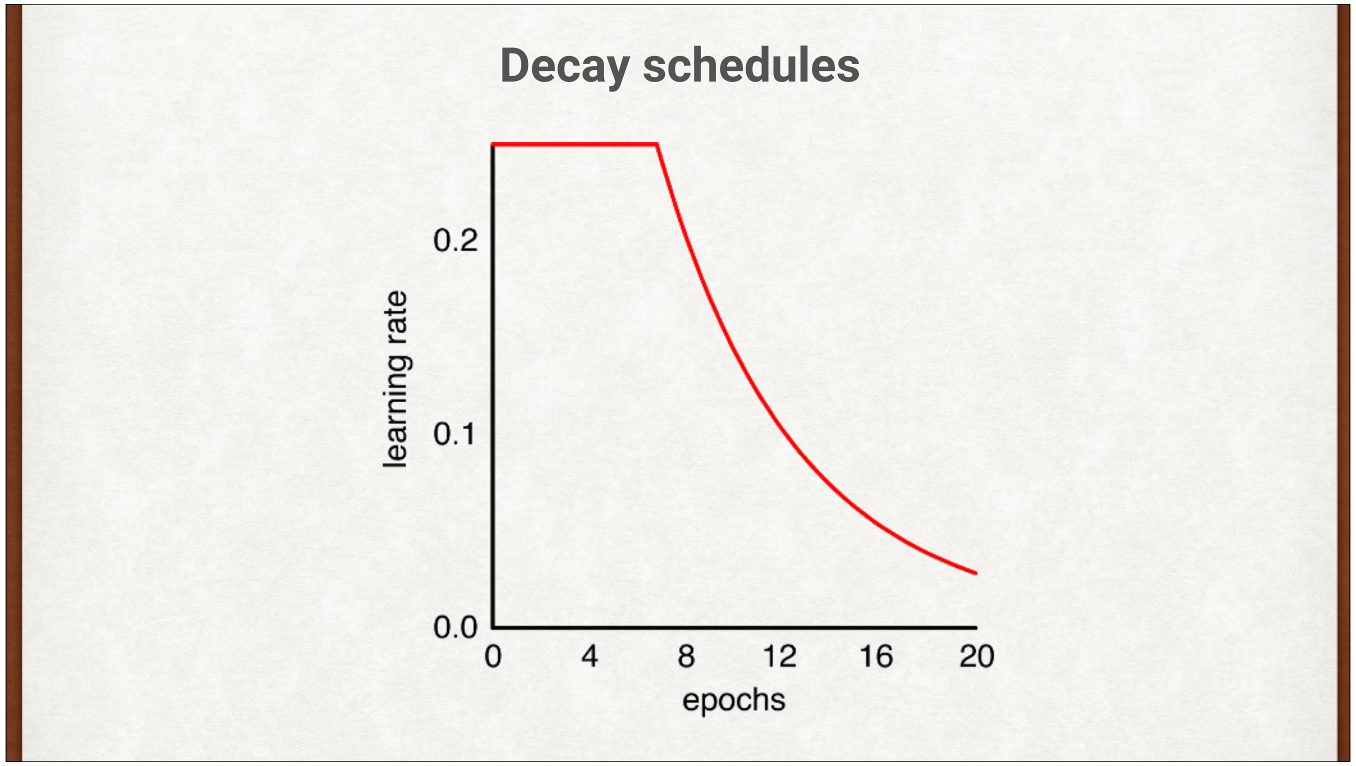
Large learning rates

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



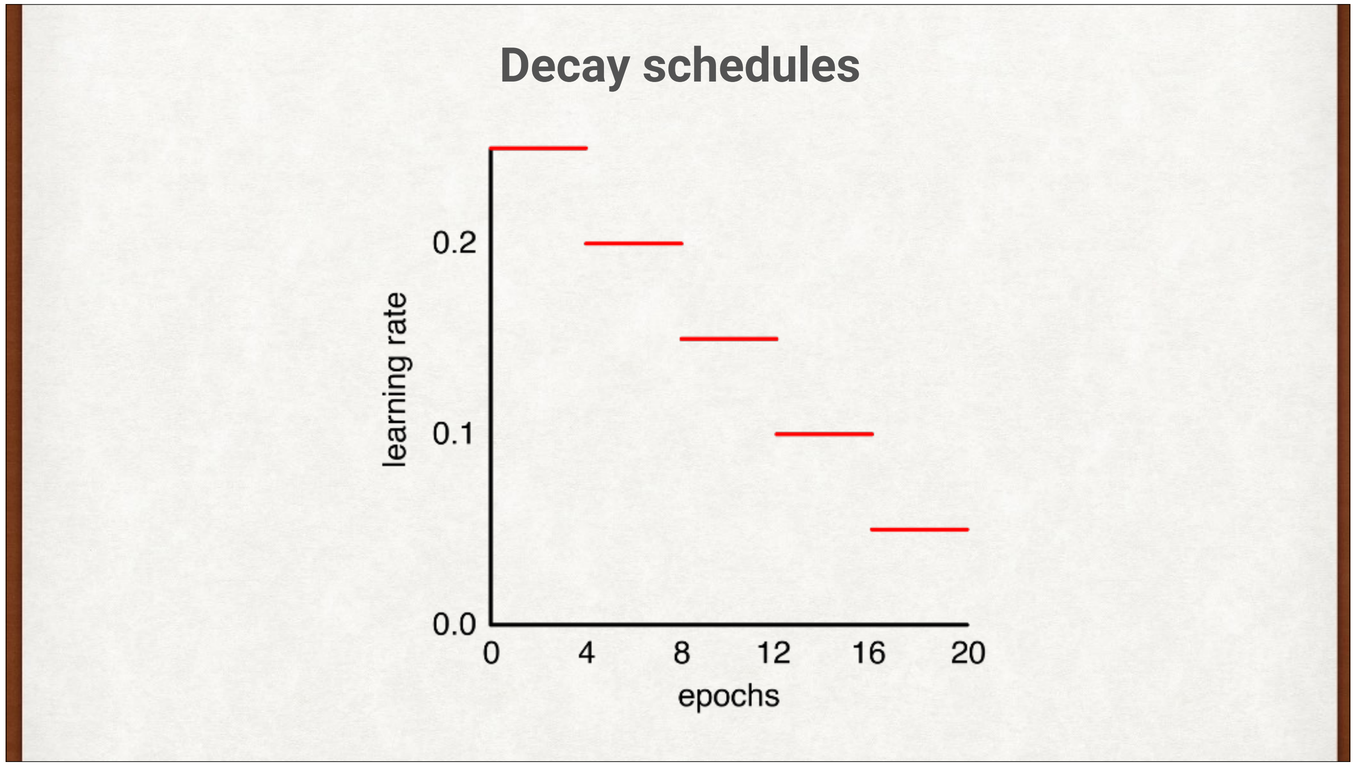
Decay schedules

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.



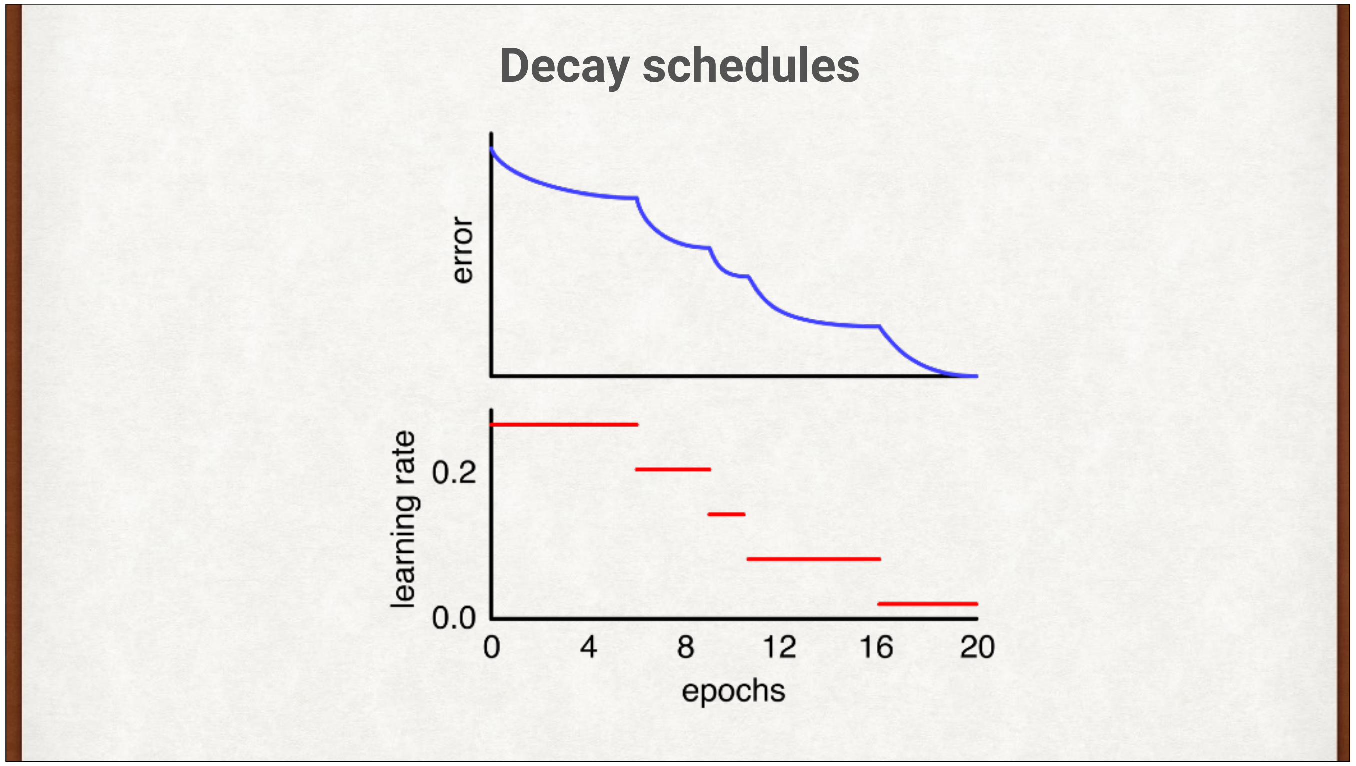
Decay schedules

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.



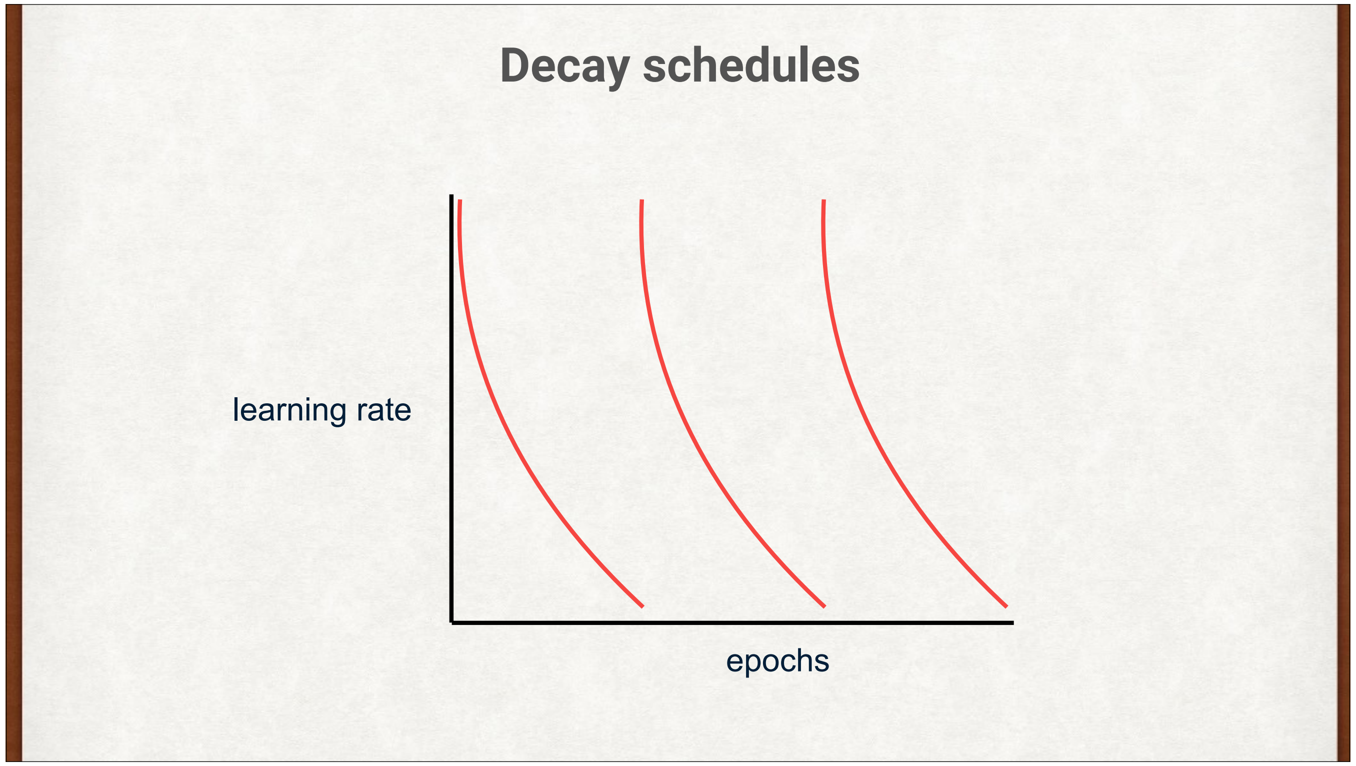
Decay schedules

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.



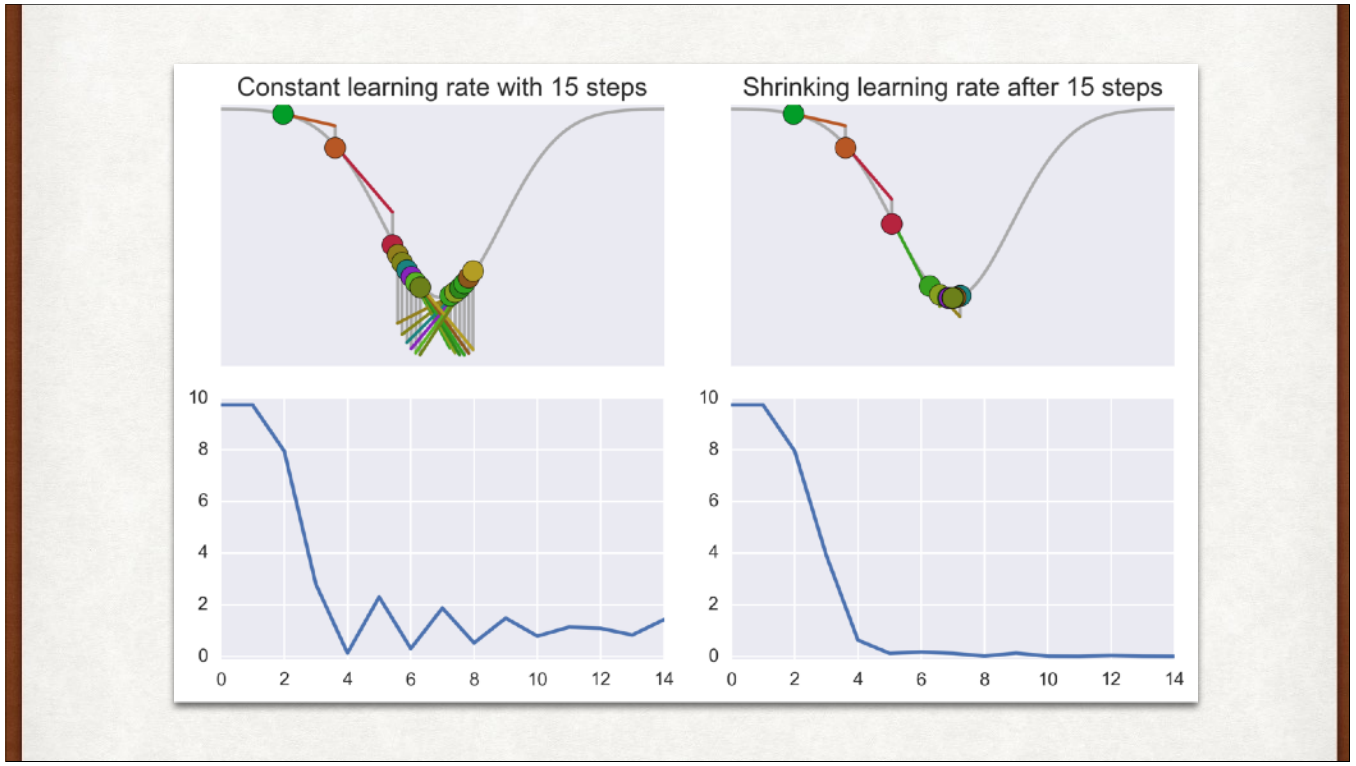
Decay schedules

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.

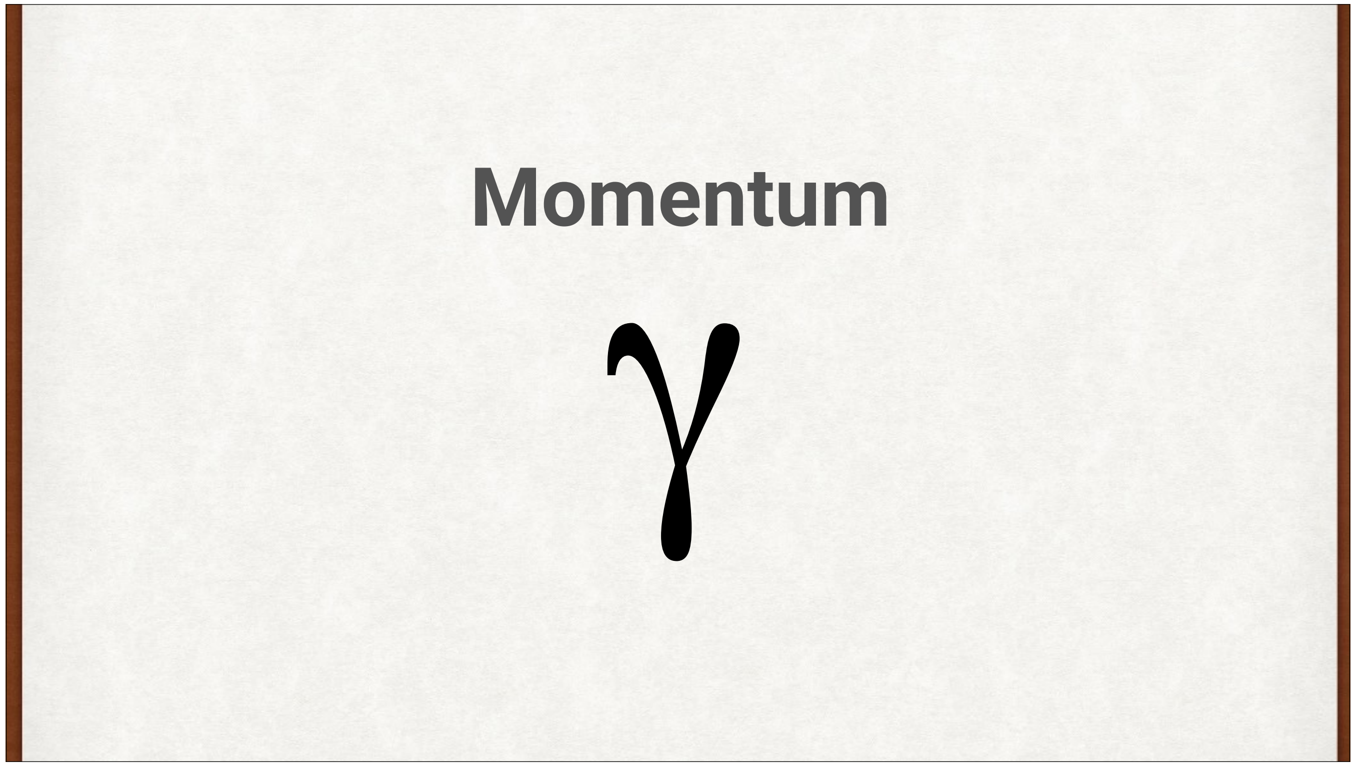


Decay schedules

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.



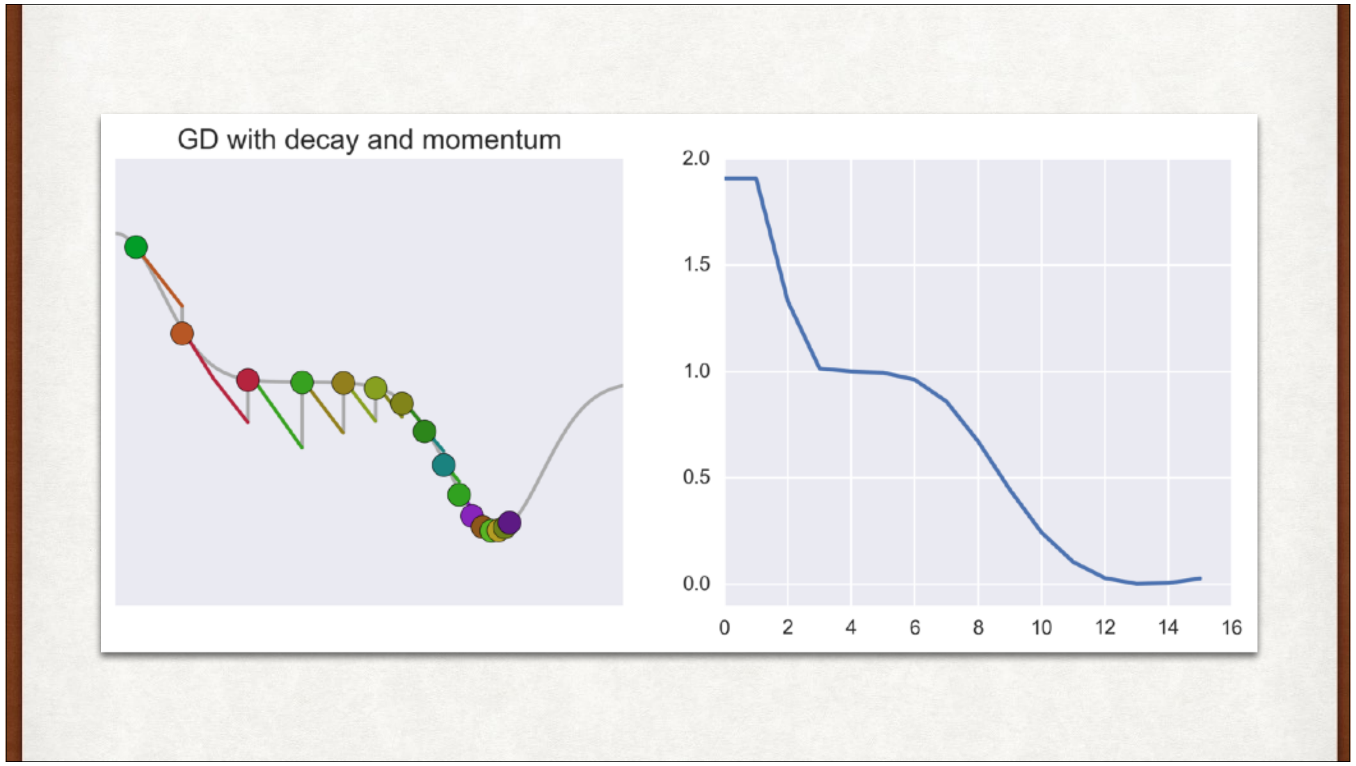
Shrinking over time works well here, and in most other situations.



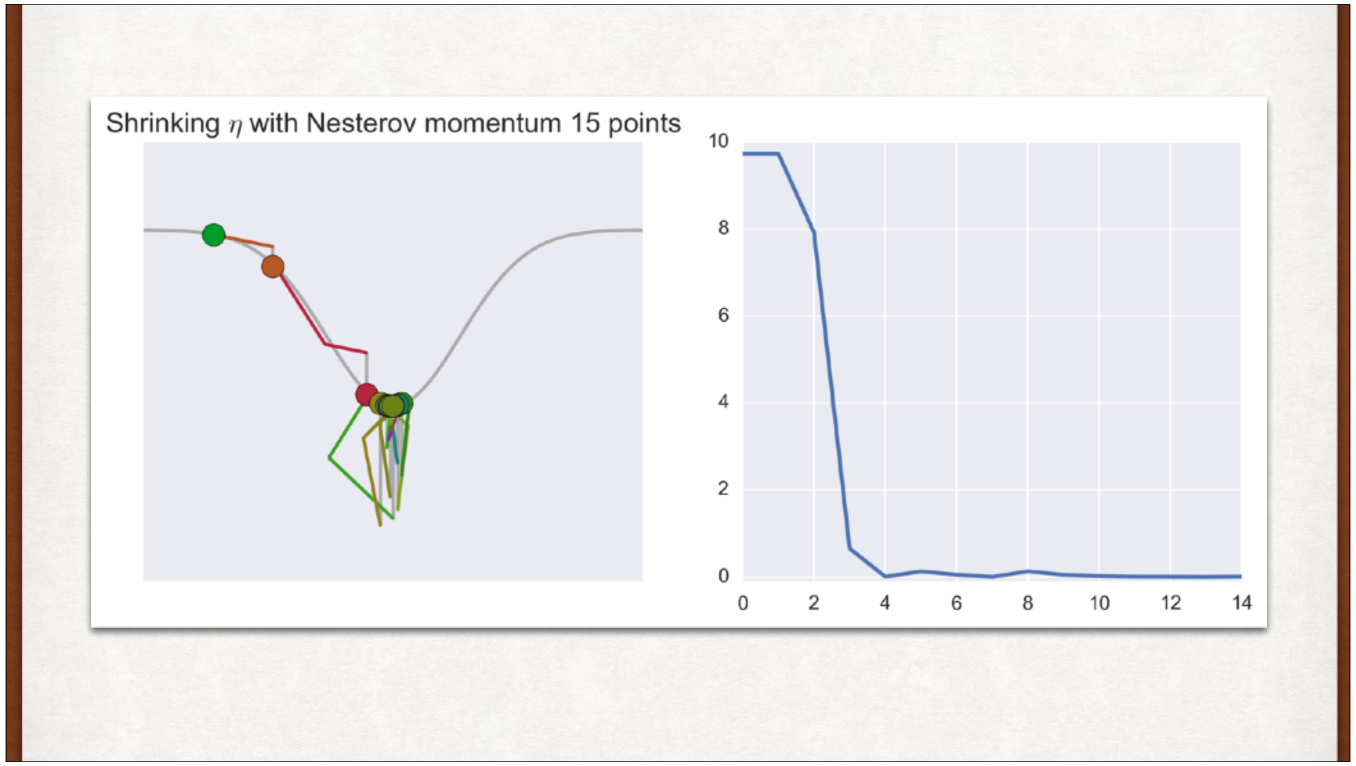
Momentum

$$\gamma$$

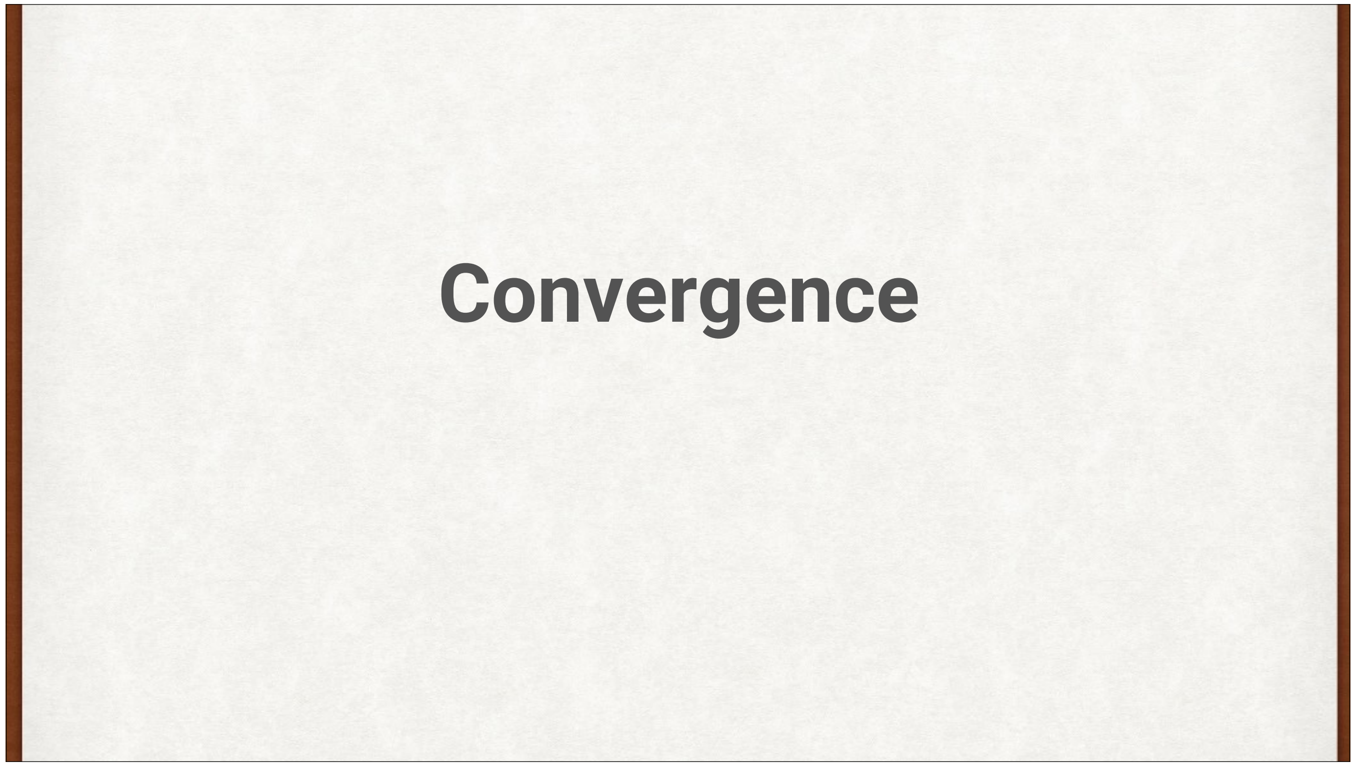
We can apply some “momentum” (actually a version of inertia) so that each change in a weight includes some percentage of its previous change as well.



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.



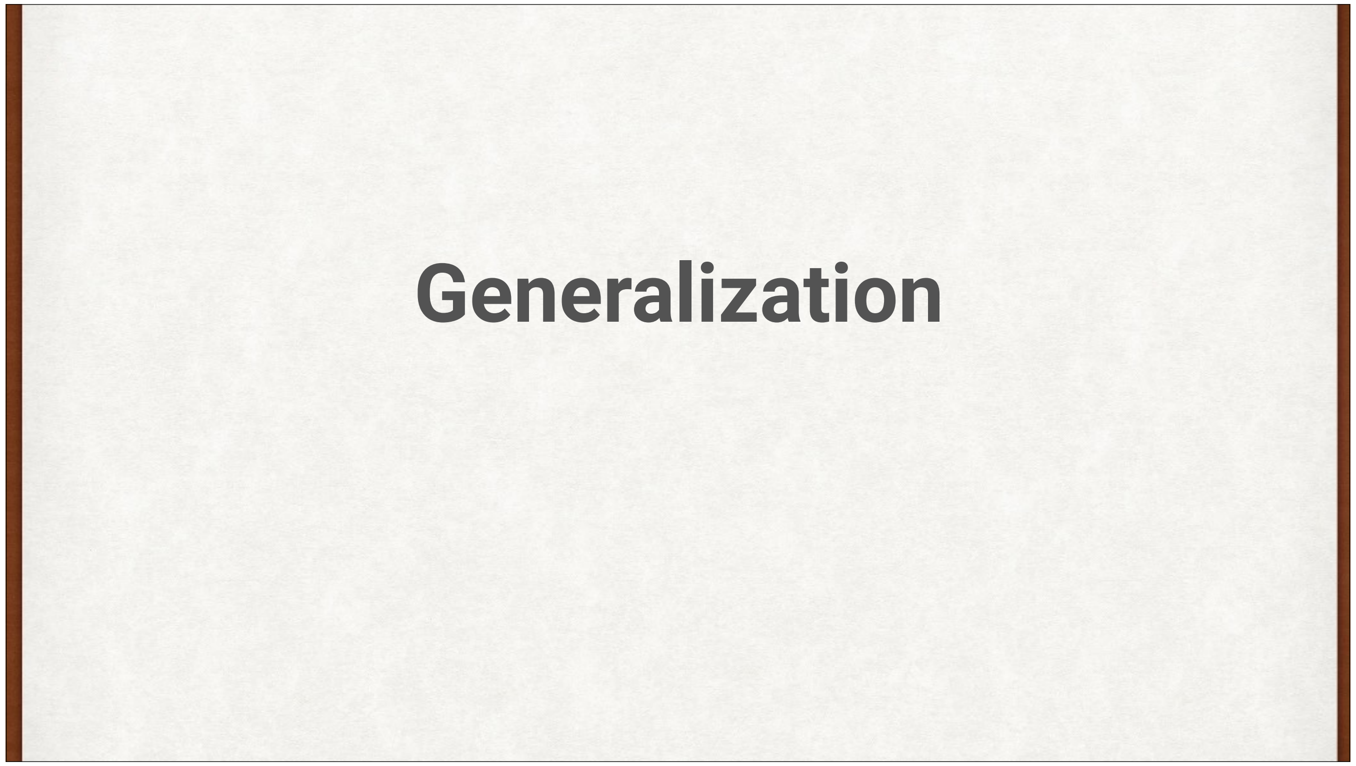
Nesterov momentum is a little more complicated, but works even better than plain momentum. It's a cool idea because it combines information from where we've been (momentum) with information from where we're going, to find out where to go now. The details aren't hard, but by now we have the general idea: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



Convergence

Will our system every learn well? That’s the question of convergence. There’s no theoretical proof or guarantee. Just experience and experimentation.

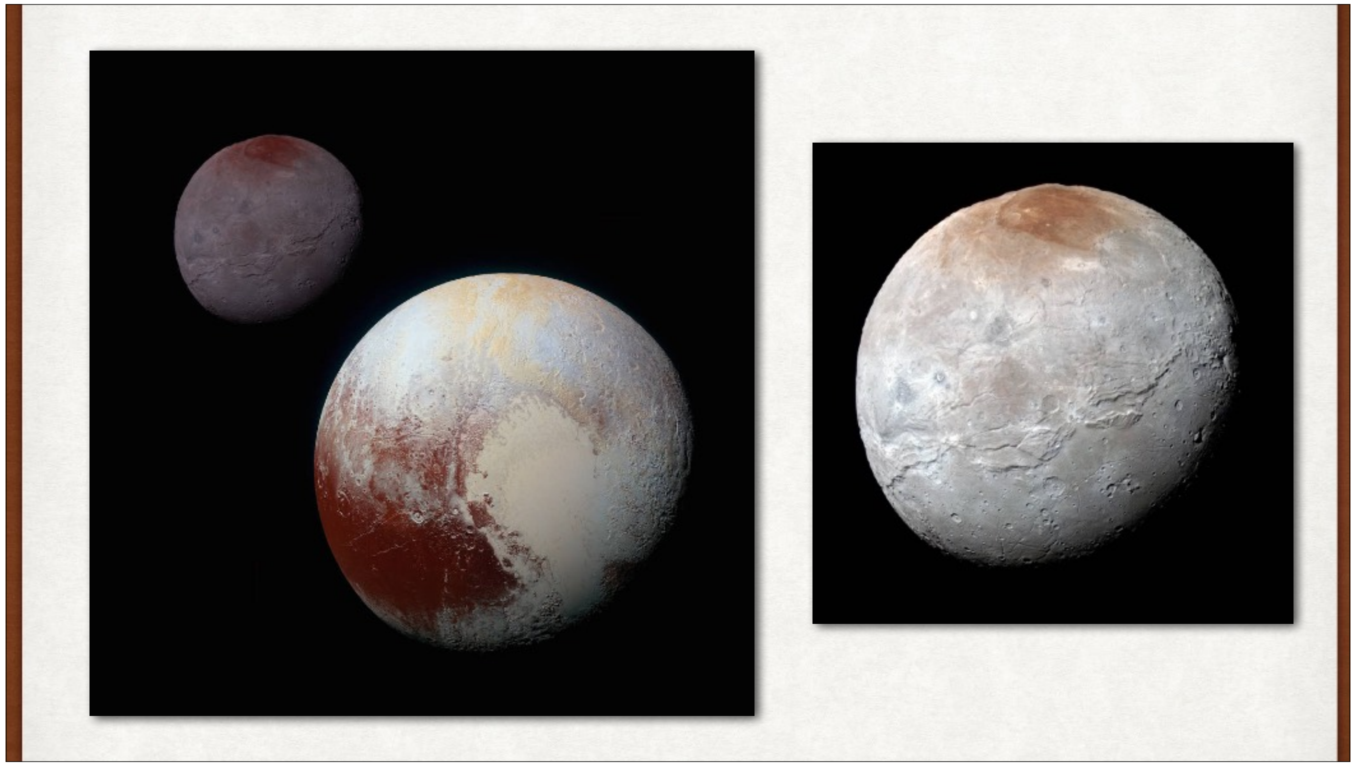
Palette cleanser. More sherbet, please!



Generalization

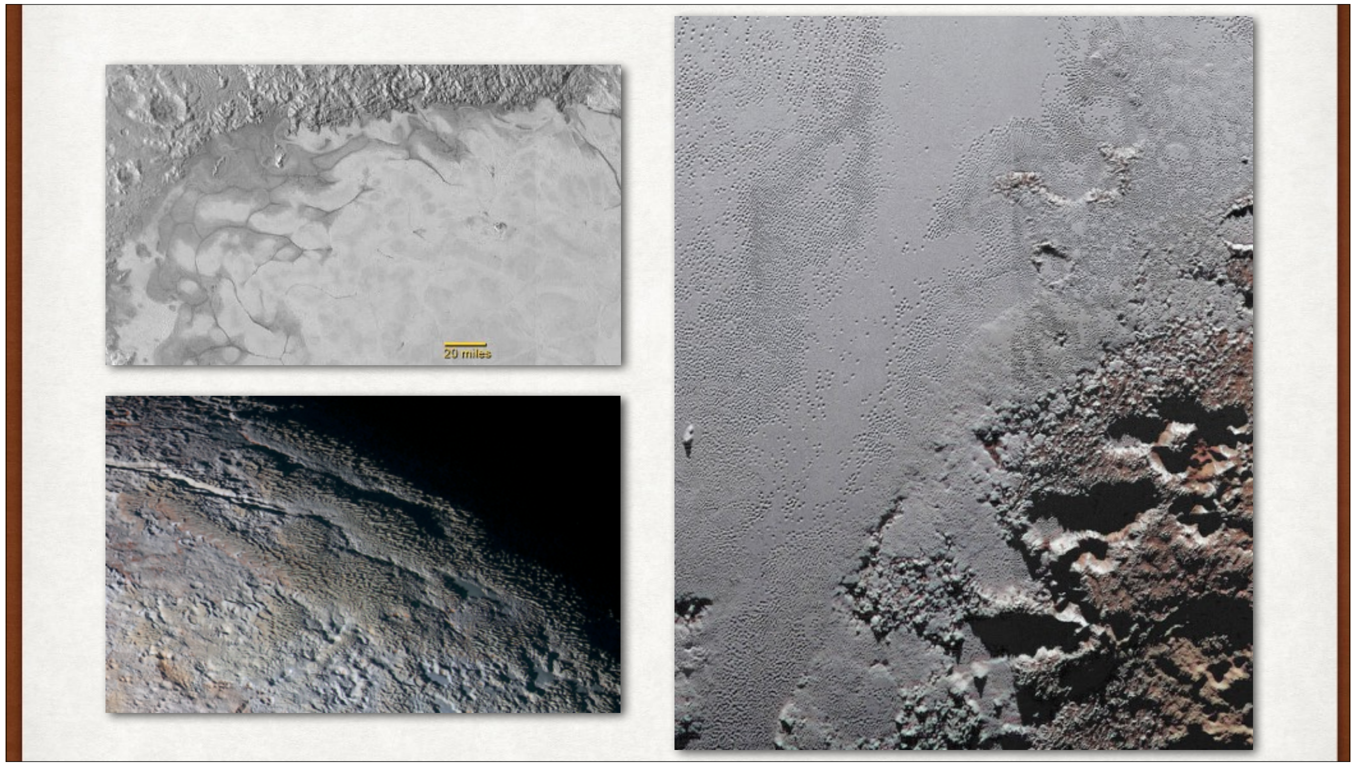
When is our system “good enough” to “deploy,” or make available in the real world? There is currently no way to predict this. The only way to know is to give it new data that it has never seen before, and see how it does. If it performs well enough, we’re set. If not, it’s up to us to somehow make it 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.”

How well does our system perform on new data? We need to set aside the test set. But what if we have only a small dataset, and every sample is precious? We definitely don’t want to NOT train with everything! Suppose we’re using pictures of Pluto to train a terrain classifier.



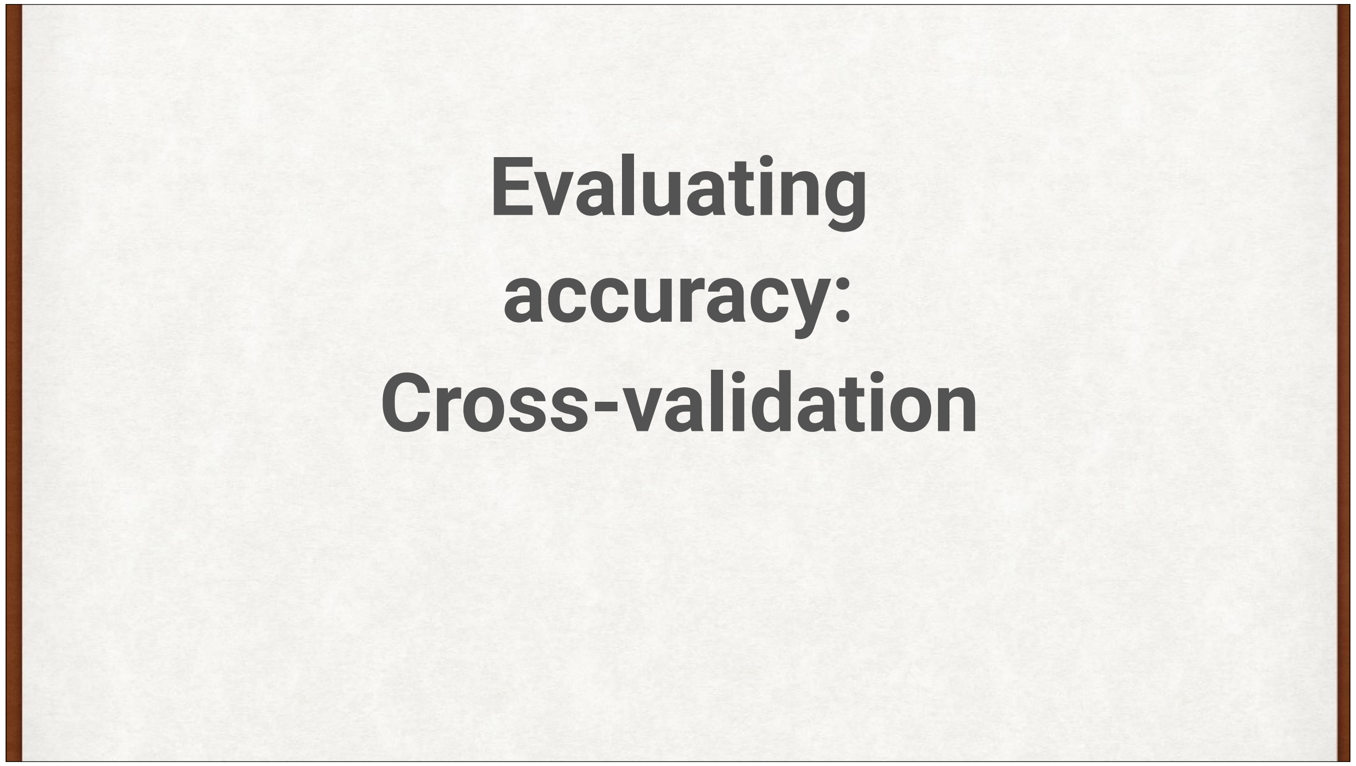
Pluto and Charon (New Horizons probe). Precious data. Can’t get more.

https://solarsystem.nasa.gov/planets/dwarf-planets/pluto/galleries/?page=0&per_page=25&order=created_at+desc&search=&href_query_params=category%3Dplanets%2Fdwarf-planets_pluto&button_class=big_more_button&tags=pluto&condition_1=1%3Ais_in_resource_list&category=51



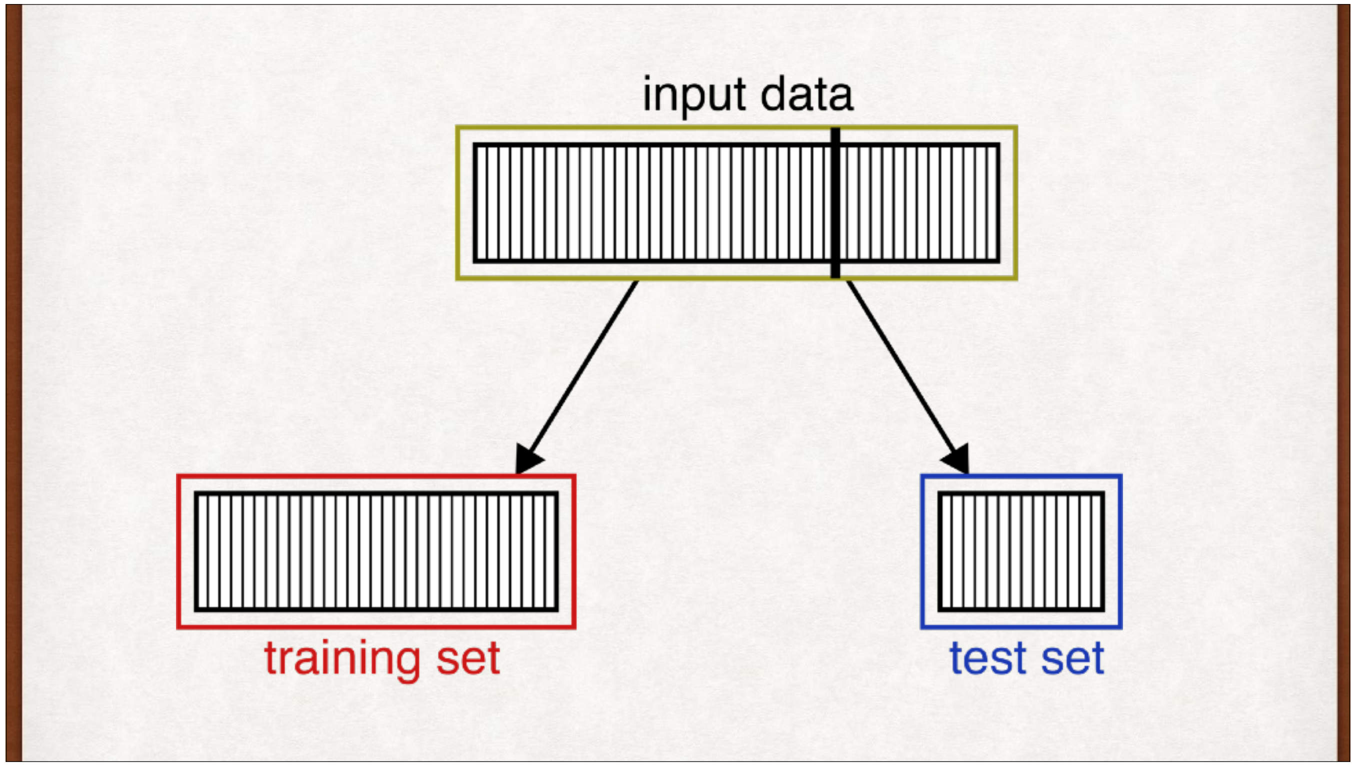
Pluto (New Horizons probe) UL Glacial flows, R icy shoreline & snowy pits, LL: “strange snakeskin terrain”

https://solarsystem.nasa.gov/planets/dwarf-planets/pluto/galleries/?page=0&per_page=25&order=created_at+desc&search=&href_query_params=category%3Dplanets%2Fdwarf-planets_pluto&button_class=big_more_button&tags=pluto&condition_1=1%3Ais_in_resource_list&category=51

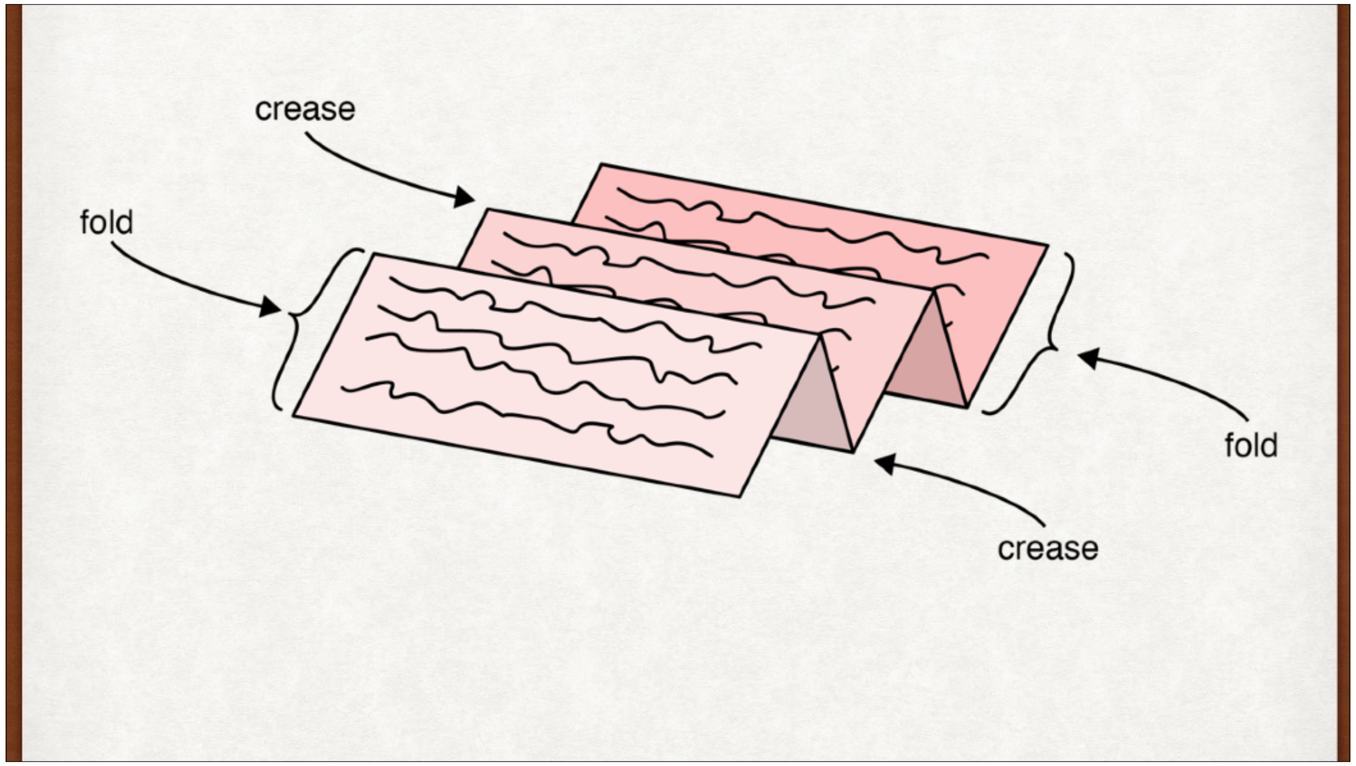


Evaluating accuracy: Cross-validation

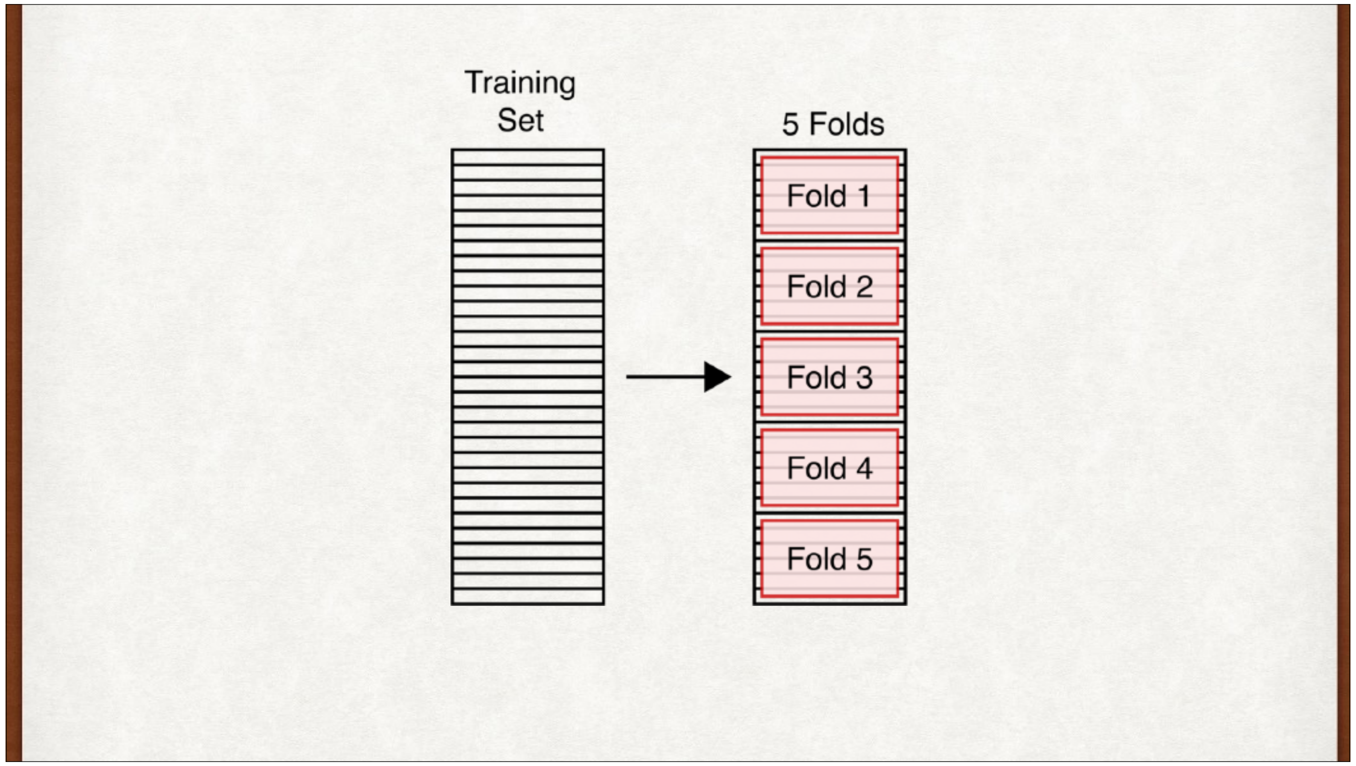
Cross-validation is a way to *estimate* how well our system generalizes, while letting us train with all the data (that is, not holding back a test set).



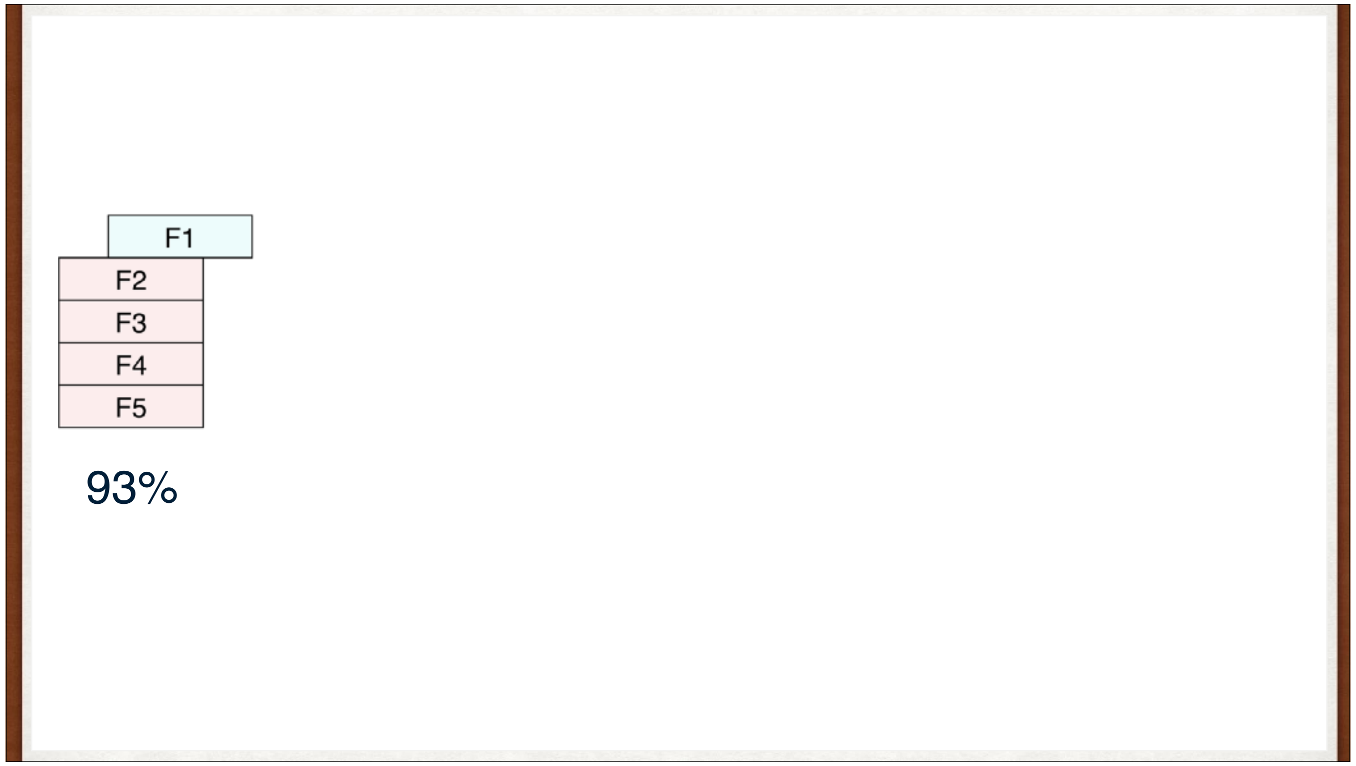
Traditionally, we split the incoming data into training and test sets. We train with one, and then evaluate performance at the very last step with the other. It's essential that the system never sees the test set during training, or we get "data leakage," and probably an overly optimistic measure of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



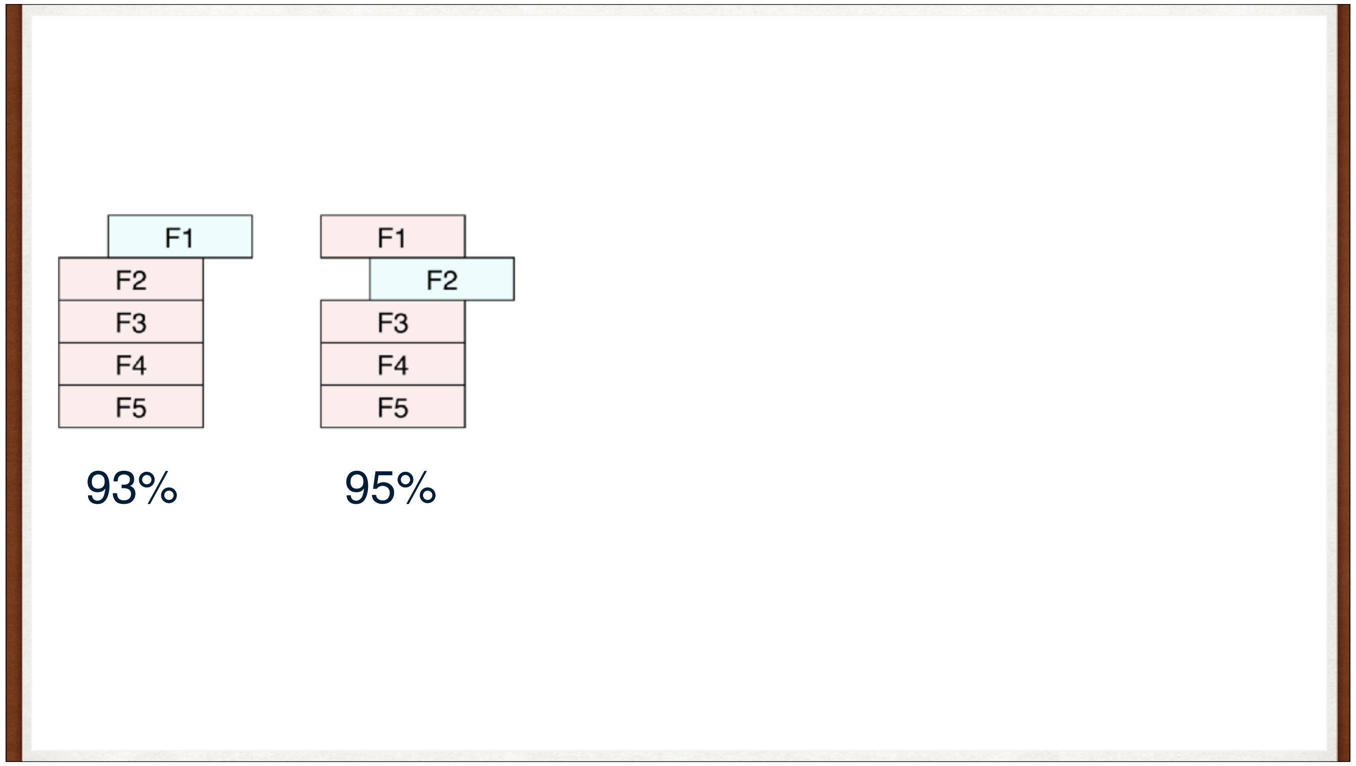
When we don’t have a lot of data, it would be a shame to set aside a separate validation set, because we want to use all of our data for training. Suppose we have 500 pictures from Pluto. We want to use every single one of them in training. And we can’t get more. To estimate accuracy, we can use cross-validation. 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.



CV is also called **rotation estimation** or **out-of-sample testing**. To evaluate the 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.



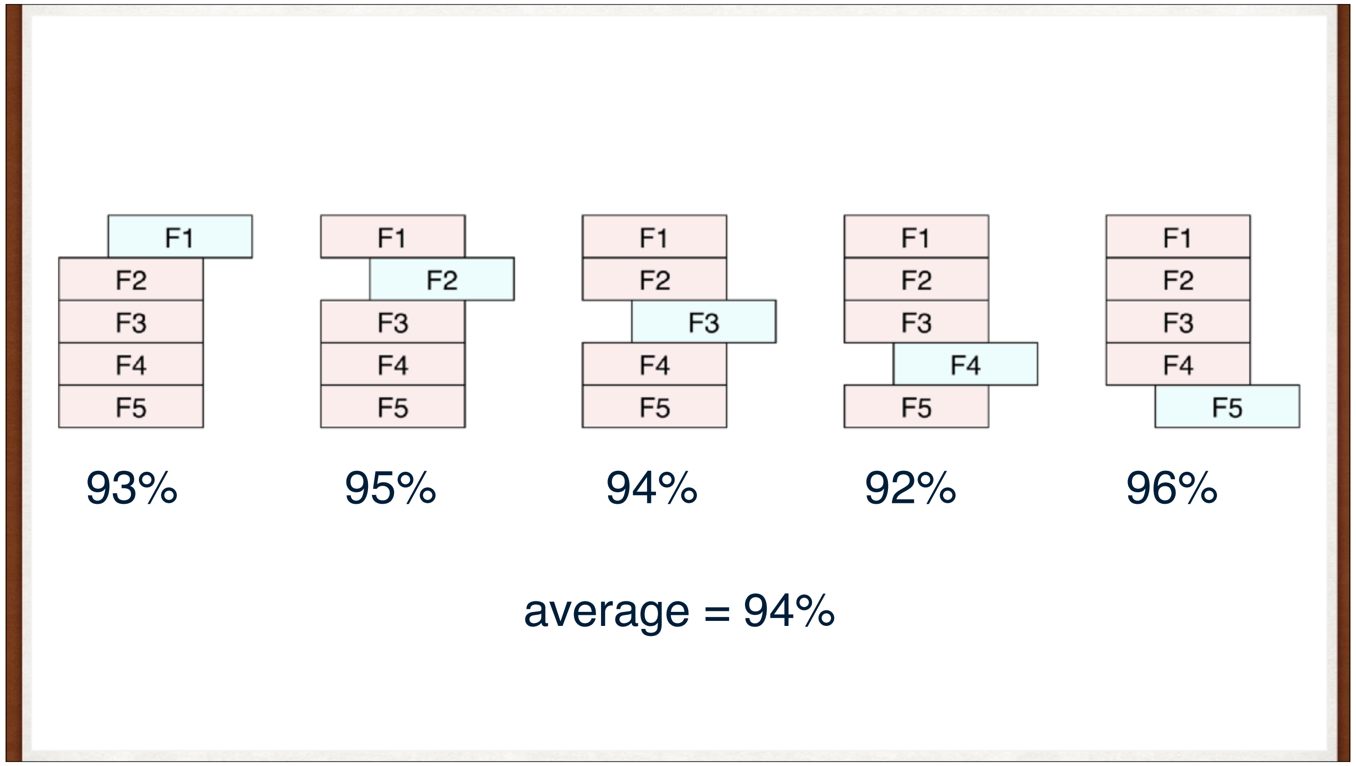
F1
F2
F3
F4
F5

93%

F1
F2
F3
F4
F5

95%

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.

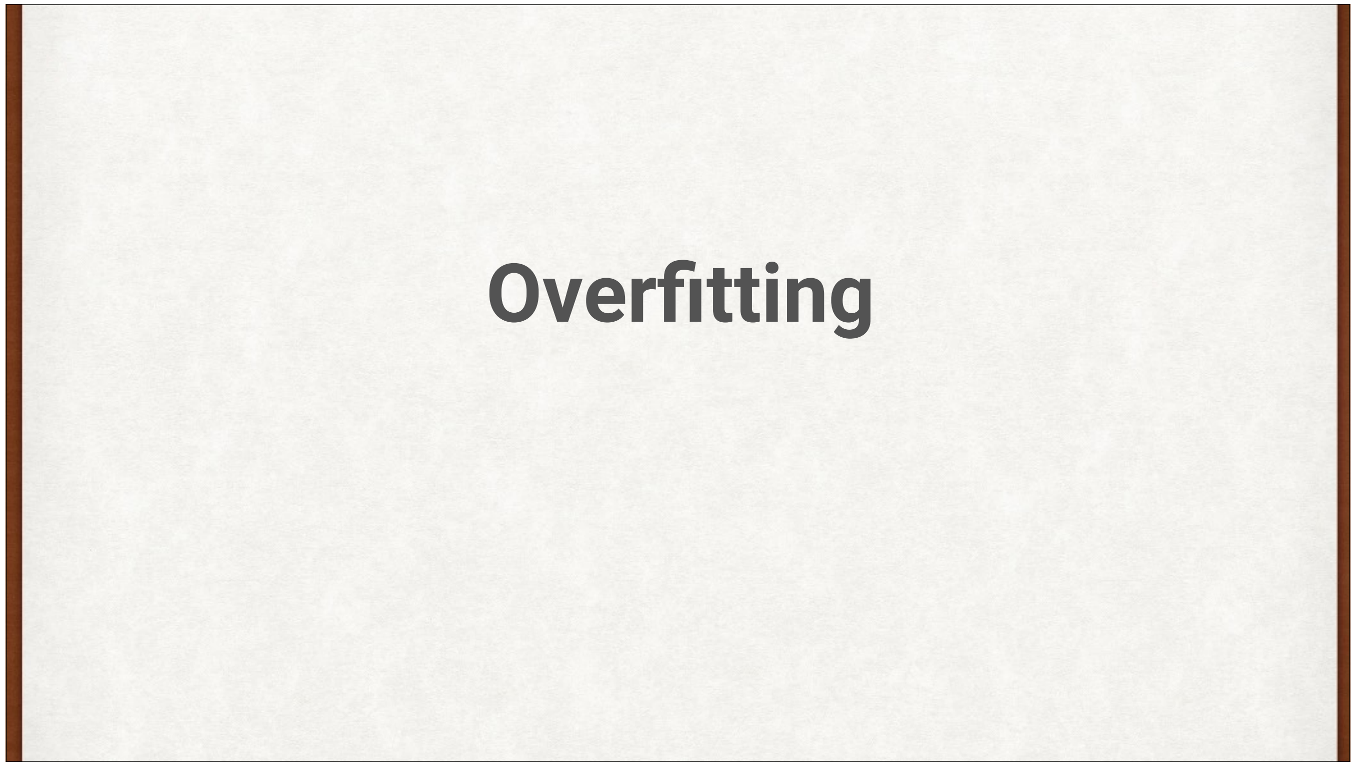


93% 95% 94% 92% 96%

average = 94%

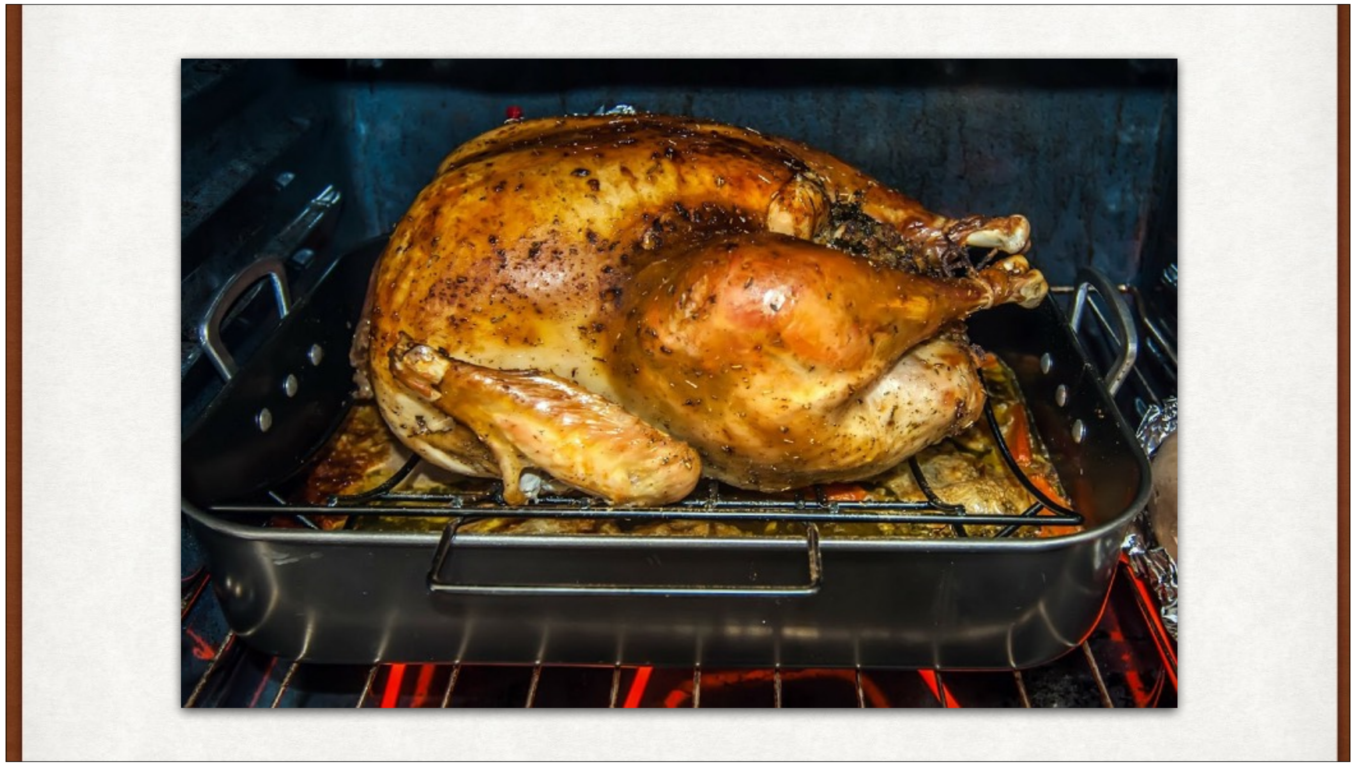
Also **rotation estimation** or **out-of-sample testing**. To evaluate performance of our network, we'll train on folds 2 through 5, then test (or validate) with fold 1. Then we'll train with folds 1 and 3-5, and test (or validate) with fold 2. When we've done this for all 5 folds, we average the performance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

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



Overfitting

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



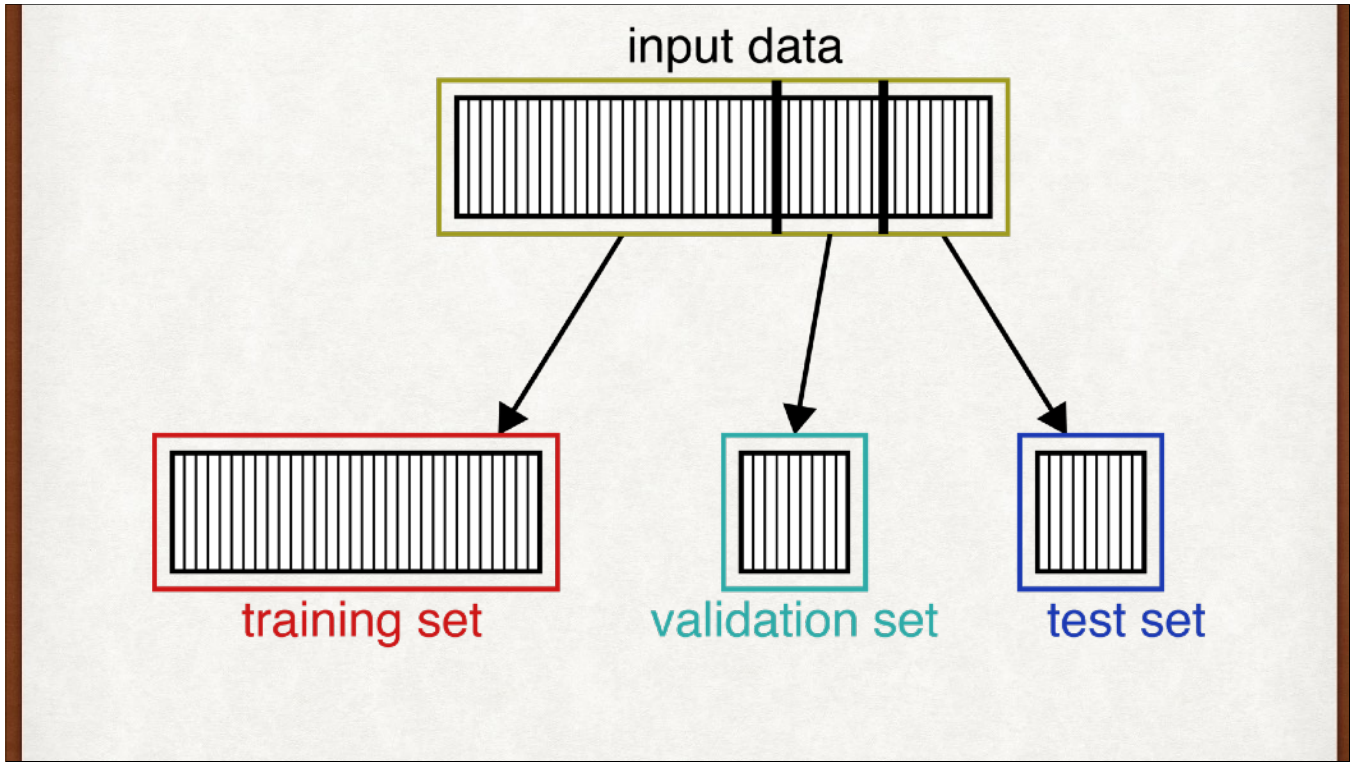
Cooking is good…

https://pixabay.com/en/turkey-oven-dinner-meal-cooking-218742/

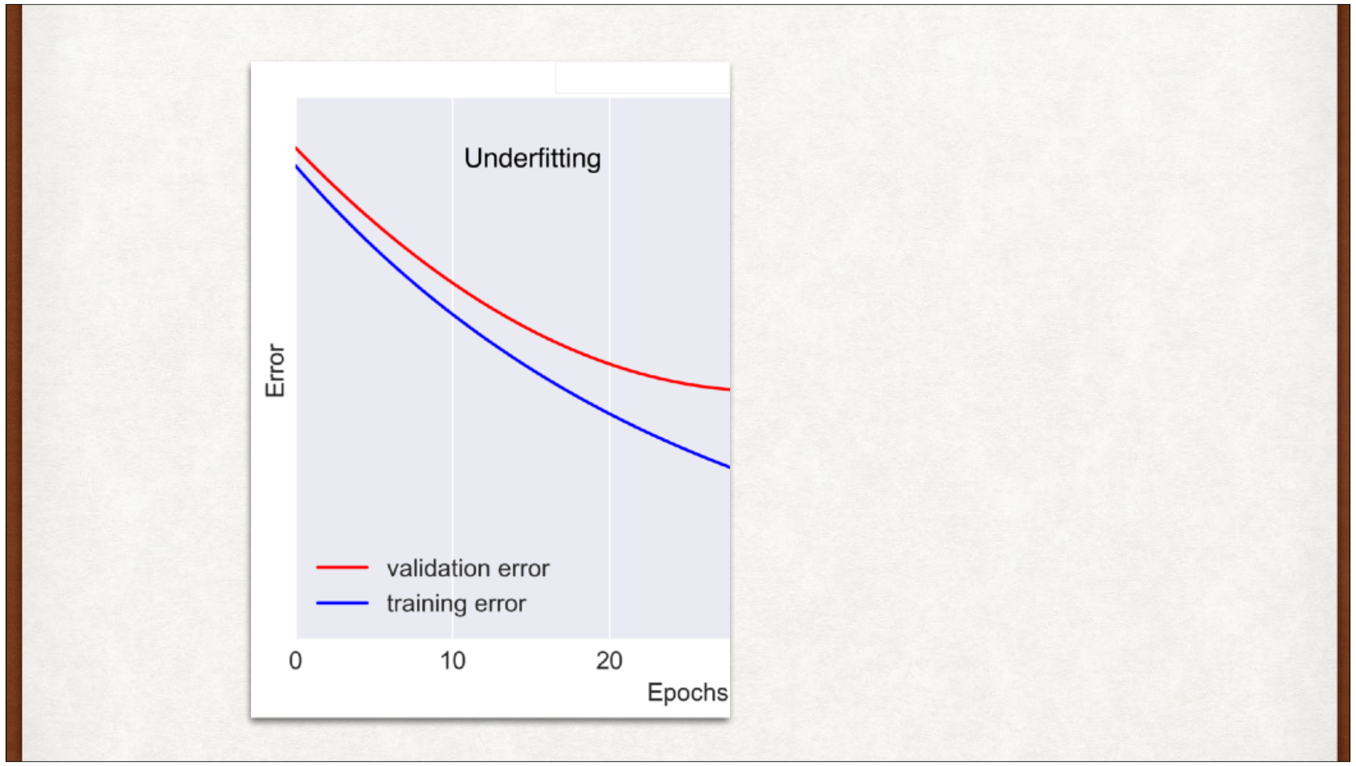
But cook too long and this will happen. It’s just the natural result of cooking too long.
https://pixabay.com/en/coal-cabbage-burned-fuel-black-842468/

You can cook for longer, without burning, by taking precautions.

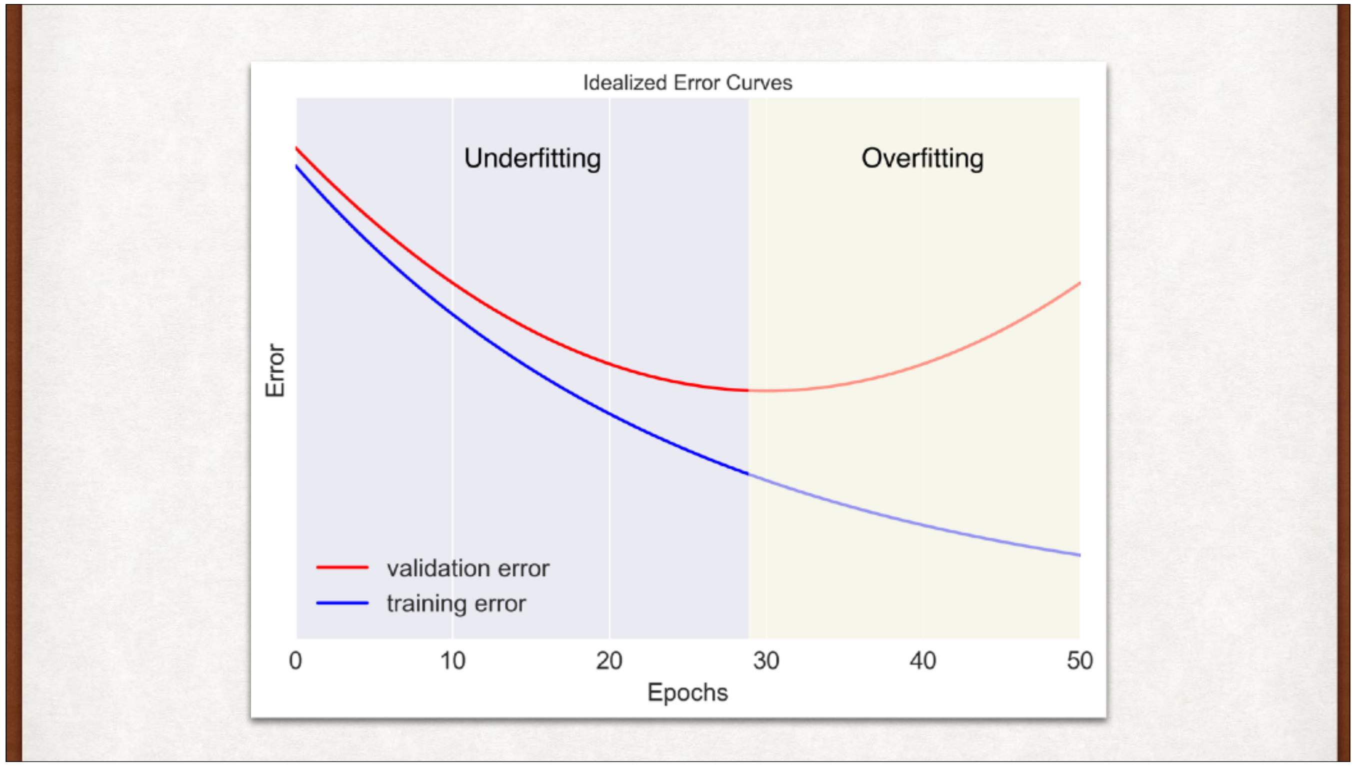
https://pixabay.com/en/grill-aluminum-foil-ash-barbecue-1853674/



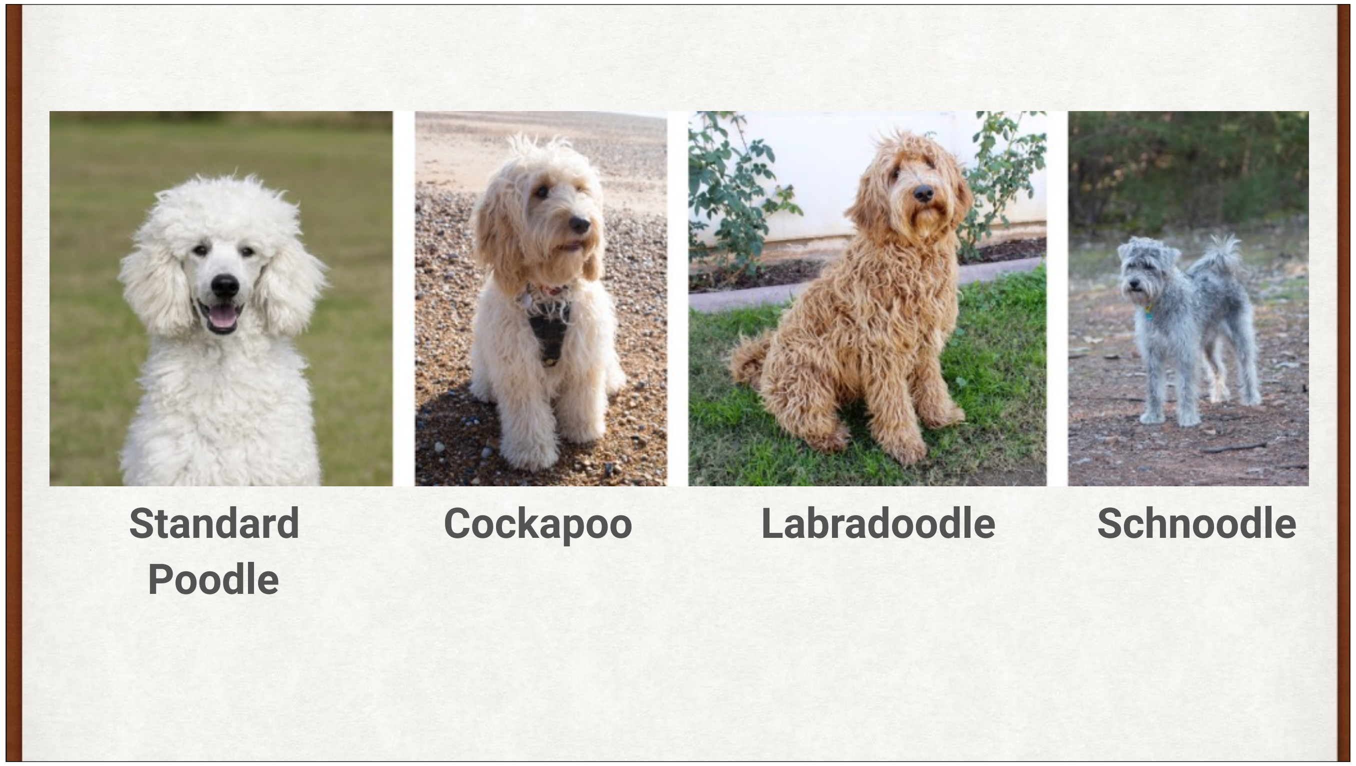
If we’re going to try a bunch of variations of our network, we can break out a separate **validation set** to let us evaluate each variation. Then we pick the one that performed best on the validation set, and *estimate* its readiness for deployment with the test set.



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?



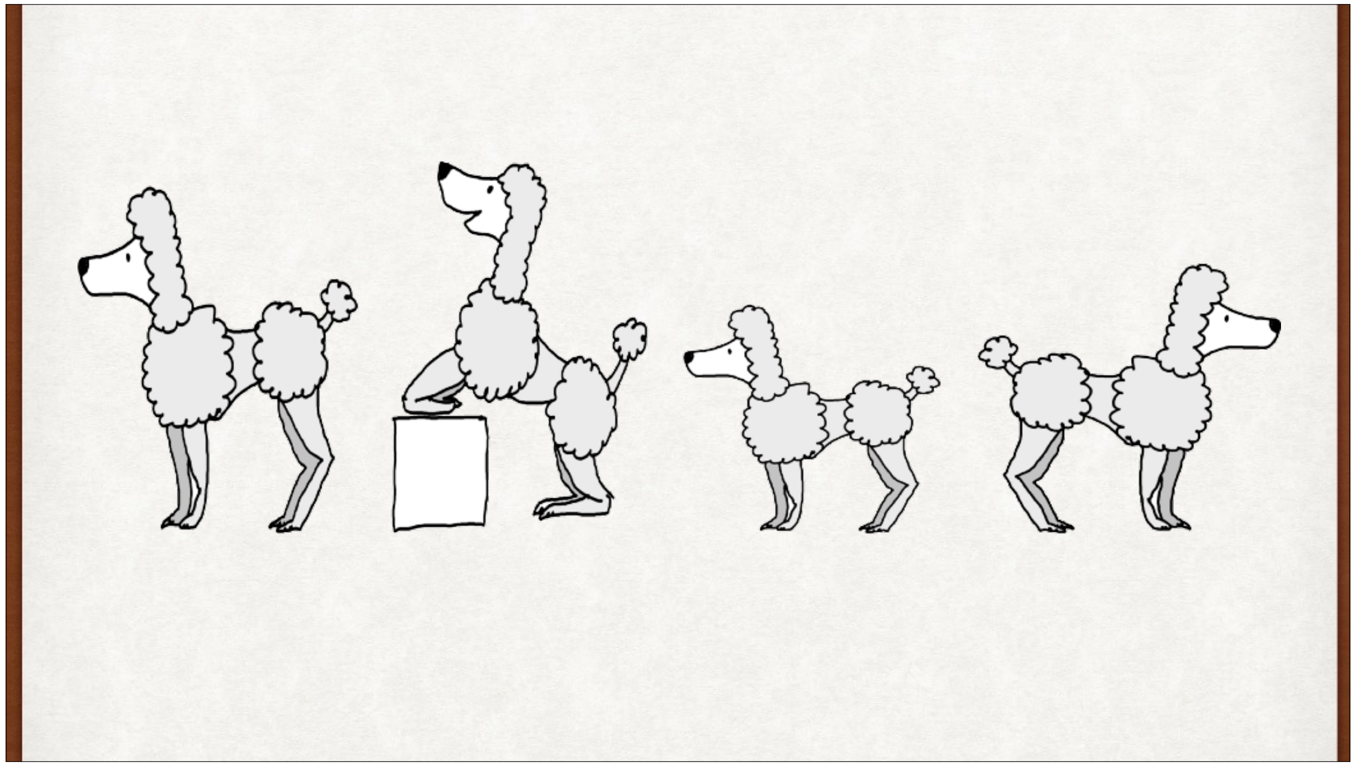
Standard Poodle

Cockapoo

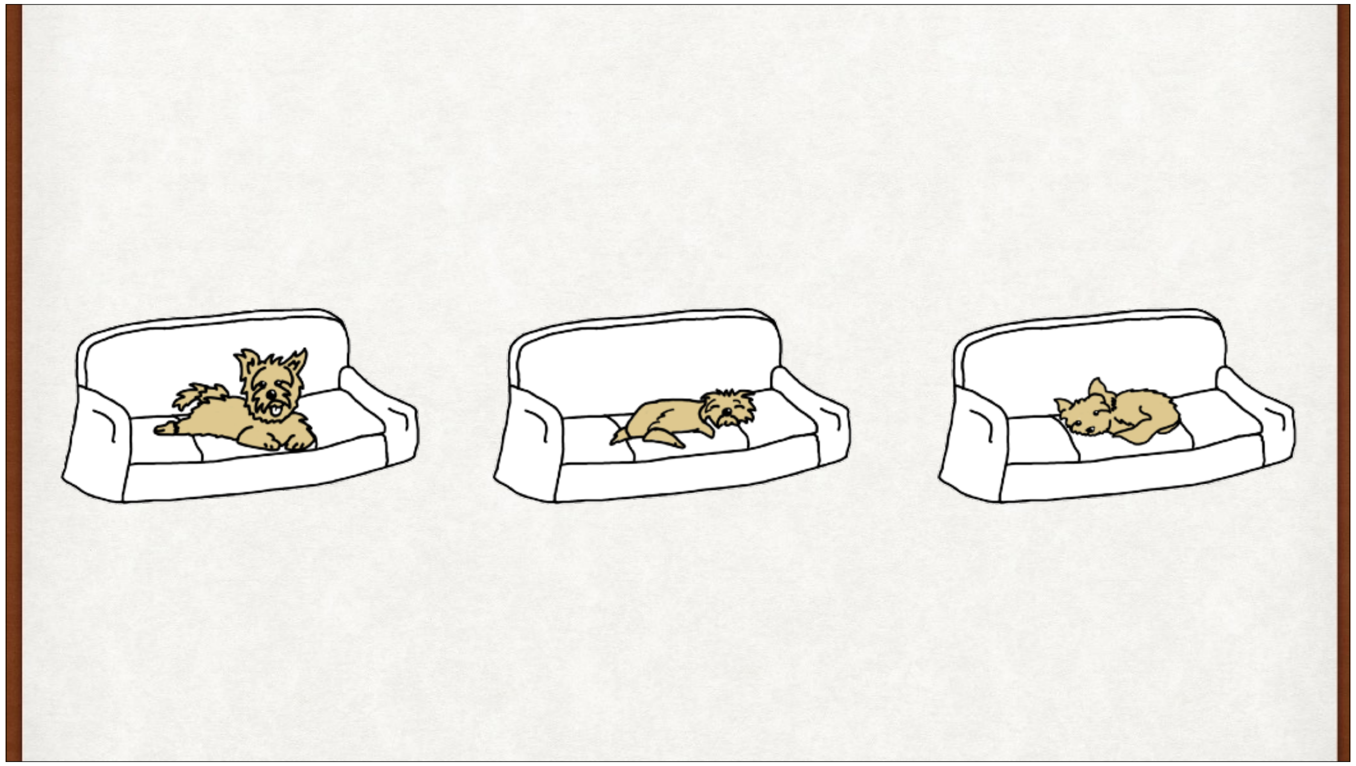
Labradoodle

Schnoodle

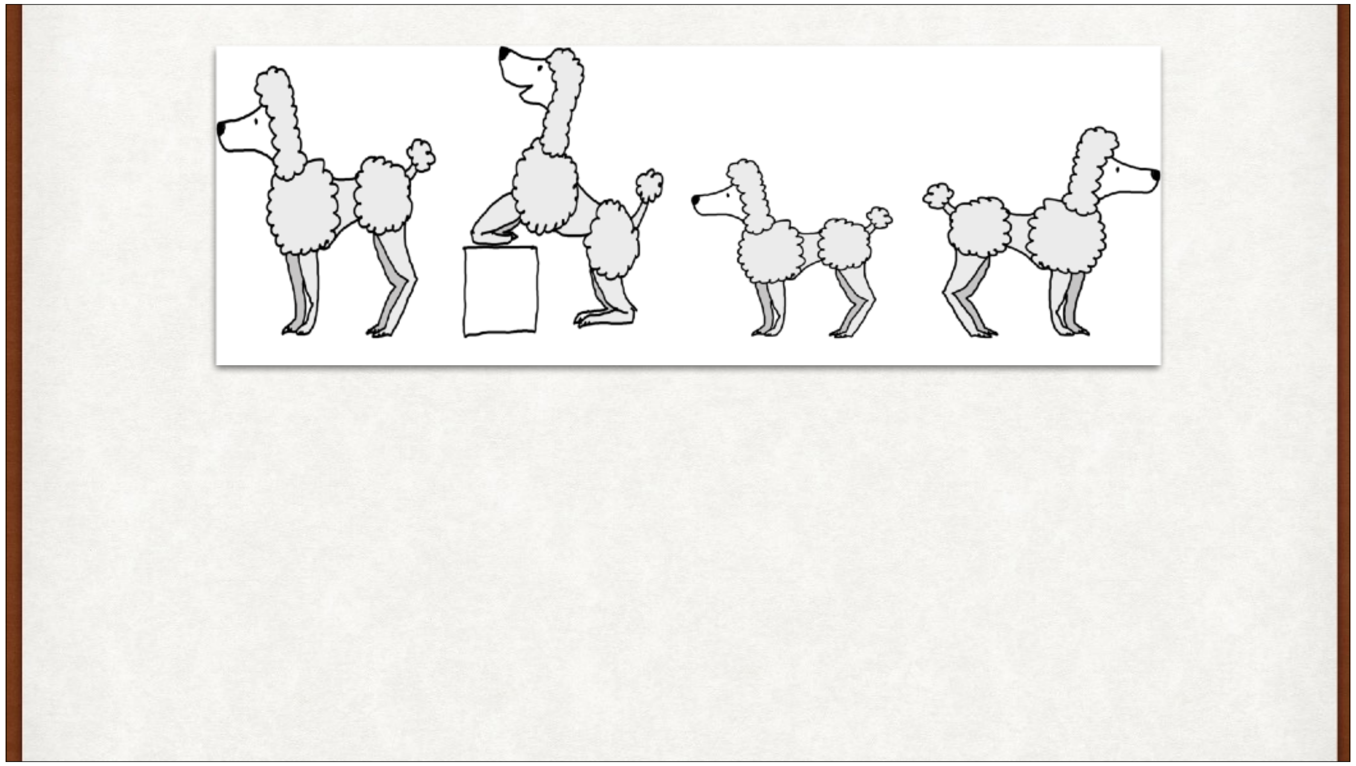
So many kinds of poodle! Let’s learn how to classify different breeds of dogs. Cocker spaniel, Labrador retriever, schnauzer.
https://pixabay.com/en/poodle-the-poodle-dog-the-dog- breed-1561405
https://pixabay.com/en/beach-dog-water-sea-cockapoo-2239440
https://pixabay.com/en/labradoodle-lab-dog-puppy-2441859
https://pixabay.com/en/dog-schnoodle-pet-animal-canine-2689514



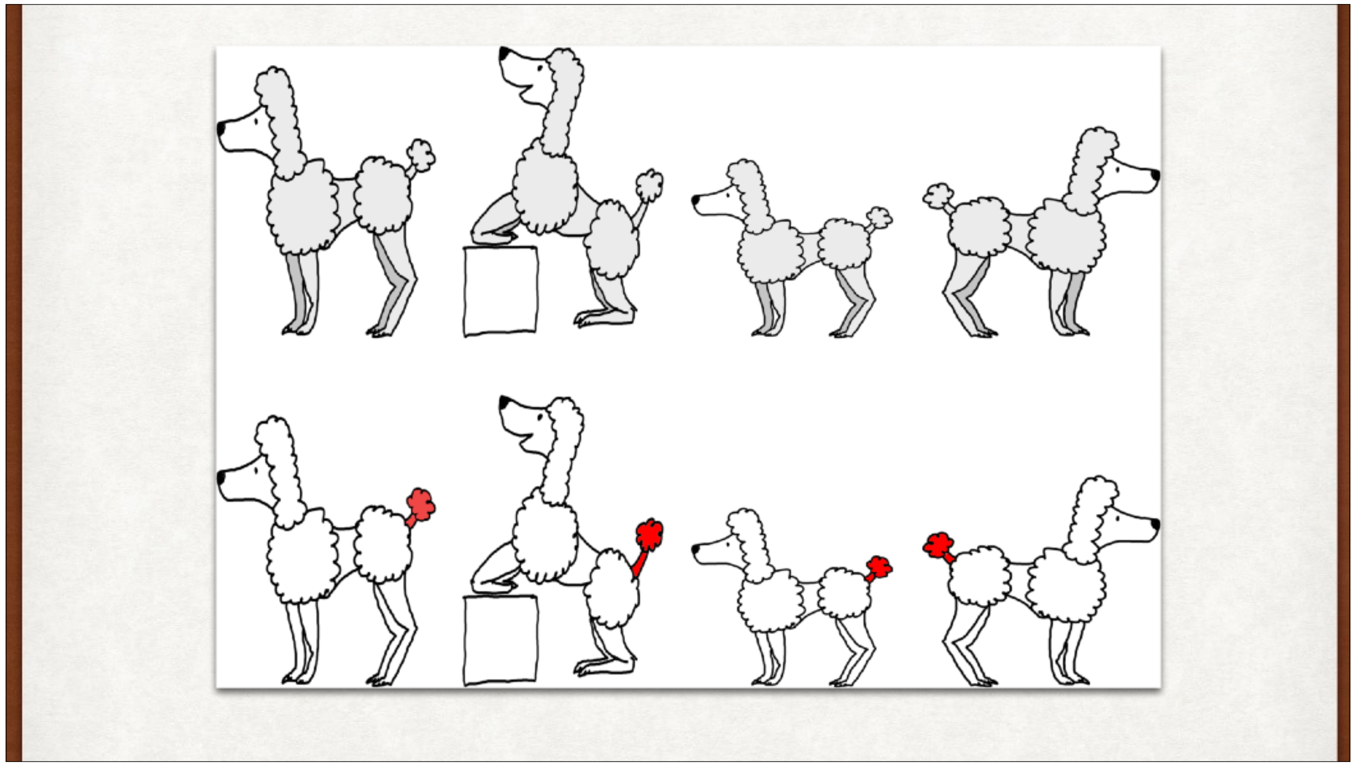
Some images of Poodles to train on.



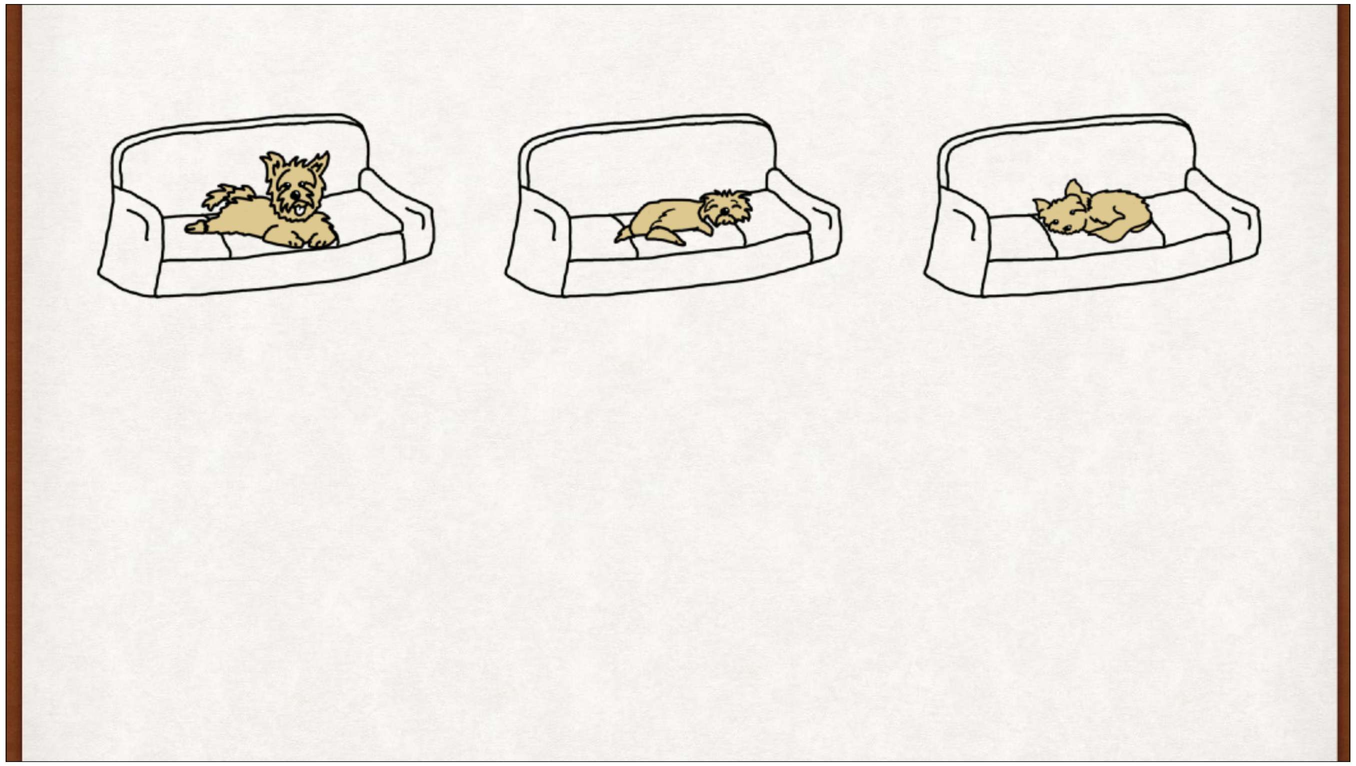
Yorkshire Terrier training data.



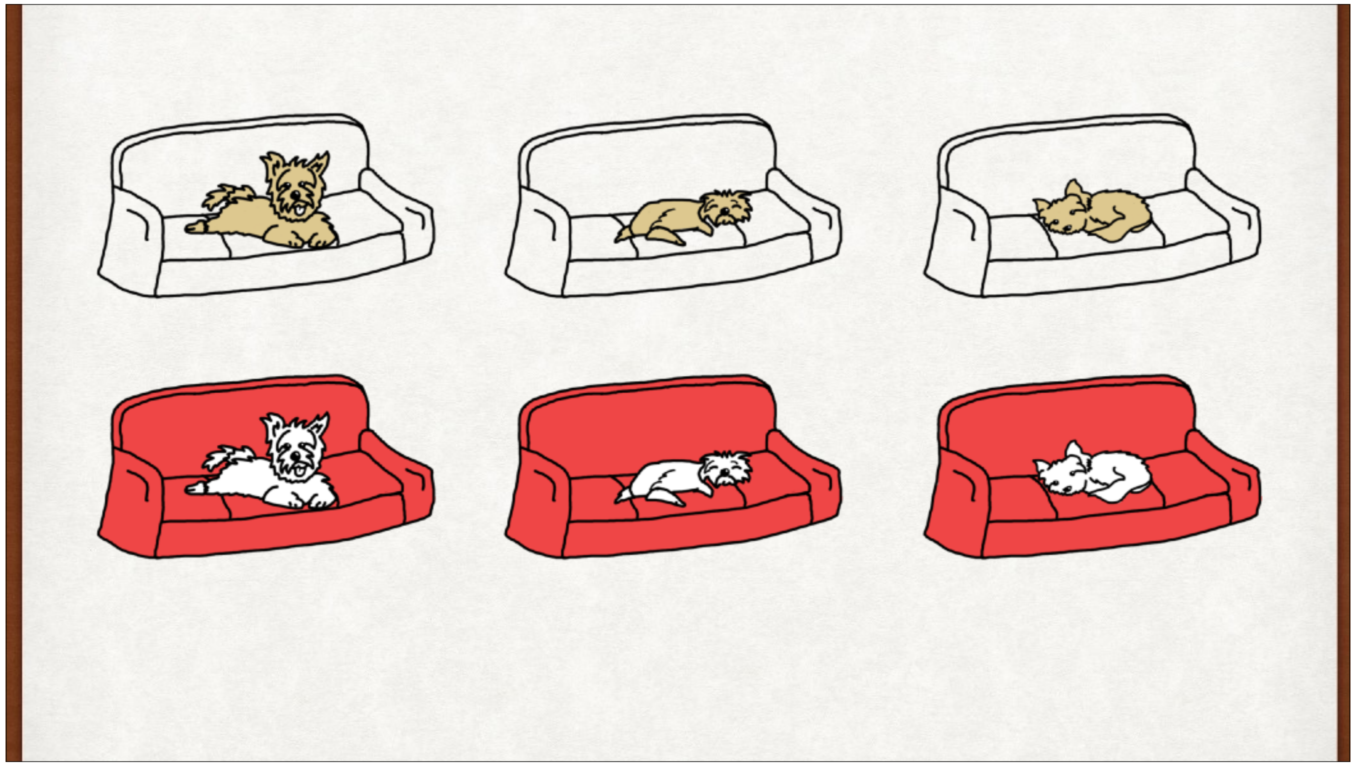
Unknown to us, the system recognizes the poodle just by the tail and fluffy ball at the end. No other dog images in our training set have this, so the system ignores everything else. This gives it great performance on the training data. But it’s just an idiosyncratic accident in the training data. We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



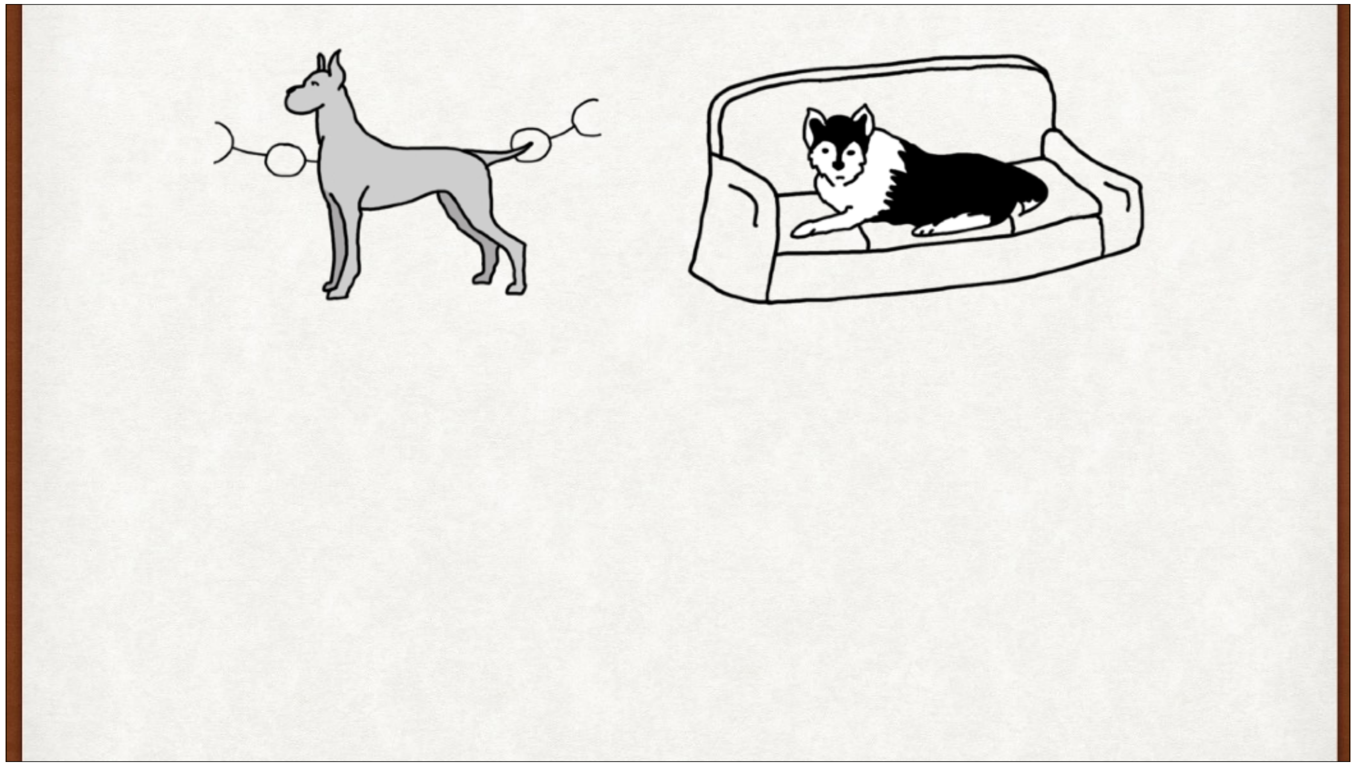
Unknown to us, the system recognizes the poodle just by the tail and fluffy ball at the end. No other dog images in our training set have this, so the system ignores everything else. This gives it great performance on the training data.



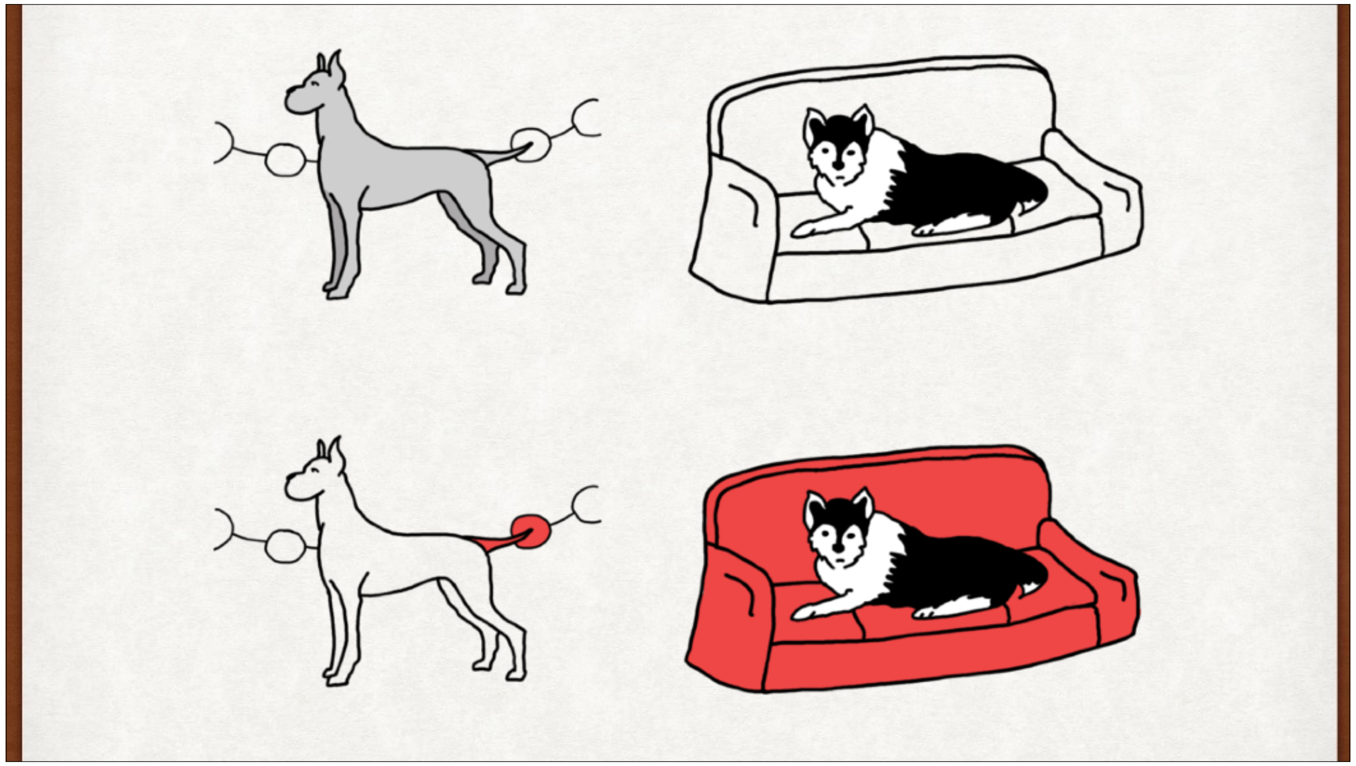
And the system isn’t recognizing Yorkshire Terriers. In our data, all Yorkies are on couches, and no other dogs are. So if the system sees a couch, it says the dog is a Yorkie. Again, that’s great for the training data, for which this just happens to be true.



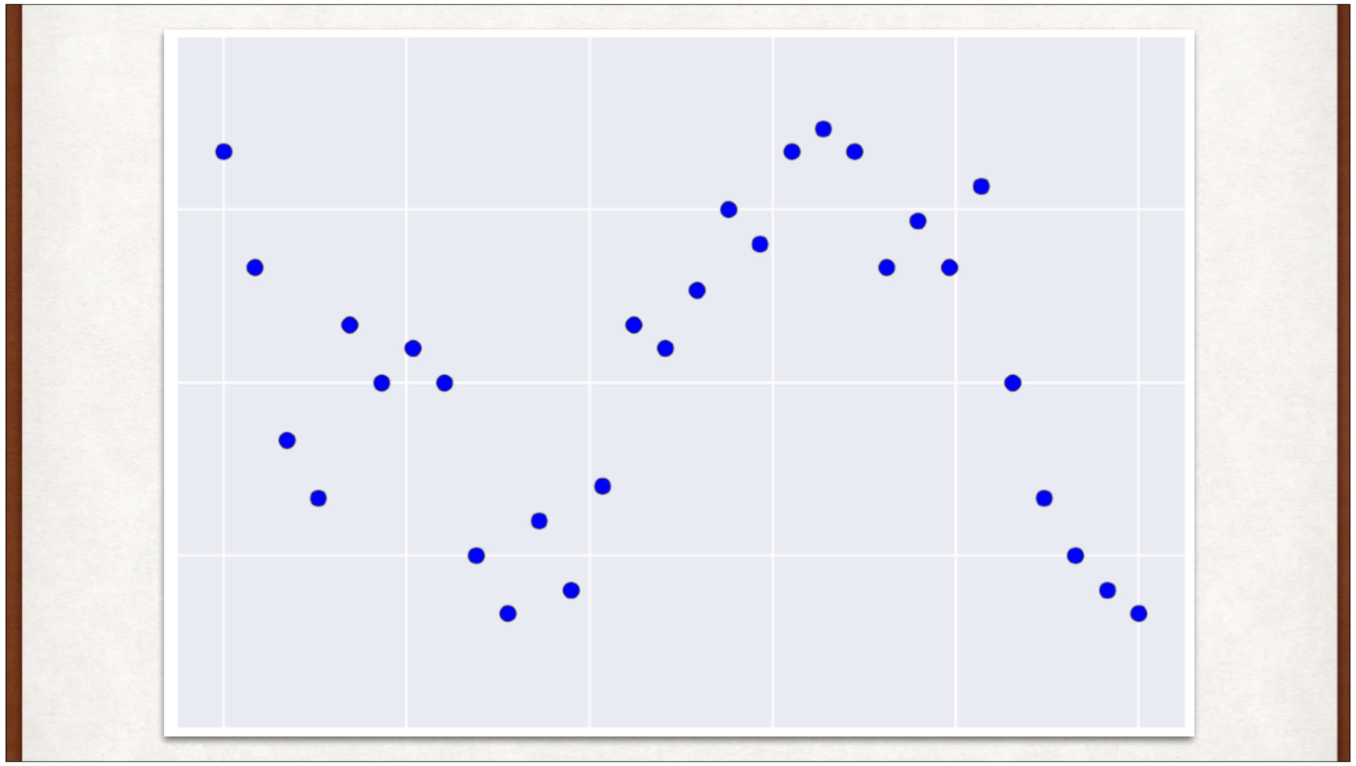
And the system isn’t recognizing Yorkshire Terriers. In our data, all Yorkies are on couches, and no other dogs are. So if the system sees a couch, it says the dog is a Yorkie. Again, that’s great for the training data, for which this just happens to be true.



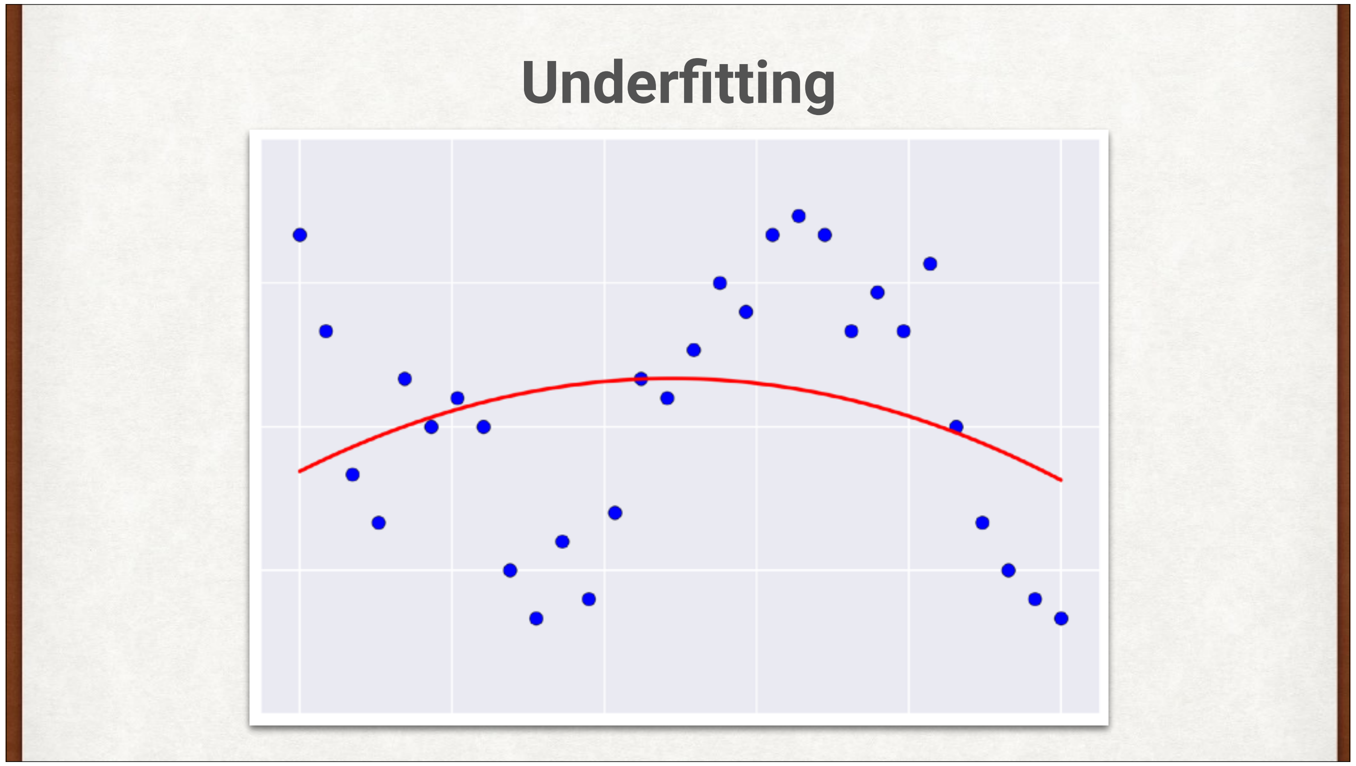
And now we get into trouble when we evaluate new images. Here, the Great Dane is in front of a holiday display with balls on a string. The system sees the tail and ball and says, “Poodle.” The Huskie is on a couch. The system sees the couch and says, “Yorkshire Terrier.” We have overtrained our system 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.

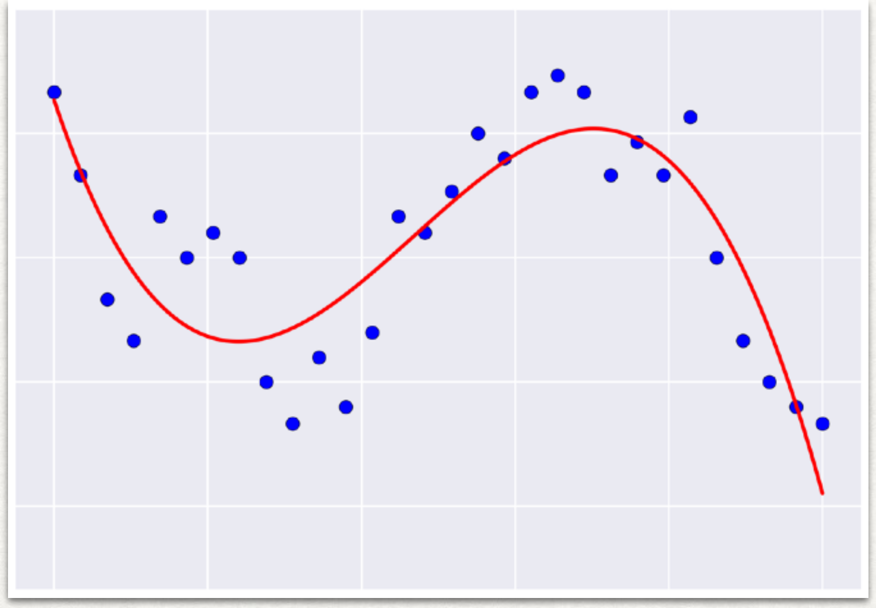


Some data we’d like to describe in a general way.

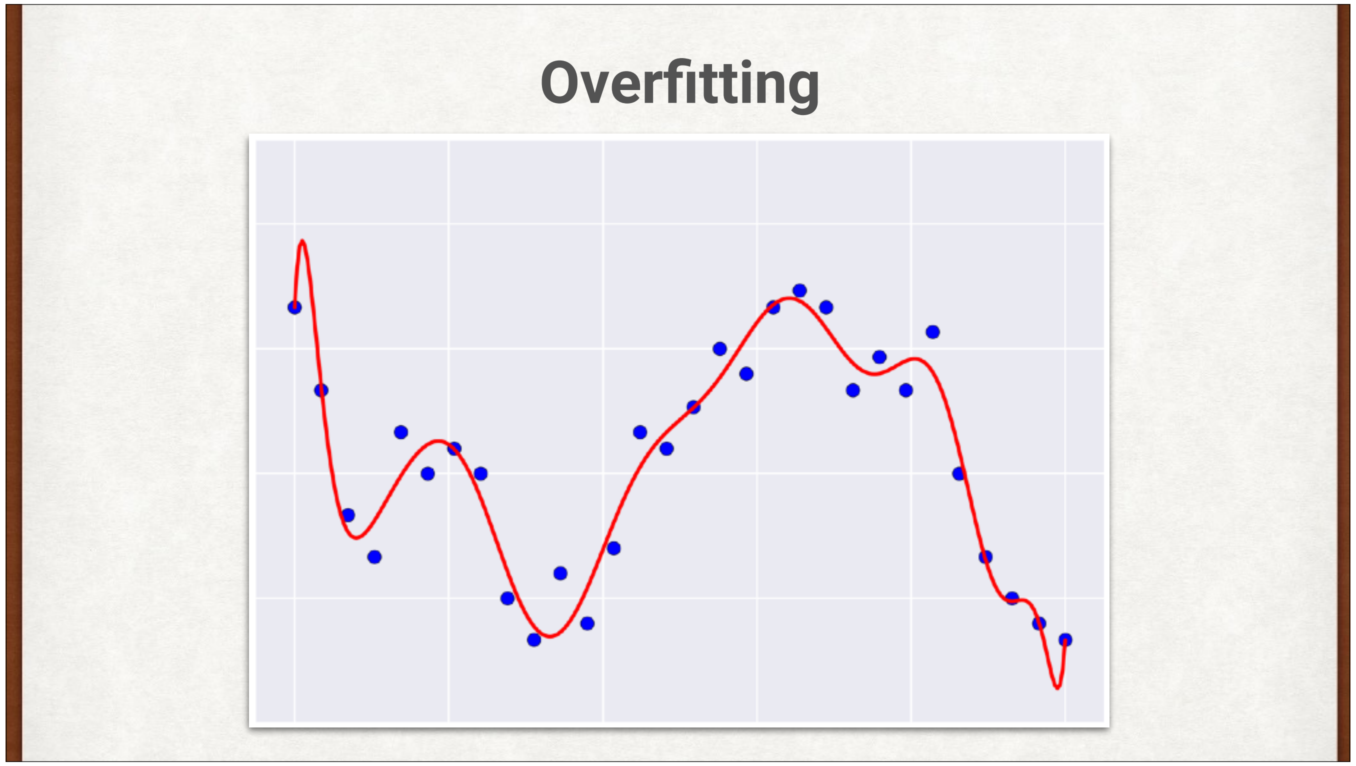


Underfitting

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.

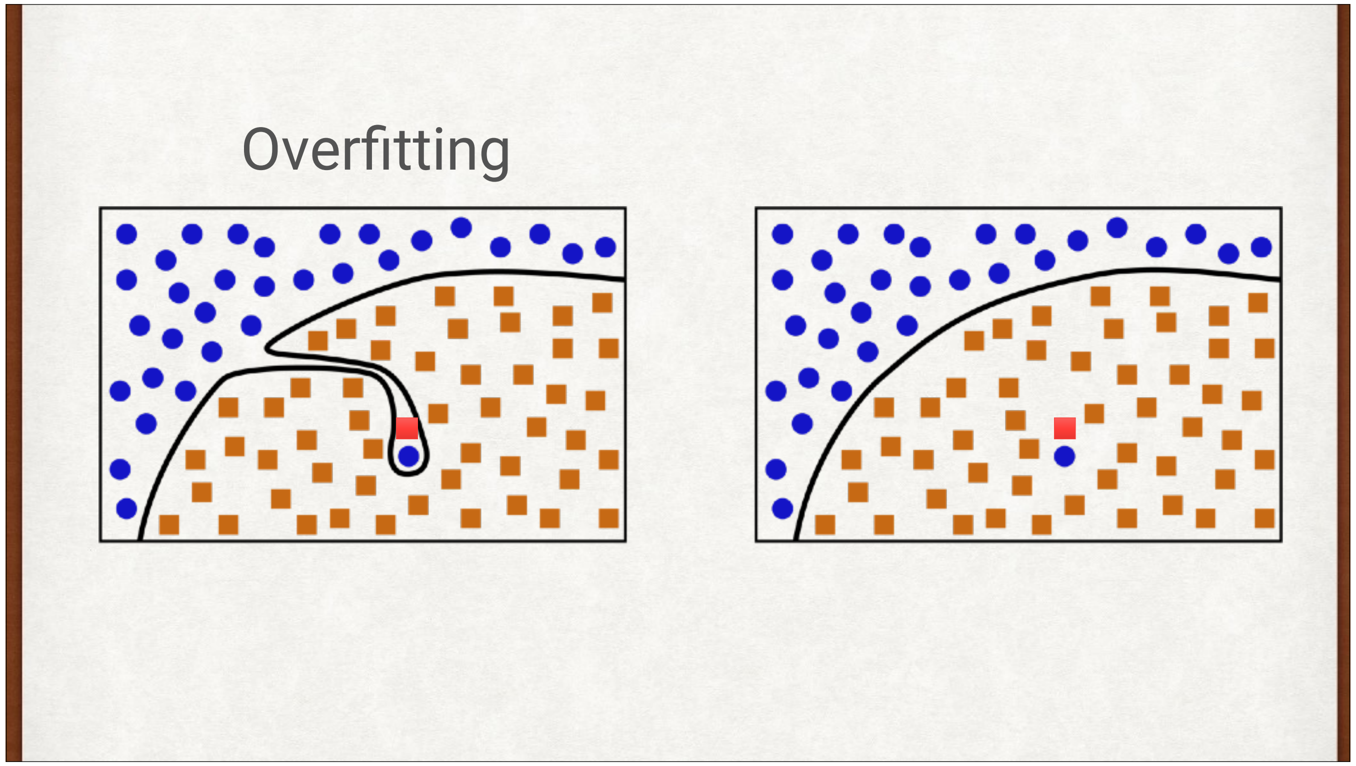


A better fit to the data.



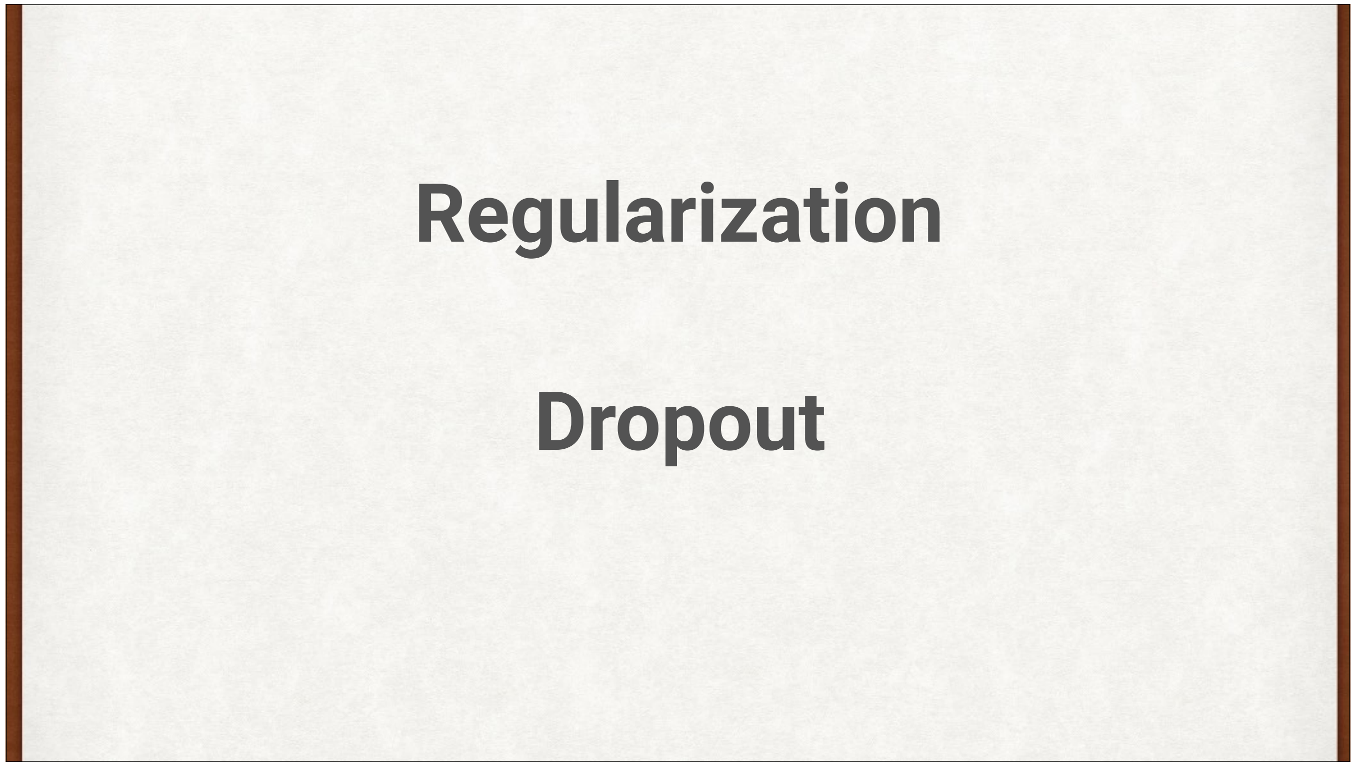
Overfitting

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



Overfitting

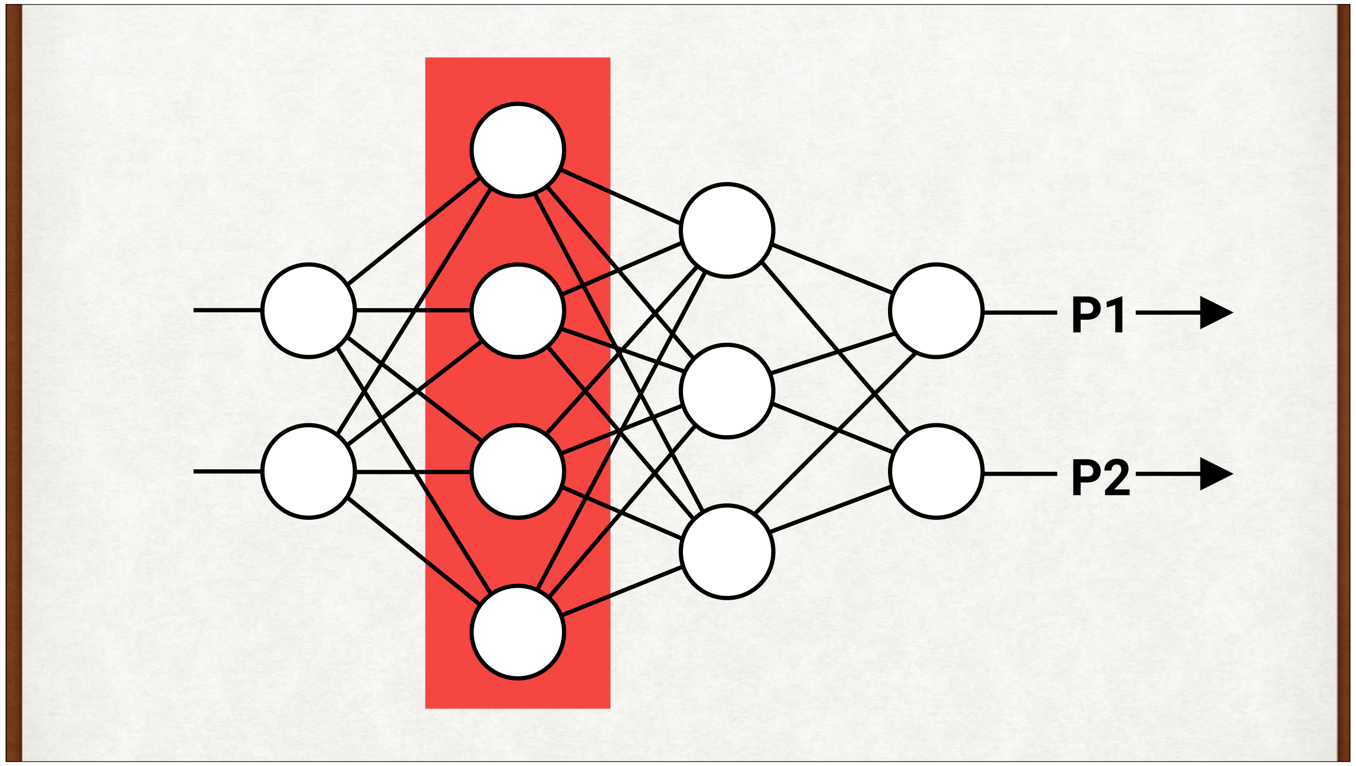
On the left, we're probably overfitting, by paying too much attention to one weird piece of data. That's likely to be an oddball event. On the right is (probably) a better curve. In this simple case we're making a subjective decision that the one lone blue circle is an anomaly, but we can automate this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



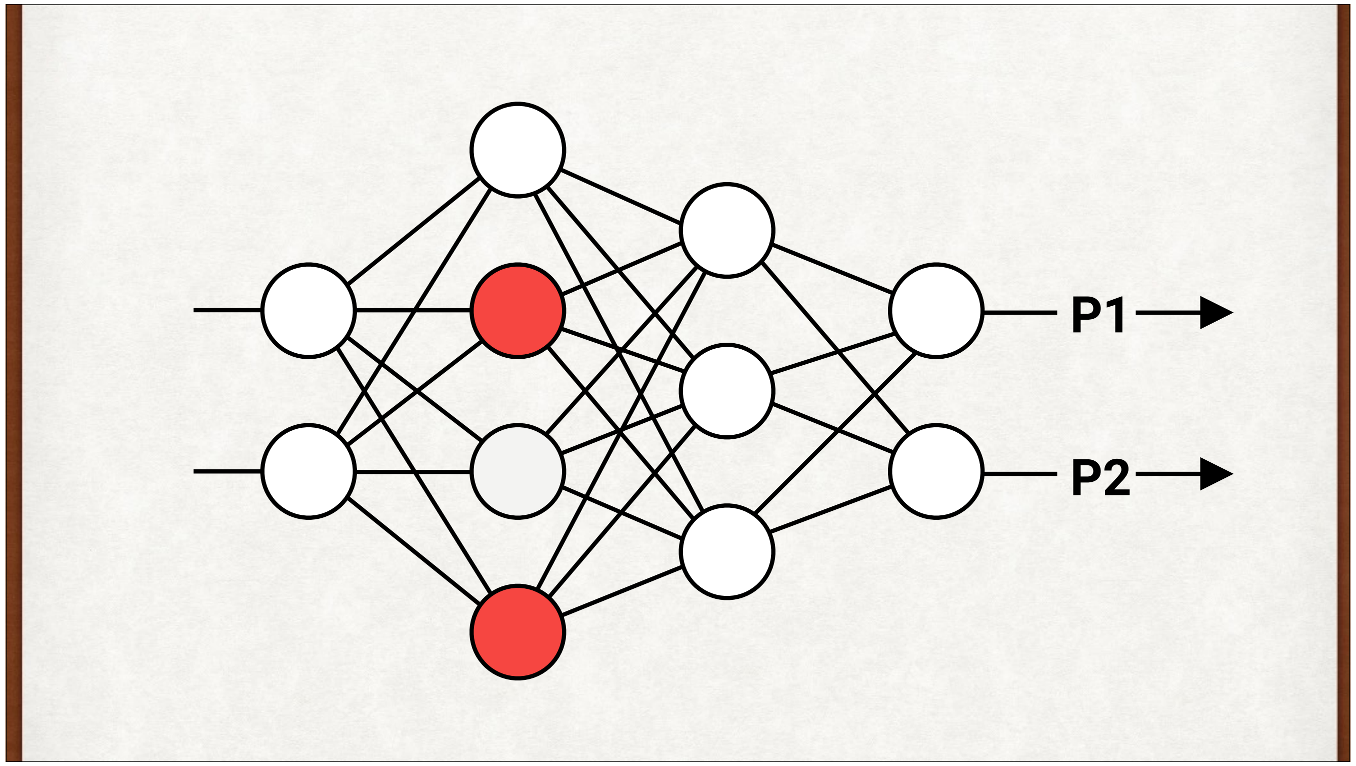
Regularization

Dropout

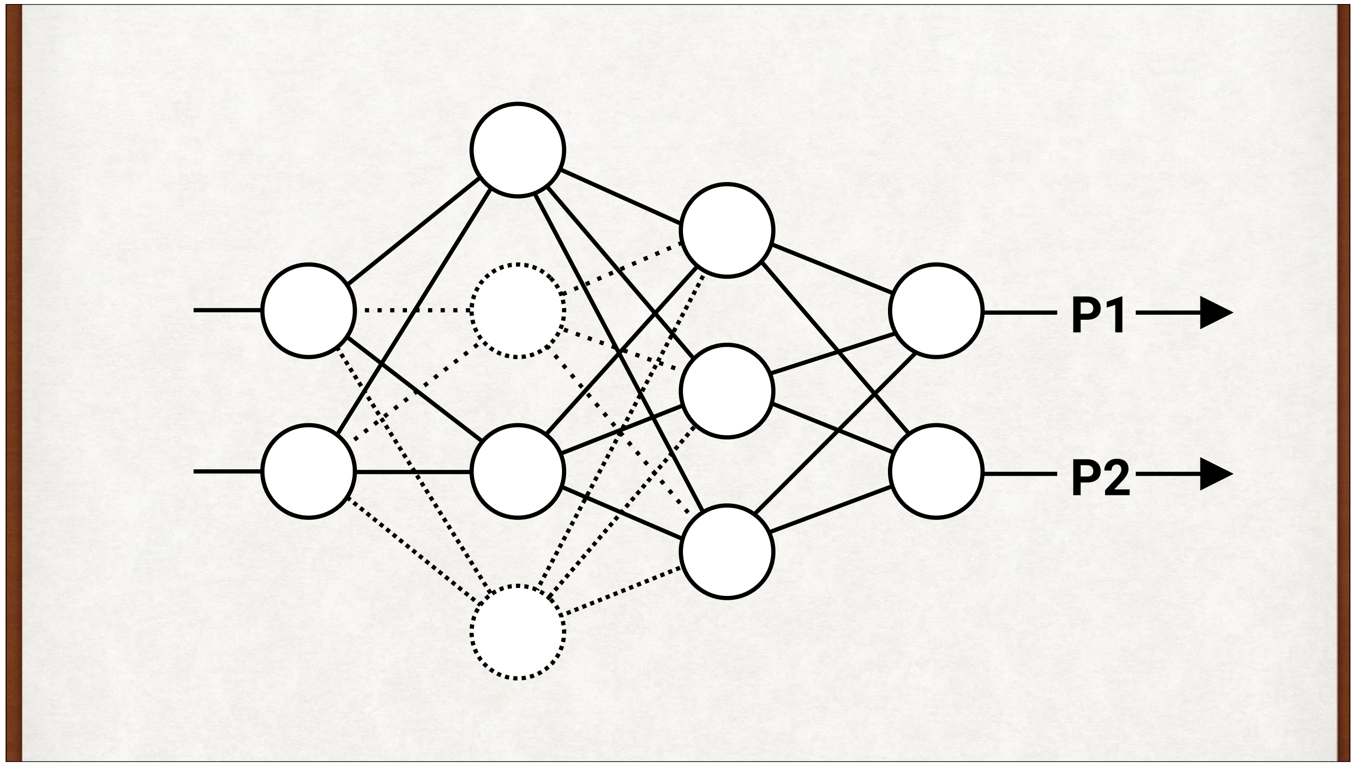
Regularization methods let us put off overfitting.



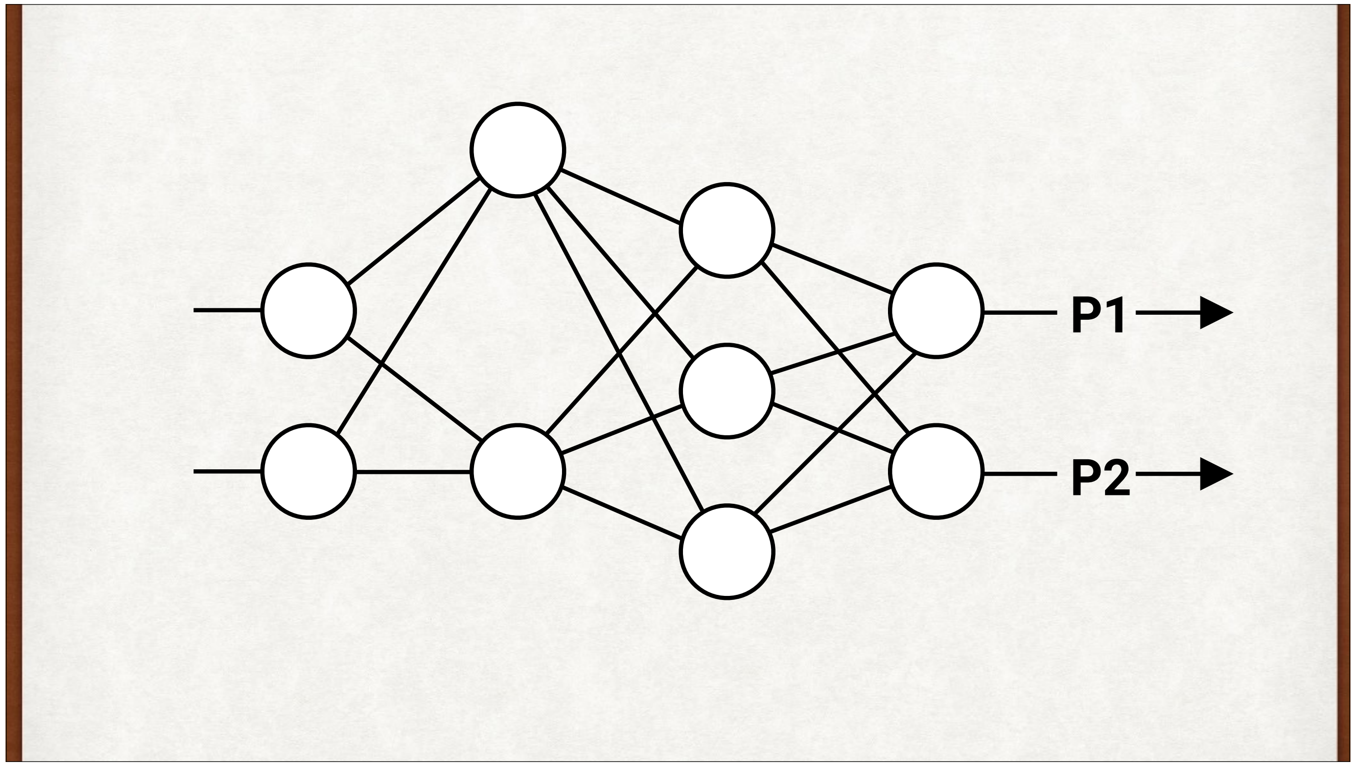
Dropout is popular, easy, and effective. At the start of each epoch, randomly pick some nodes from a given layer and disconnect them. Put them back in at the end of the epoch, and randomly pick some others.



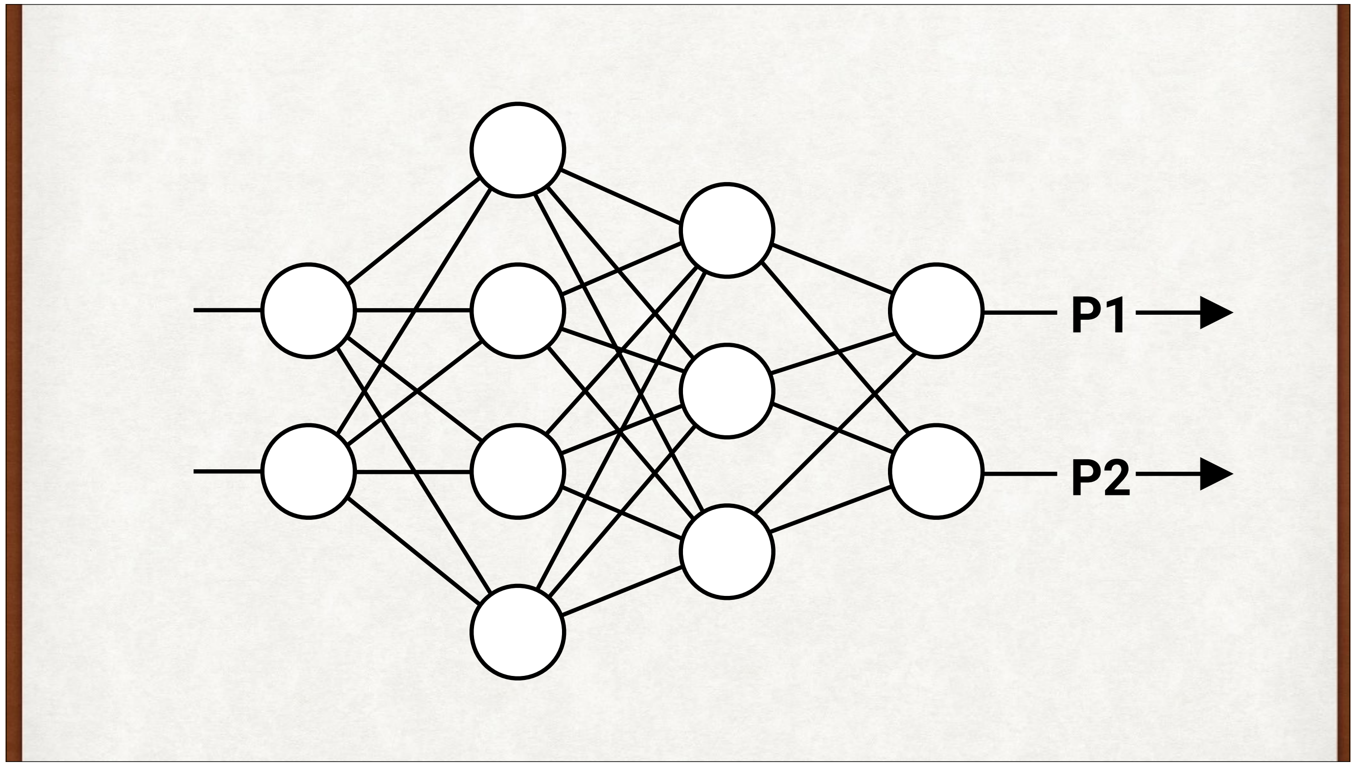
Dropout is popular, easy, and effective.



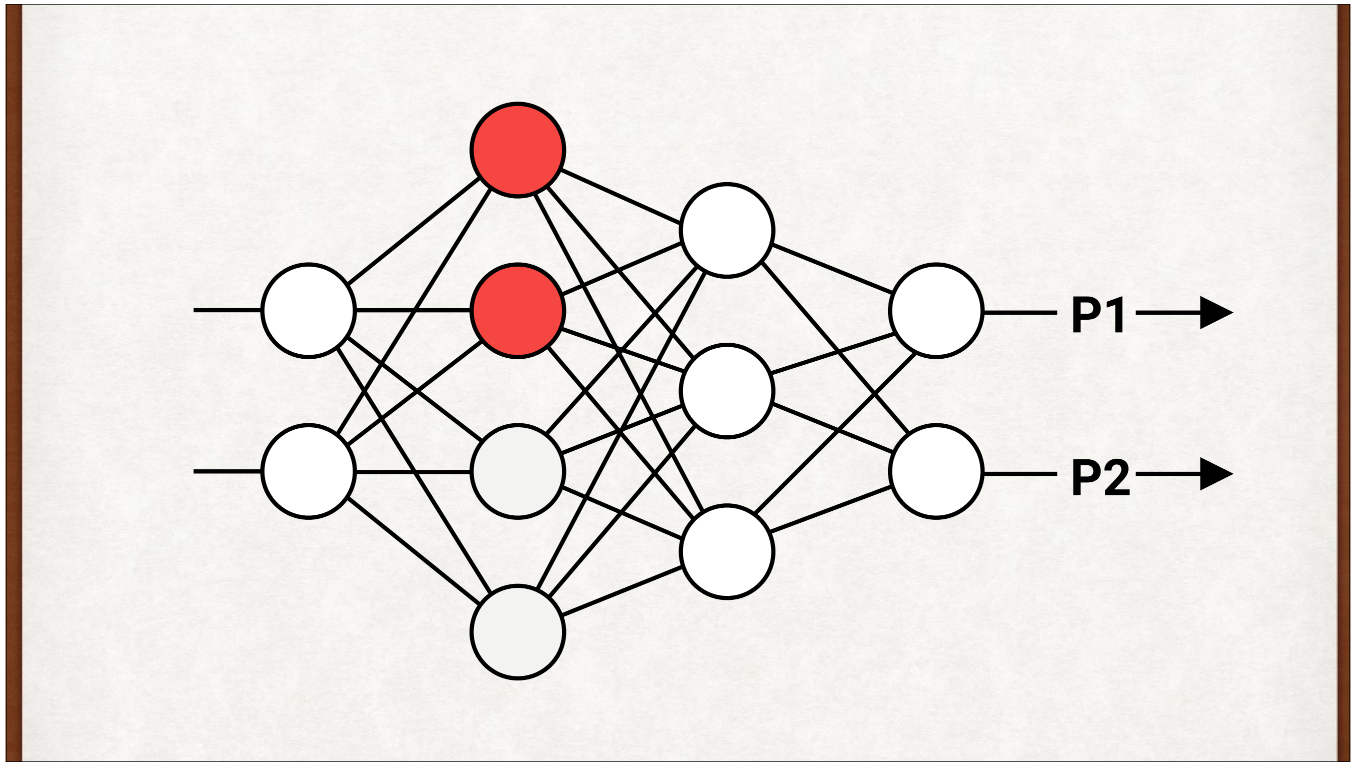
Dropout is popular, easy, and effective.



Dropout is popular, easy, and effective.



Dropout is popular, easy, and effective.



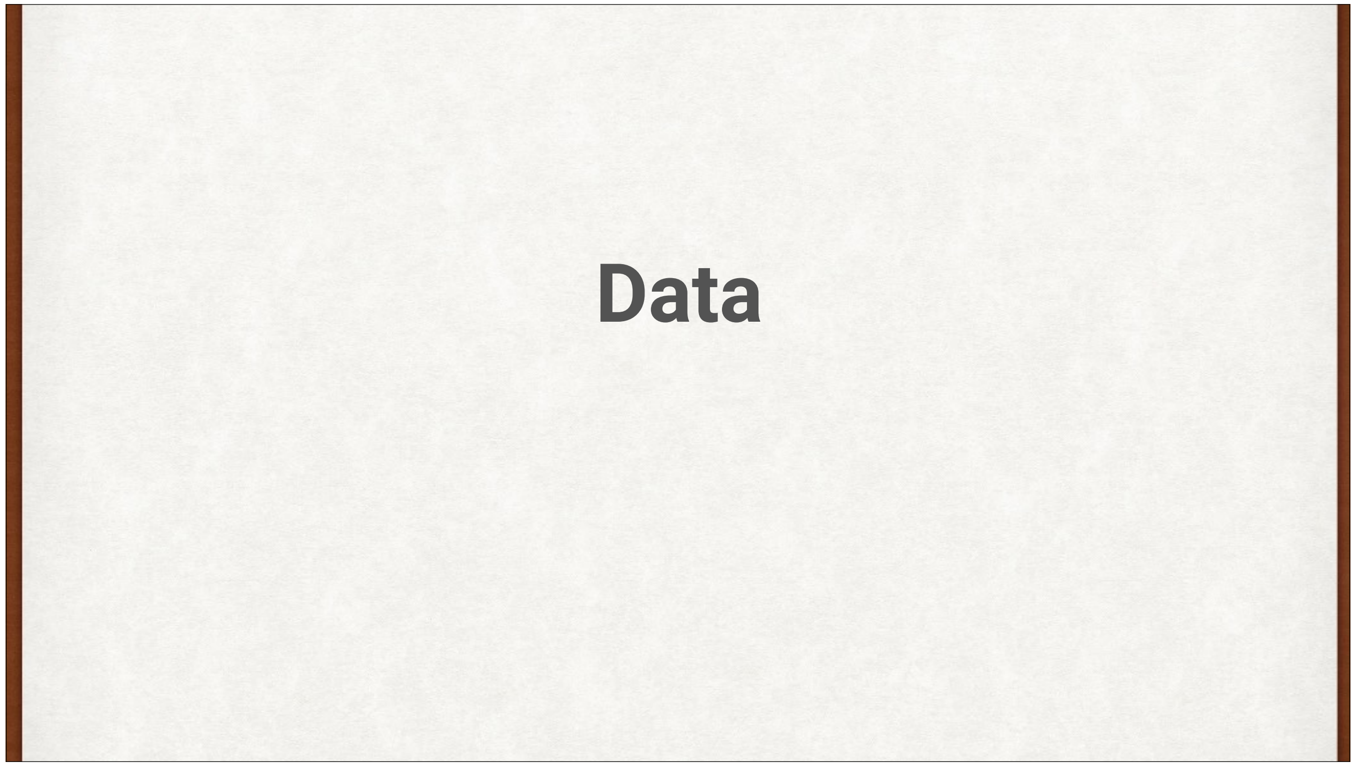
Dropout is popular, easy, and effective.

Regularization

Batchnorm

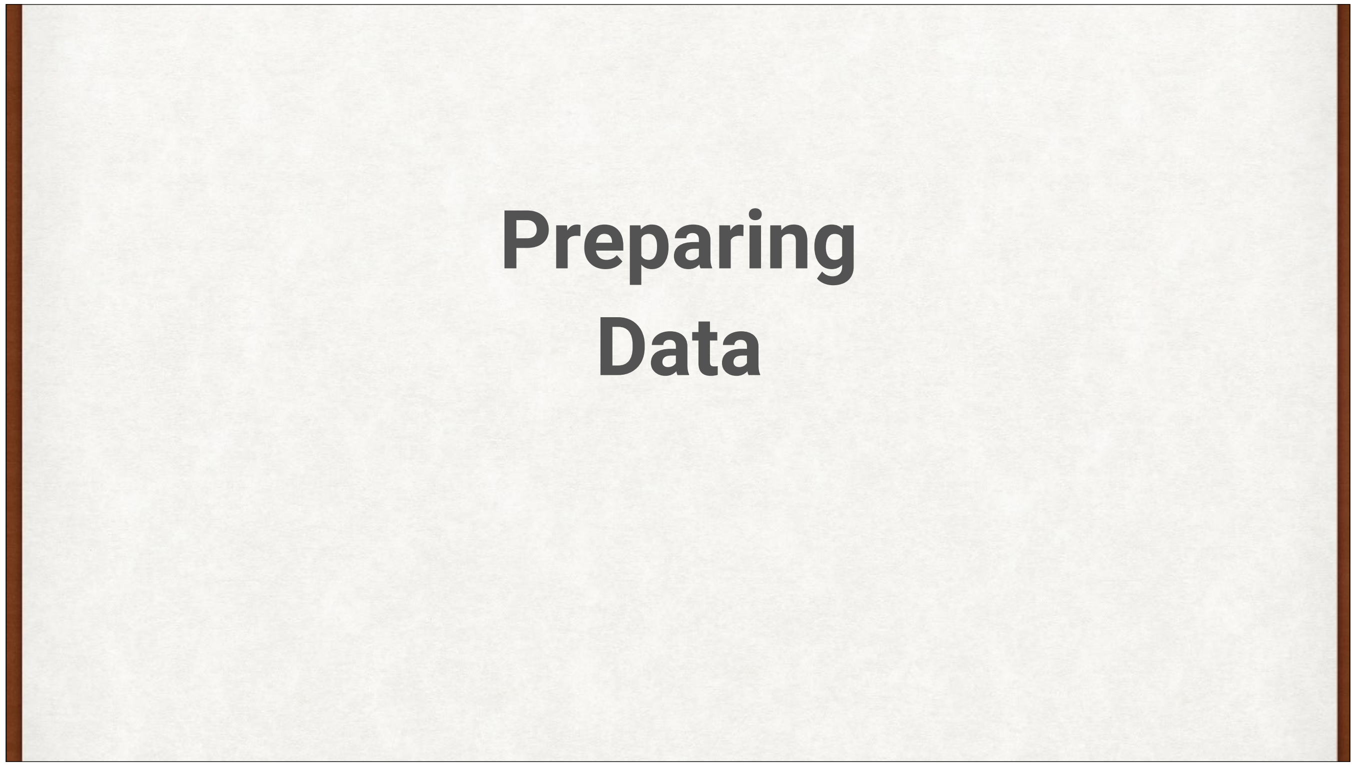
Batchnorm is another popular regularization method. Warning: batchnorm and dropout don’t usually play together nicely. For simplicity, it’s usually best to use one or the other.

Palette cleanser. New topic. Ahhhhh.



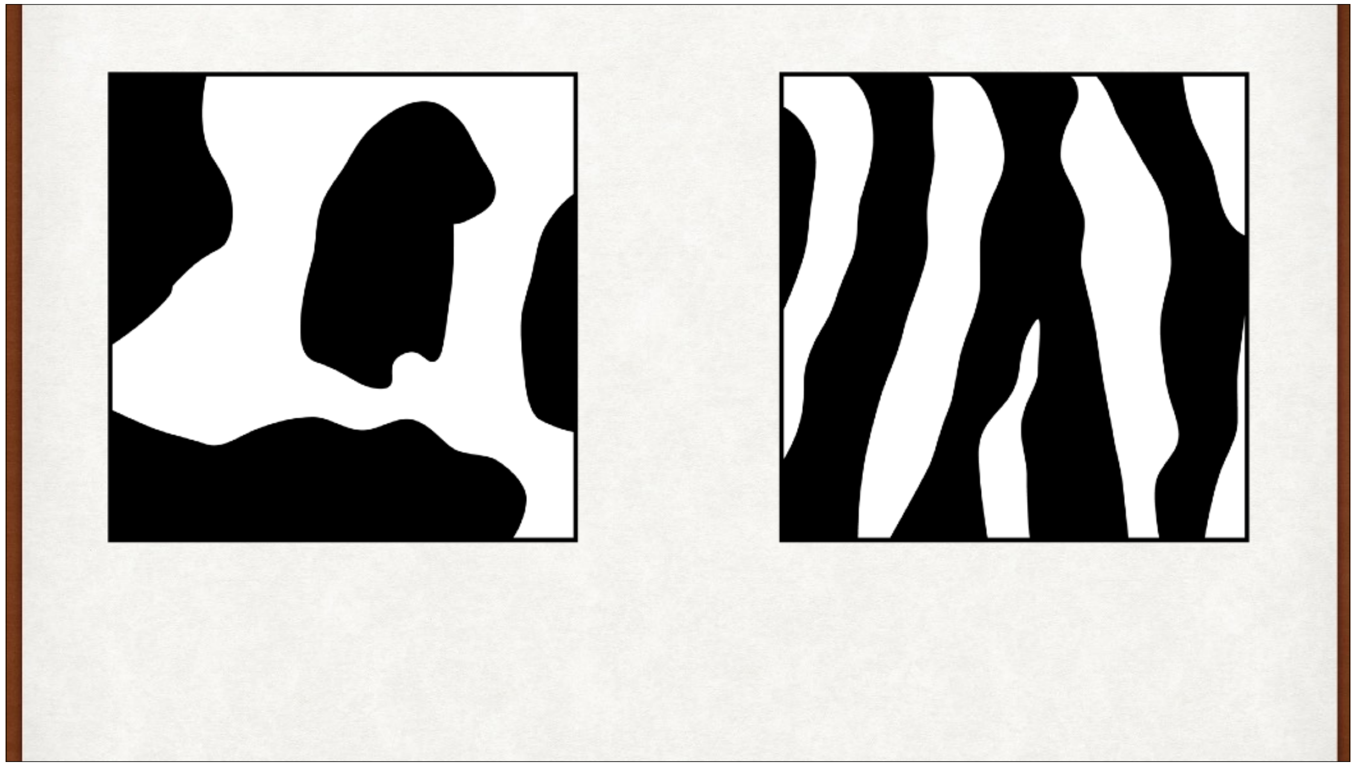
Data

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.

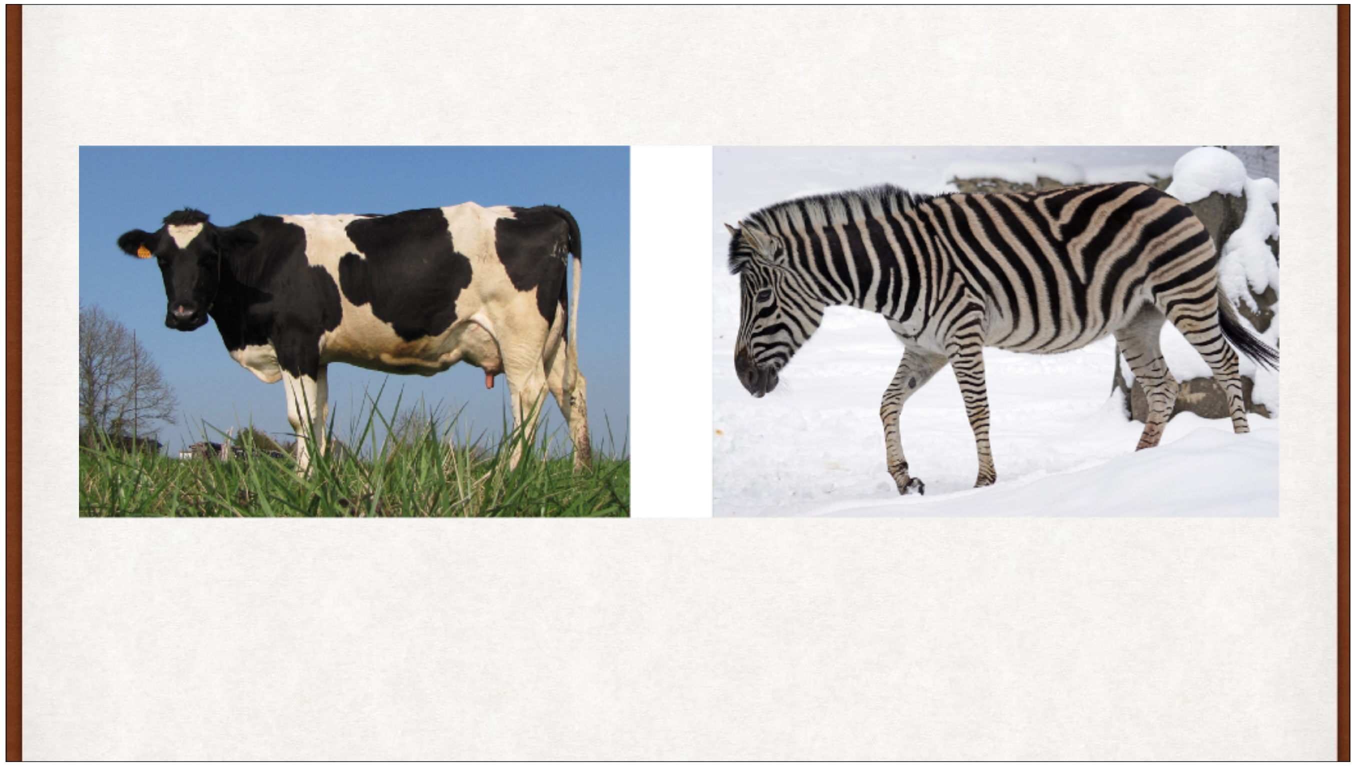


Preparing Data

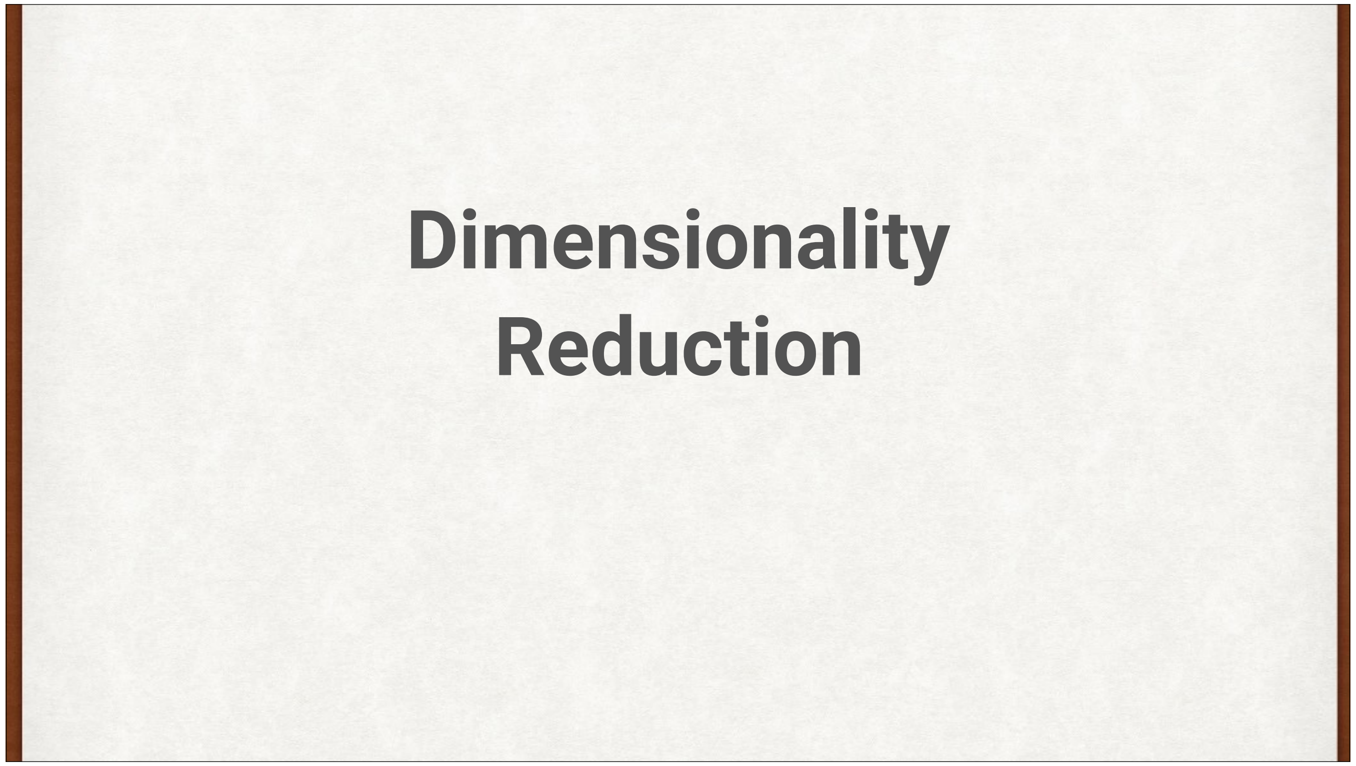
We almost never get “clean” data to train our systems. Making sure our data is consistent and well-formatted (or “clean”) is essential, and often a big piece of the overall work.



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

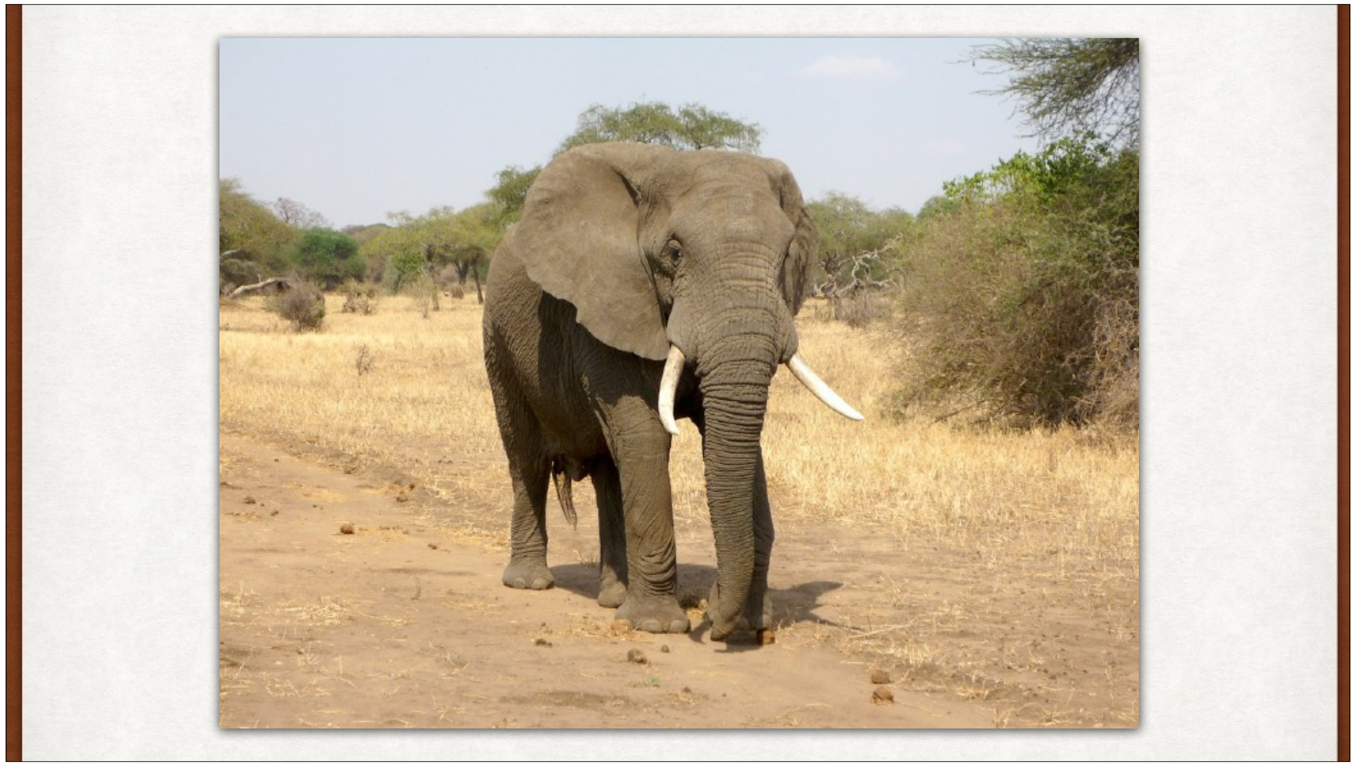


…and later, some well-meaning user gives us pictures of the whole animals. These aren’t the kind of images we trained on, and the results probably won’t be great. So we must be vigilant to keep all of our data, from training to real-world use, structured consistently.
https://pixabay.com/en/cow-field-normande-800306
https://pixabay.com/en/zebra-chapman-steppe-zebra-1975794



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.



What qualities describe an elephant?

https://pixabay.com/en/elephant-african-bush-elephant-114543/

Age

Length

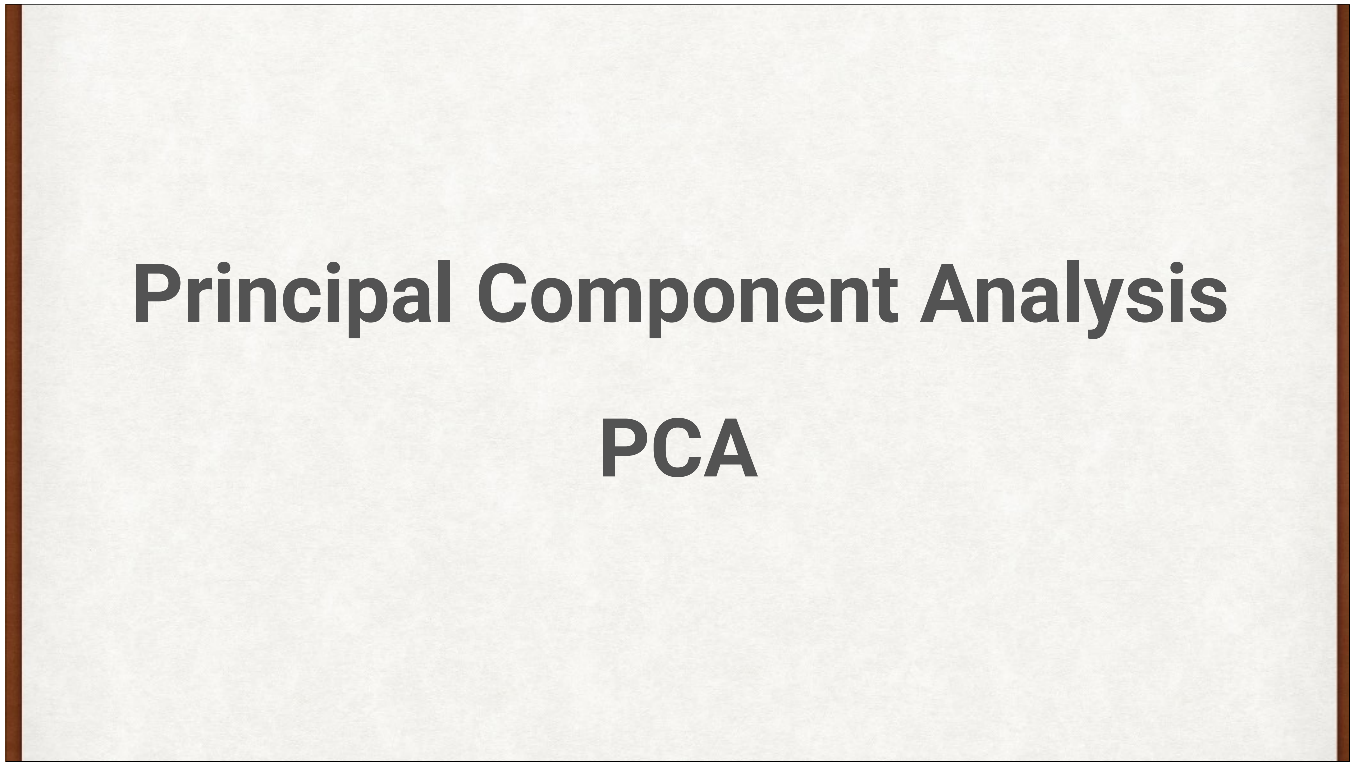
~~Ear size~~

Tusk length

Weight

Maybe ear size is redundant with age (for an imaginary species of elephant)

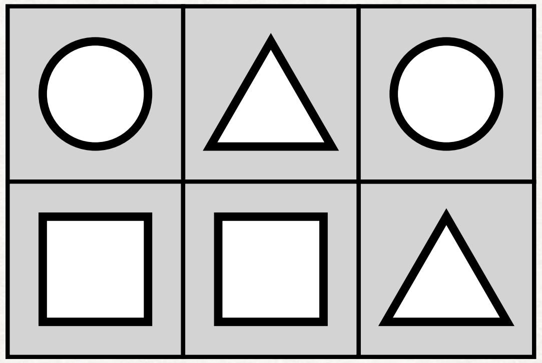
https://pixabay.com/en/elephant-african-bush-elephant-114543/



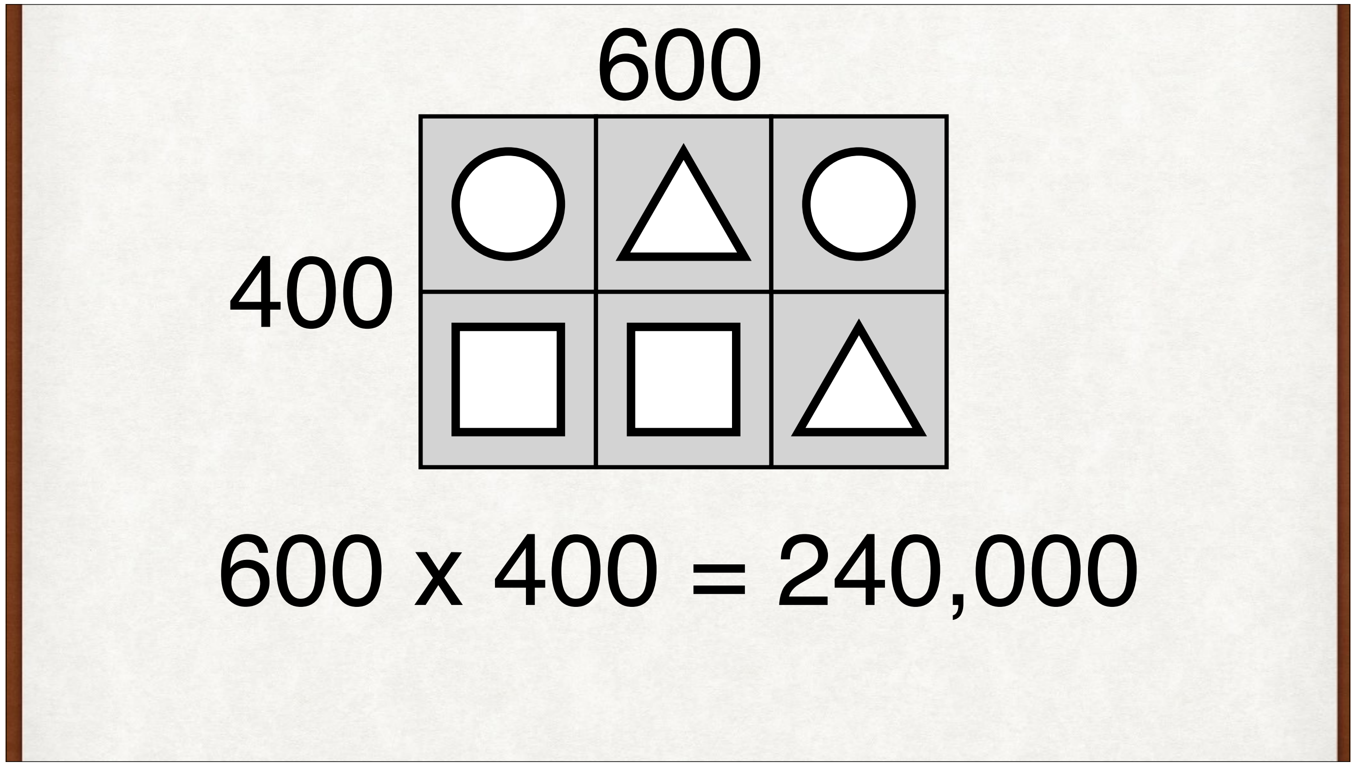
Principal Component Analysis

PCA

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.

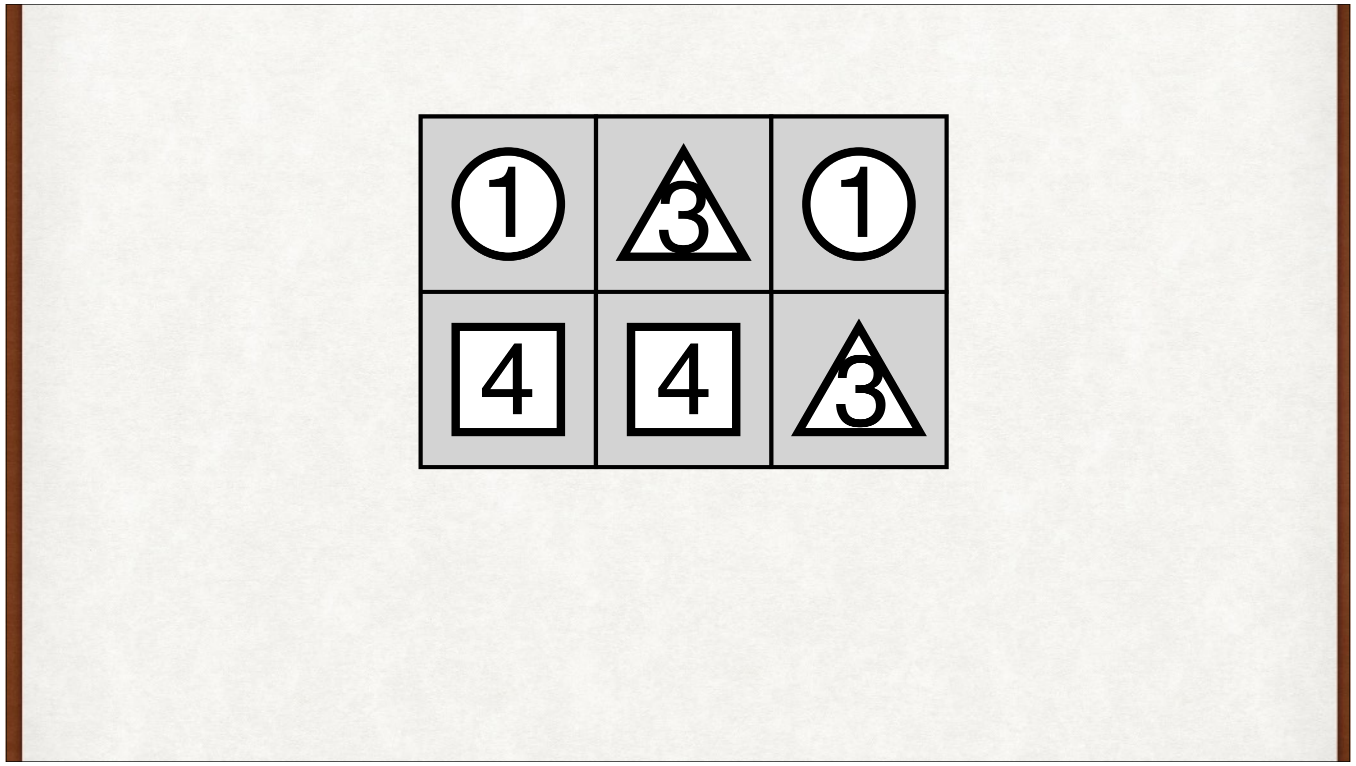


Are there circles in the upper corners? Given a high-res image, there’s a lot of pixel-examining to be done to answer this question.

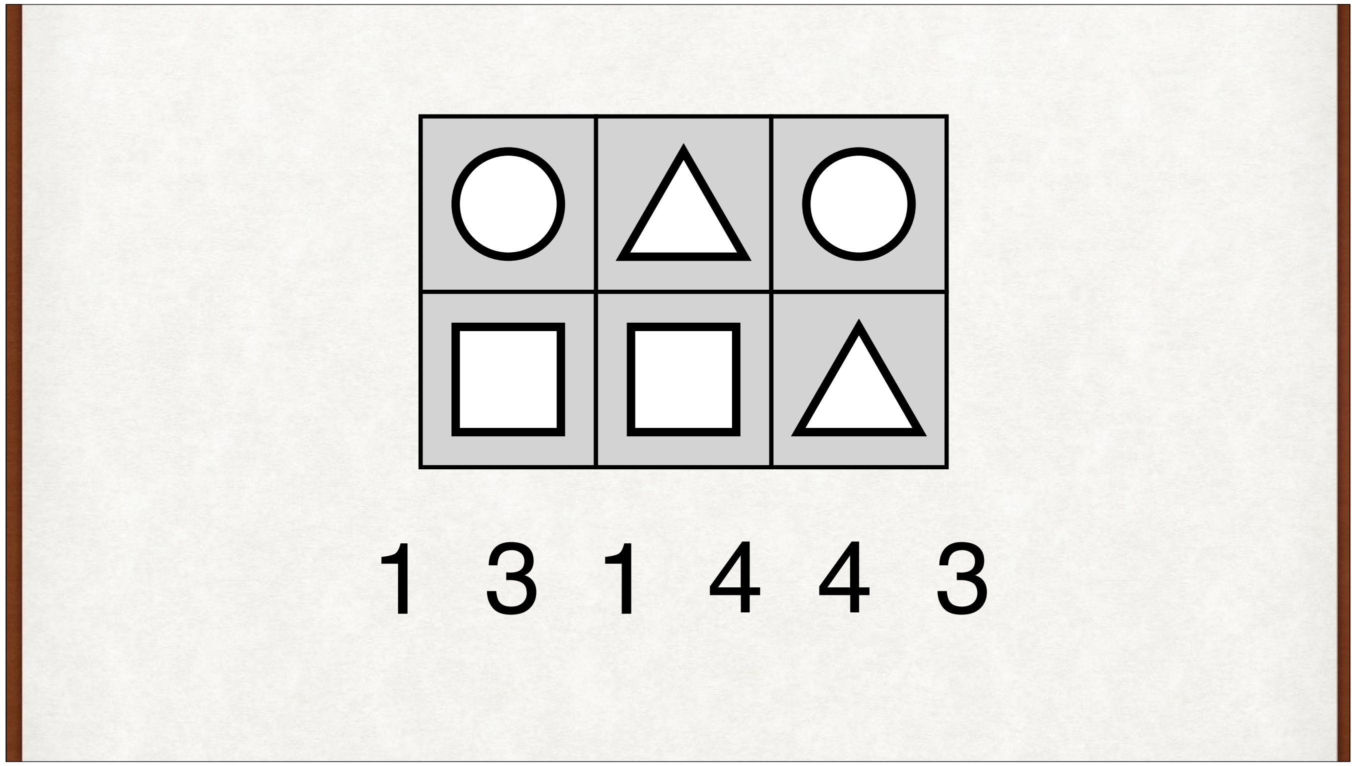


600 x 400 = 240,000

Even at low res, there are a lot of pixels per image

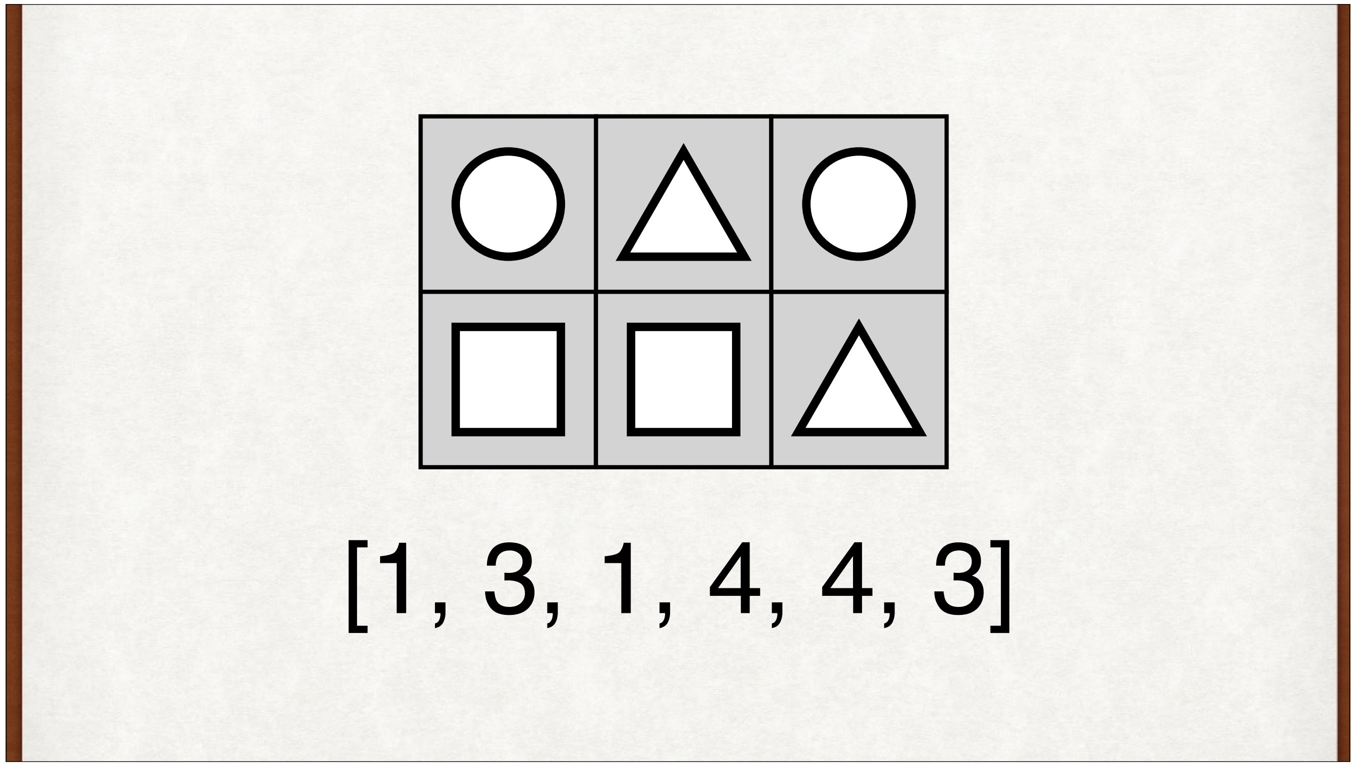


Suppose we had a magic pre-processor that assigned a number to each shape



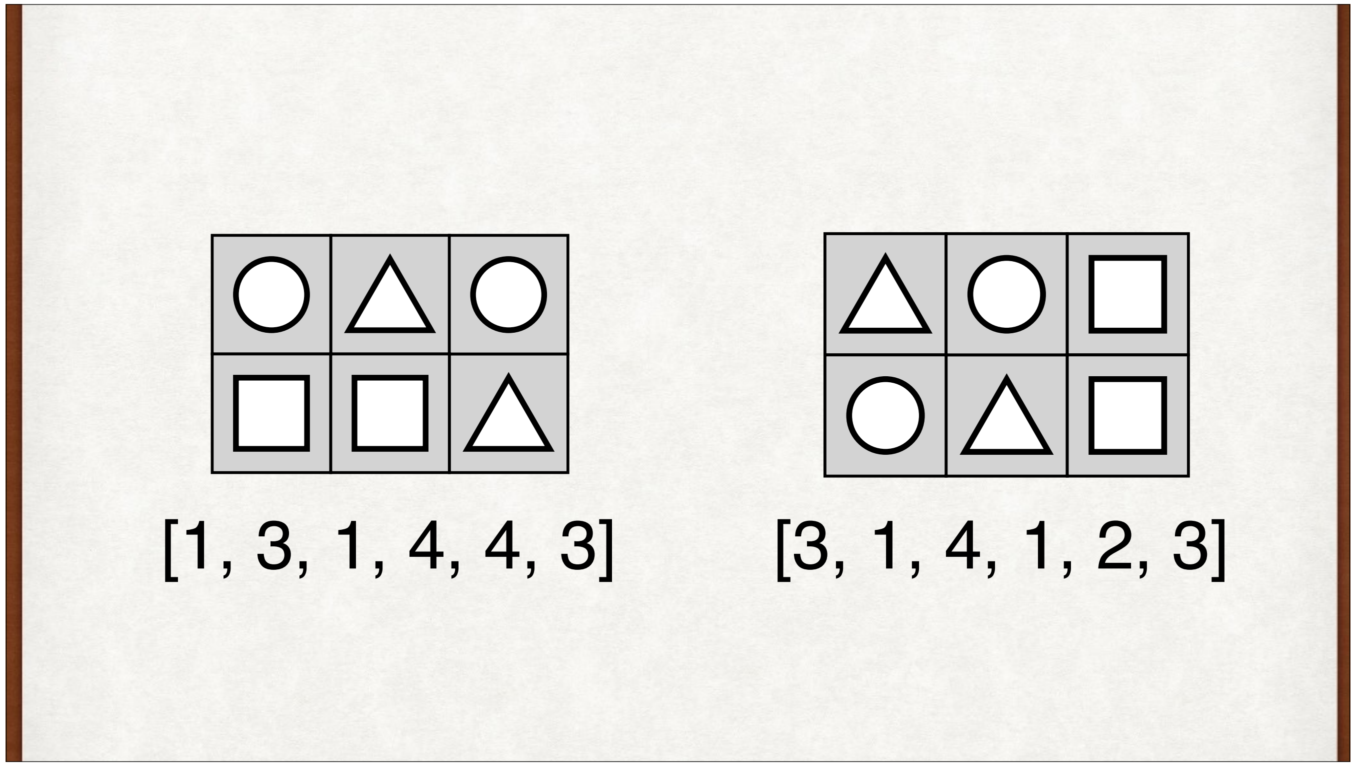
1 3 1 4 4 3

Now we can represent our picture with just the numbers. That means we can train on lists of just 6 numbers, not a quarter-million.



[1, 3, 1, 4, 4, 3]

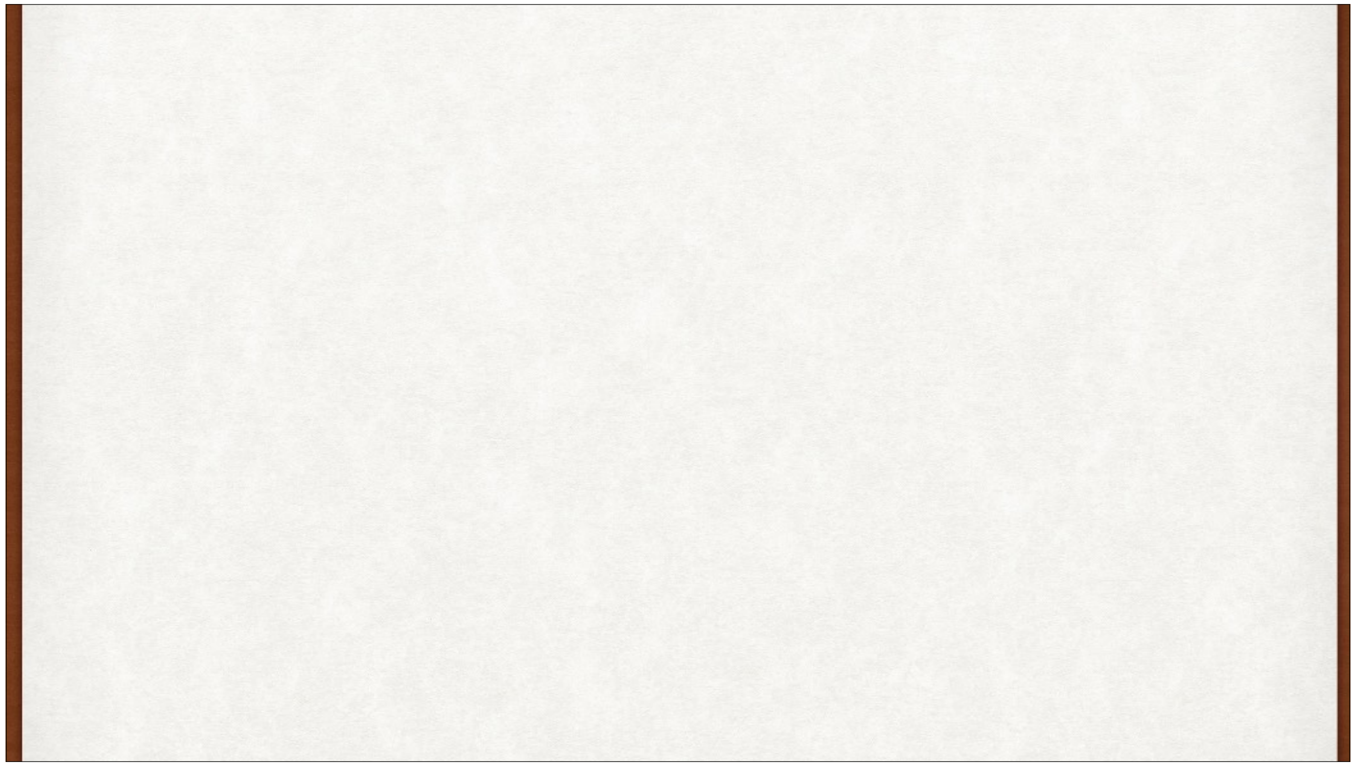
The list of numbers.



[1, 3, 1, 4, 4, 3]

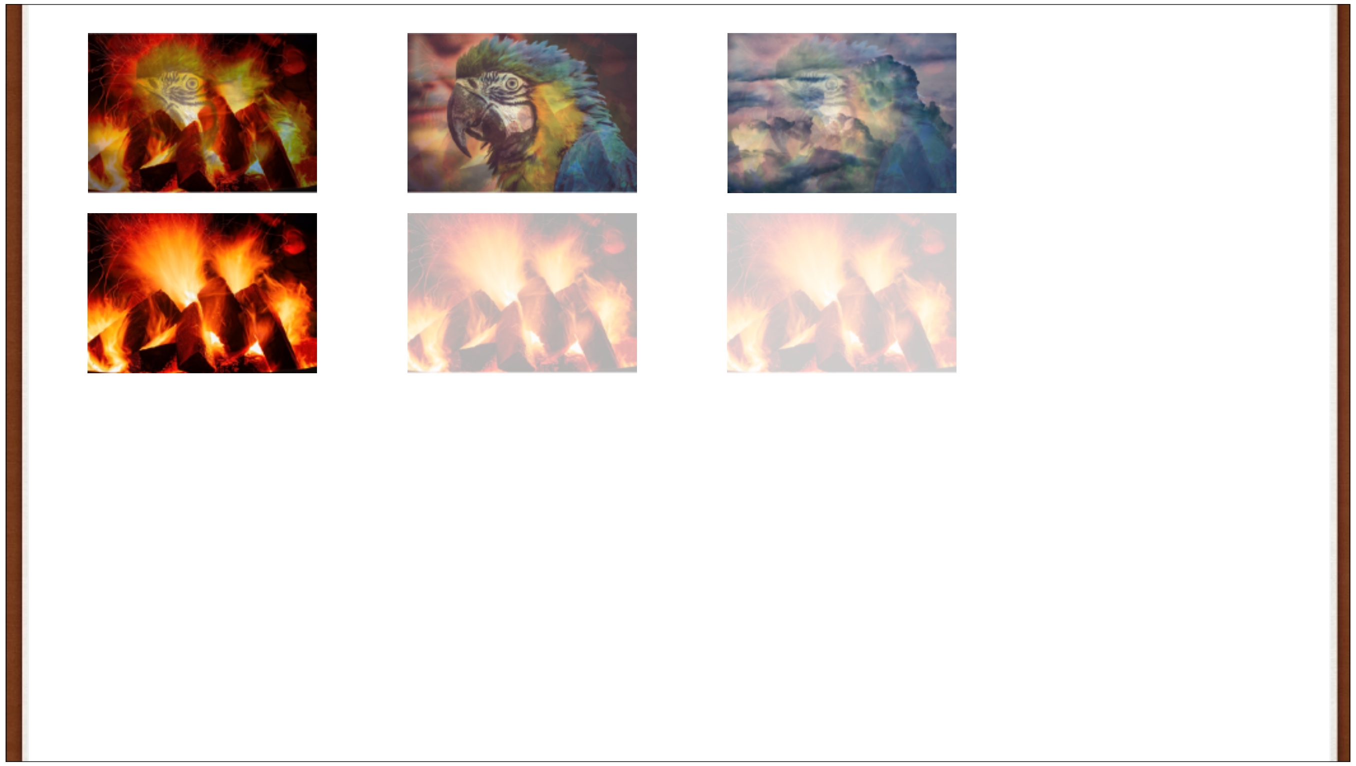
[3, 1, 4, 1, 2, 3]

Any of these images can be represented with just six numbers, not 240,000

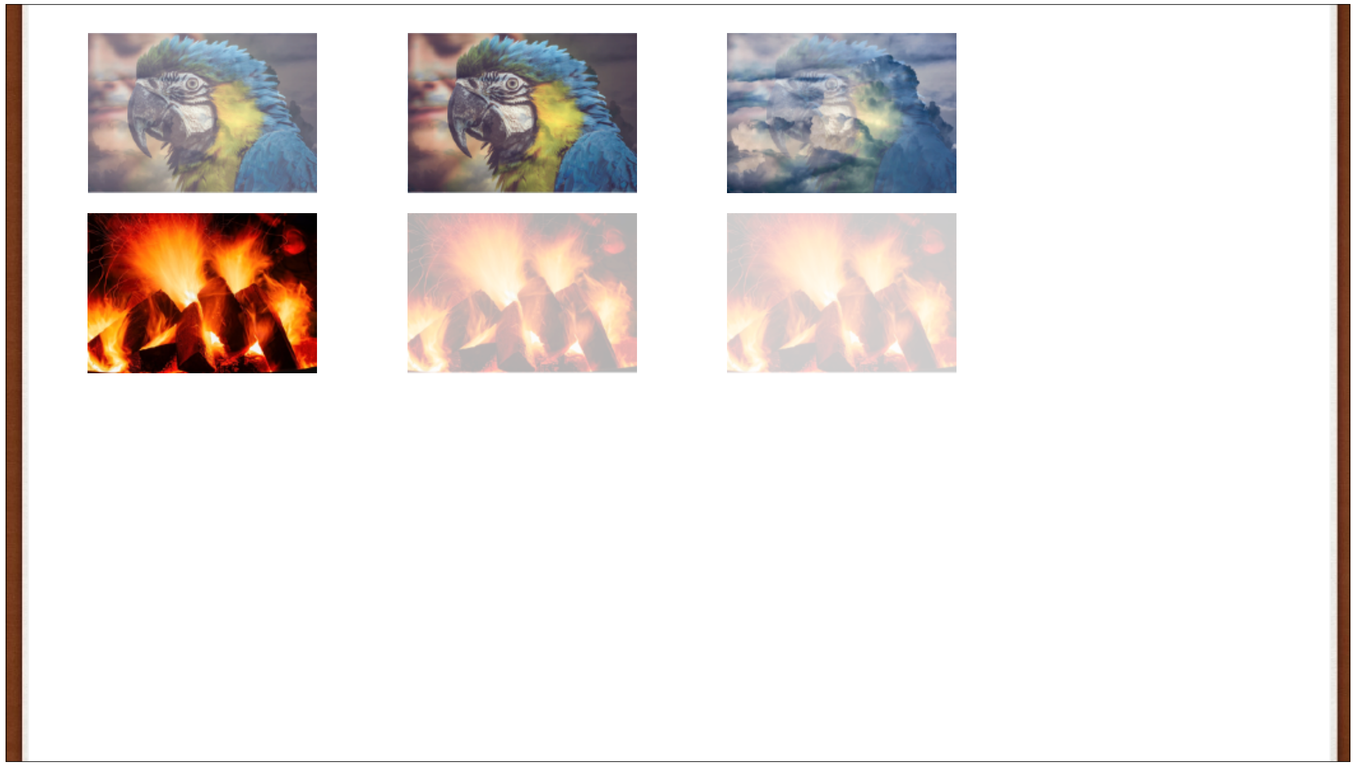


Any of these images can be represented with just 3 numbers, not 240,000

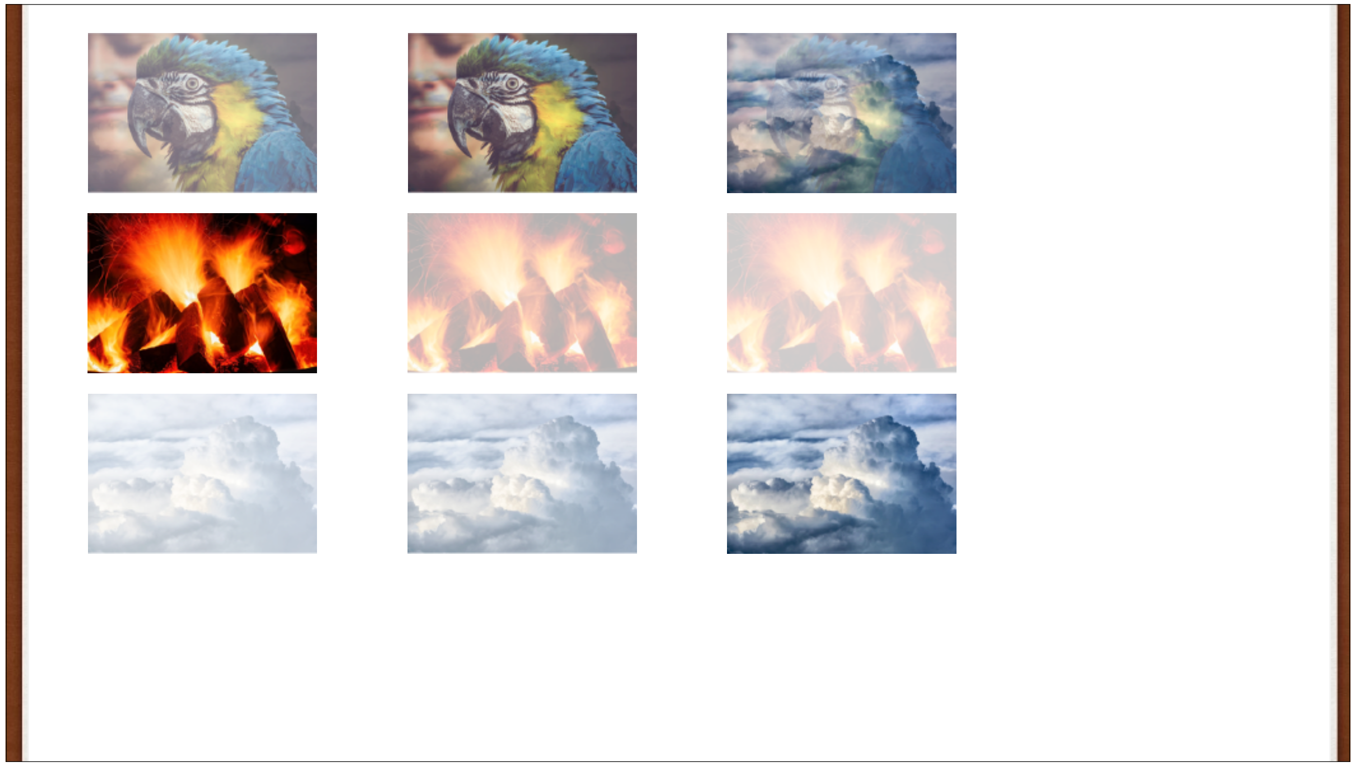
Here are three pictures. Can we represent them in some simpler form, rather than gigantic lists of pixel colors?
https://pixabay.com/photos/ara-parrot-animal-world-3695678/
https://pixabay.com/photos/nature-outdoor-sky-cloud-cloudy-3294543/
https://pixabay.com/photos/open-fire-fire-embers-flame-hot-3879031/



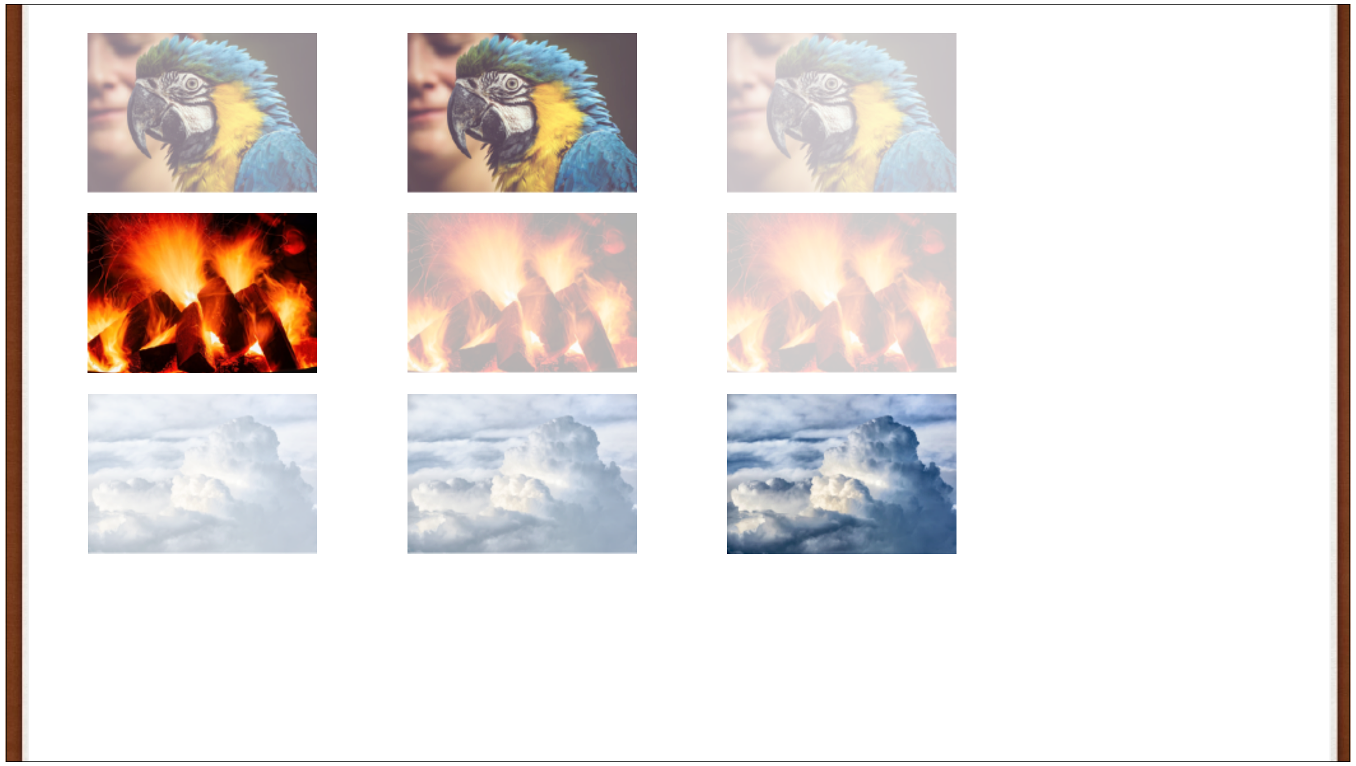
Each of the starting images has some of this fire image in it.



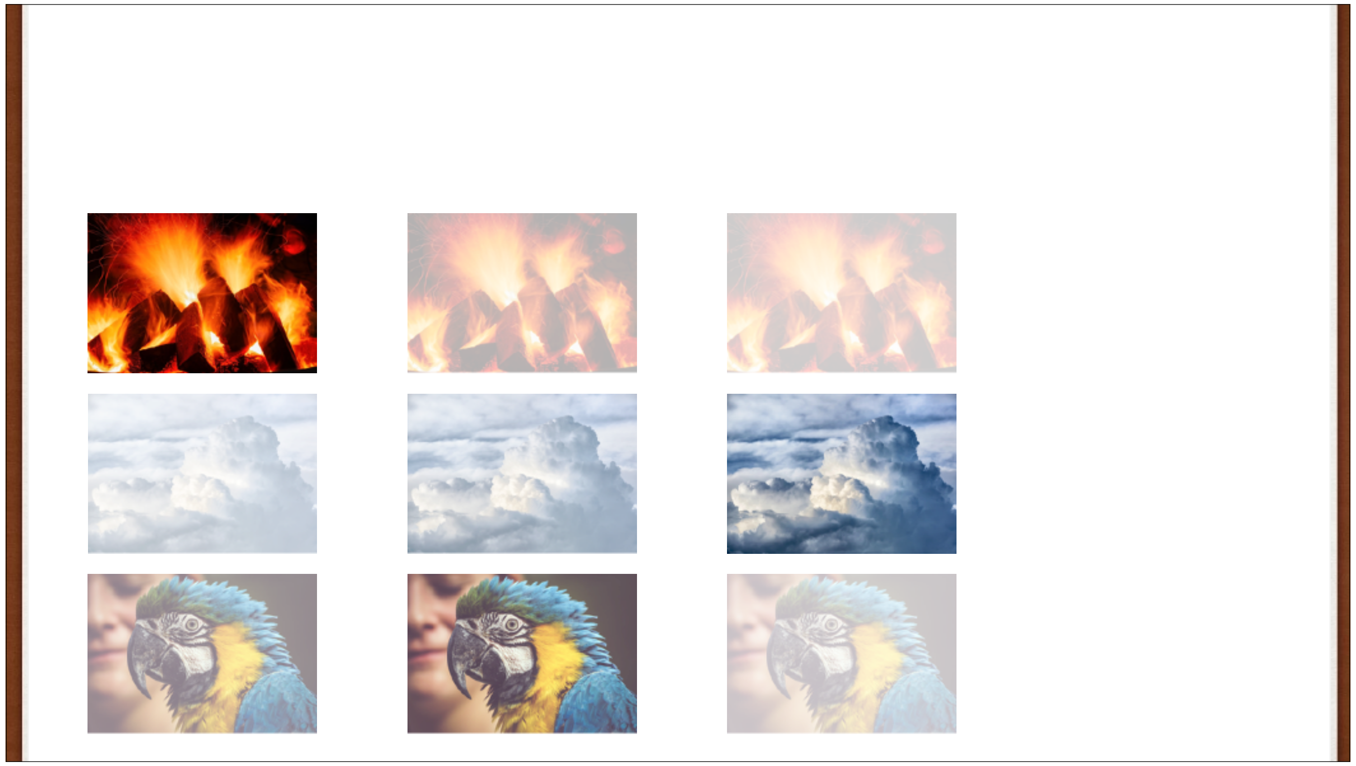
Each of the starting images has some of this fire image in it.



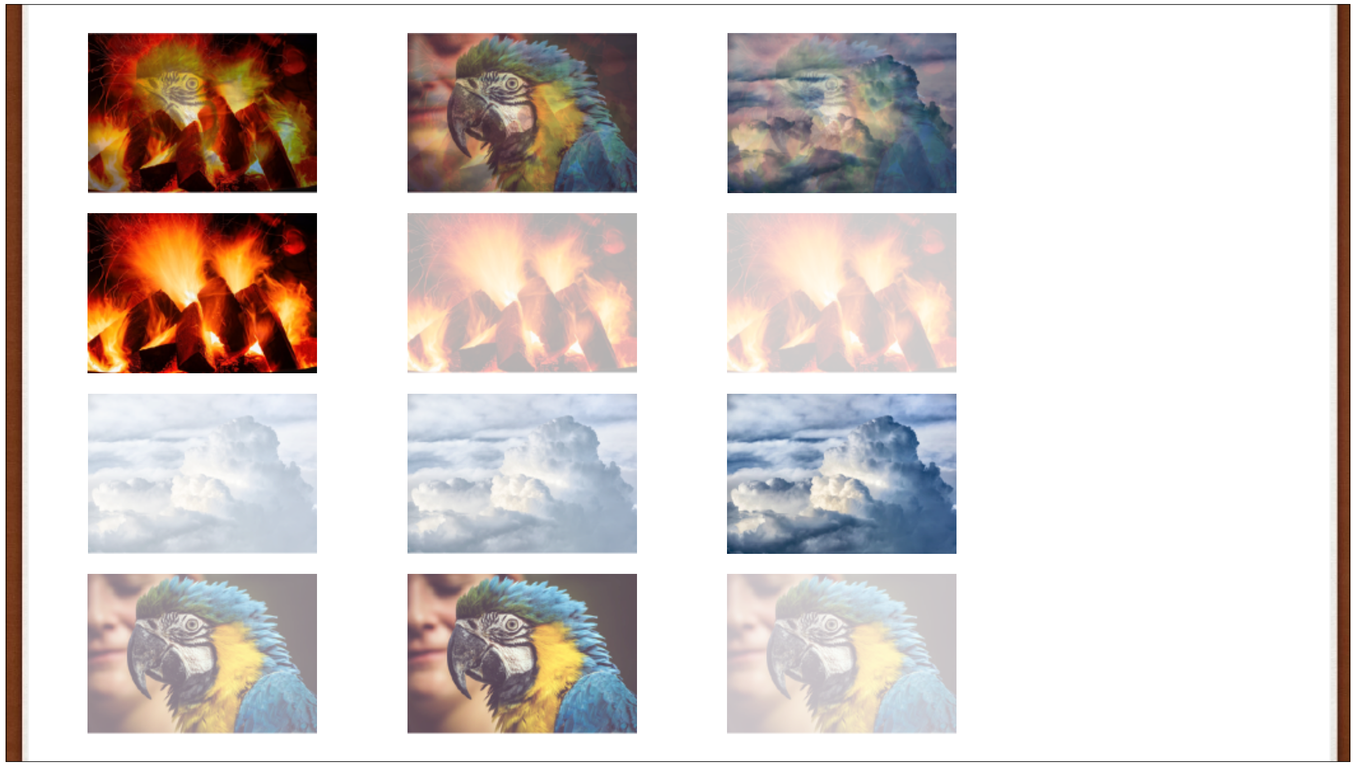
Each of the starting images has some of this cloud image in it.



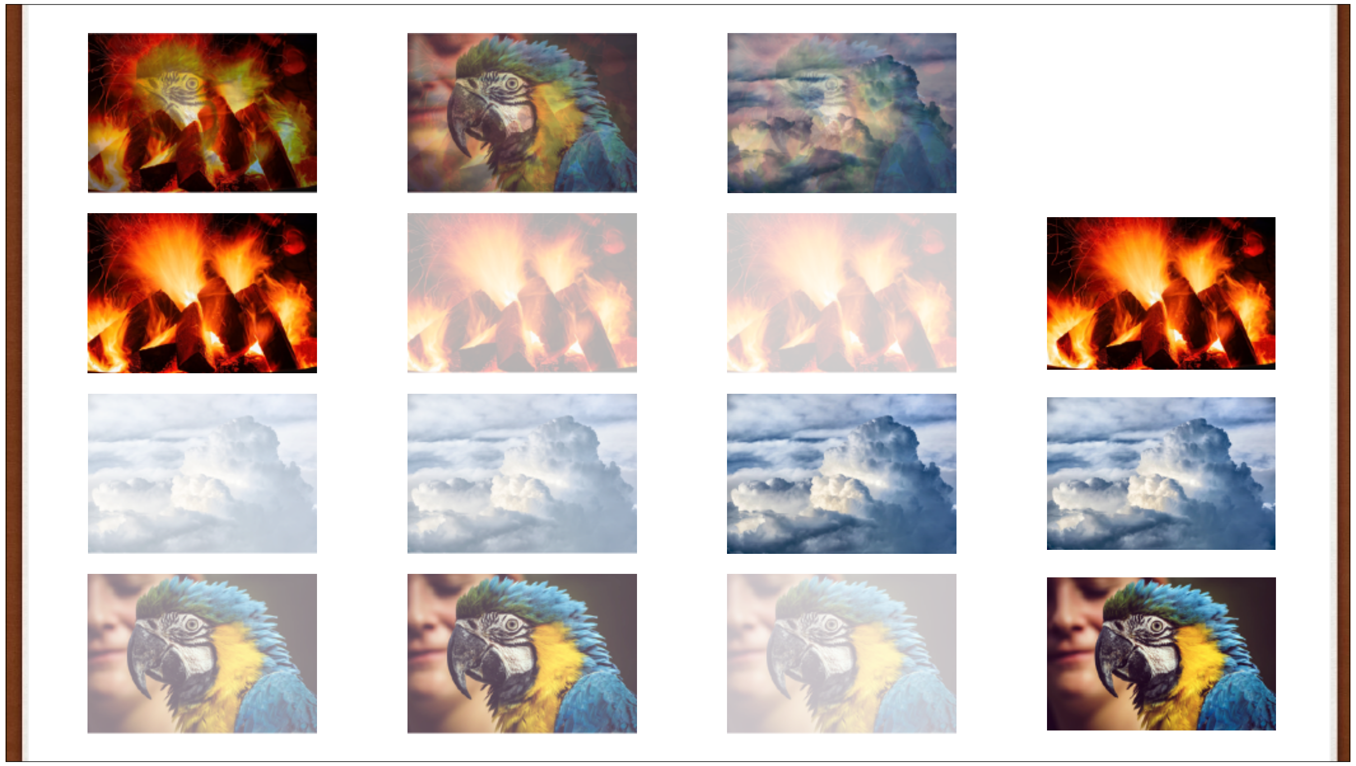
Each of the starting images has some of this cloud image in it.



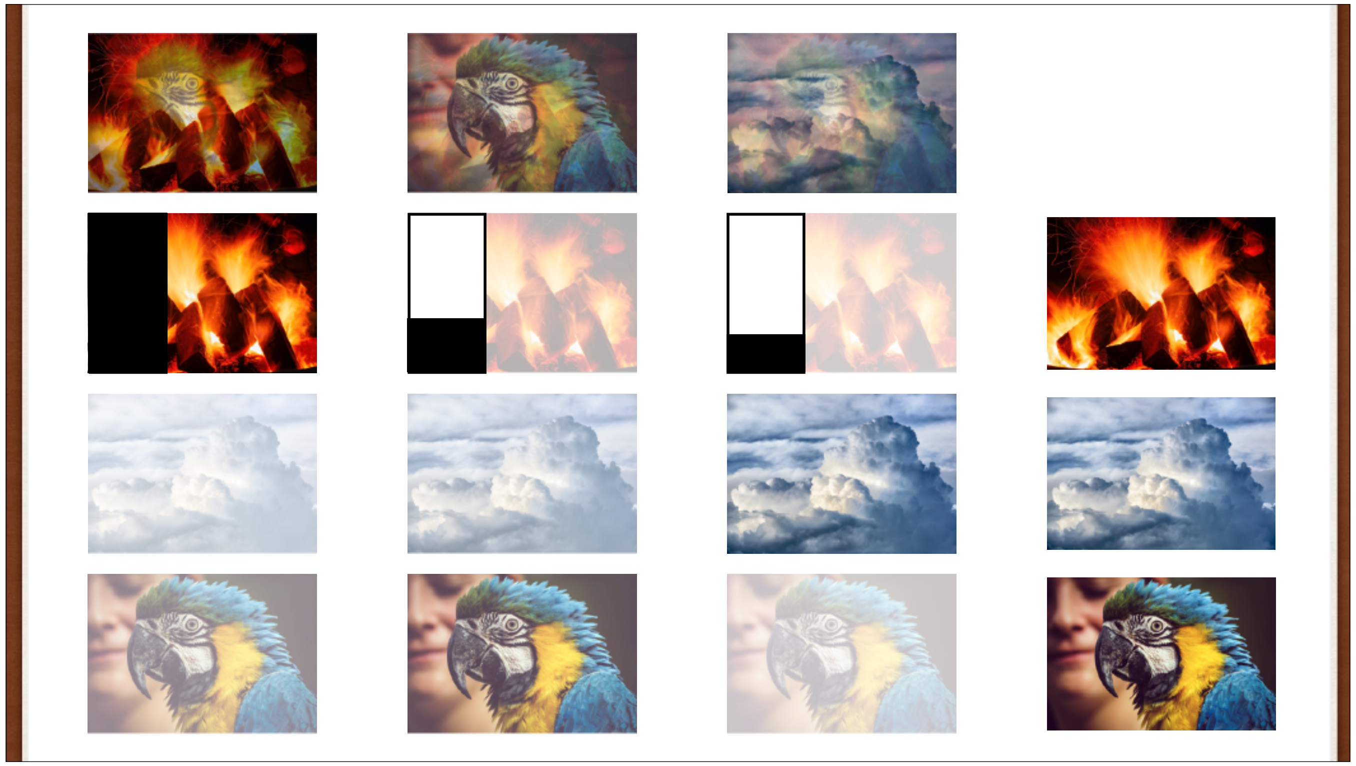
Each of the starting images has some of this parrot image in it.



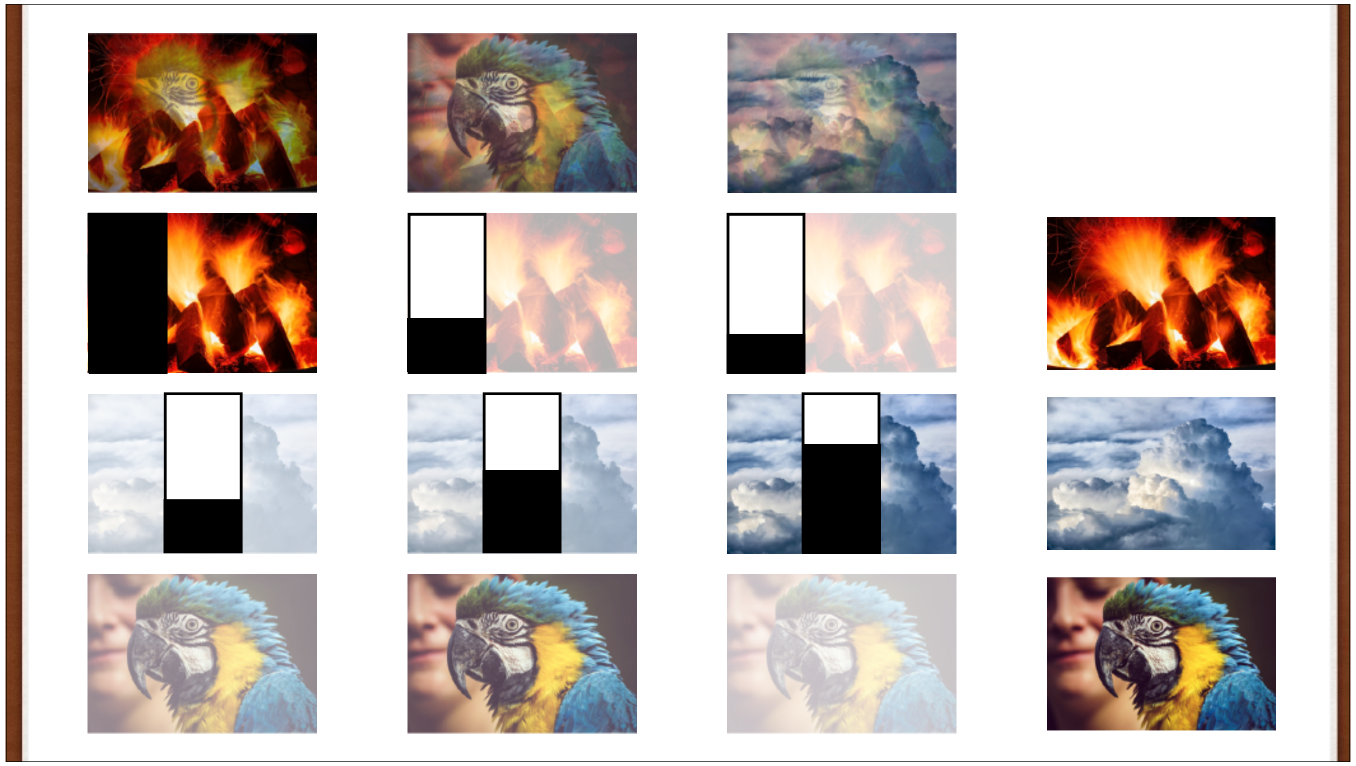
Each of the starting images has some of this parrot image in it.



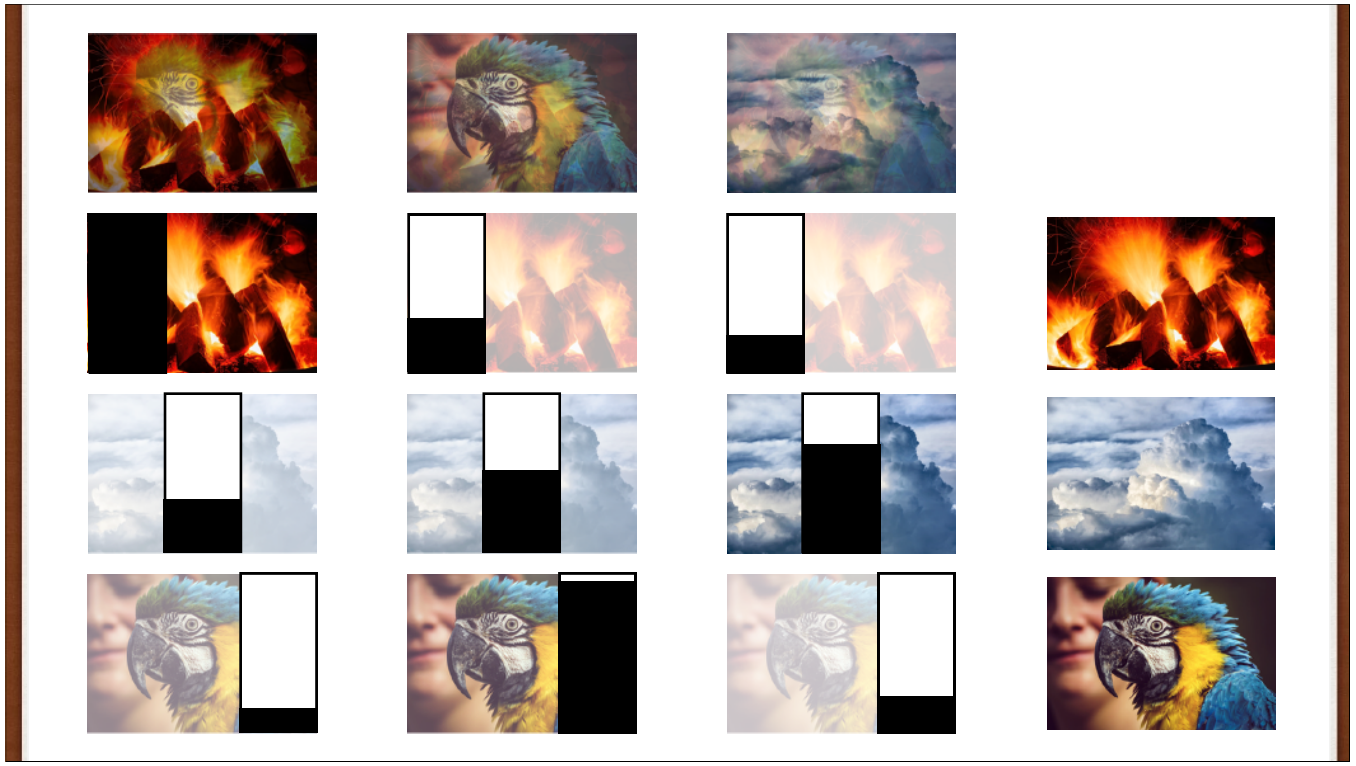
The starting images at top, their components below them, the originals of the components at the right.



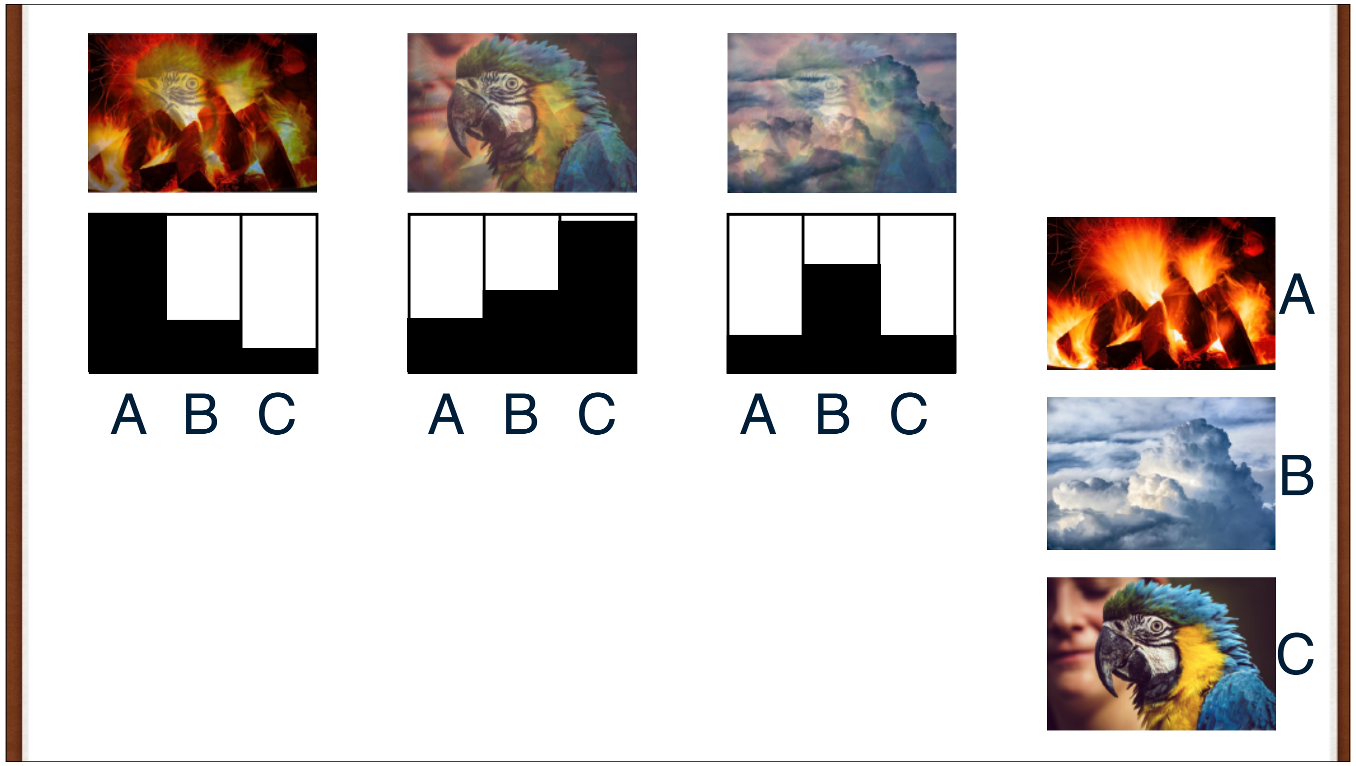
The bars show how much of the fire is used in the picture at the top.



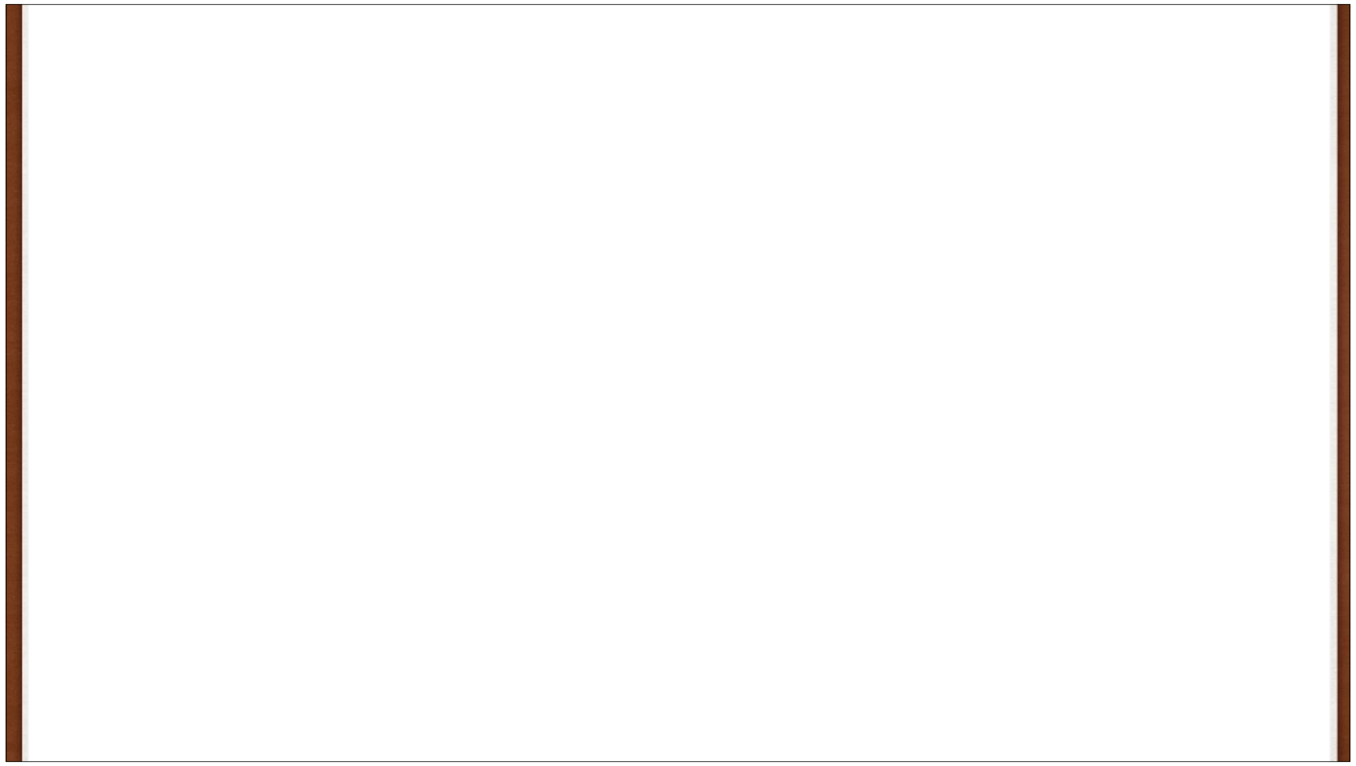
And how much cloud.



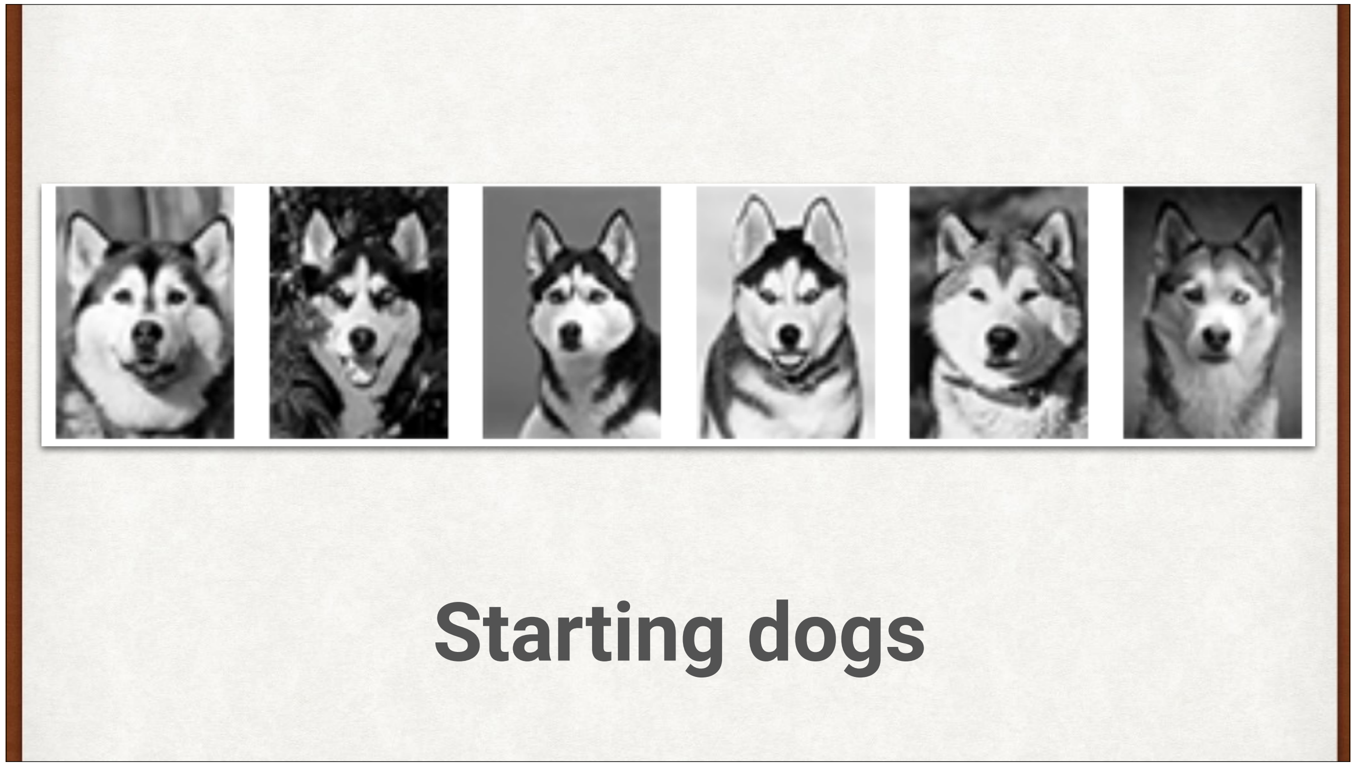
And how much parrot.



Now any of these images can be represented with just 3 numbers, not 240,000. The computer can be trained on just 3 numbers, and as long as we have the three component images around, we can always recreate the pictures at the top by scaling and adding the components using the 3 numbers.



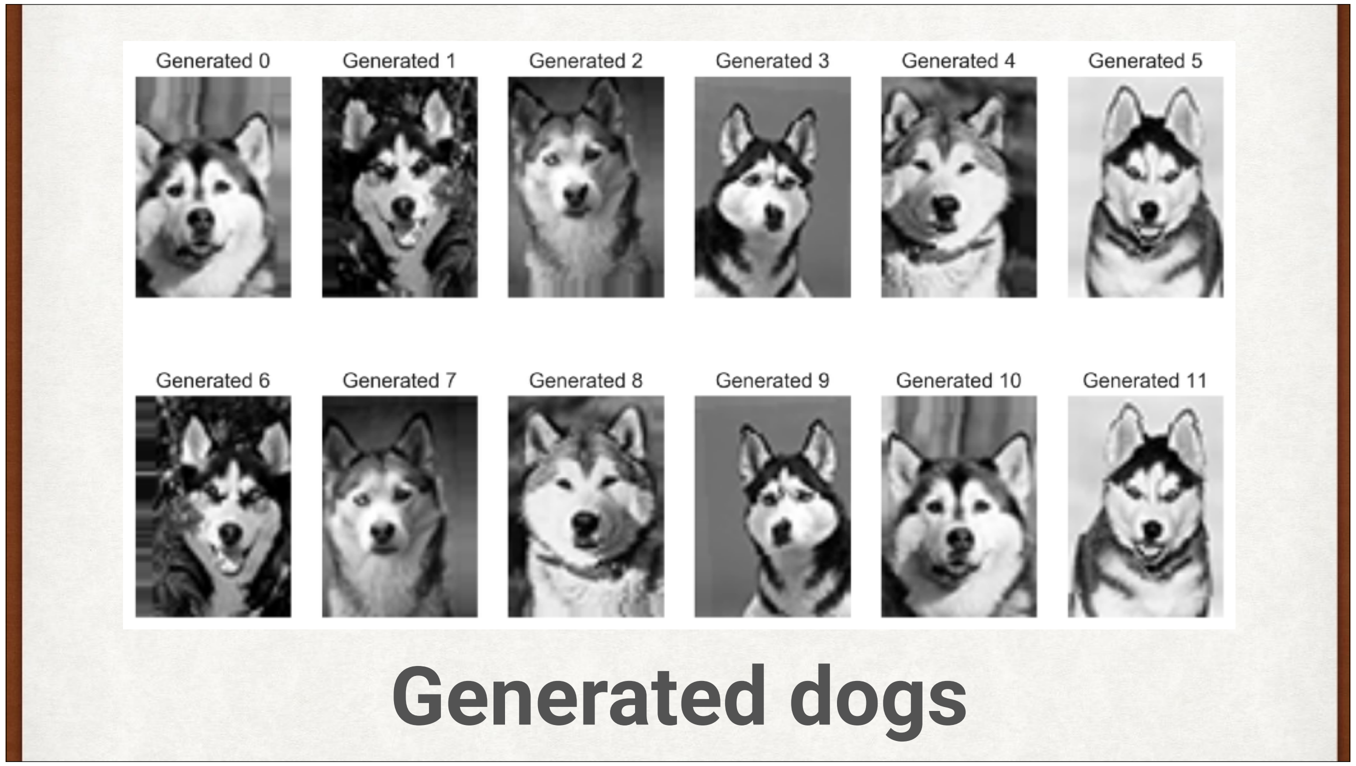
Let’s look at this again with pictures of dogs and see the component images that PCA creates for us.



Starting dogs

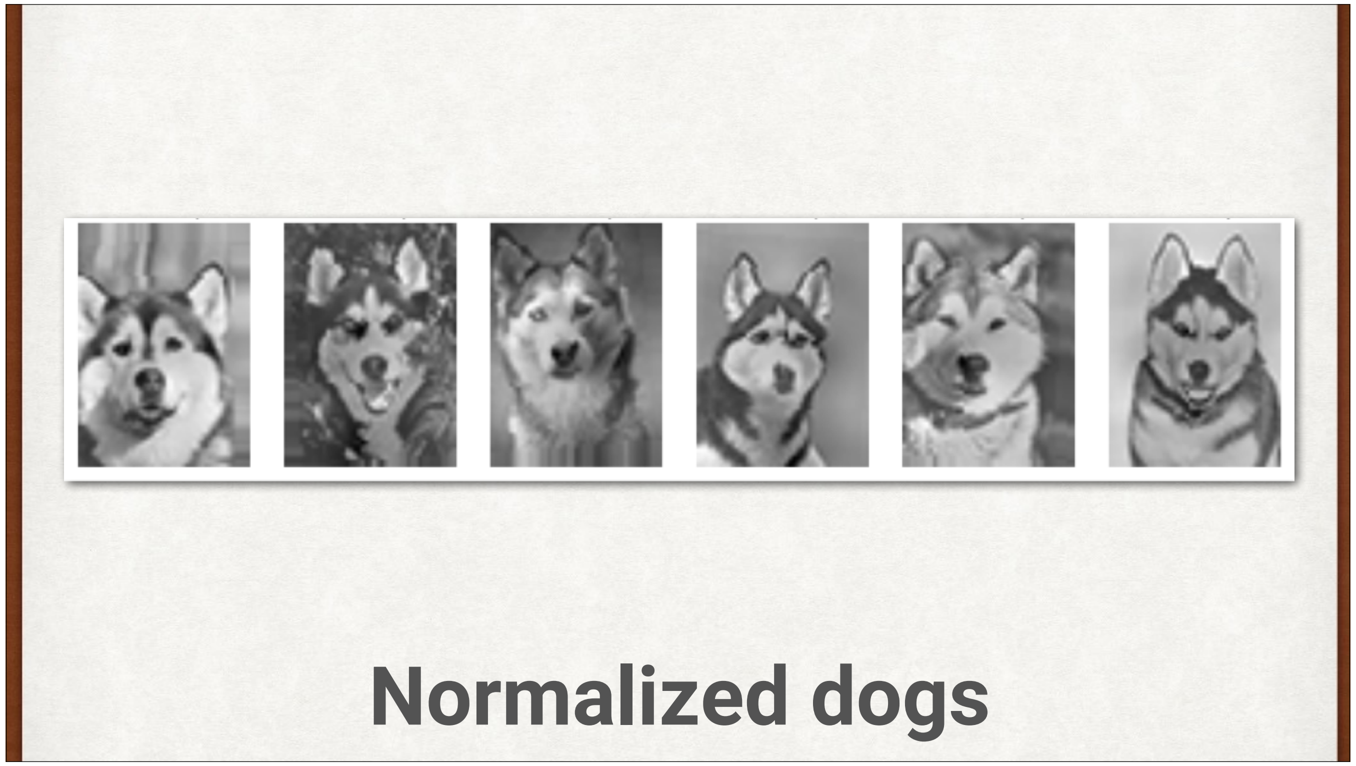
Pictures contain a lot of data. Big pictures can contains millions of numbers. We can save time and resources by crunching the images down into smaller numbers. We could do that by just making the image smaller, but when our data is more abstract, it’s hard to work out what that can mean. Luckily, there 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.

https://pixabay.com/en/husky-sled-dogs-adamczak-1105338
https://pixabay.com/en/husky-dog-outdoor-siberian-breed-1328899
https://pixabay.com/en/dog-husky-sled-dog-animal-2016708
https://pixabay.com/en/dog-husky-friend-2332240
https://pixabay.com/en/green-grass-playground-nature-2562252
https://pixabay.com/en/husky-dog-siberian-husky-sled-dog-2671006



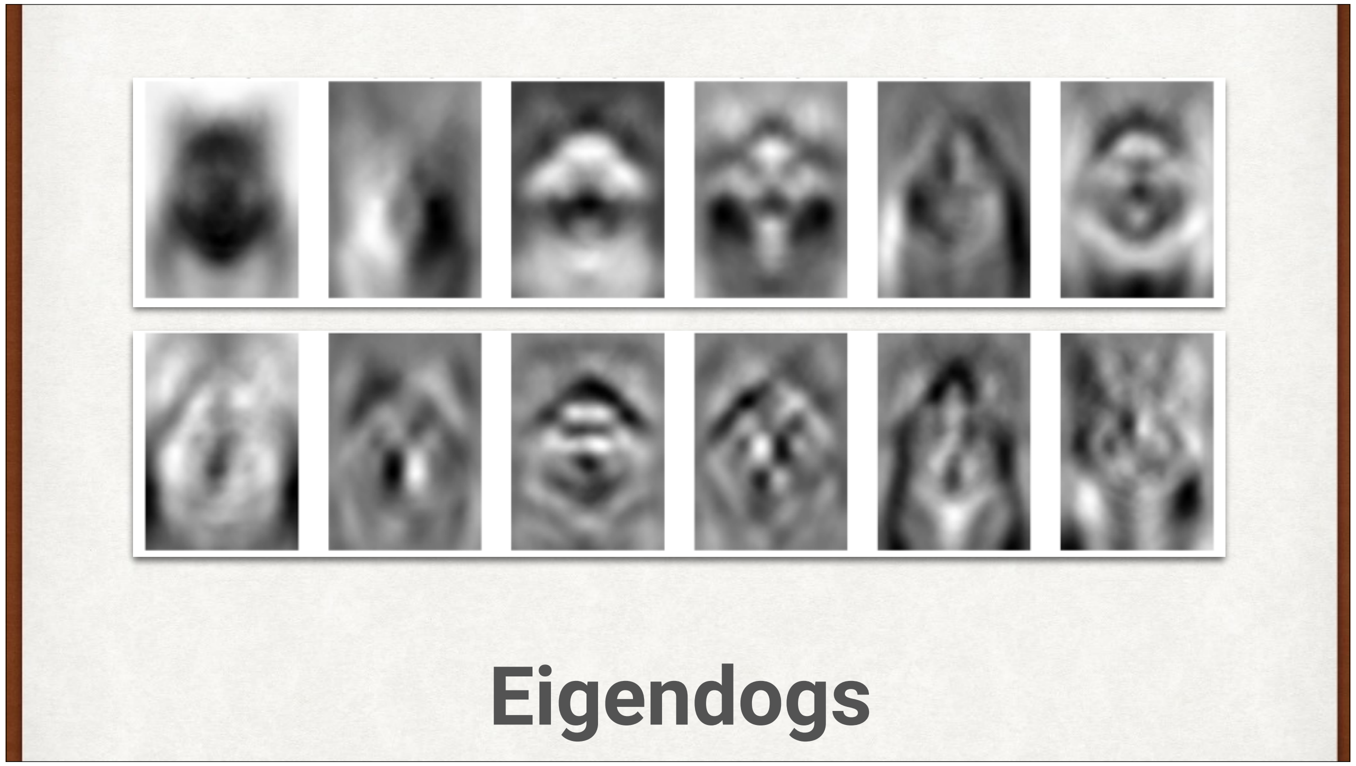
Generated dogs

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.



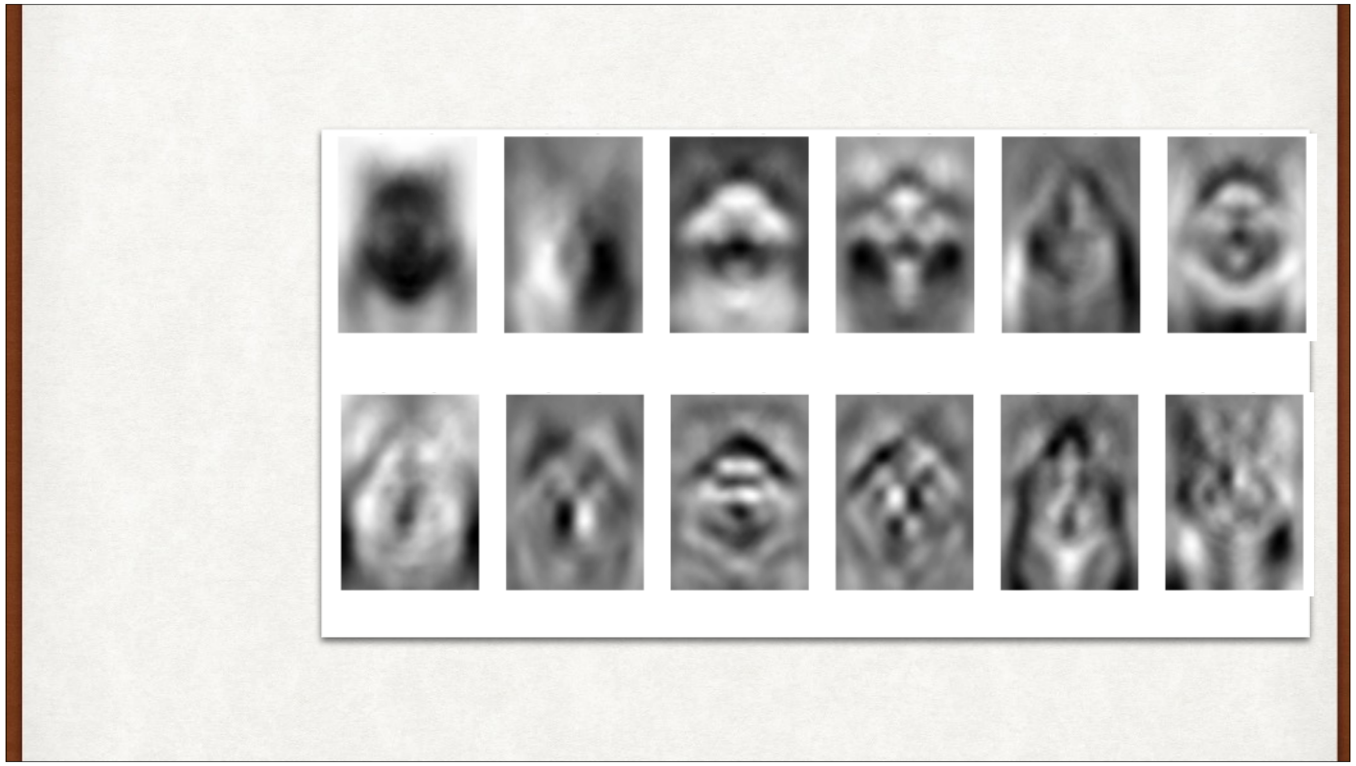
Normalized dogs

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).

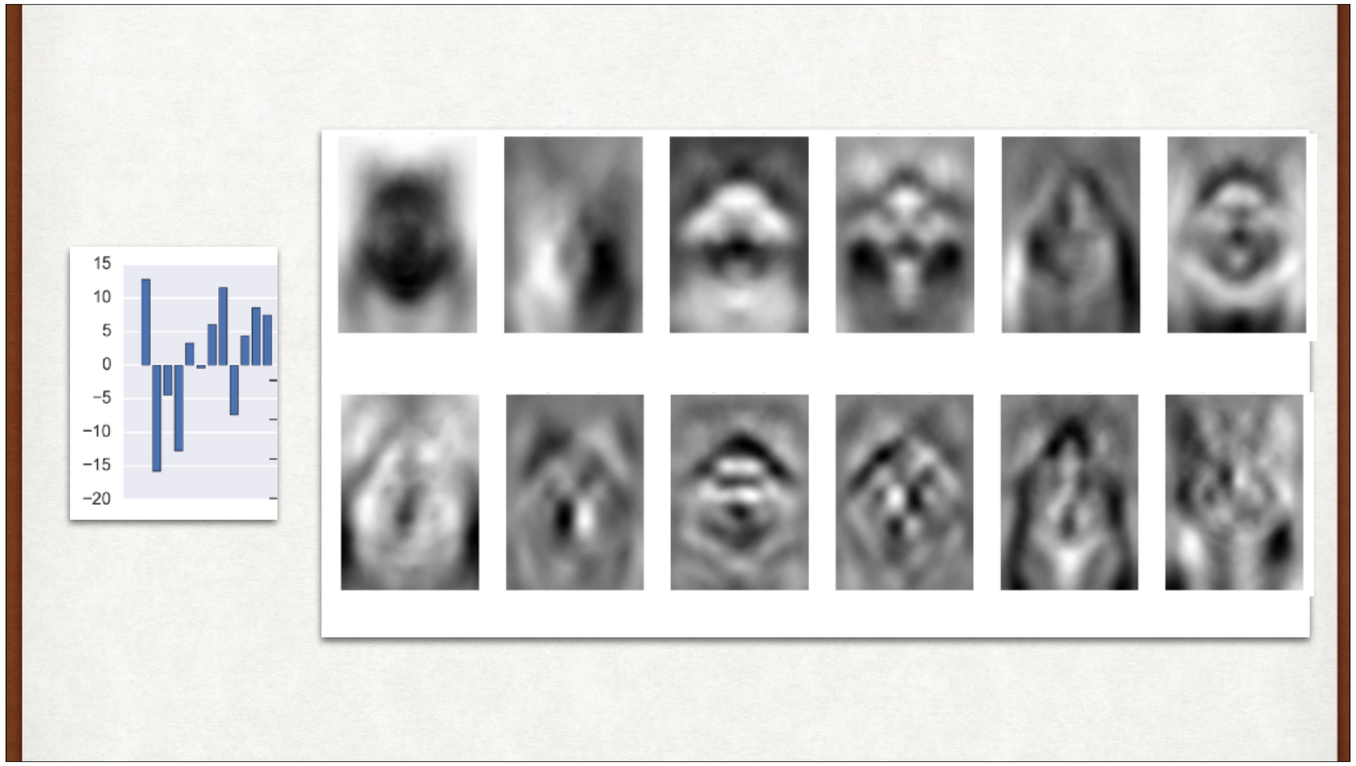


Eigendogs

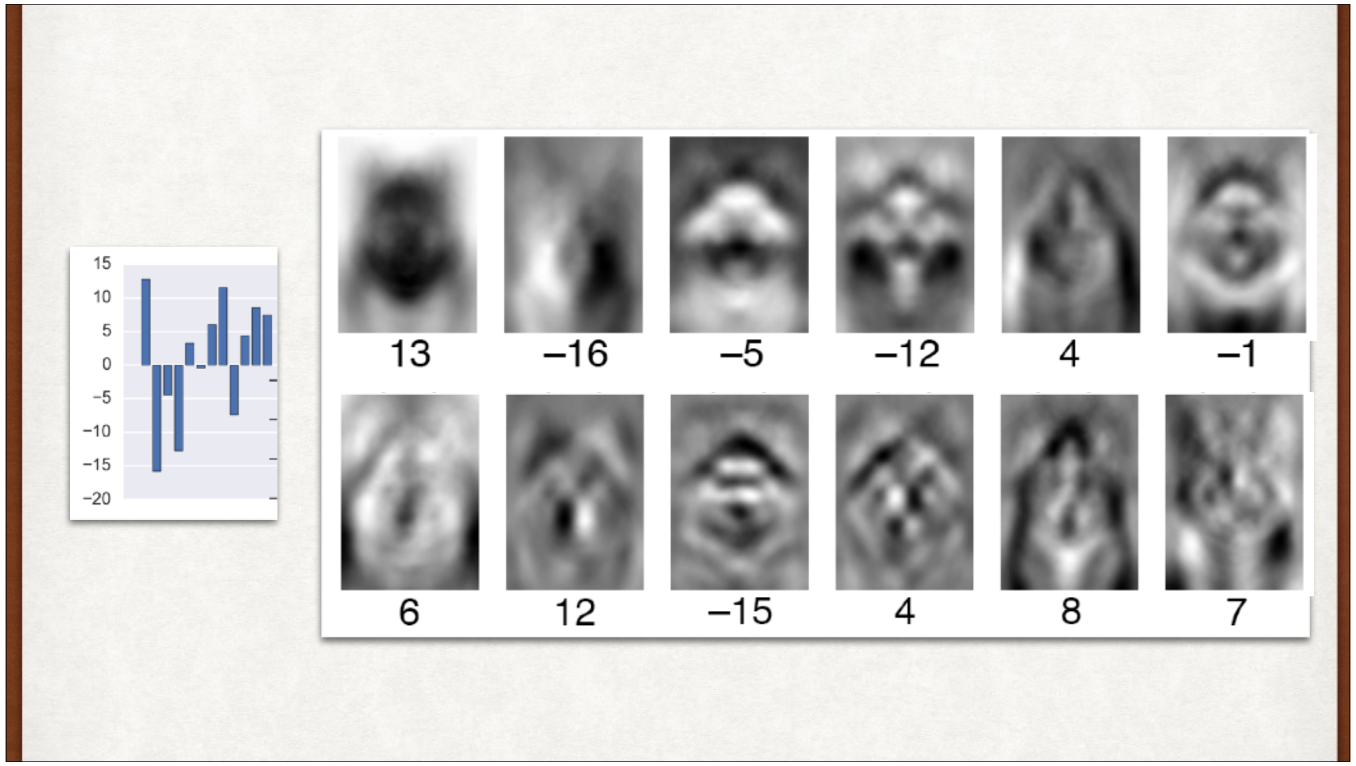
We will find the components of the dog images with an algorithm that finds their “eigenvectors,” or important components. These are dogs, so we’ll call the eigenvectors by the much better name eigendogs. Here are the first 12 eigendogs from our data. By mixing these together with different weights, we 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…



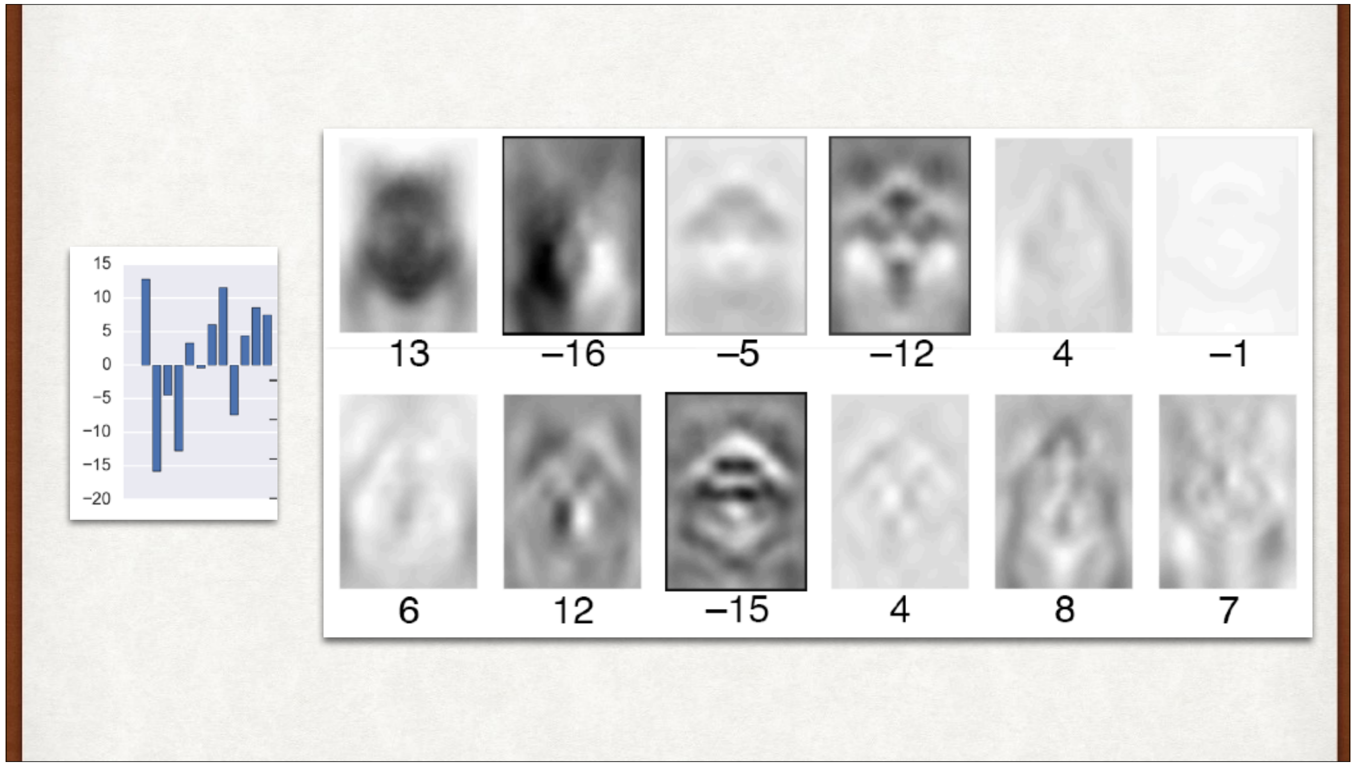
We will find the “most important” parts of the dog images with an algorithm that finds their “eigenvectors,” or important components. These are dogs, so we’ll call the eigenvectors by the much better name eigendogs. Here are the first 12 eigendogs from our data. By mixing these together with different 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…



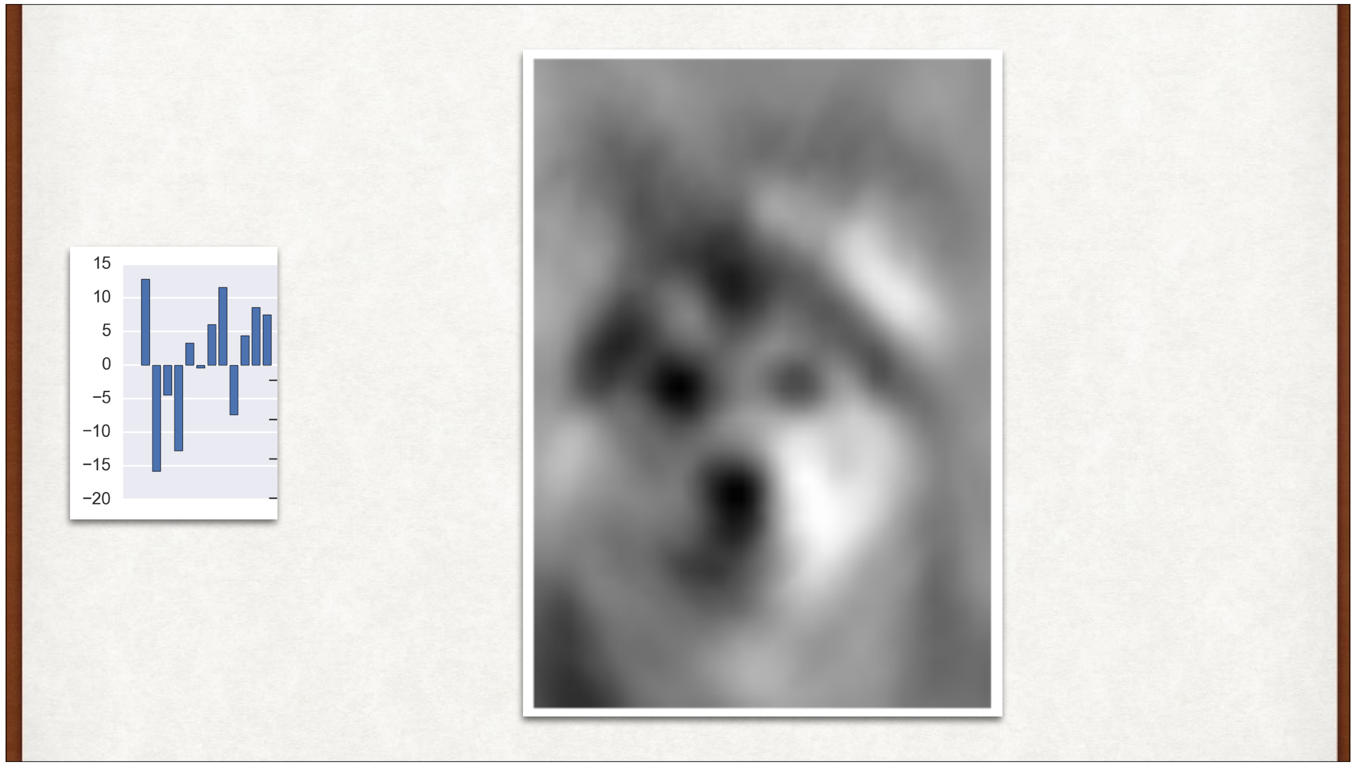
We will find the “most important” parts of the dog images with an algorithm that finds their “eigenvectors,” or important components. These are dogs, so we’ll call the eigenvectors by the much better name eigendogs. Here are the first 12 eigendogs from our data. By mixing these together with different 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…



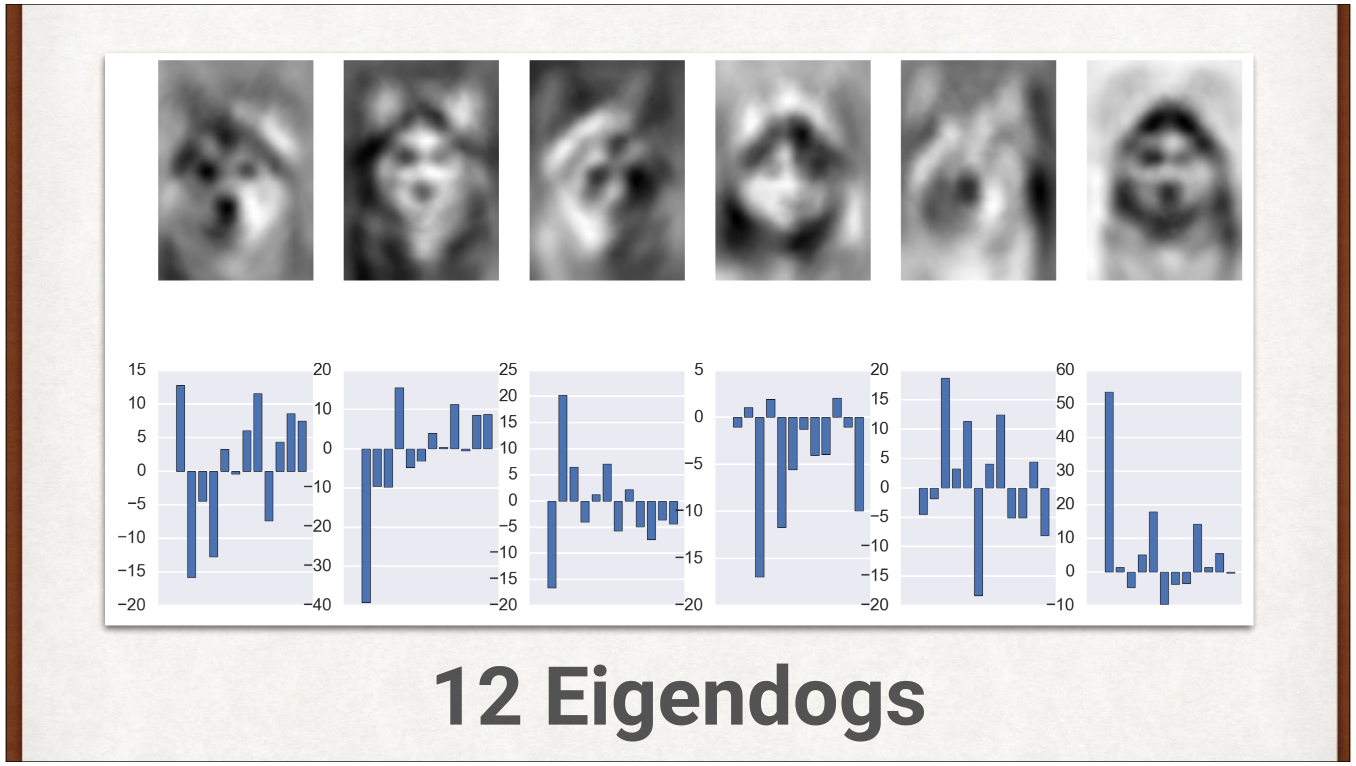
We will find the “most important” parts of the dog images with an algorithm that finds their “eigenvectors,” or important components. These are dogs, so we’ll call the eigenvectors by the much better name eigendogs. Here are the first 12 eigendogs from our data. By mixing these together with different 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…



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…

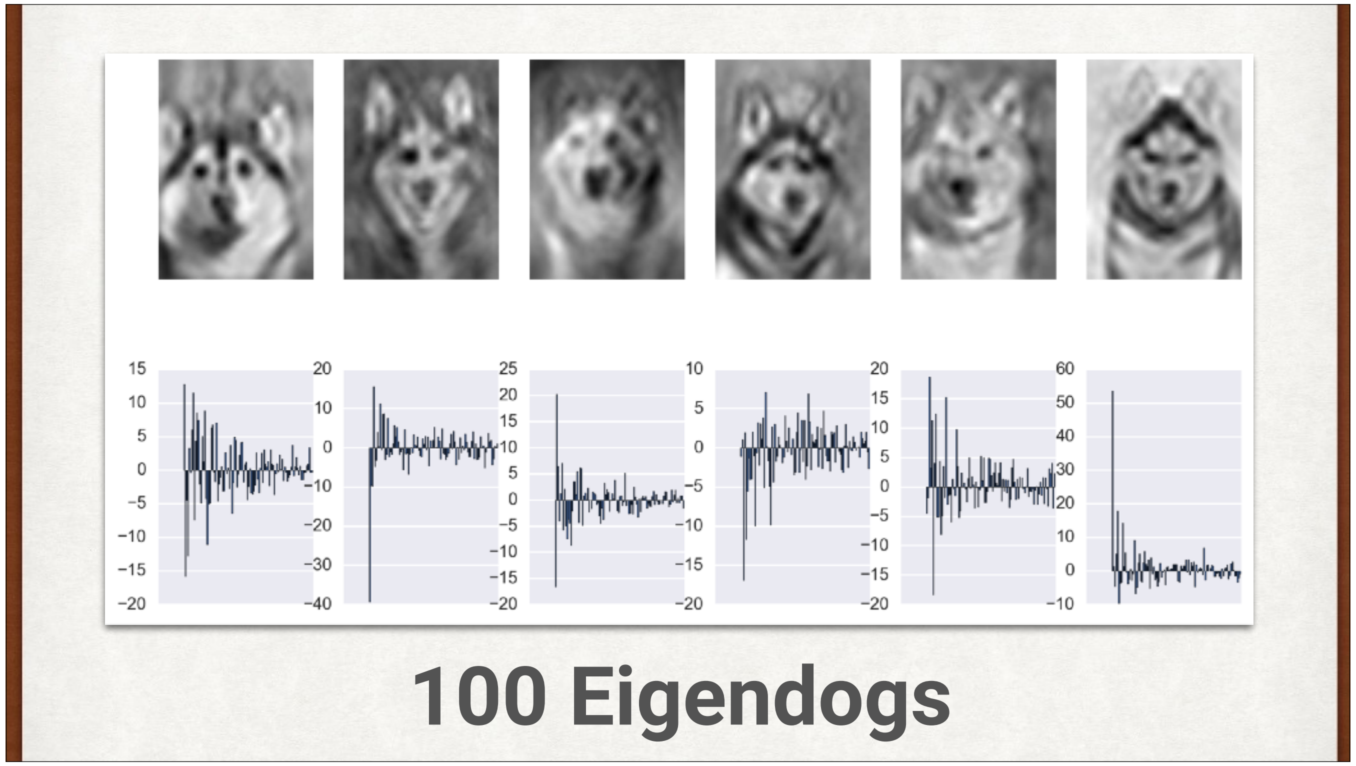


OK, it’s doggish. Not great. Let’s look at all six…



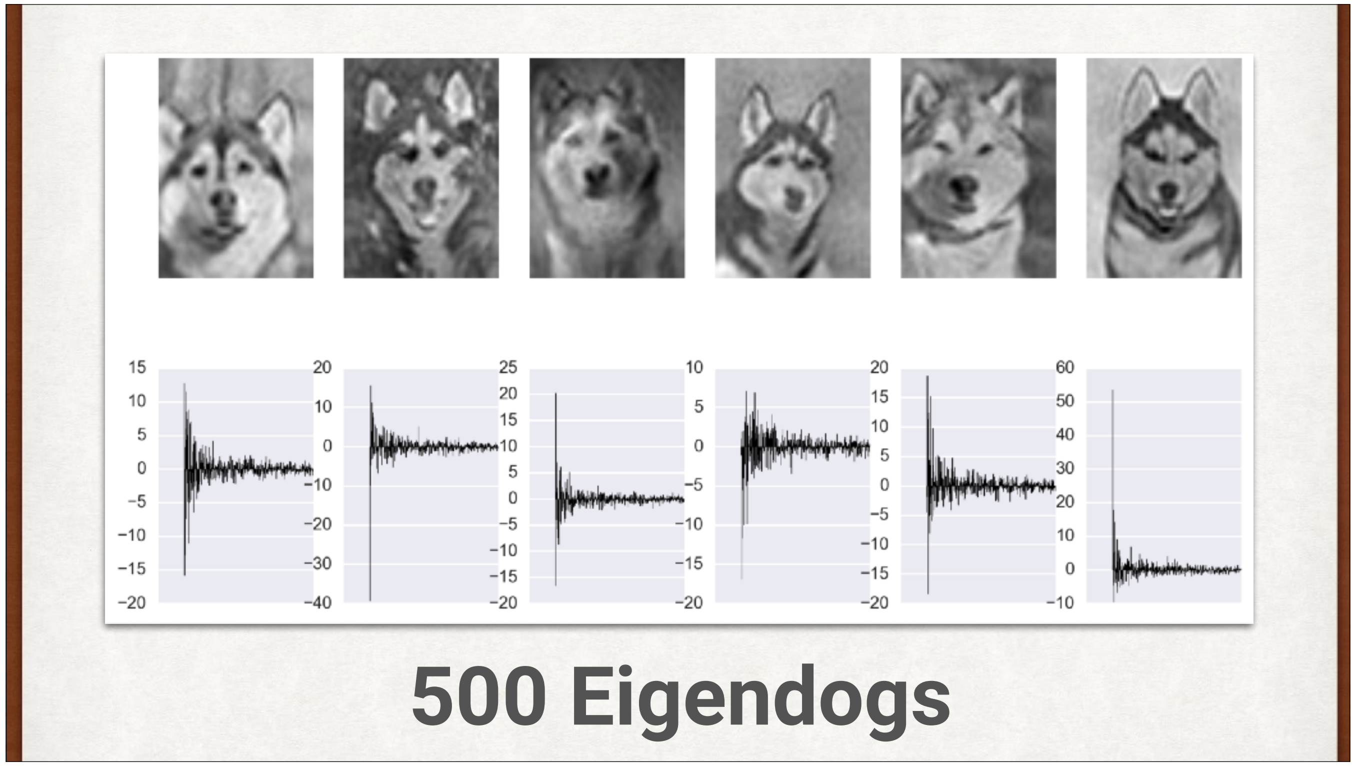
12 Eigendogs

…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.



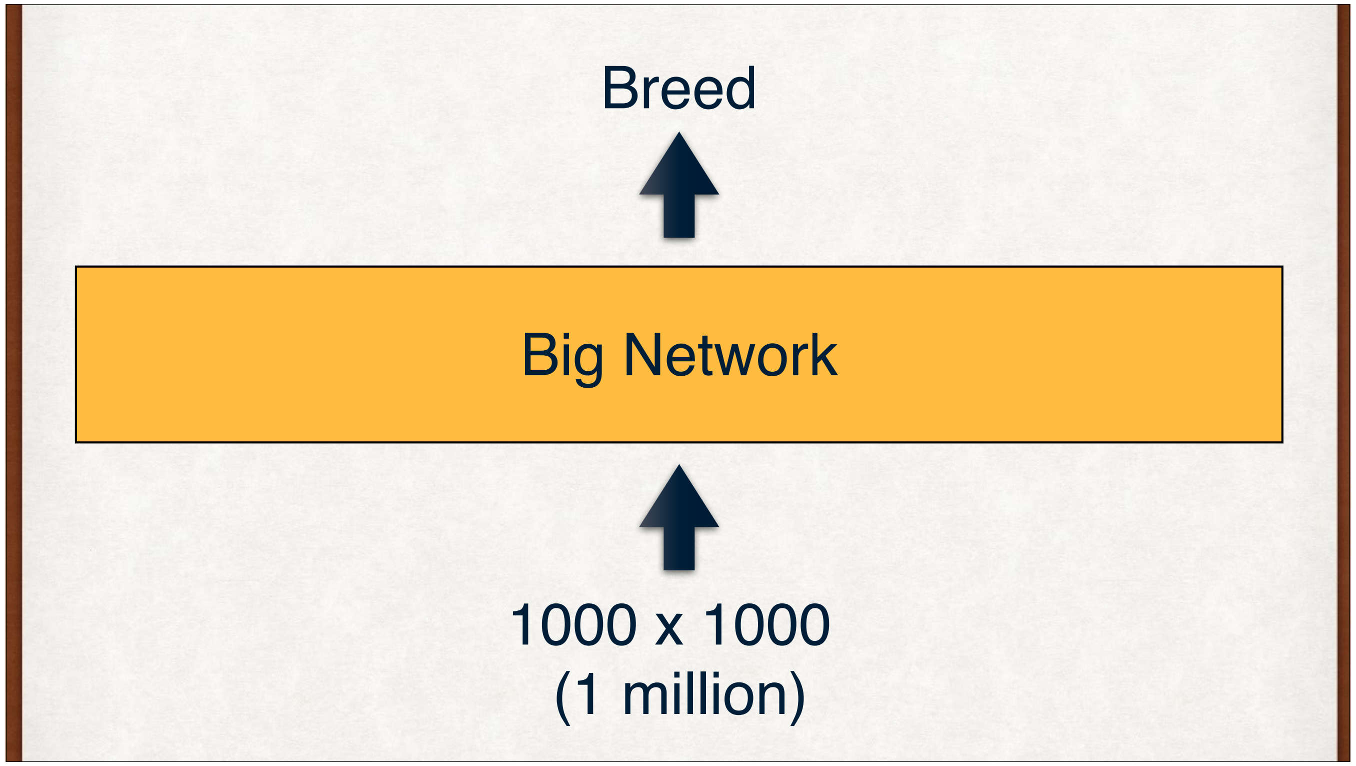
100 Eigendogs

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 tens of thousands (or more) that make up each image. We also need the 500 eigendog images to recover the dog 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. The learner can be smaller and will run faster.

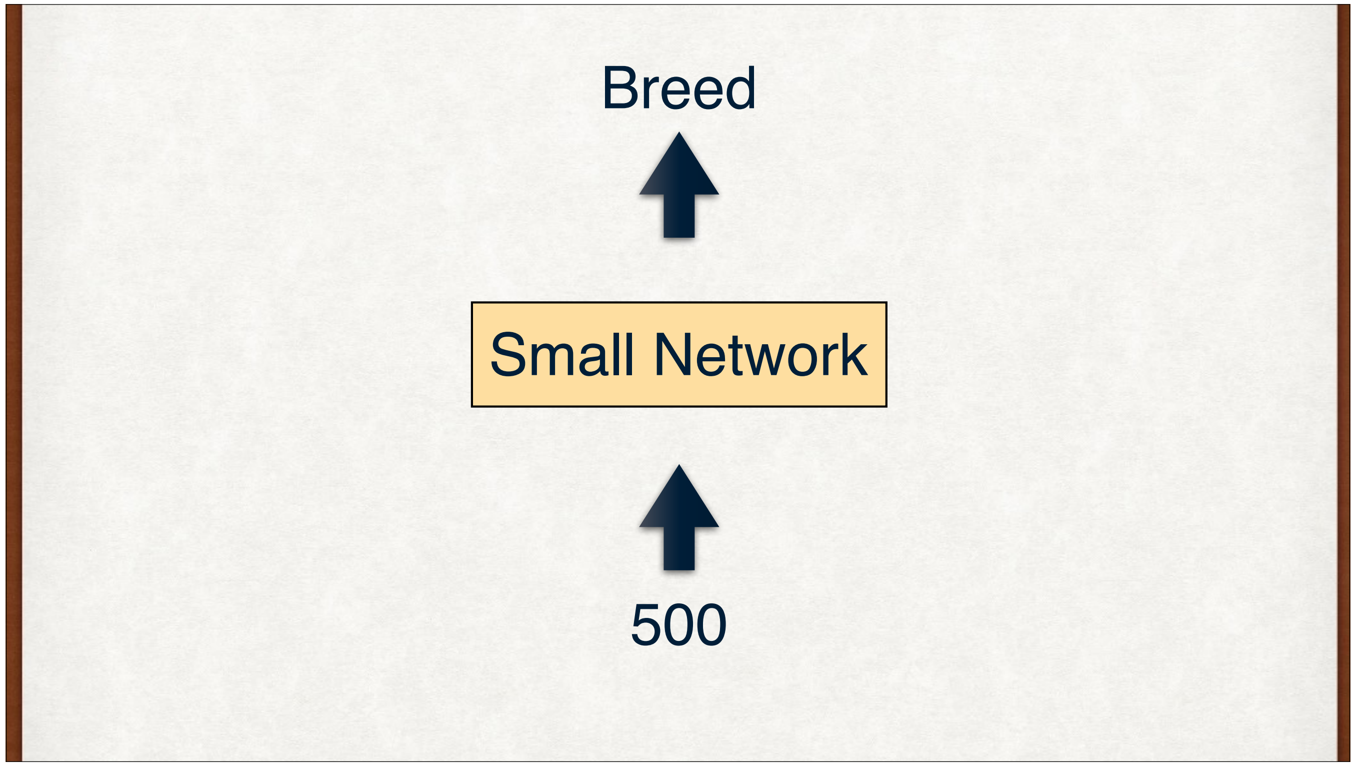


Breed

Big Network

1000 x 1000
(1 million)

So instead of some big neural network…

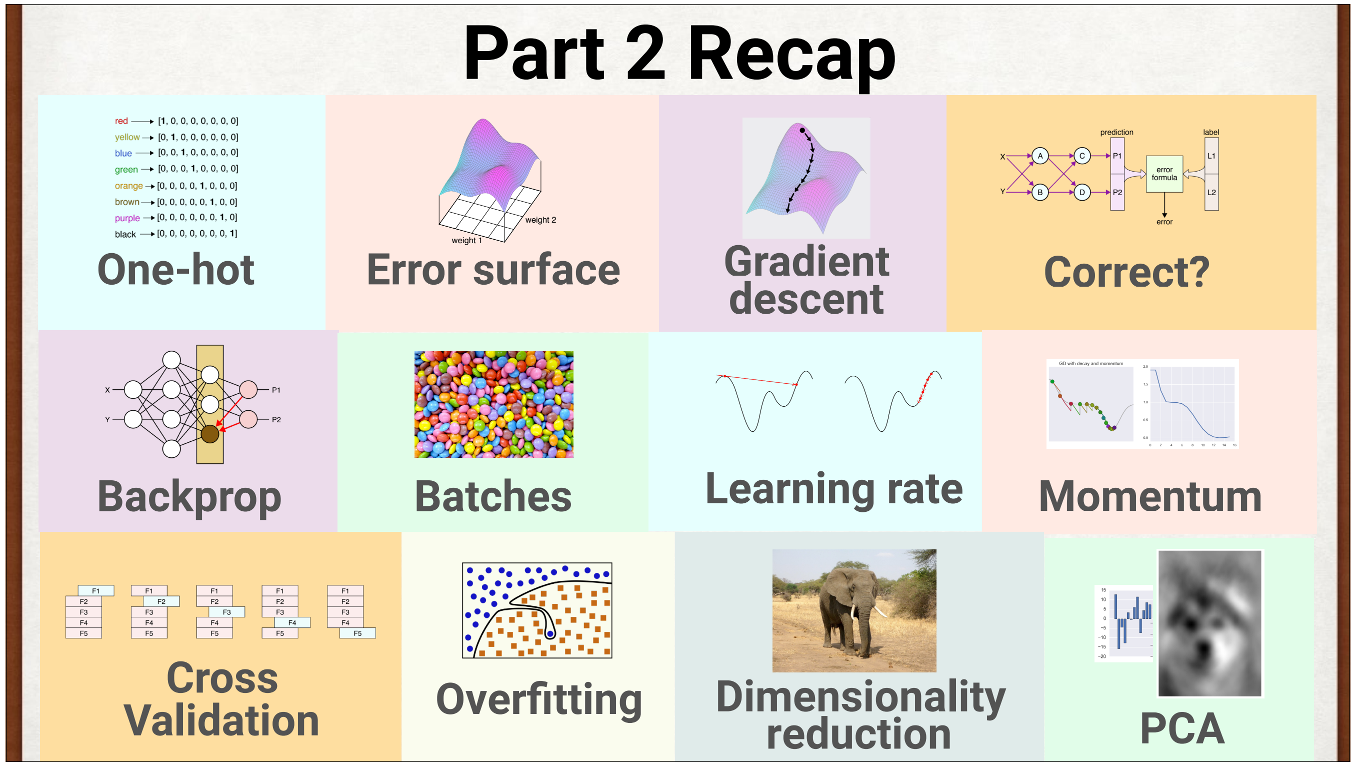


Breed

Small Network

500

We can use a smaller one. Faster. Less memory. And maybe it will learn better, too.



Part 2 Recap

One-hot

Error surface

Gradient descent

Correct?

Backprop

Batches

Learning rate

Momentum

Cross Validation

Overfitting

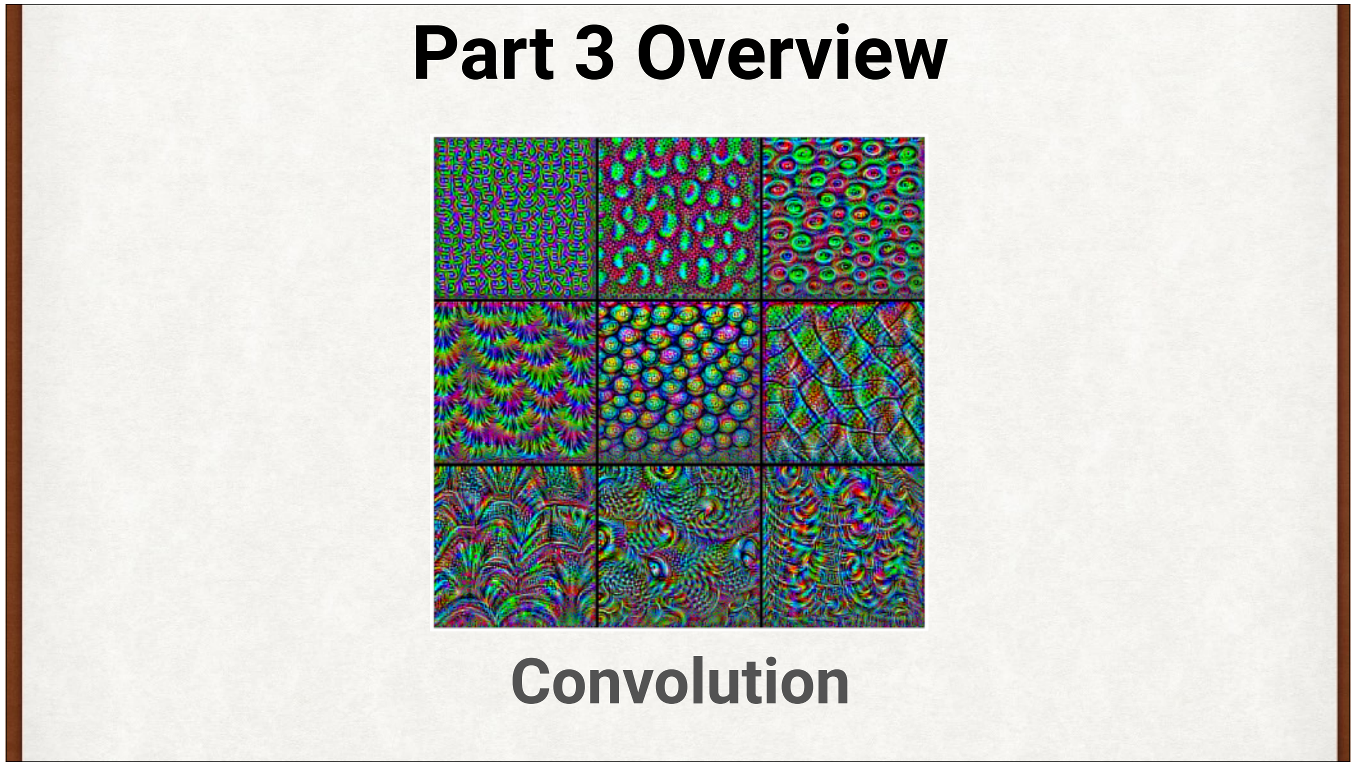
Dimensionality reduction

PCA

red → [1, 0, 0, 0, 0, 0, 0, 0]
yellow → [0, 1, 0, 0, 0, 0, 0, 0]
blue → [0, 0, 1, 0, 0, 0, 0, 0]
green → [0, 0, 0, 1, 0, 0, 0, 0]
orange → [0, 0, 0, 0, 1, 0, 0, 0]
brown → [0, 0, 0, 0, 0, 1, 0, 0]
purple → [0, 0, 0, 0, 0, 0, 1, 0]
black → [0, 0, 0, 0, 0, 0, 0, 1]
weight 1
weight 2
prediction
label
error formula
error
F1
F2
F3
F4
F5

Where we’ve been.

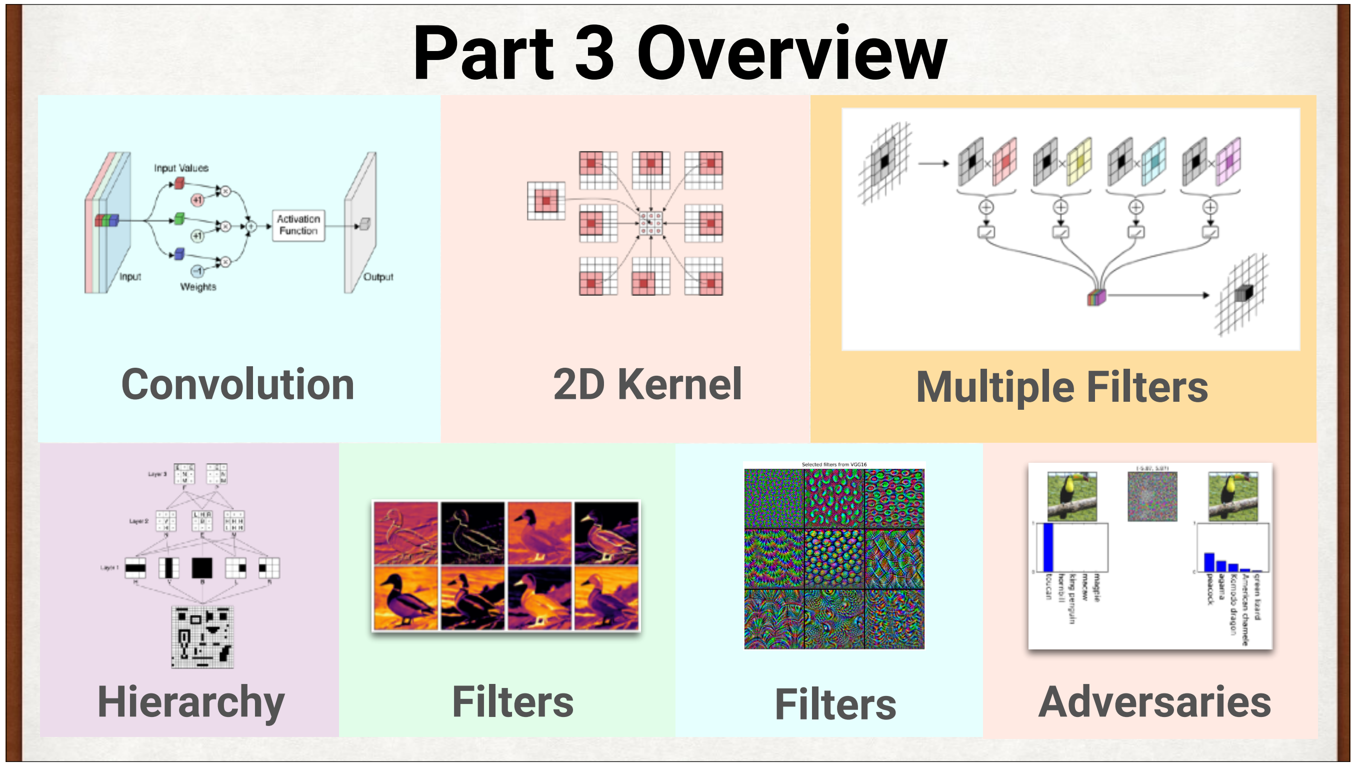
A cleanse of our mental palette. Some visual sherbet between courses. Let’s talk about something new.



Part 3 Overview

Convolution

Convolution!



Part 3 Overview

Convolution

2D Kernel

Multiple Filters

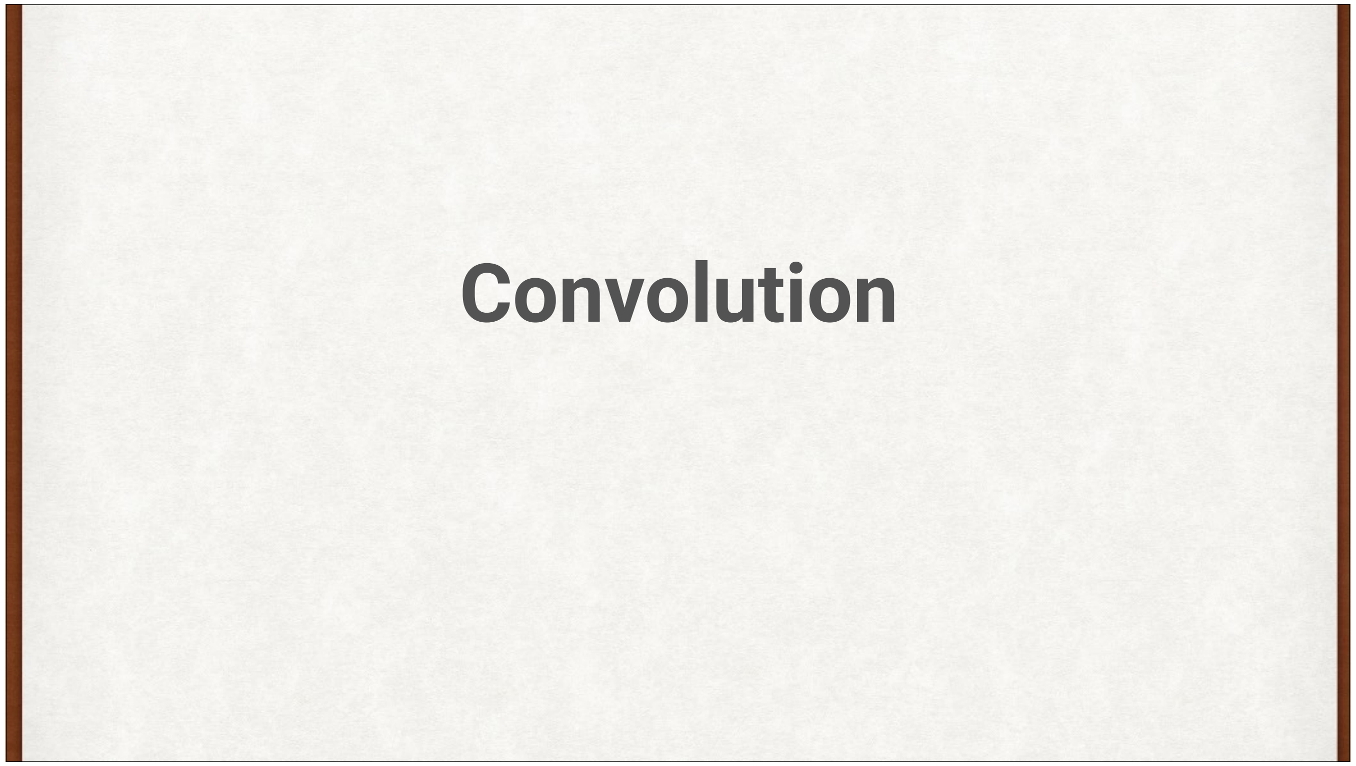
Hierarchy

Filters

Filters

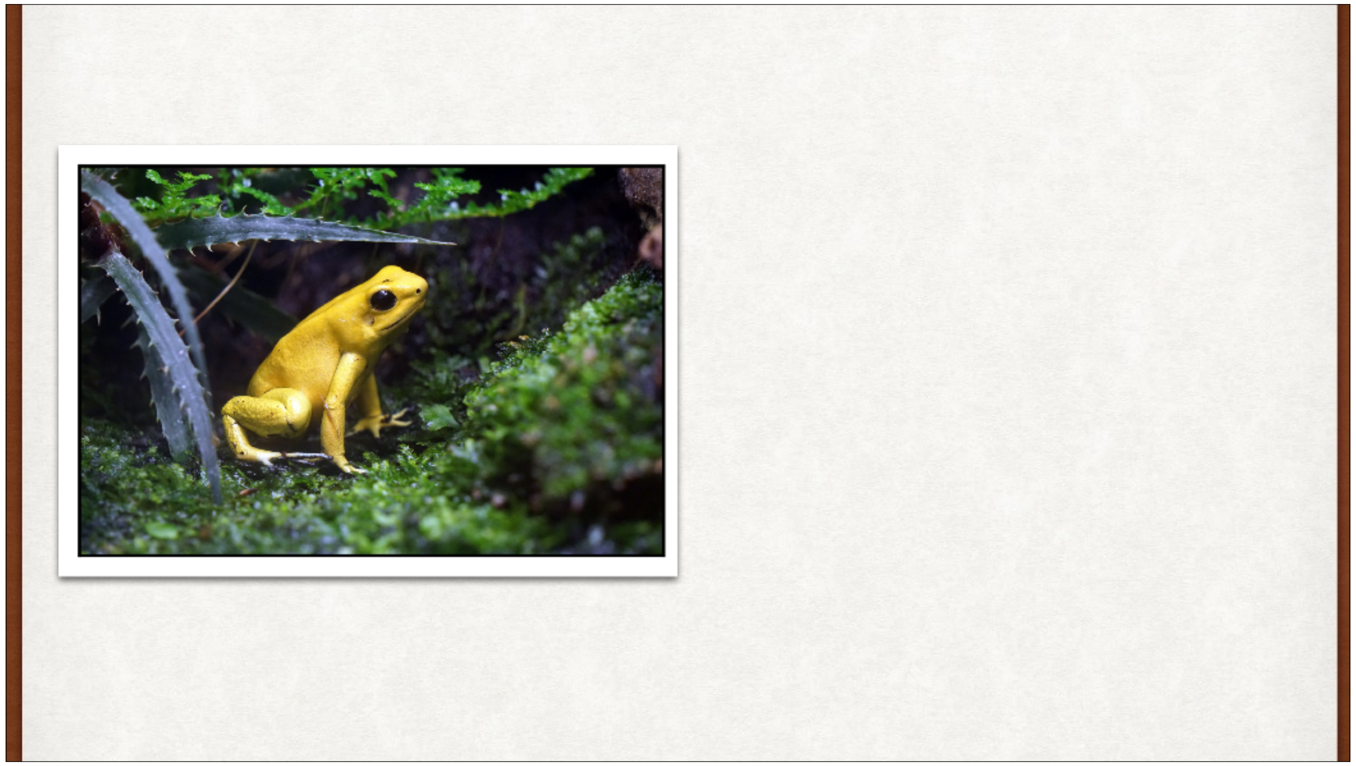
Adversaries

What we’re gonna do next.



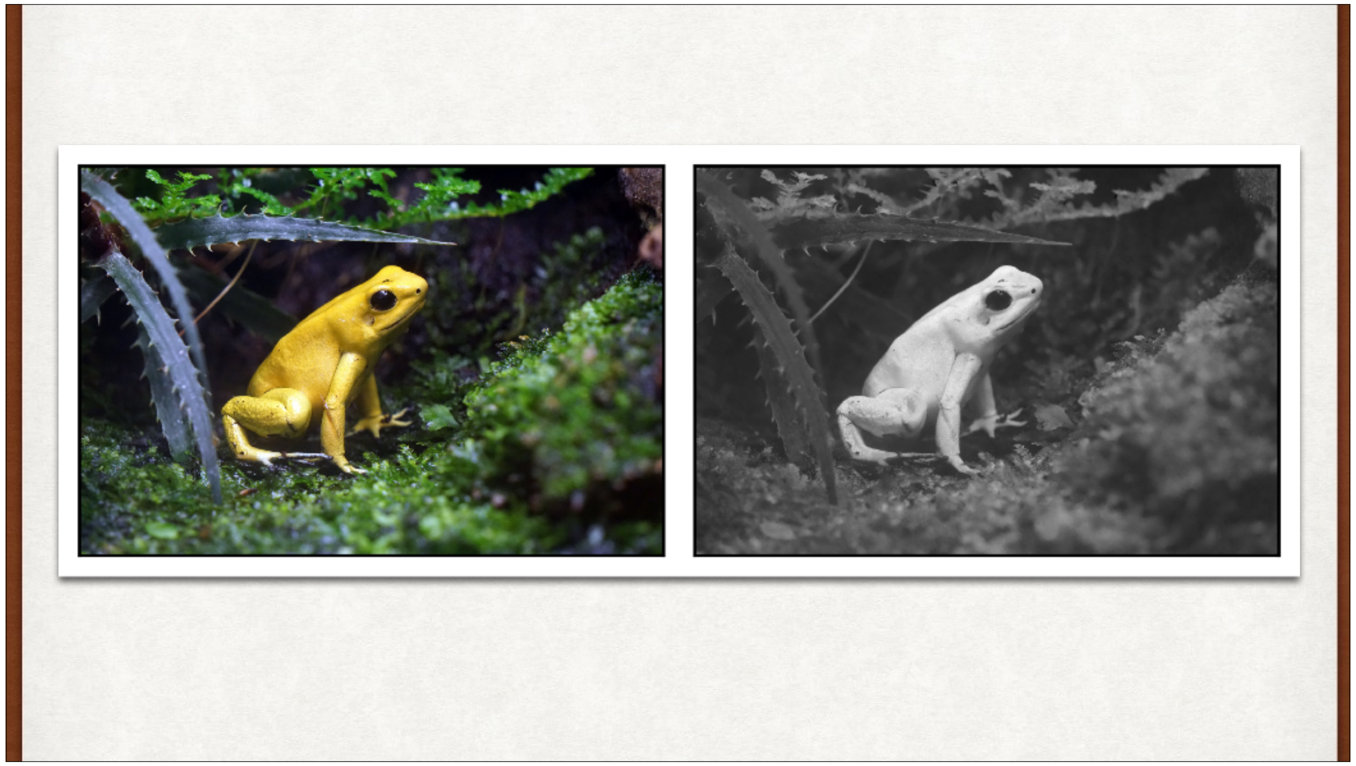
Convolution

Convolution is hugely popular for image-based systems. Let’s see what it’s all about, without math!

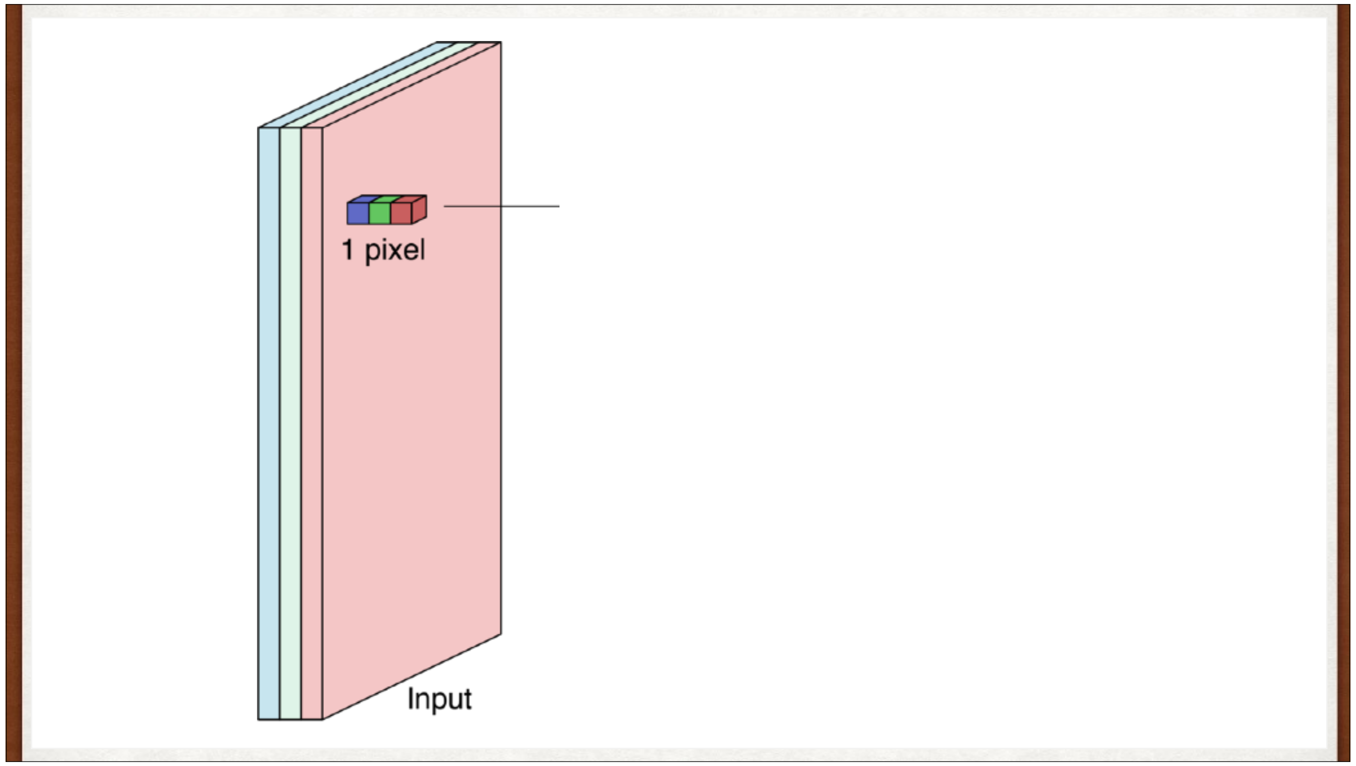


Our yellow filter applied to a frog, with the results scaled to 0=black, 1=white.

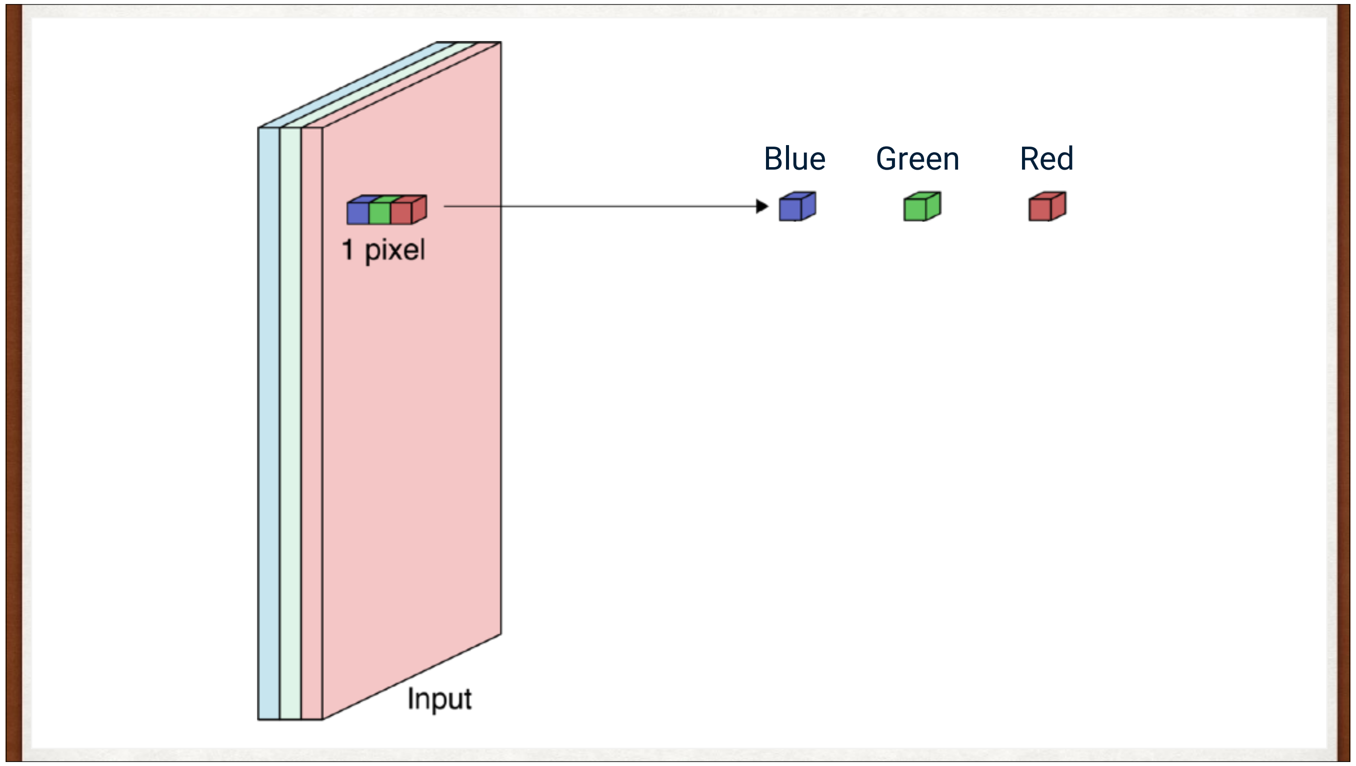
https://pixabay.com/en/frog-toxic-yellow-netherlands-1463831



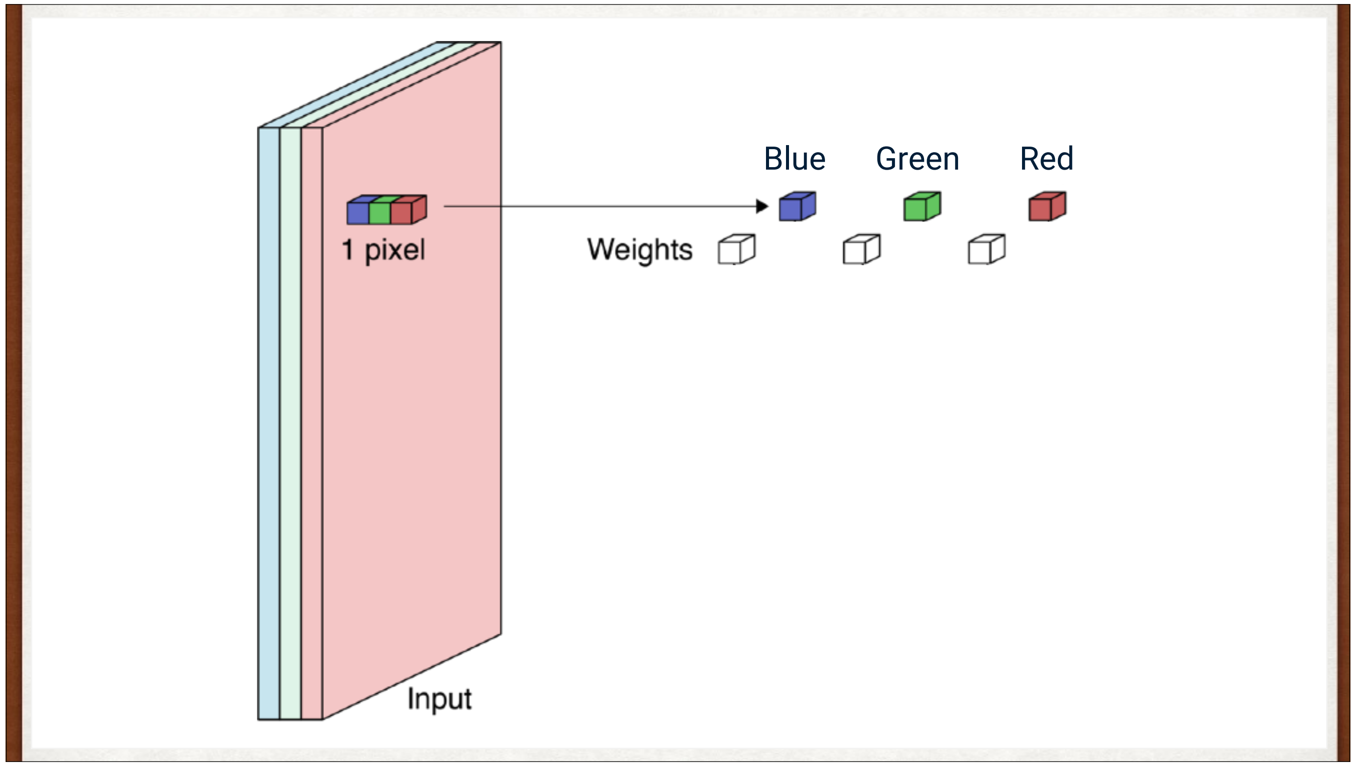
Our yellow filter applied to a frog, with the results scaled to 0=black, 1=white.



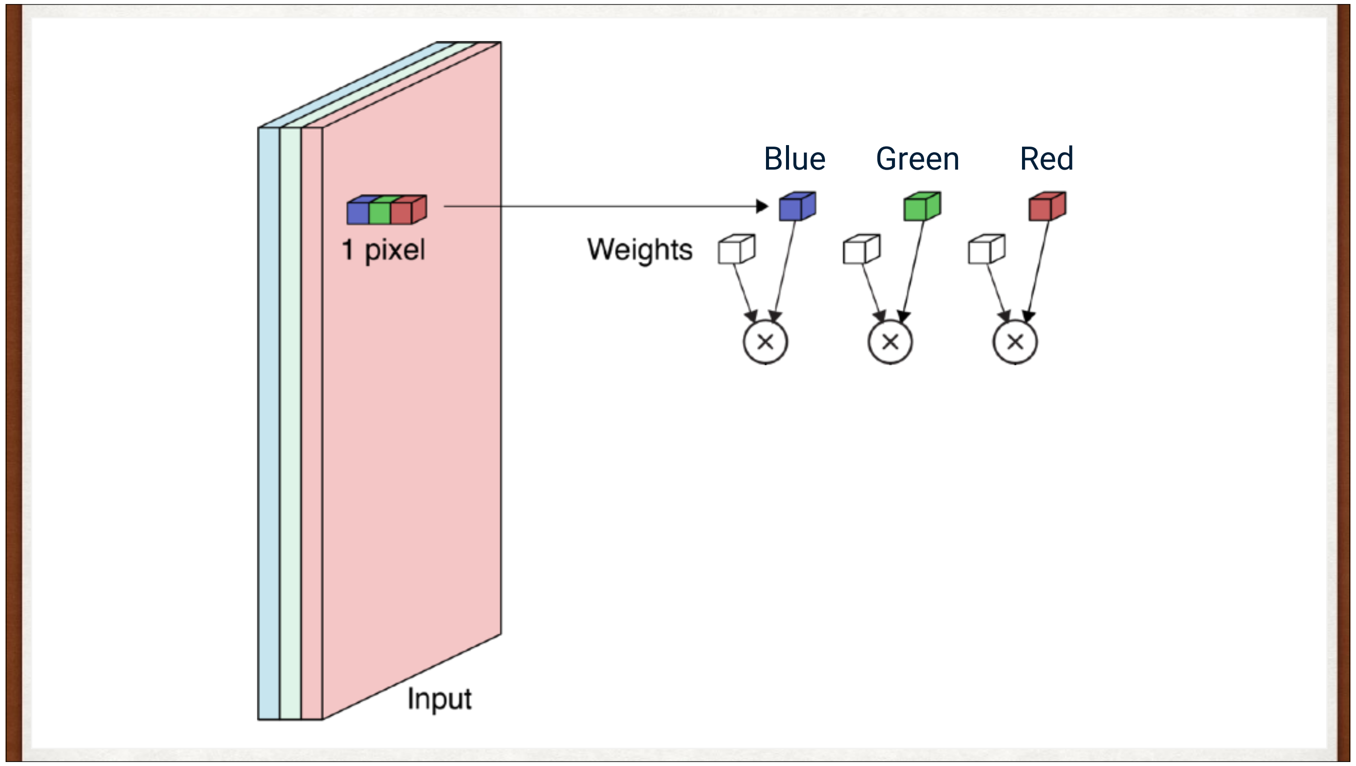
An RGB picture at left. We take a “core sample” of 1 pixel. We multiply the red, green, and blue values by associated weights. Then we add up the results. For our purposes, this is “convolution”. We say that we have “convolved” the input (the core sample) with a filter (the three weights).



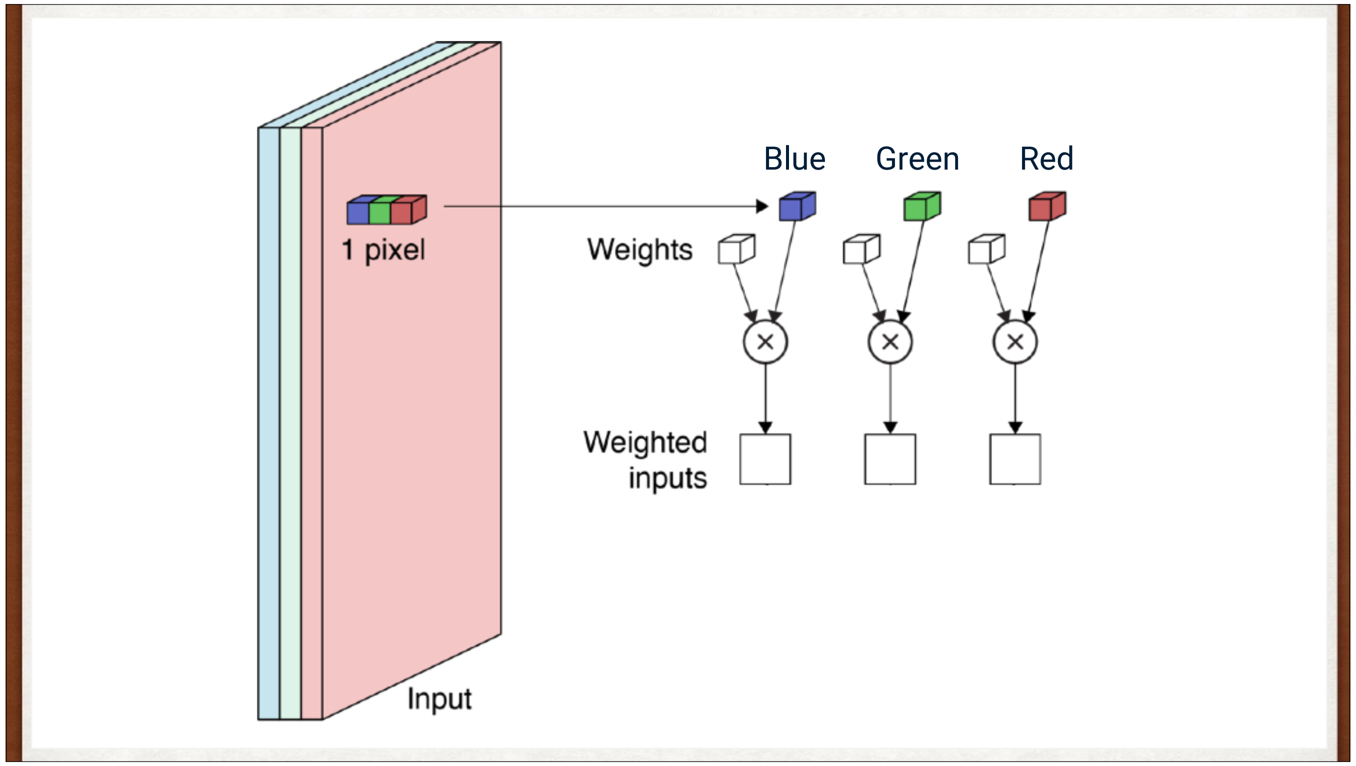
An RGB picture at left. We take a “core sample” of 1 pixel. We multiply the red, green, and blue values by associated weights. Then we add up the results. For our purposes, this is “convolution”. We say that we have “convolved” the input (the core sample) with a filter (the three weights).



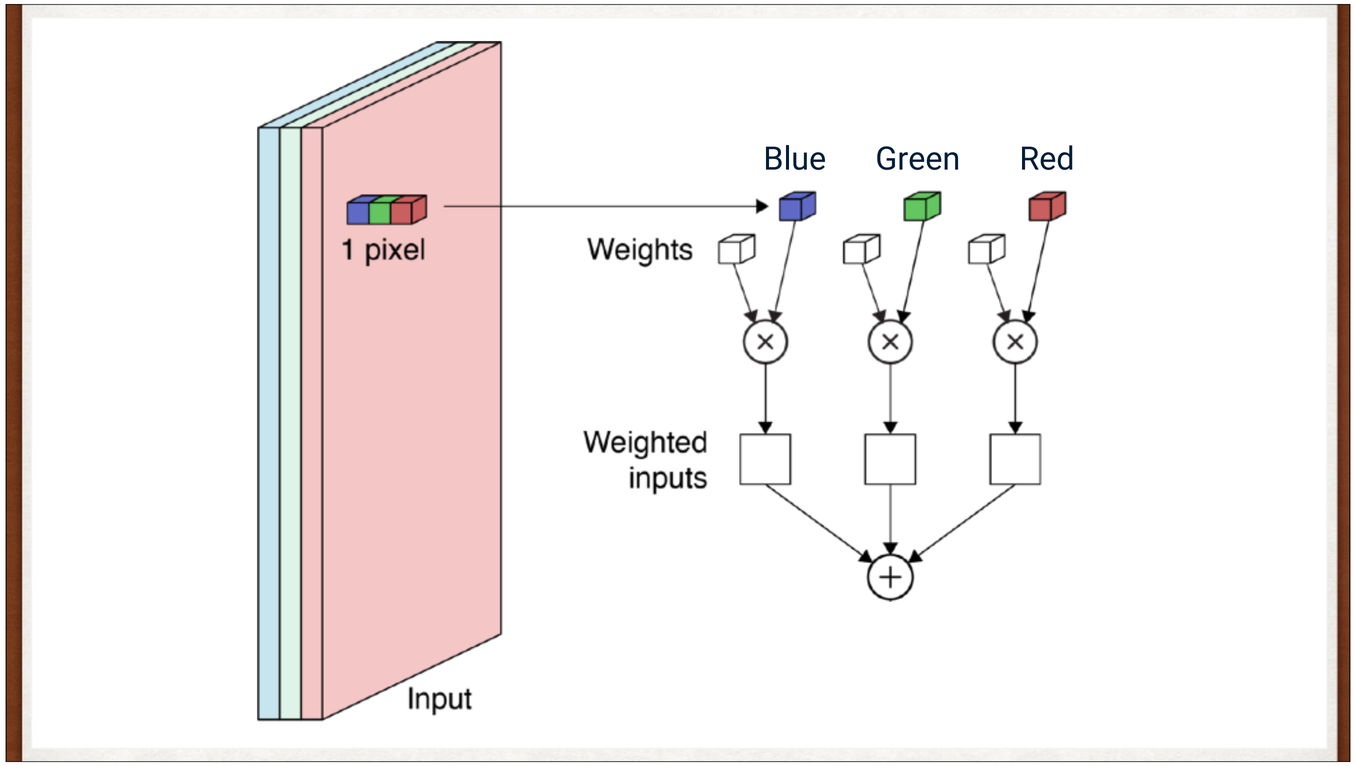
An RGB picture at left. We take a “core sample” of 1 pixel. We multiply the red, green, and blue values by associated weights. Then we add up the results. For our purposes, this is “convolution”. We say that we have “convolved” the input (the core sample) with a filter (the three weights).



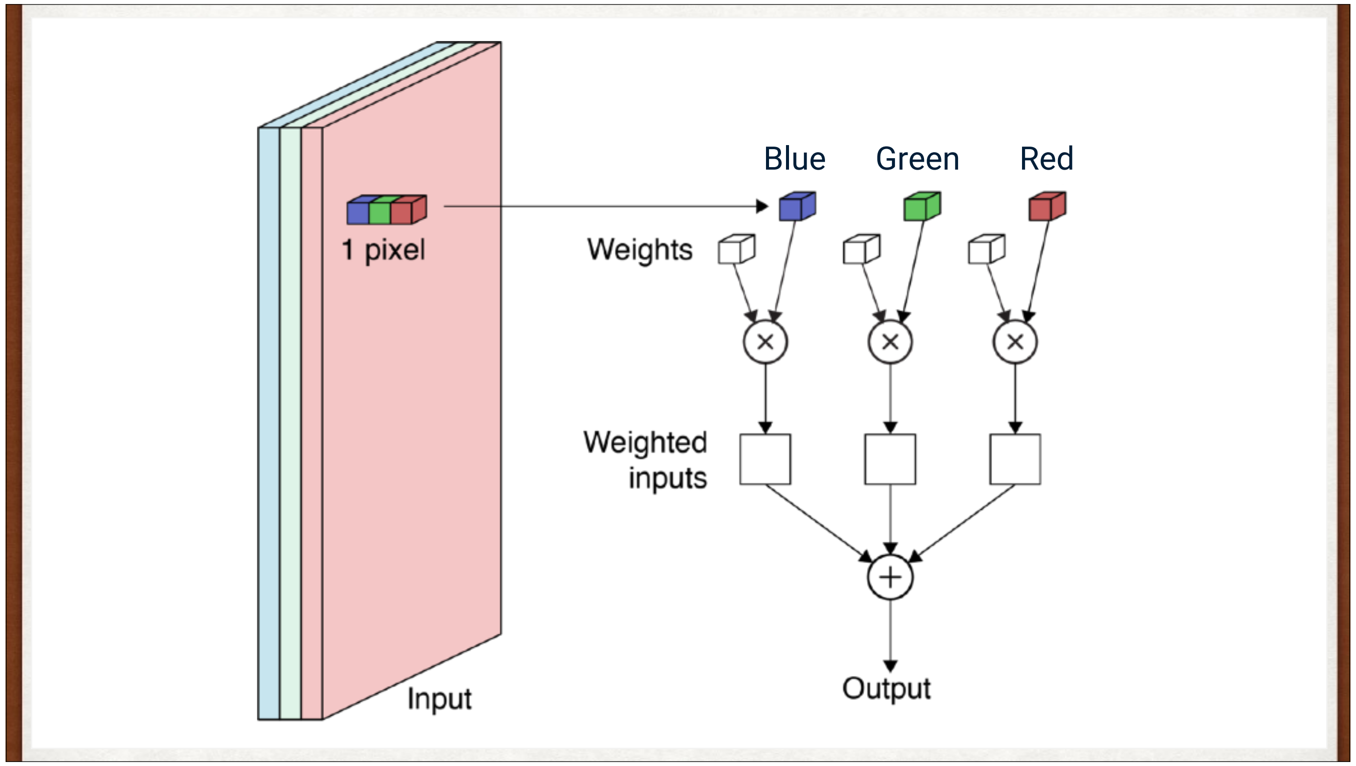
An RGB picture at left. We take a “core sample” of 1 pixel. We multiply the red, green, and blue values by associated weights. Then we add up the results. For our purposes, this is “convolution”. We say that we have “convolved” the input (the core sample) with a filter (the three weights).



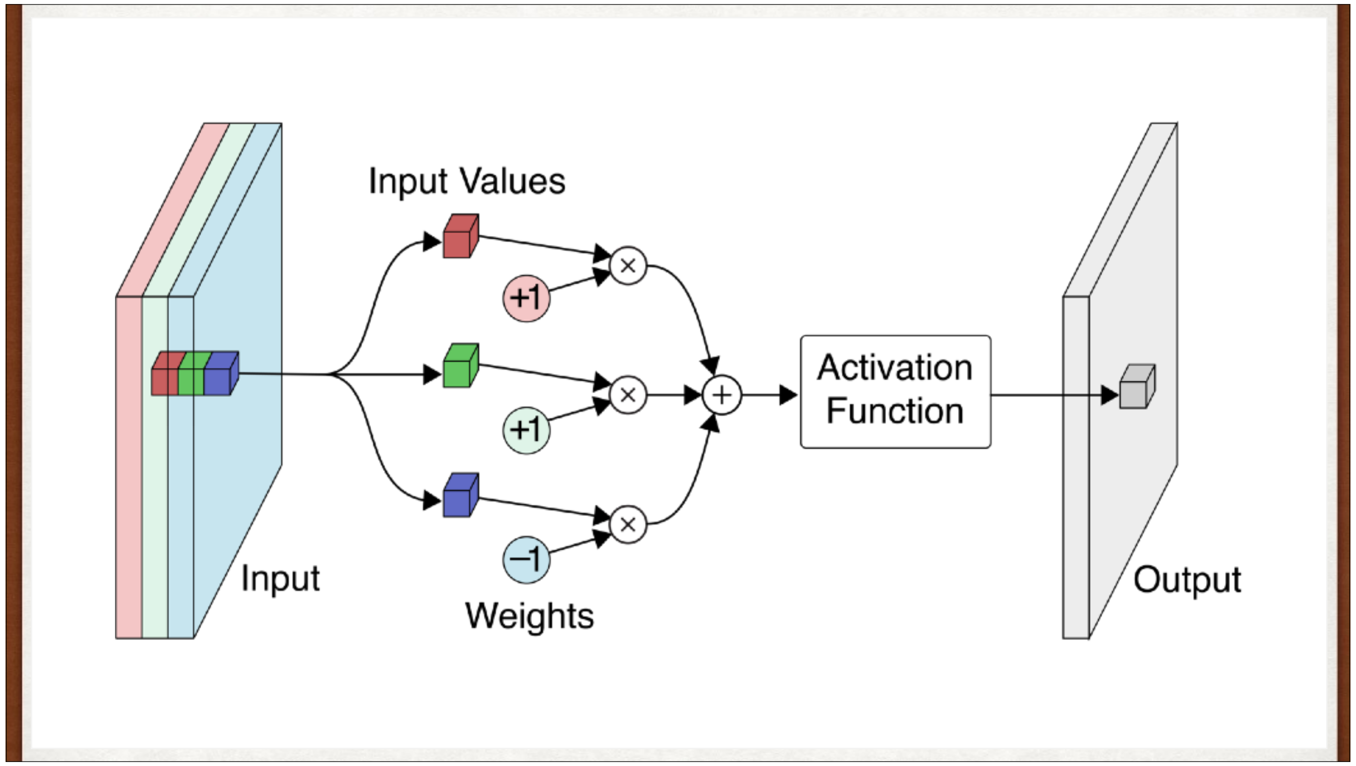
An RGB picture at left. We take a “core sample” of 1 pixel. We multiply the red, green, and blue values by associated weights. Then we add up the results. For our purposes, this is “convolution”. We say that we have “convolved” the input (the core sample) with a filter (the three weights).



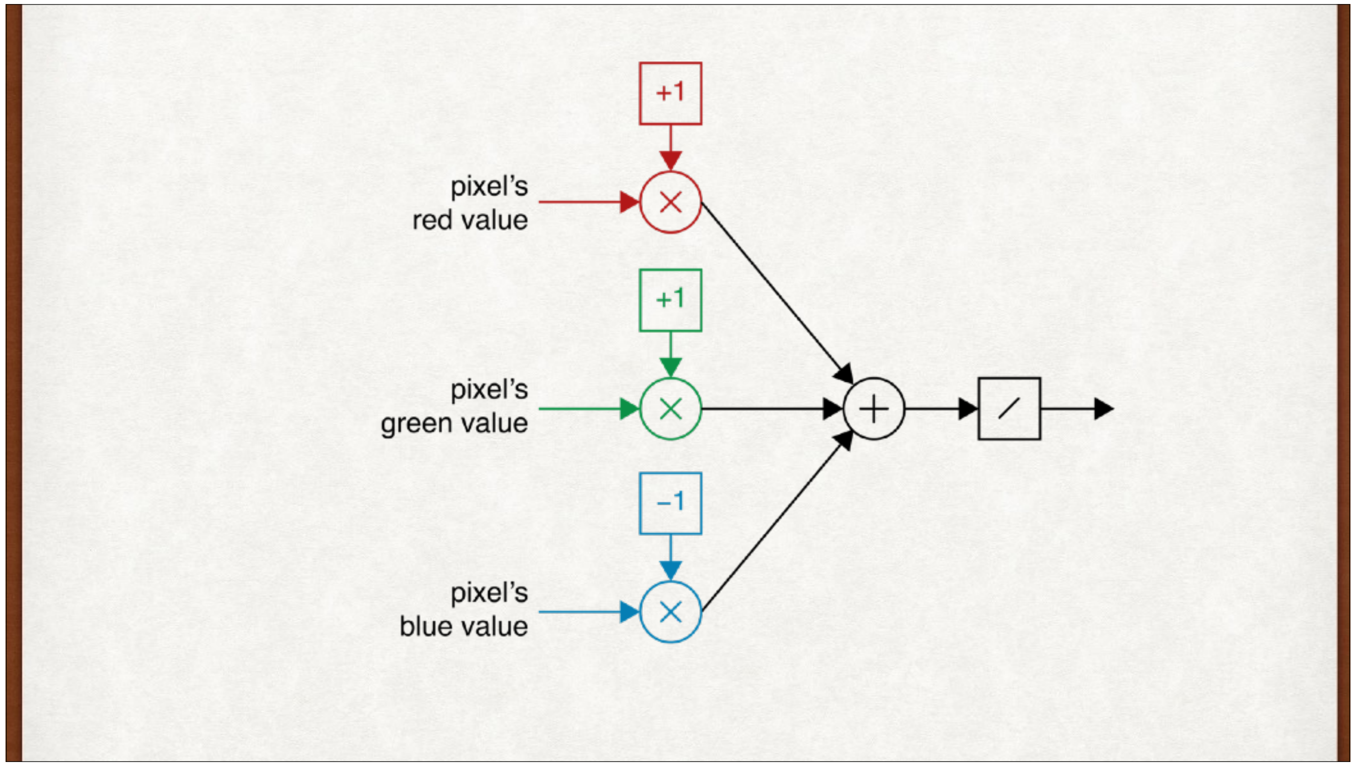
An RGB picture at left. We take a “core sample” of 1 pixel. We multiply the red, green, and blue values by associated weights. Then we add up the results. For our purposes, this is “convolution”. We say that we have “convolved” the input (the core sample) with a filter (the three weights).



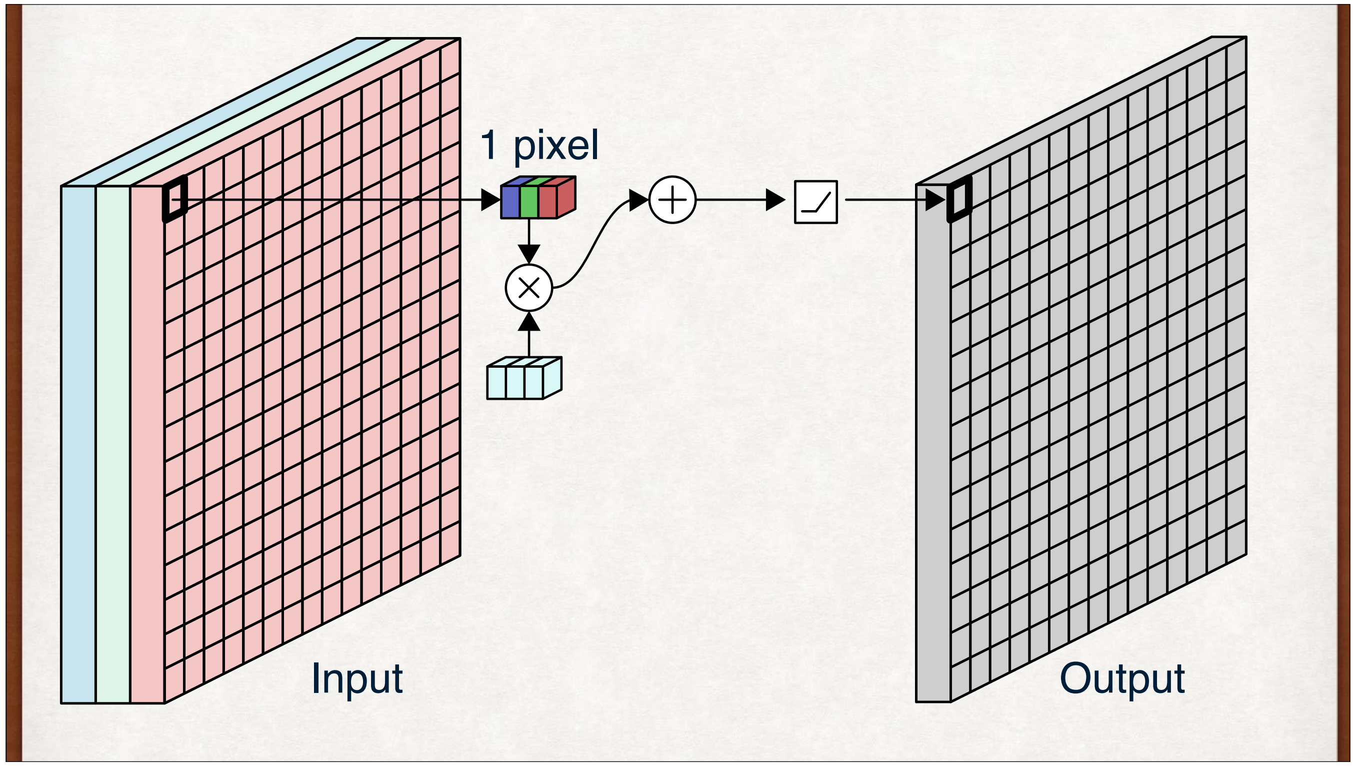
An RGB picture at left. We take a “core sample” of 1 pixel. We multiply the red, green, and blue values by associated weights. Then we add up the results. For our purposes, this is “convolution”. We say that we have “convolved” the input (the core sample) with a filter (the three weights). Hey, this looks like a perceptron.



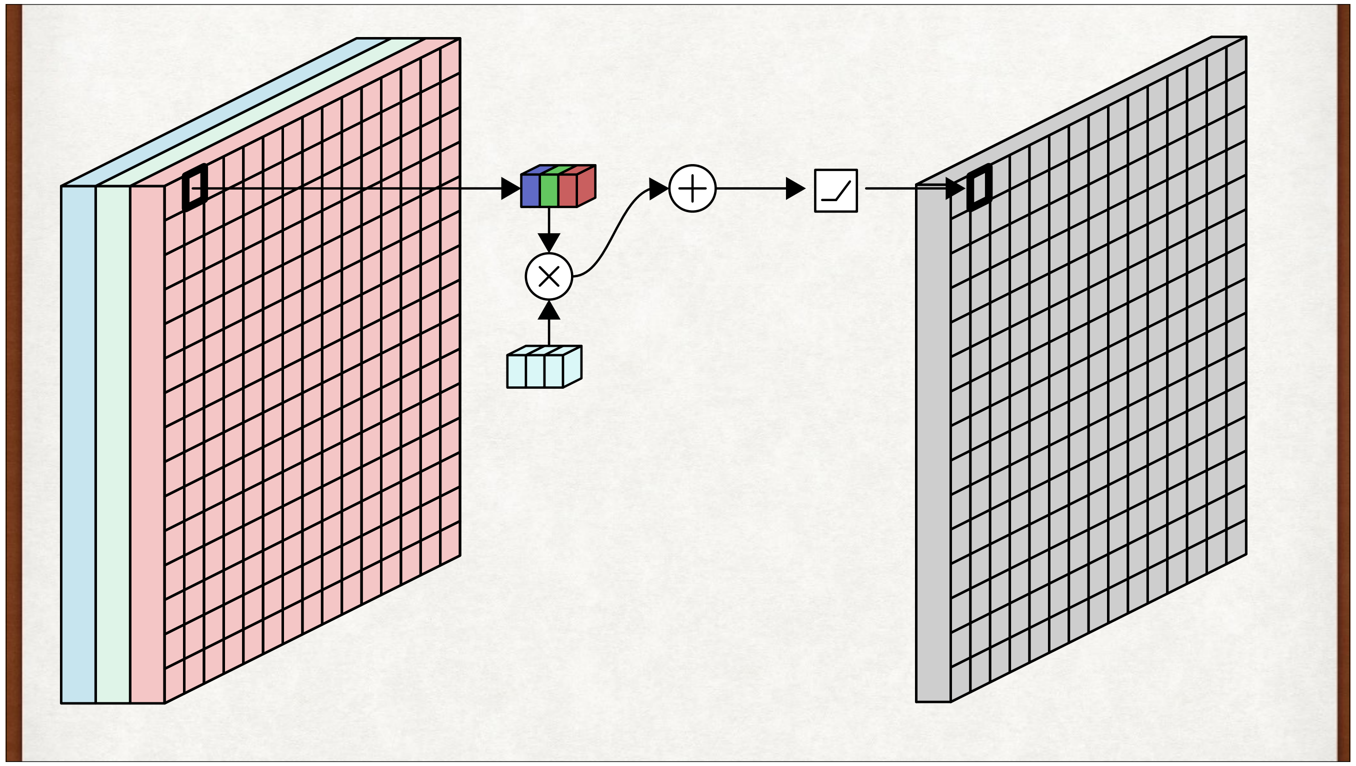
Yup, it is a perceptron.



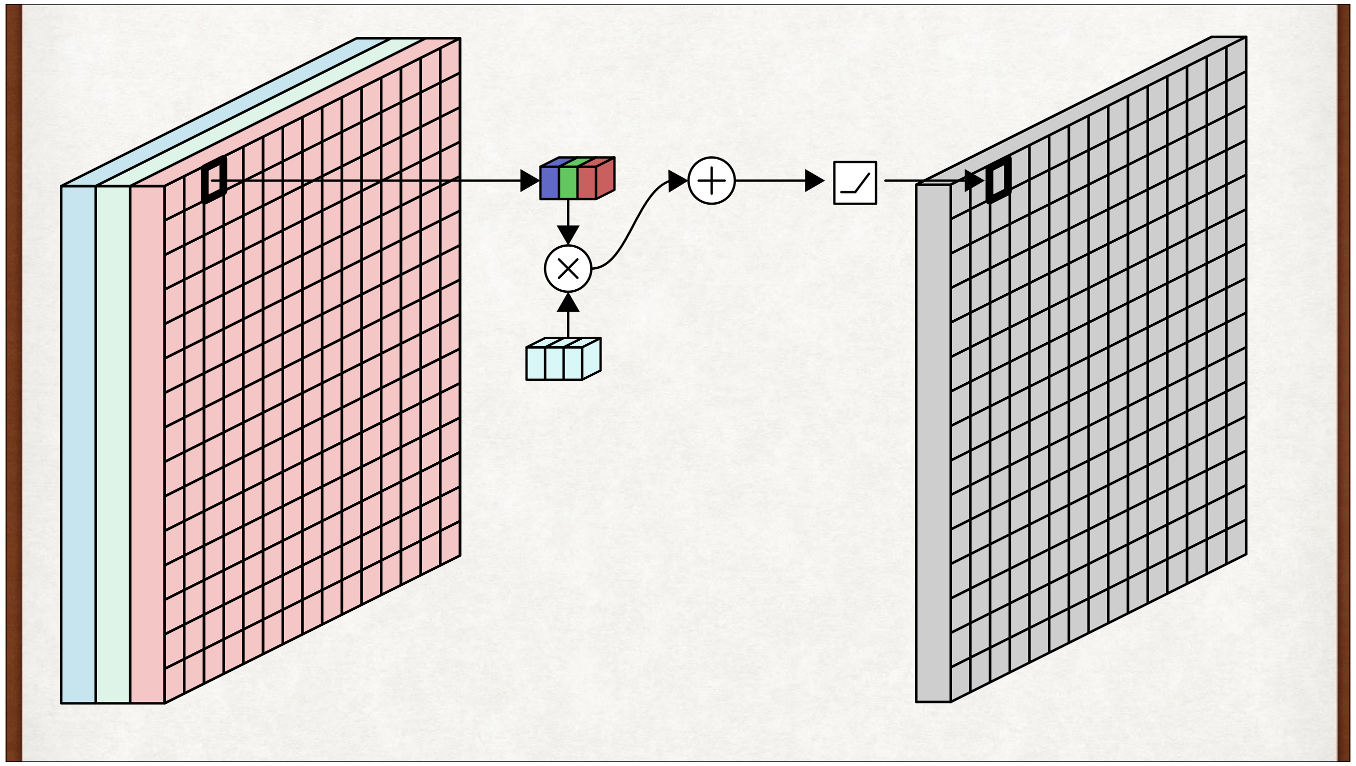
Using convolution to find yellow pixels (red=1, green=1, blue=0). The more yellow a pixel, the greater the output, to a maximum of +2. This looks a lot like an artificial neuron, or perceptron! The activation function in this case is just a line at 45 degrees (which is a bad idea in general, since it lets the neurons collapse, but it makes things easier when first getting the hang of this step). In practice, we'd use a real activation function like ReLU.



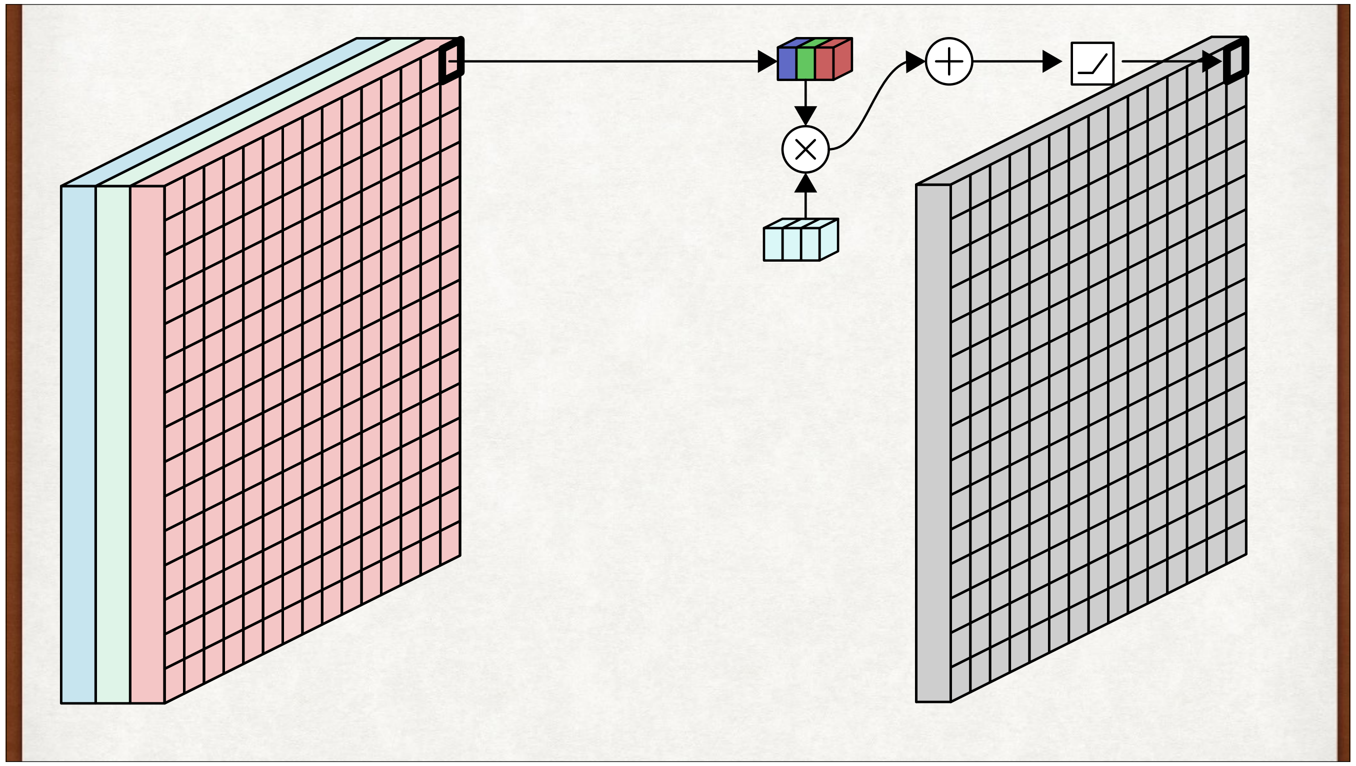
We’ll save our weighted, summed core sample in an output image of just one channel (such as grayscale). Here’s the core sample, the weights (or filter, or kernel) that multiply the RGB, then we add up the results, go through an activation function, and that gives us a single value in the output.



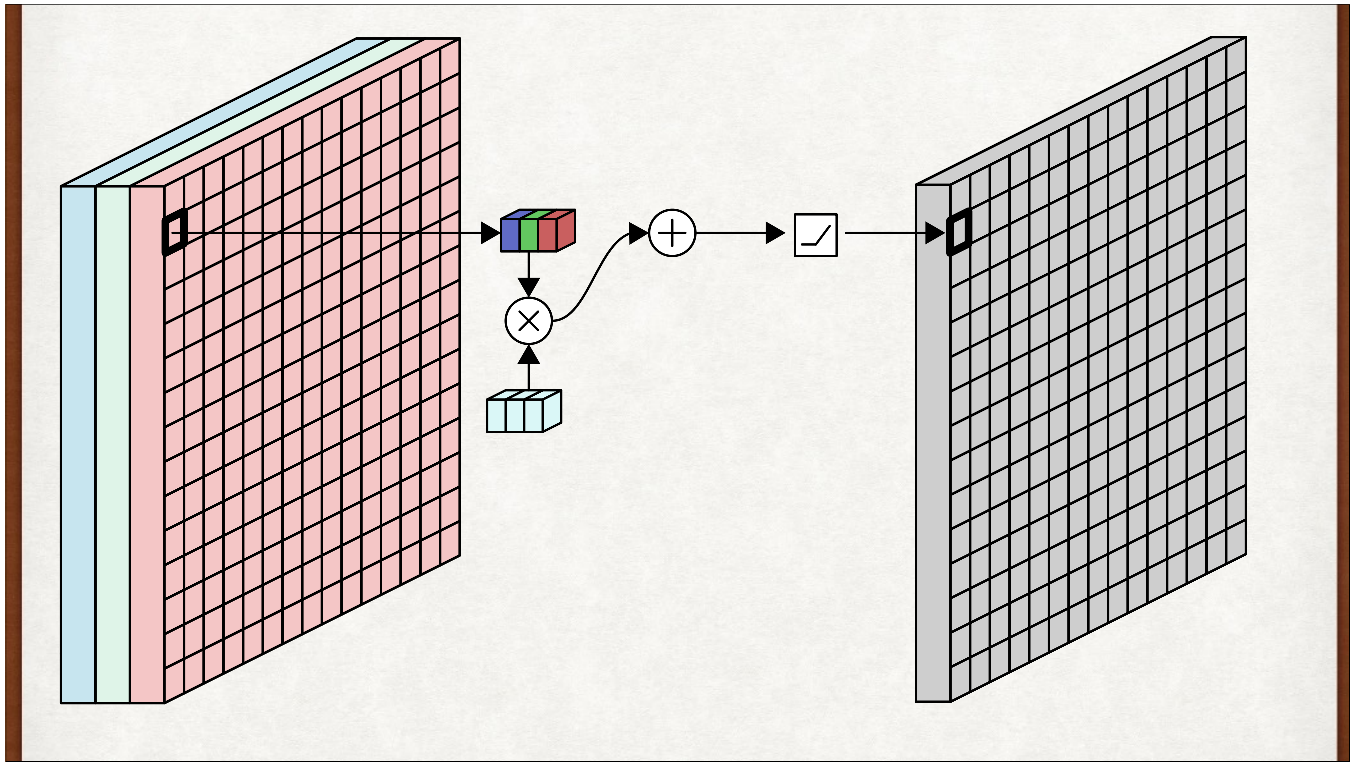
Convolving means sweeping our machinery over each pixel in the input.



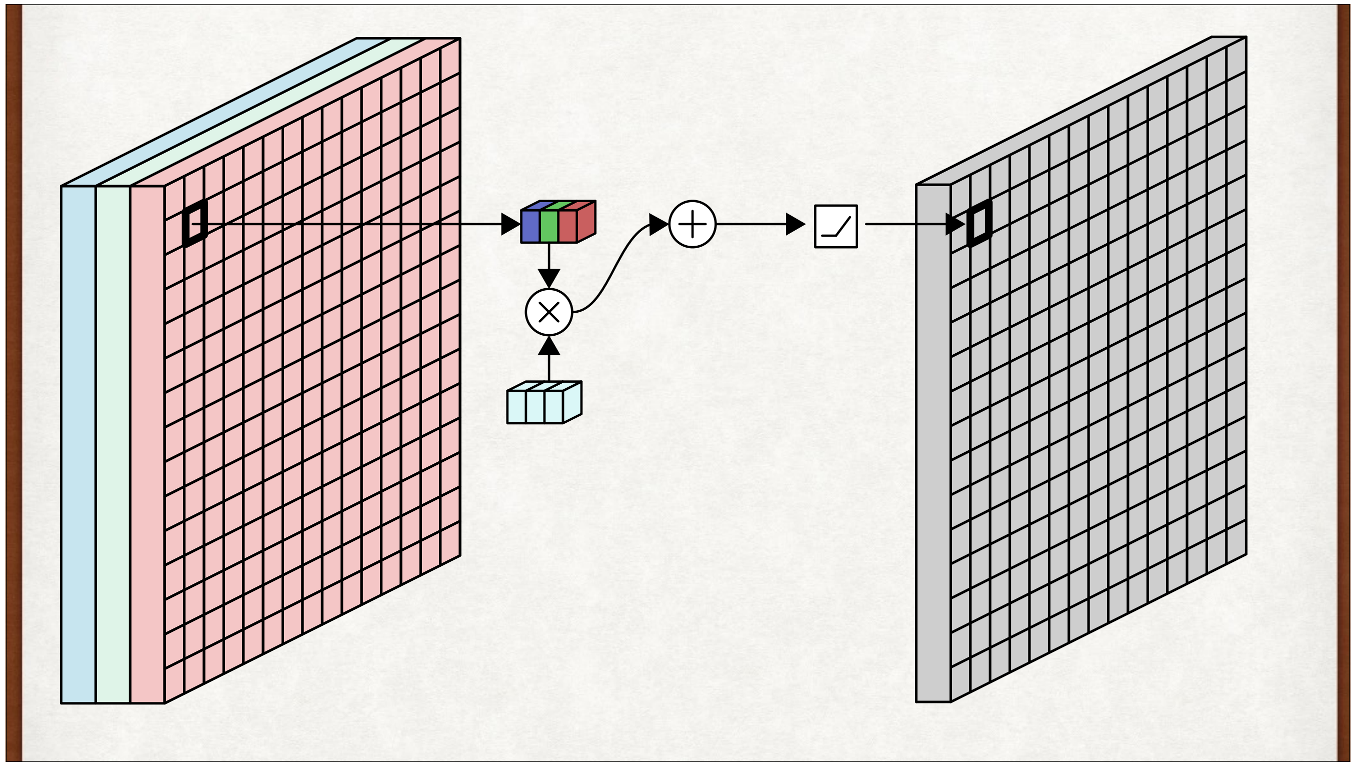
Convolving means sweeping our machinery over each pixel in the input.



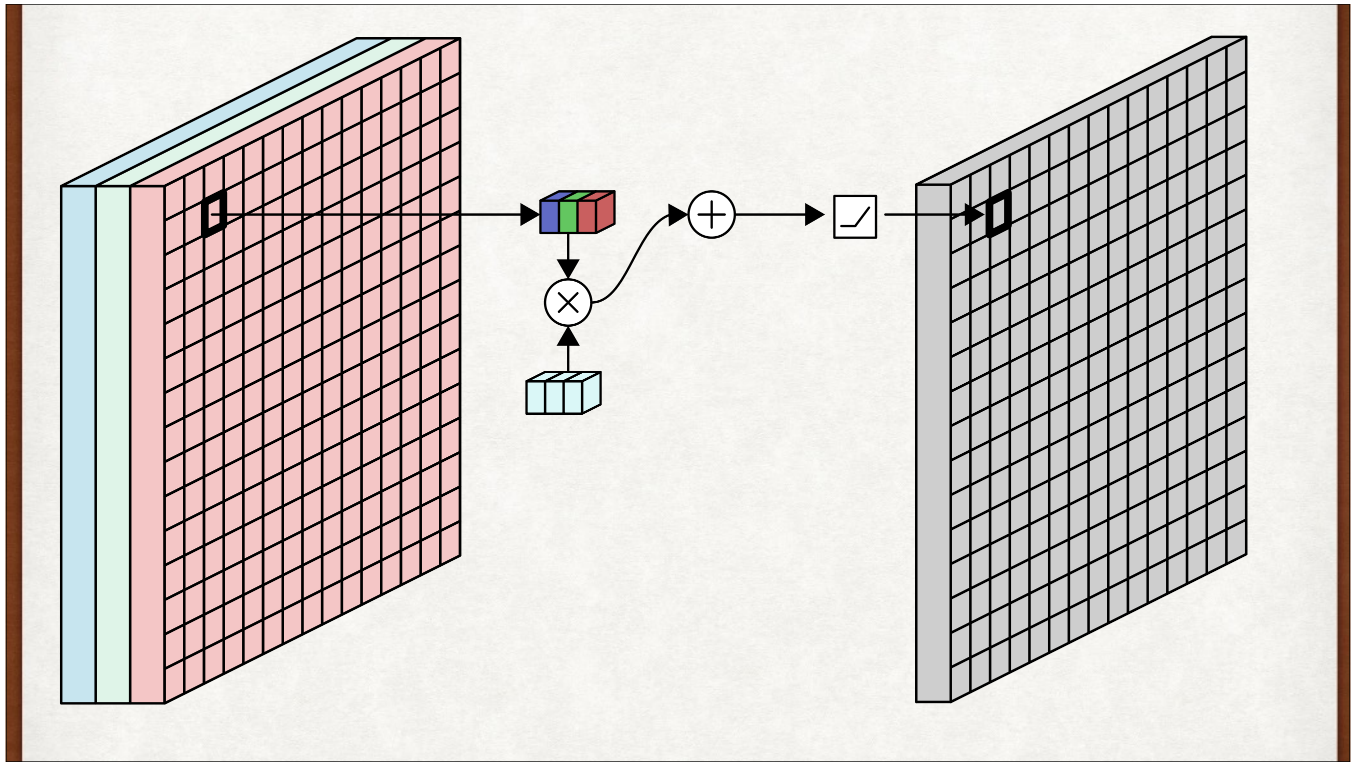
Convolving means sweeping our machinery over each pixel in the input.



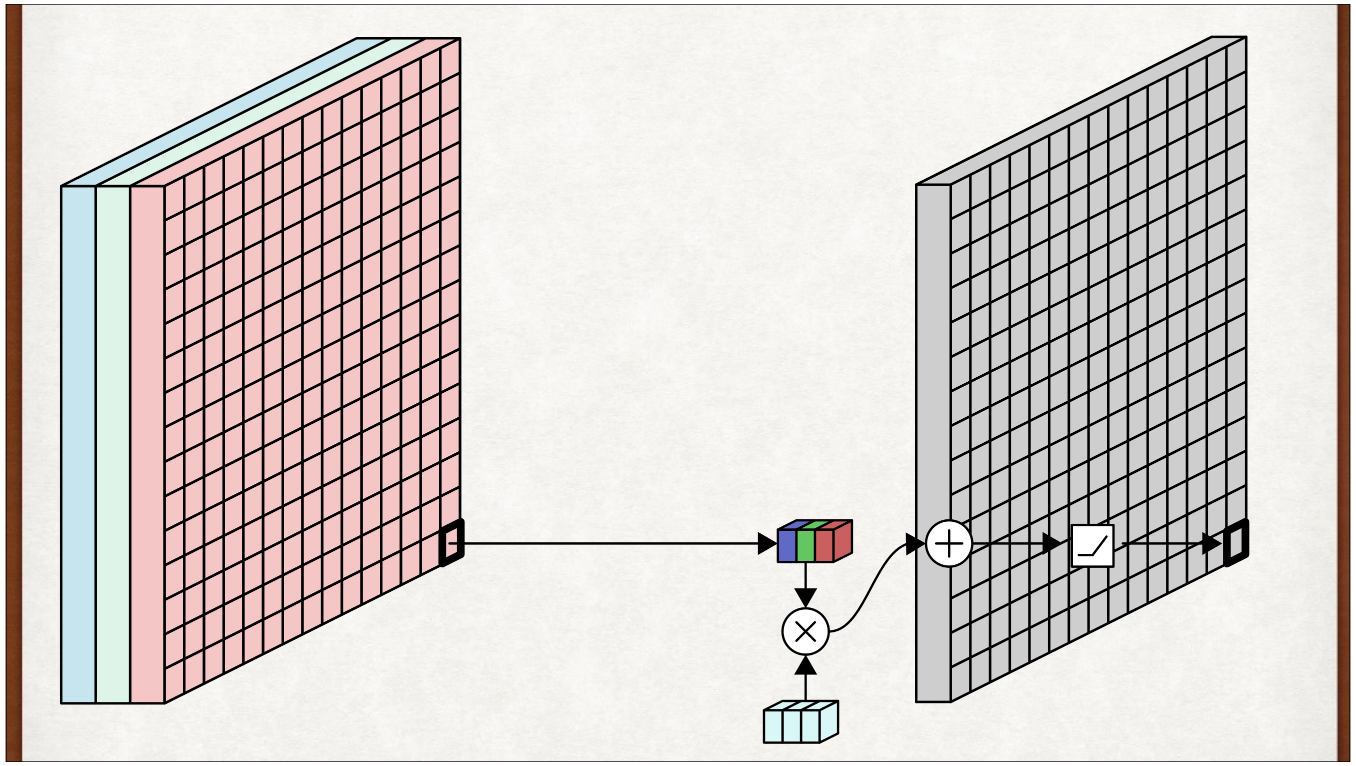
Convolving means sweeping our machinery over each pixel in the input.



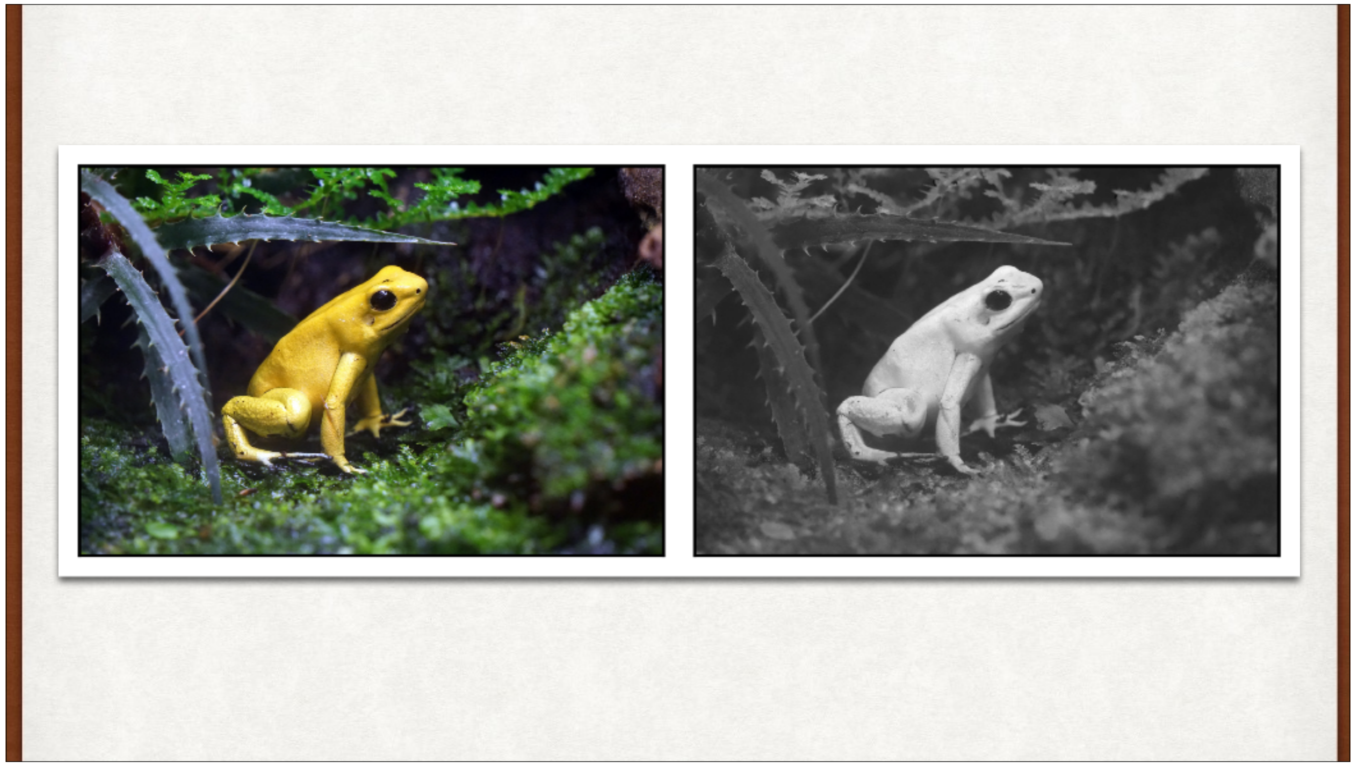
Convolving means sweeping our machinery over each pixel in the input.



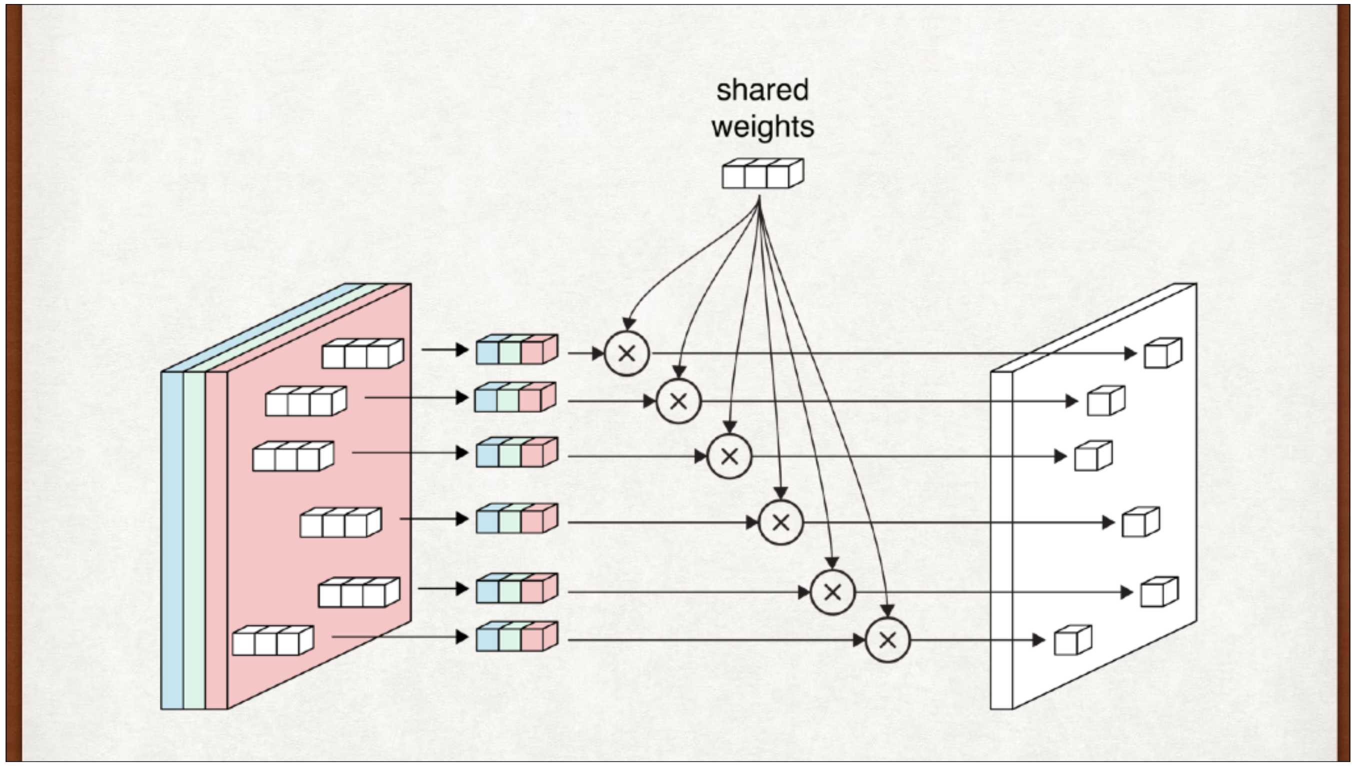
Convolving means sweeping our machinery over each pixel in the input.



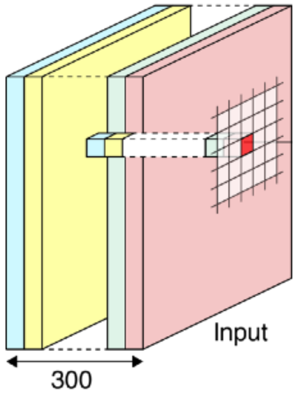
Convolving means sweeping our machinery over each pixel in the input.



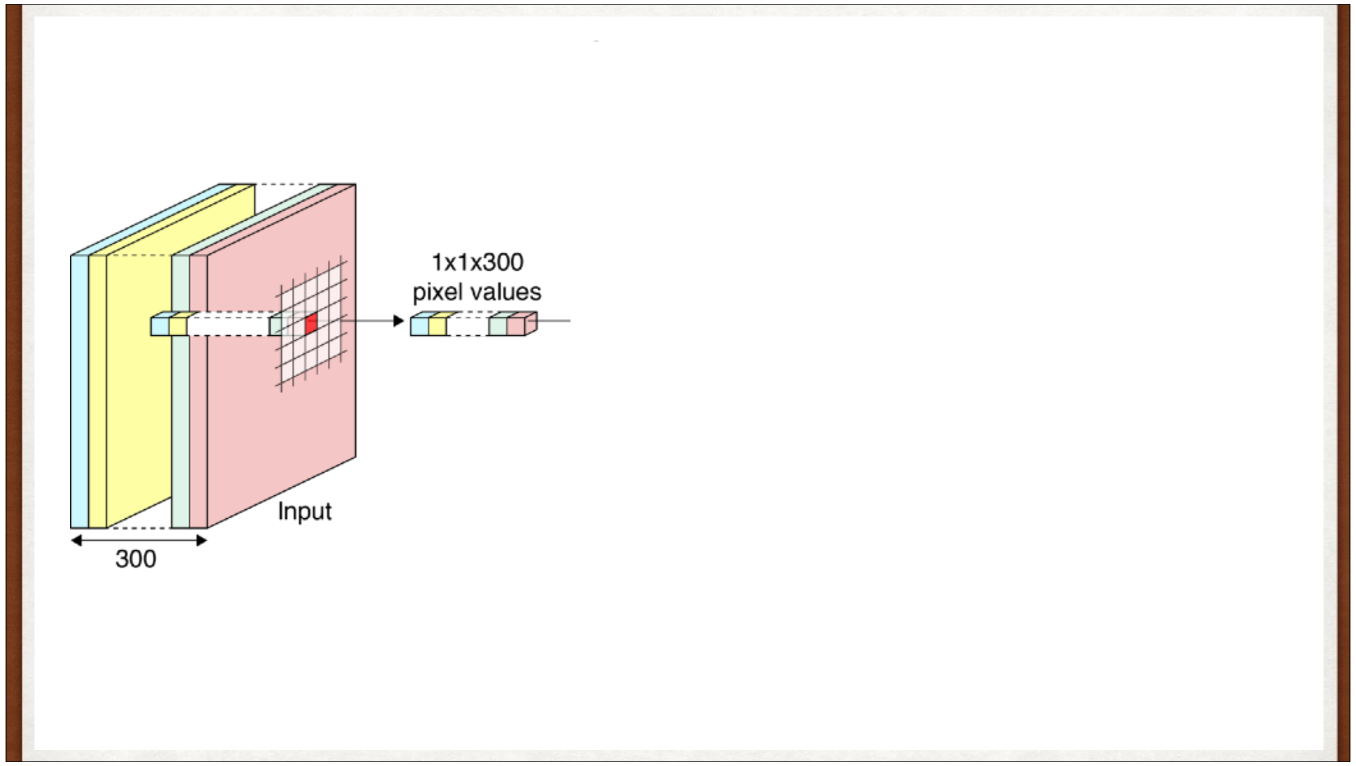
Our yellow filter applied to a frog, with the results scaled to 0=black, 1=white.



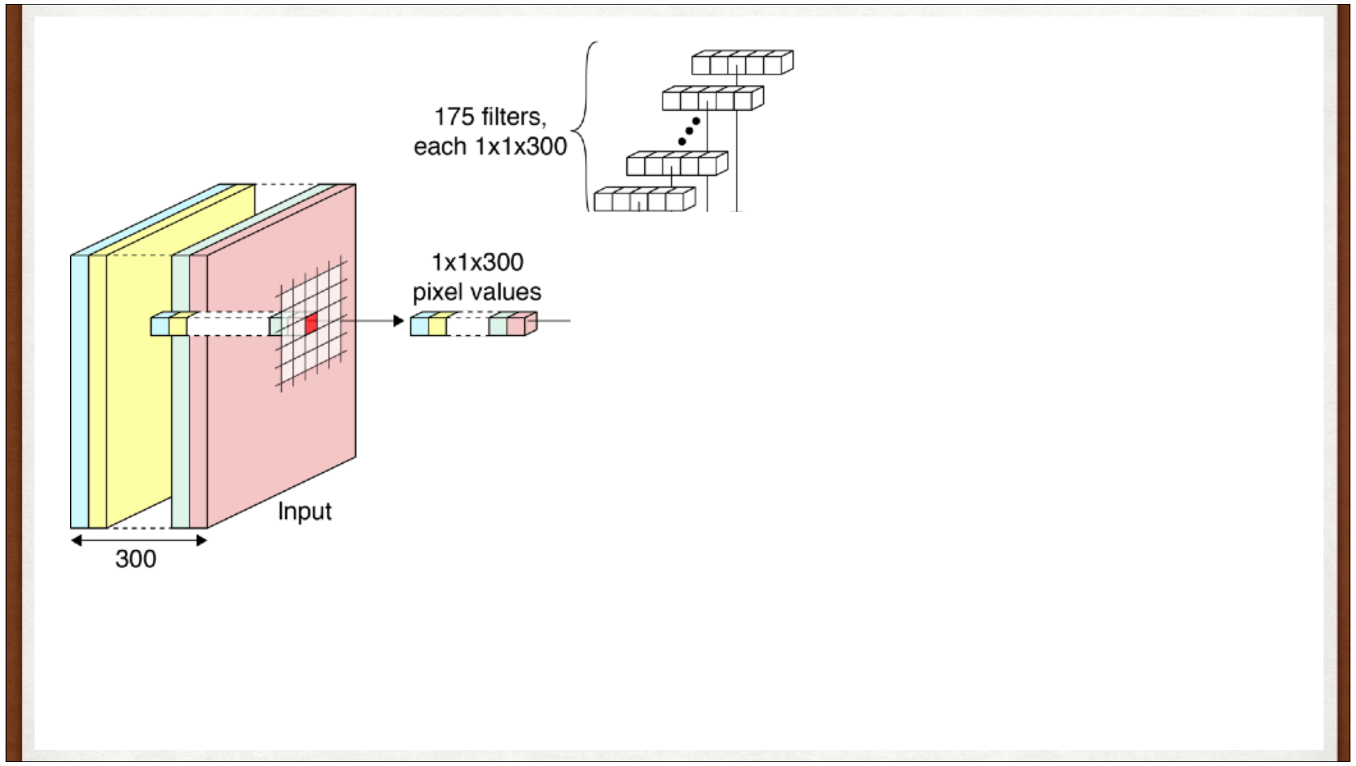
We apply the same weights to every pixel. Each of these “weight and add” convolution steps is using the same weights. We say they’re “sharing” those weights.



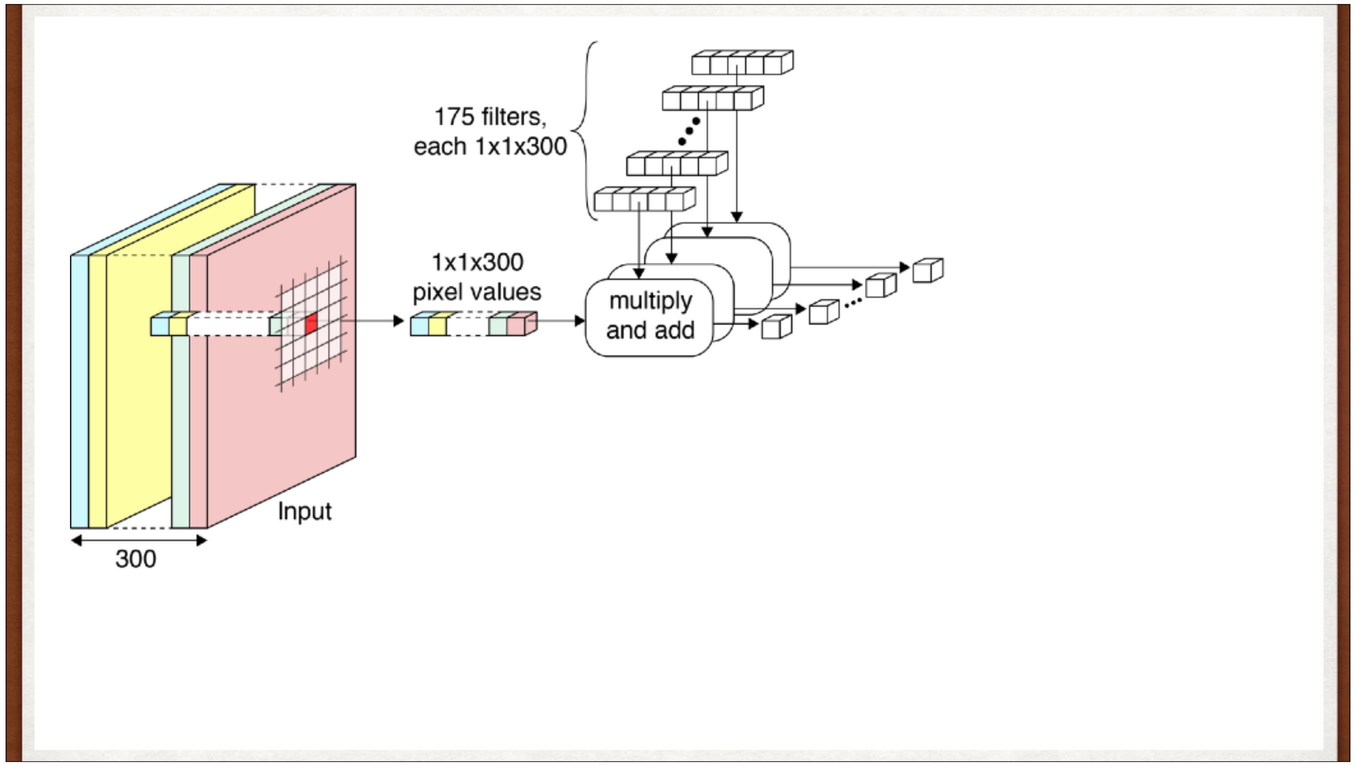
We’re not limited to RGB inputs with 3 layers. The input here has 300 layers, so each “core sample” is 1 by 1 by 300. So each filter also has 300 values. We multiply and add, or convolve, the input and filter to get a single number. Here we have 175 filters, so we’ll get back 175 numbers. Those get stored in an output that has the same height and width as the input, but 175 channels, one for each filter.



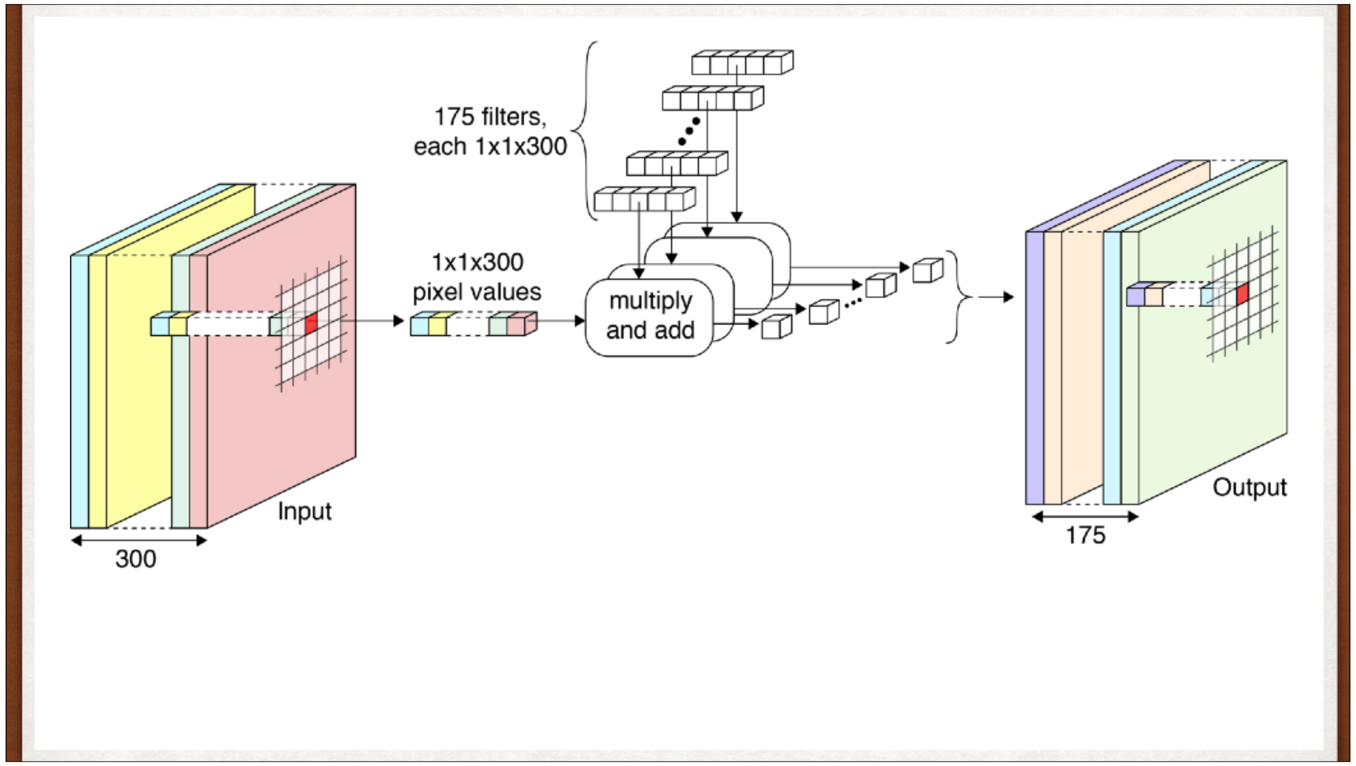
We’re not limited to RGB inputs with 3 layers. The input here has 300 layers, so each “core sample” is 1 by 1 by 300. So each filter also has 300 values. We multiply and add, or convolve, the input and filter to get a single number. Here we have 175 filters, so we’ll get back 175 numbers. Those get stored in an output that has the same height and width as the input, but 175 channels, one for each filter.



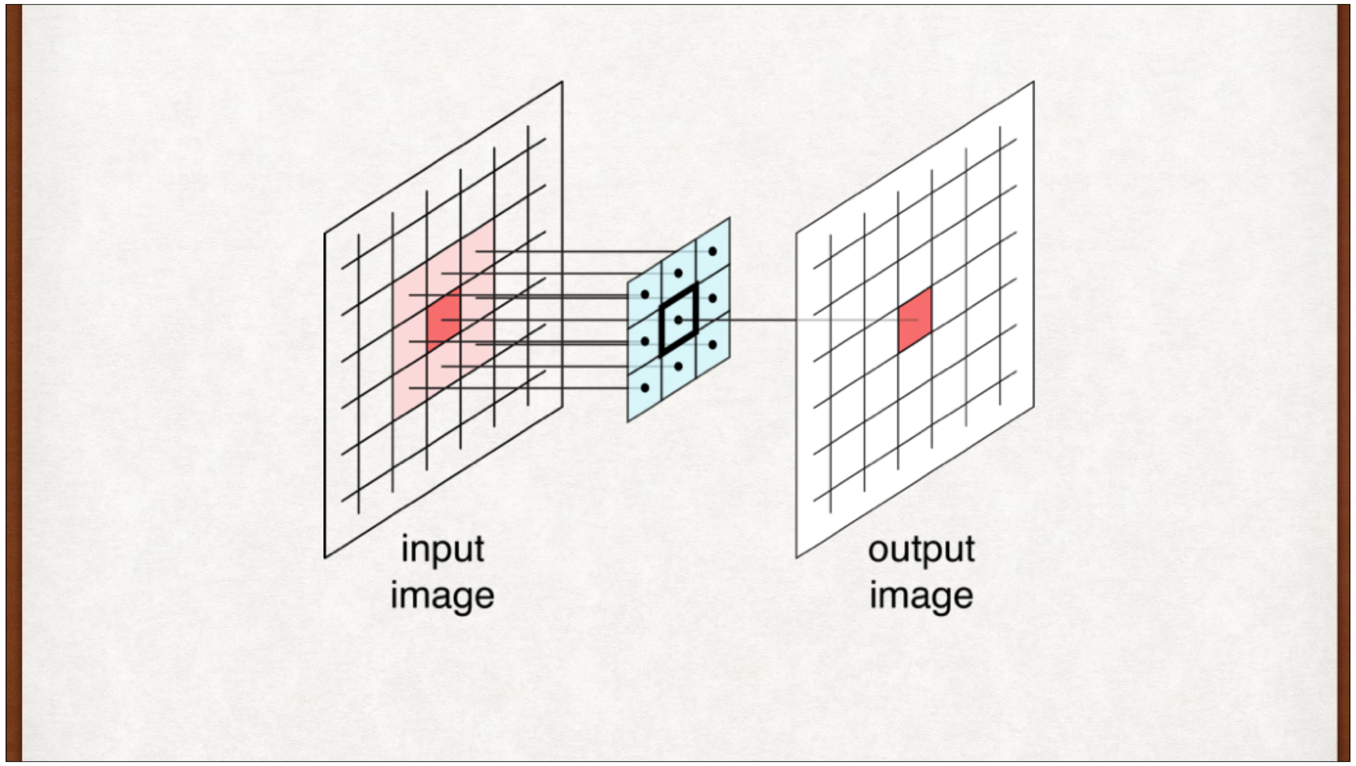
We’re not limited to RGB inputs with 3 layers. The input here has 300 layers, so each “core sample” is 1 by 1 by 300. So each filter also has 300 values. We multiply and add, or convolve, the input and filter to get a single number. Here we have 175 filters, so we’ll get back 175 numbers. Those get stored in an output that has the same height and width as the input, but 175 channels, one for each filter.



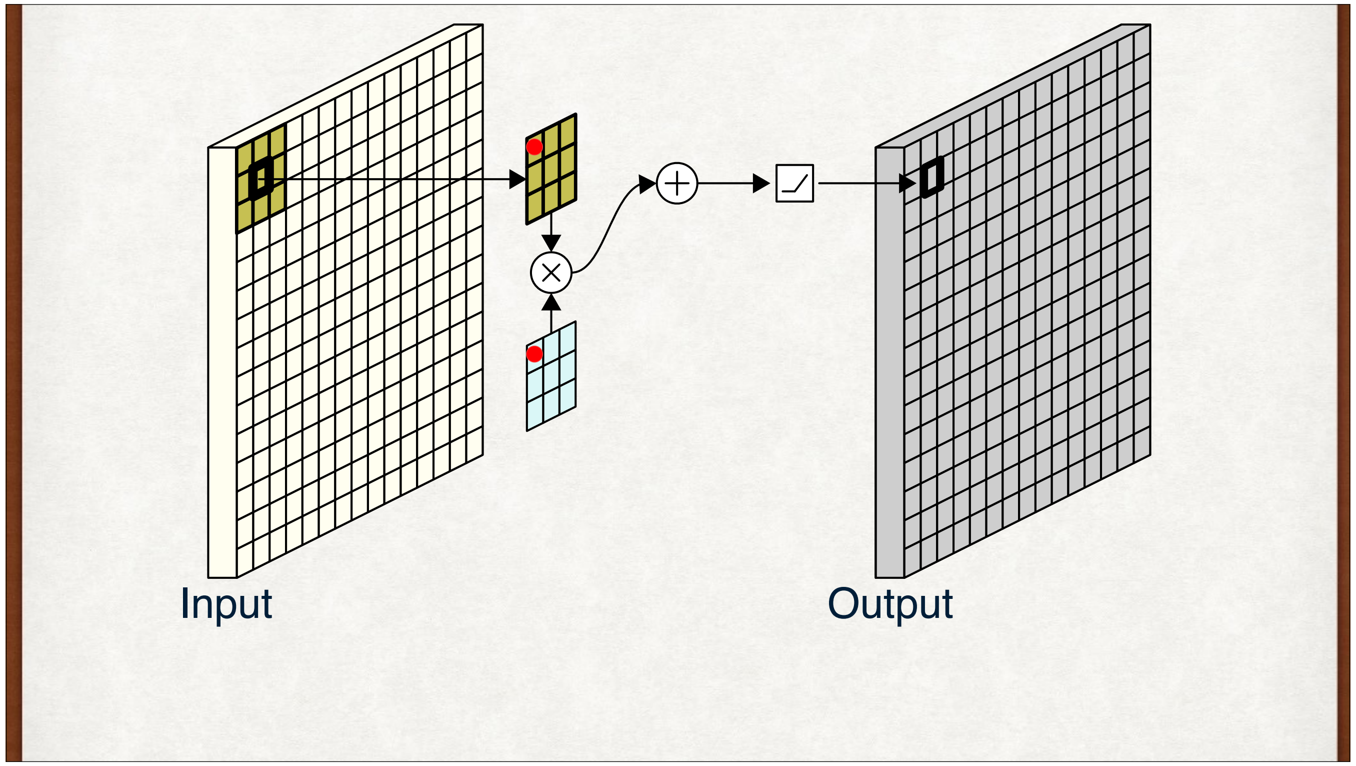
We’re not limited to RGB inputs with 3 layers. The input here has 300 layers, so each “core sample” is 1 by 1 by 300. So each filter also has 300 values. We multiply and add, or convolve, the input and filter to get a single number. Here we have 175 filters, so we’ll get back 175 numbers. Those get stored in an output that has the same height and width as the input, but 175 channels, one for each filter.



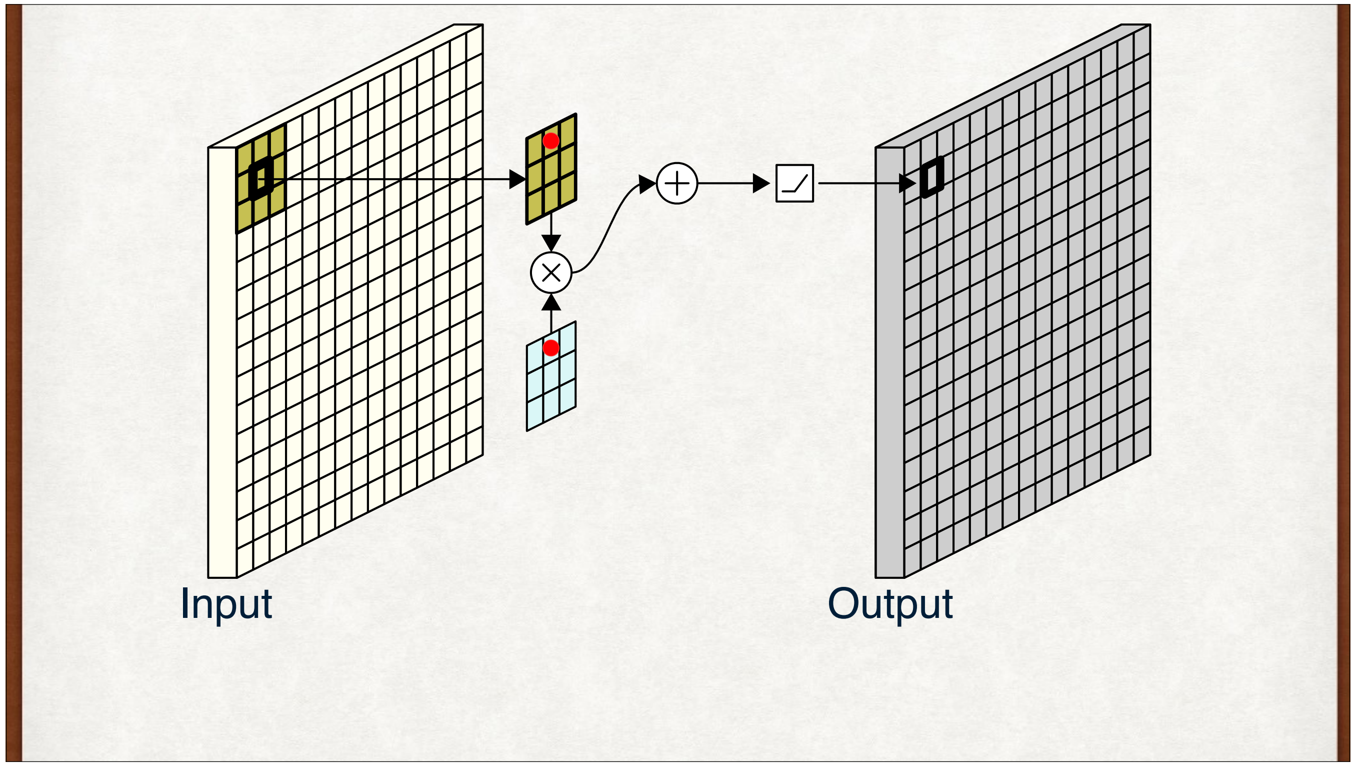
We’re not limited to RGB inputs with 3 layers. The input here has 300 layers, so each “core sample” is 1 by 1 by 300. So each filter also has 300 values. We multiply and add, or convolve, the input and filter to get a single number. Here we have 175 filters, so we’ll get back 175 numbers. Those get stored in an output that has the same height and width as the input, but 175 channels, one for each filter.



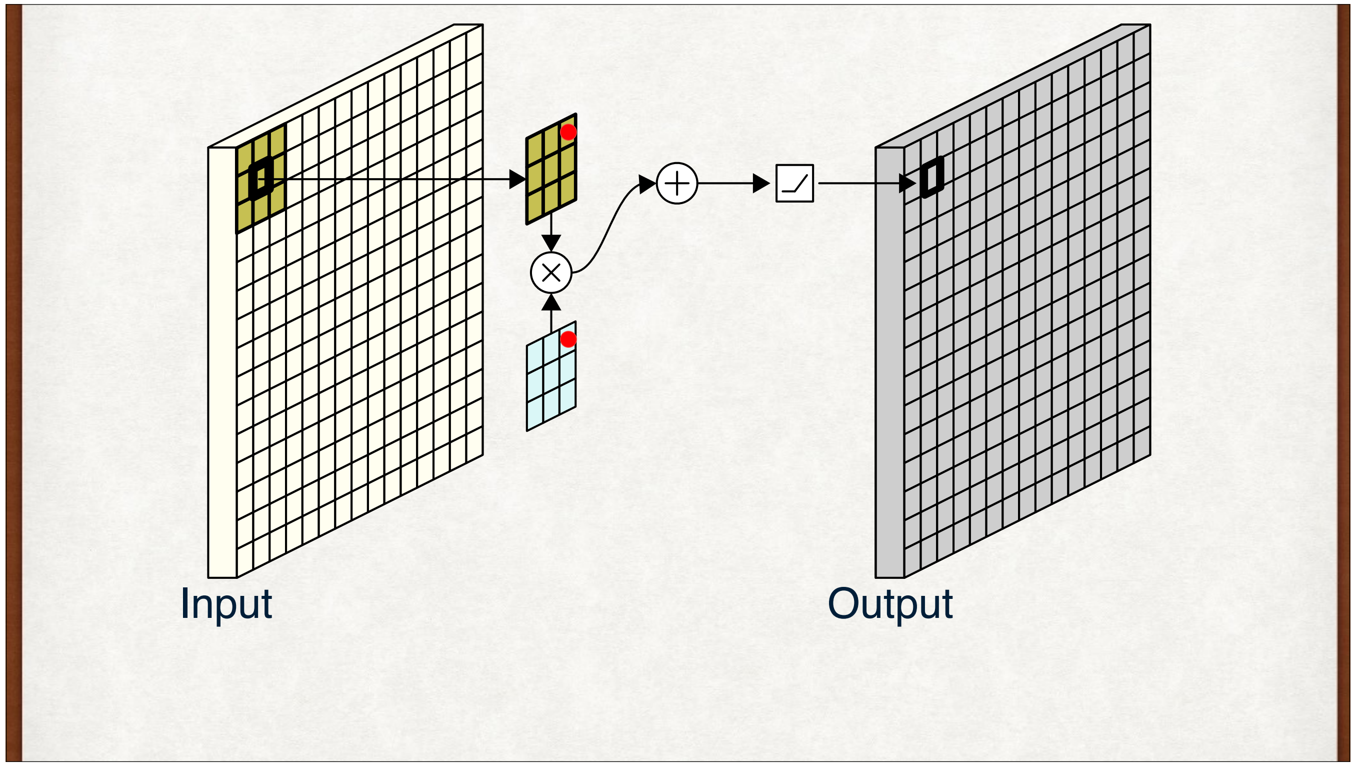
We can use spatial filters, too. Each filter has a “footprint,” or the region it covers in its input. The cyan filter’s footprint is 3 by 3.



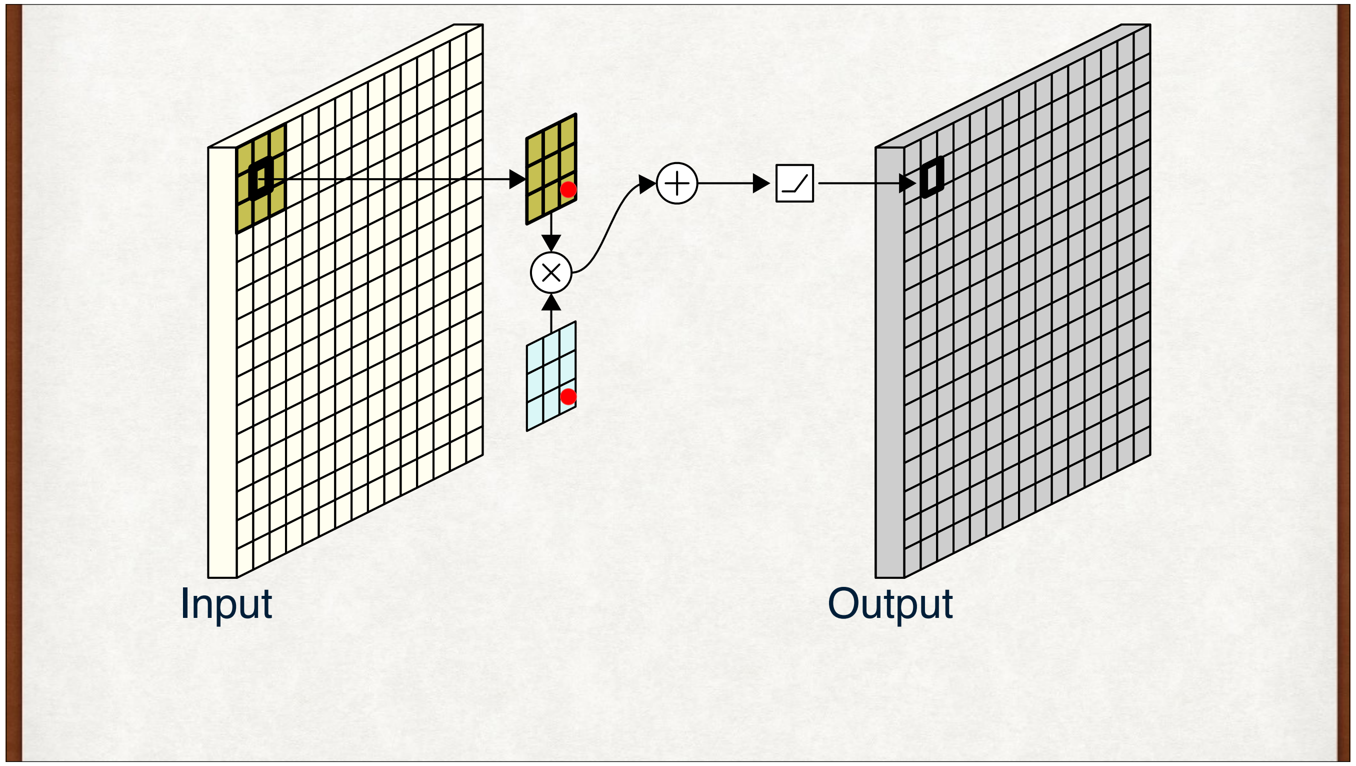
The process is just like before. Our input is grayscale, so the “core sample” has just 9 values. Thus our filter has 9 values. We multiply together the corresponding values, here shown in red…



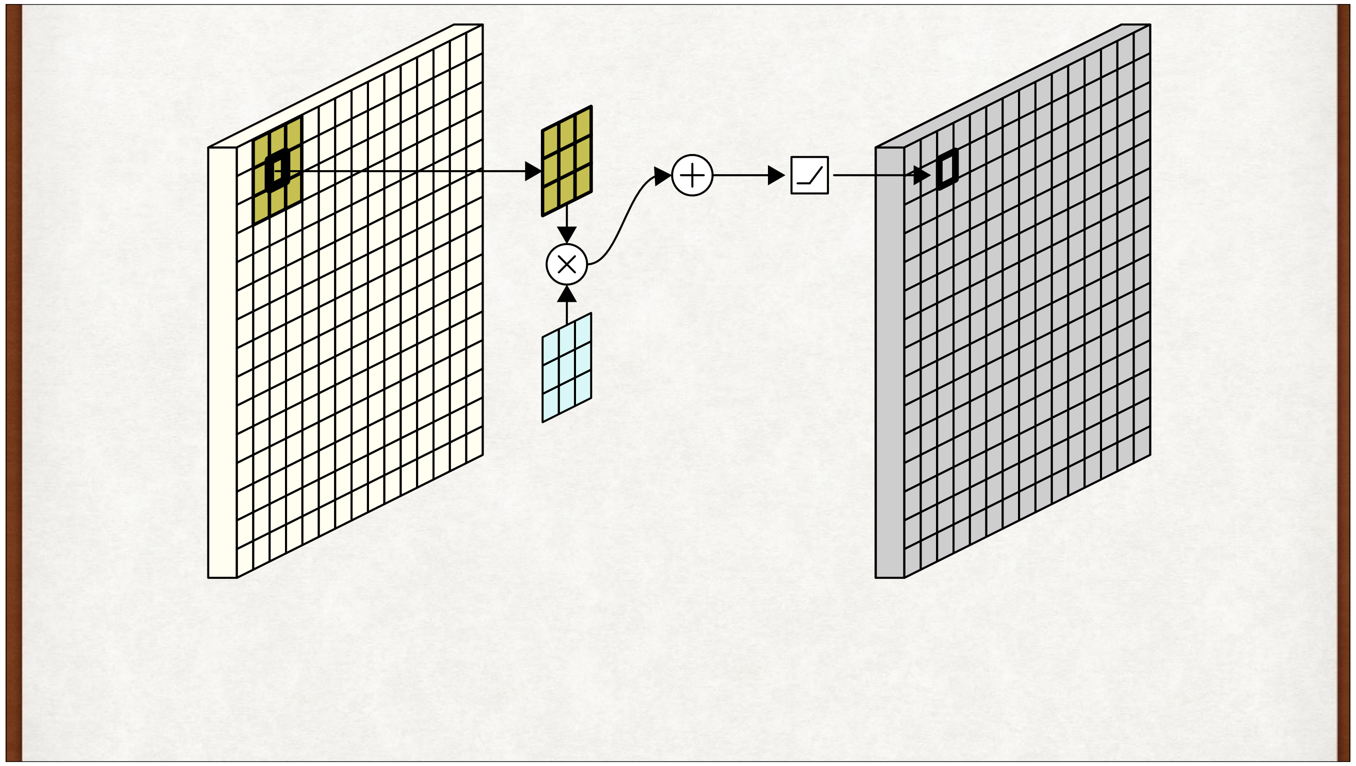
The process is just like before. Our input is grayscale, so the “core sample” has just 9 values. Thus our filter has 9 values. We multiply together the corresponding values, here shown in red…



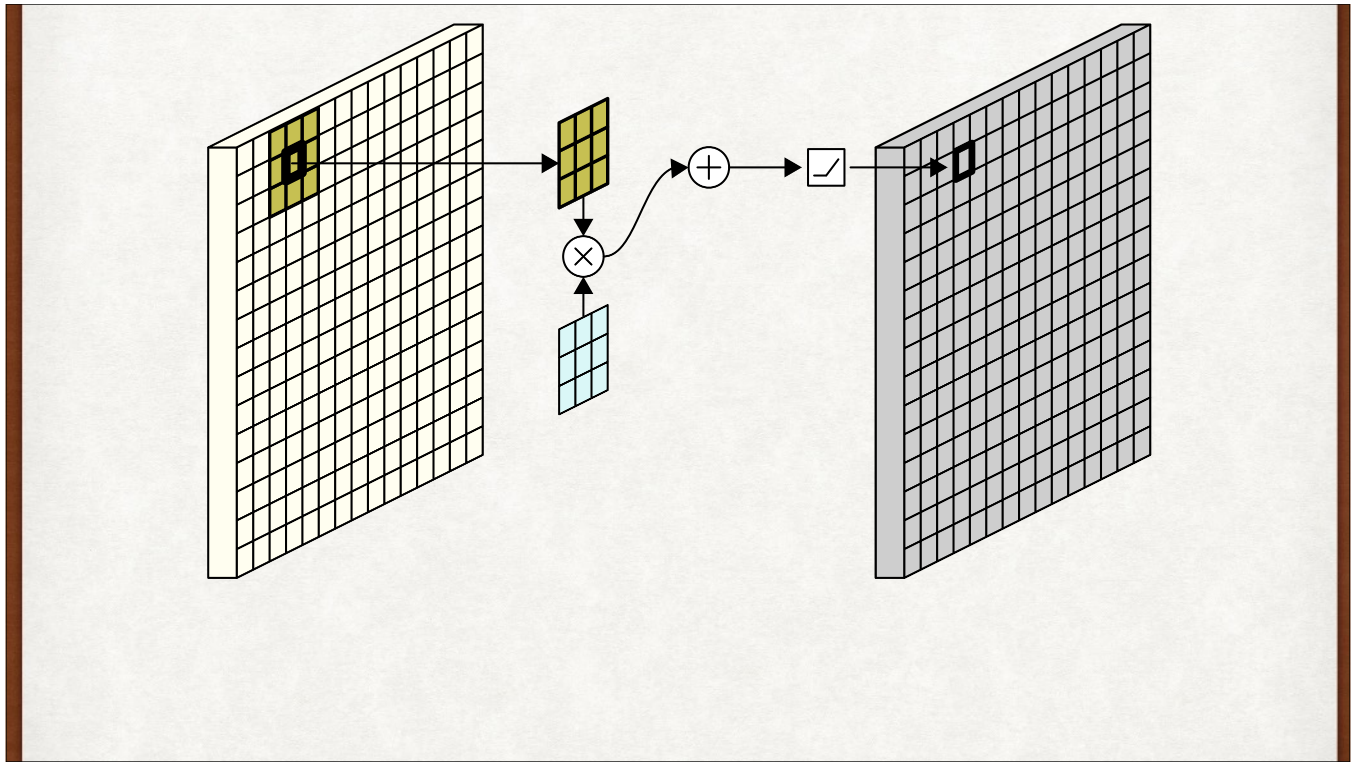
The process is just like before. Our input is grayscale, so the “core sample” has just 9 values. Thus our filter has 9 values. We multiply together the corresponding values, here shown in red…



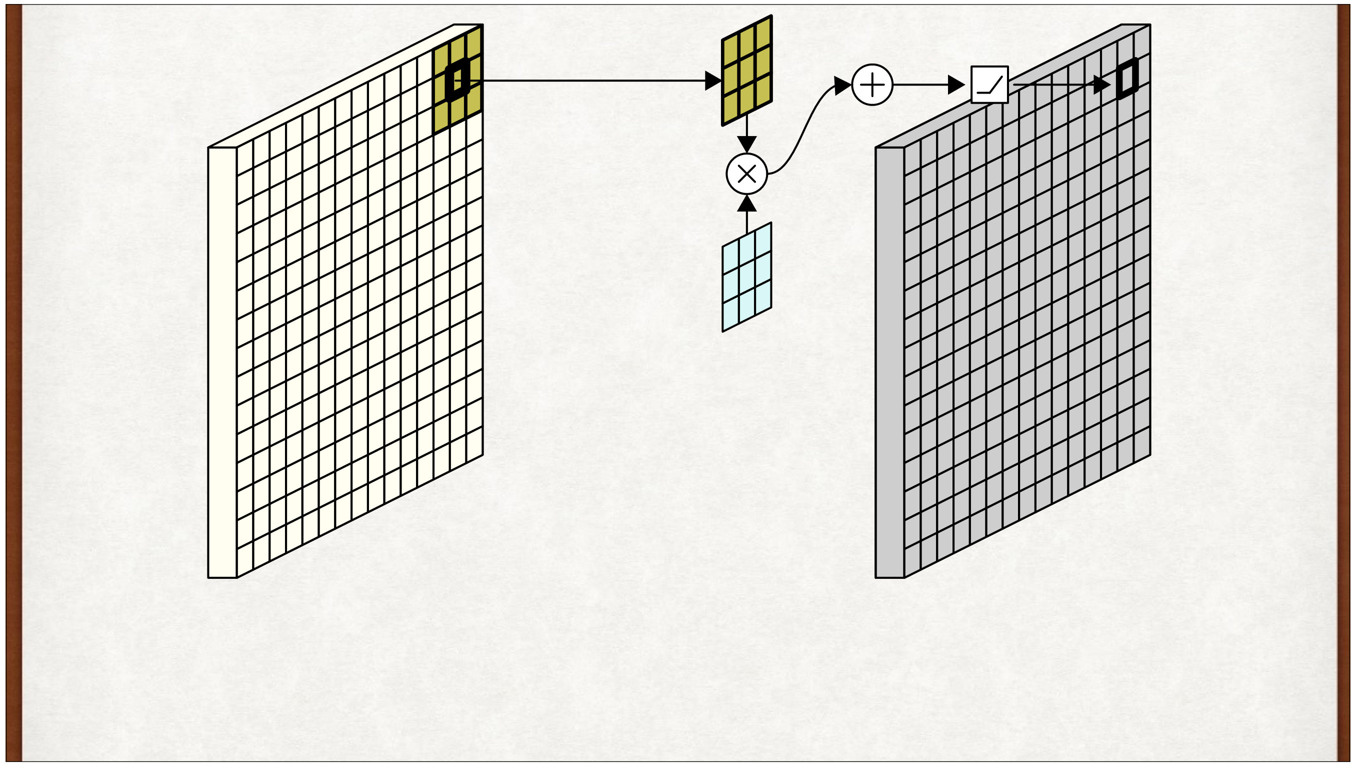
The process is just like before. Our input is grayscale, so the “core sample” has just 9 values. Thus our filter has 9 values. We multiply together the corresponding values, here shown in red. They’re added together, go through the activation function, and we have an output value.



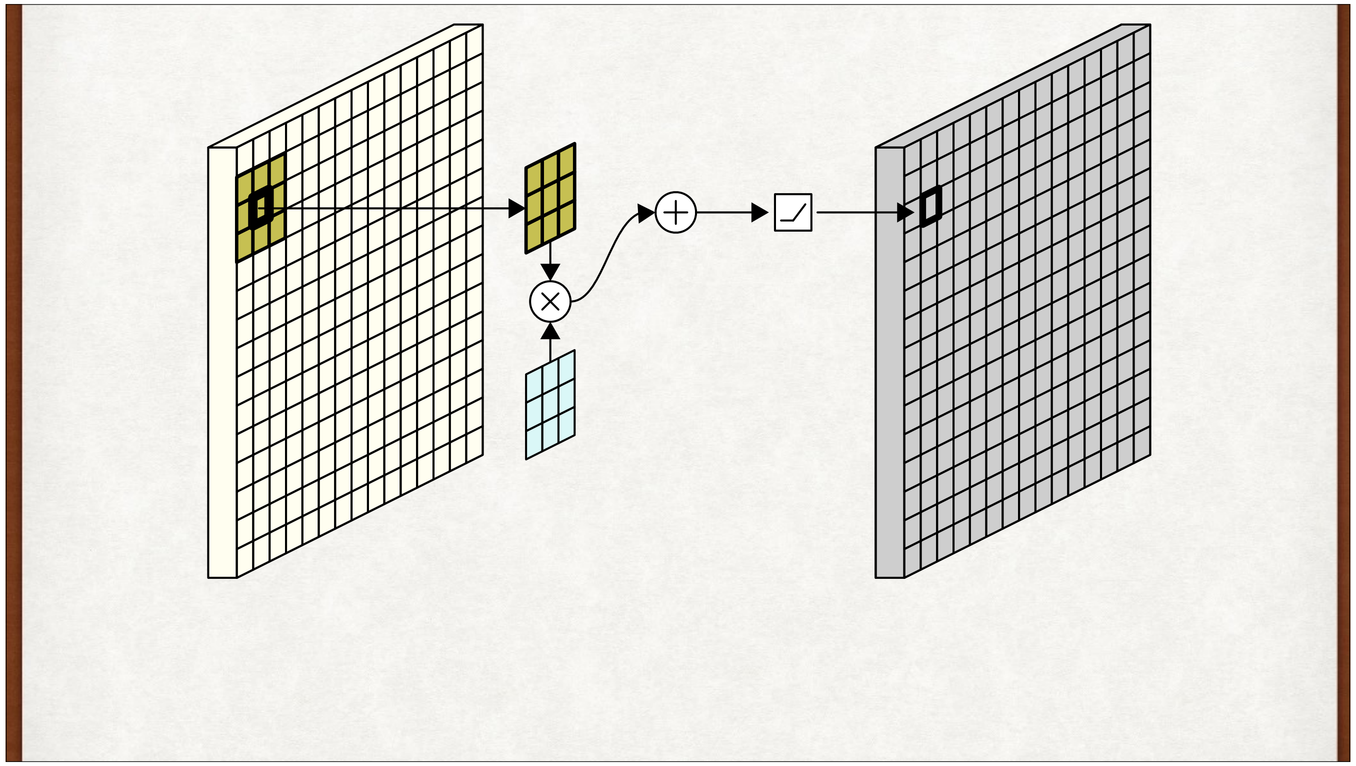
We convolve this 2D filter like before, sweeping it over the input.



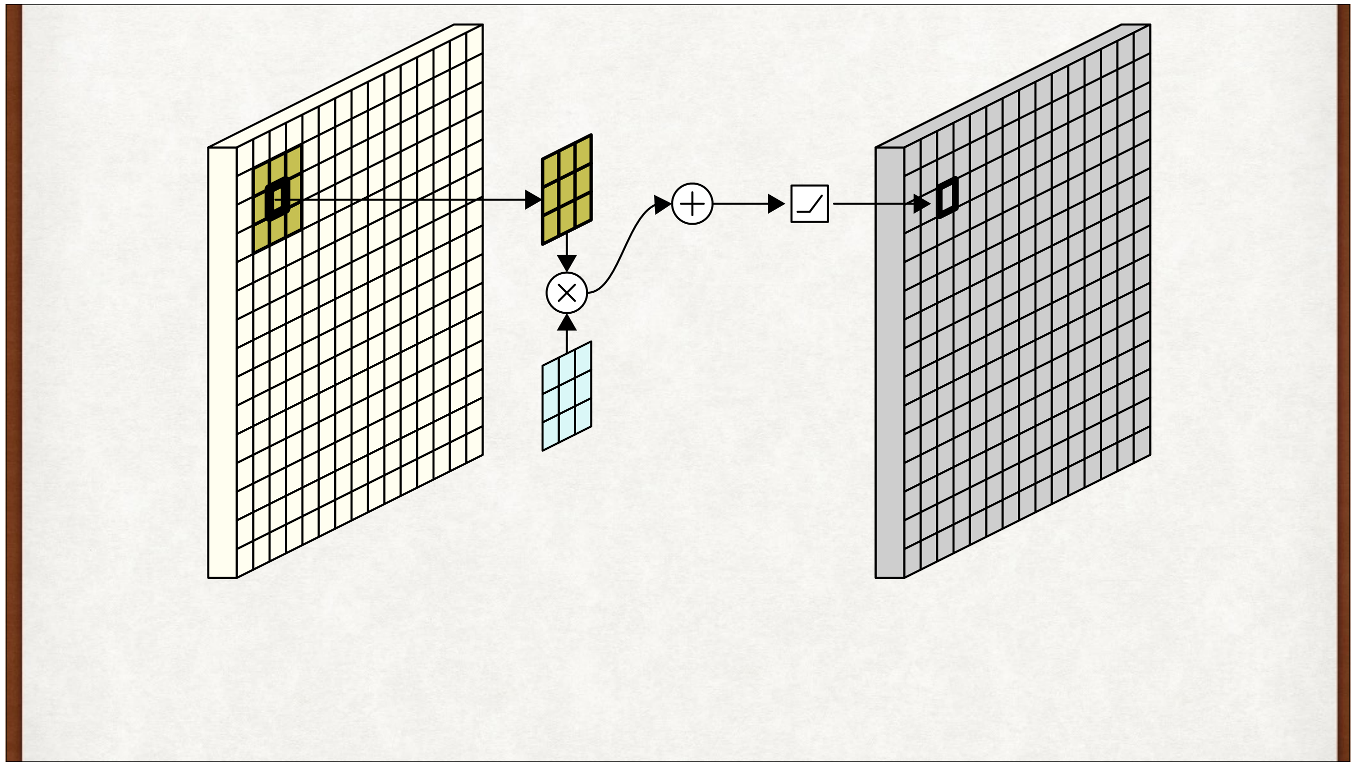
We convolve this 2D filter like before, sweeping it over the input.



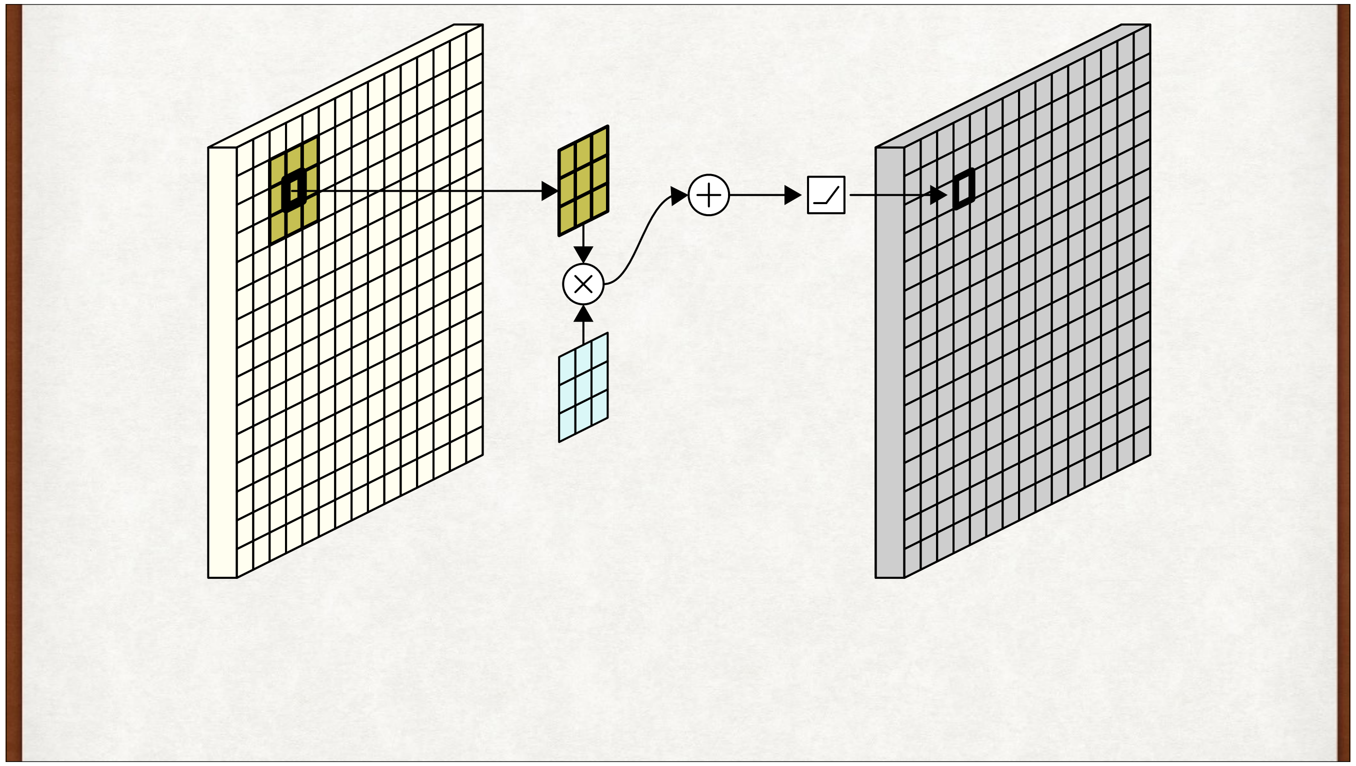
We convolve this 2D filter like before, sweeping it over the input.



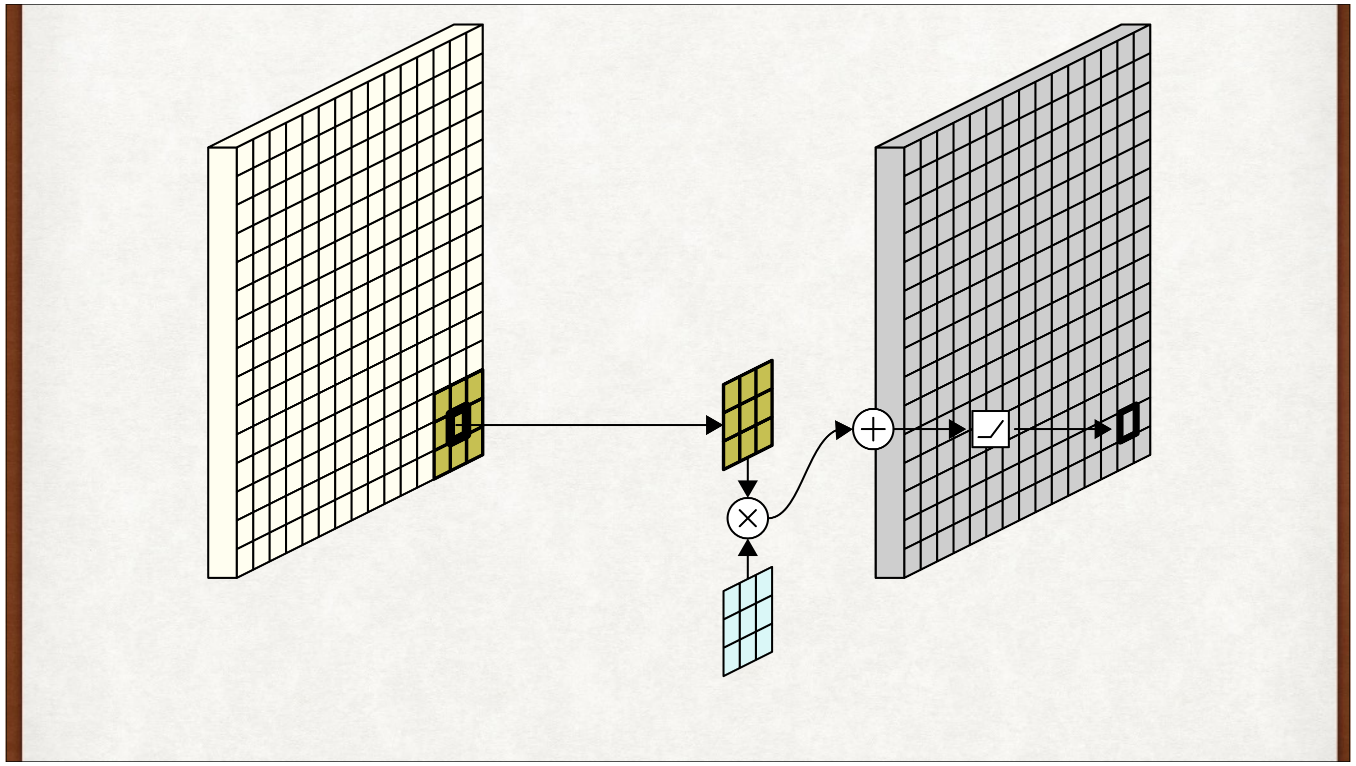
We convolve this 2D filter like before, sweeping it over the input.



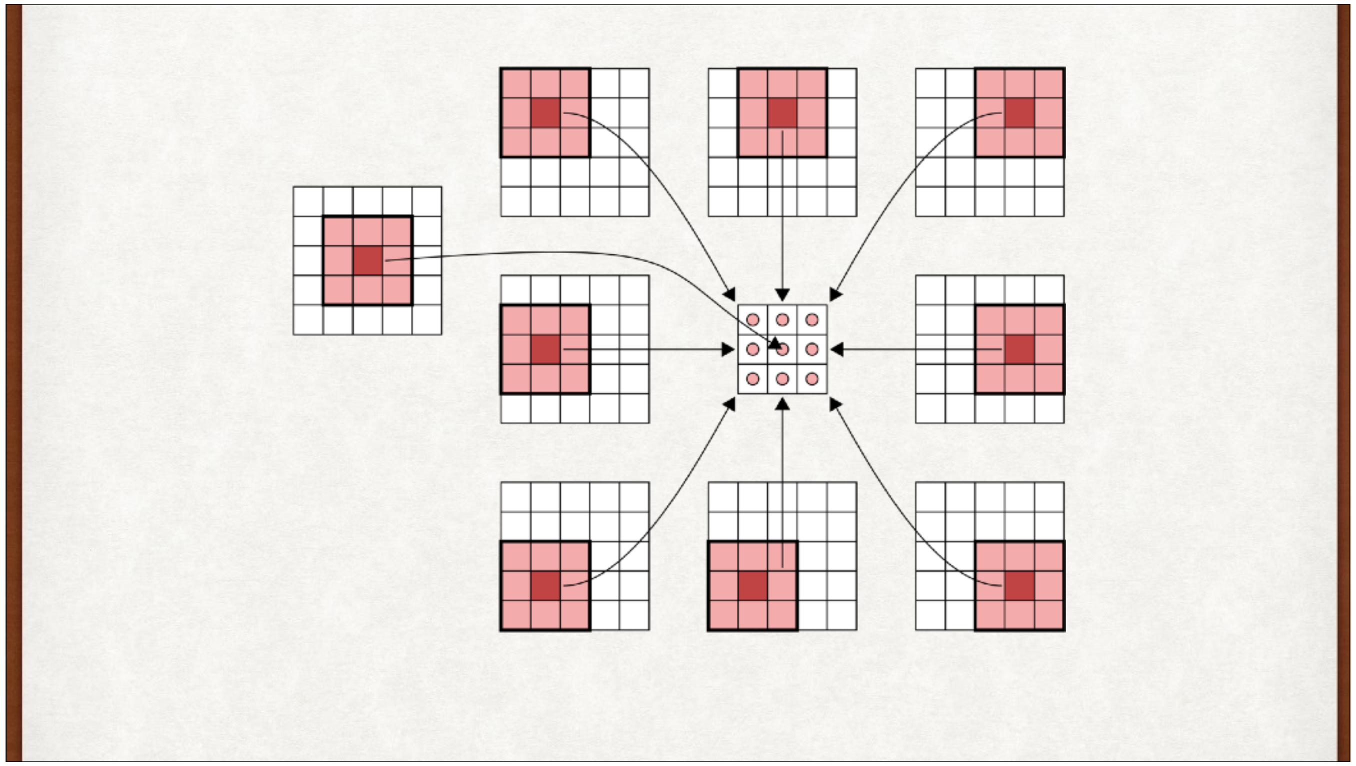
We convolve this 2D filter like before, sweeping it over the input.



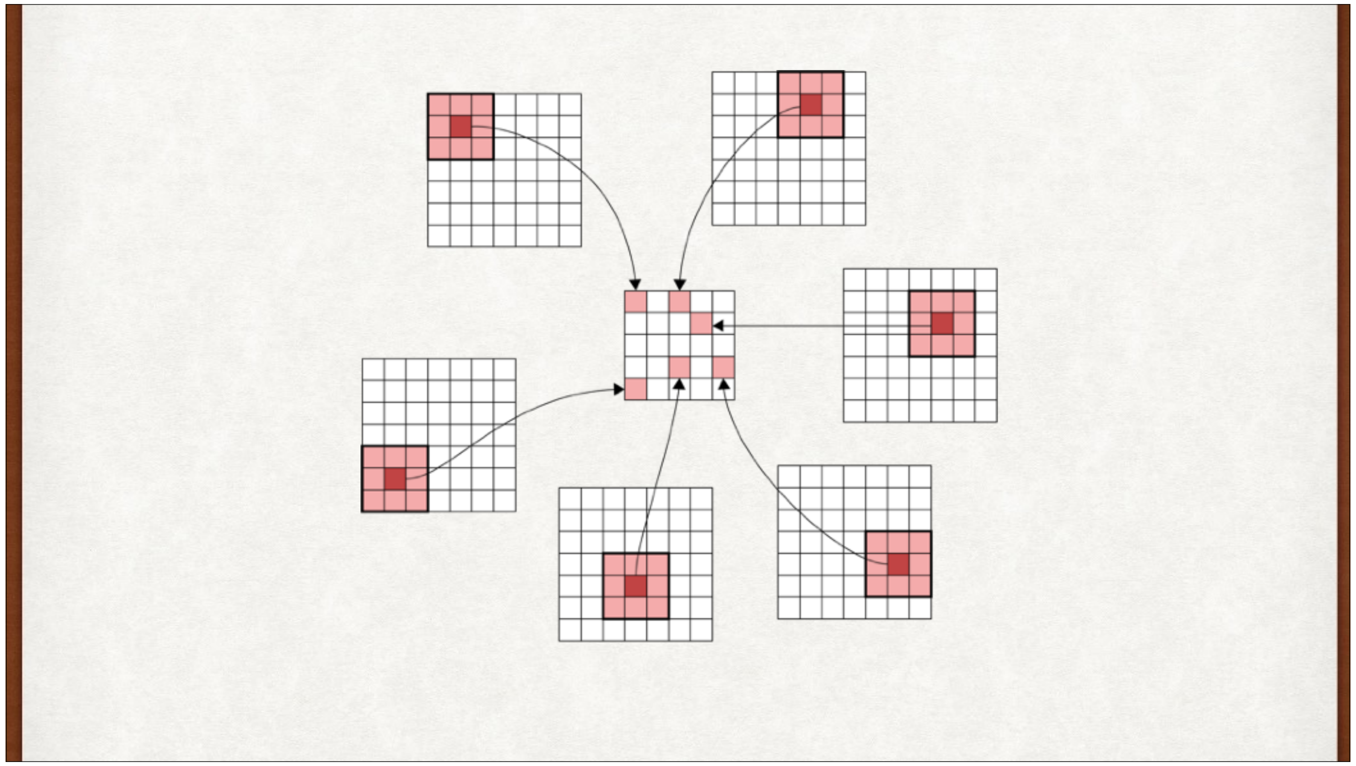
We convolve this 2D filter like before, sweeping it over the input.



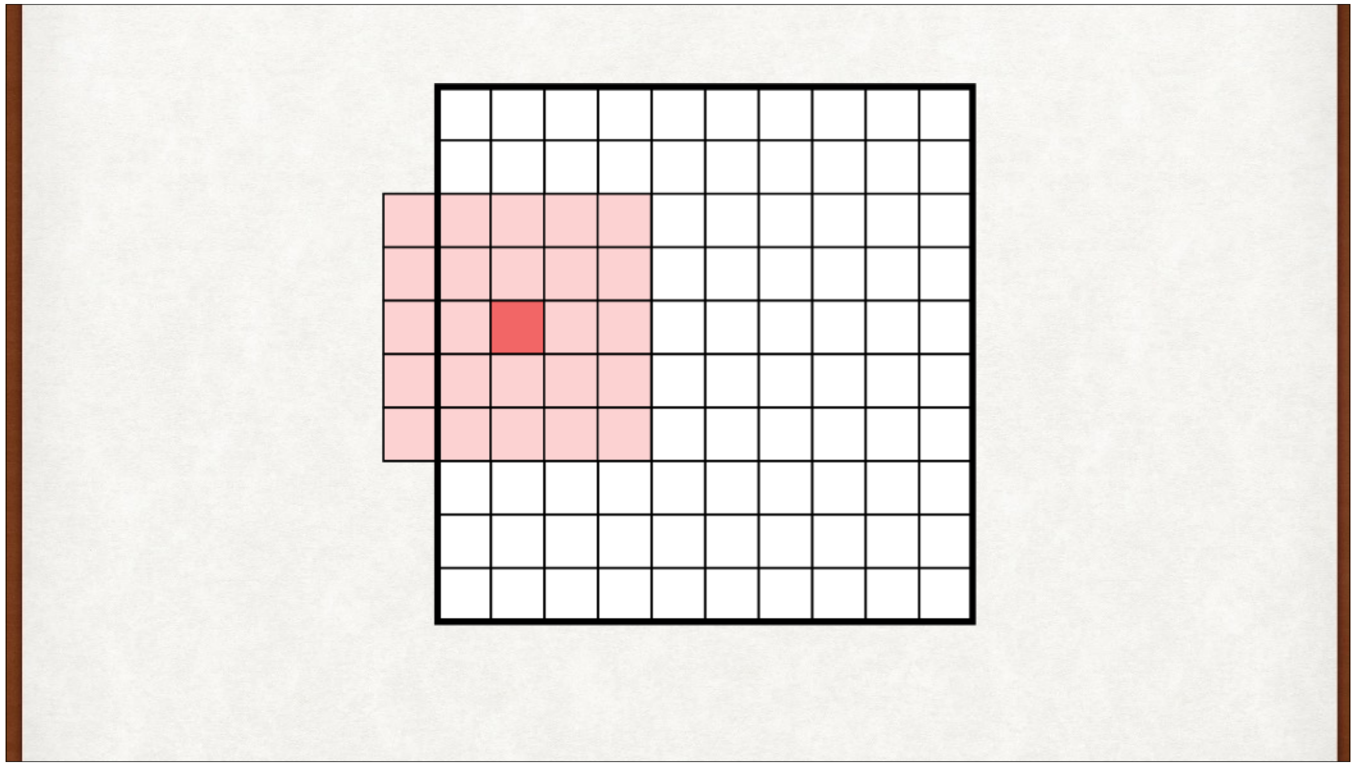
We convolve this 2D filter like before, sweeping it over the input.



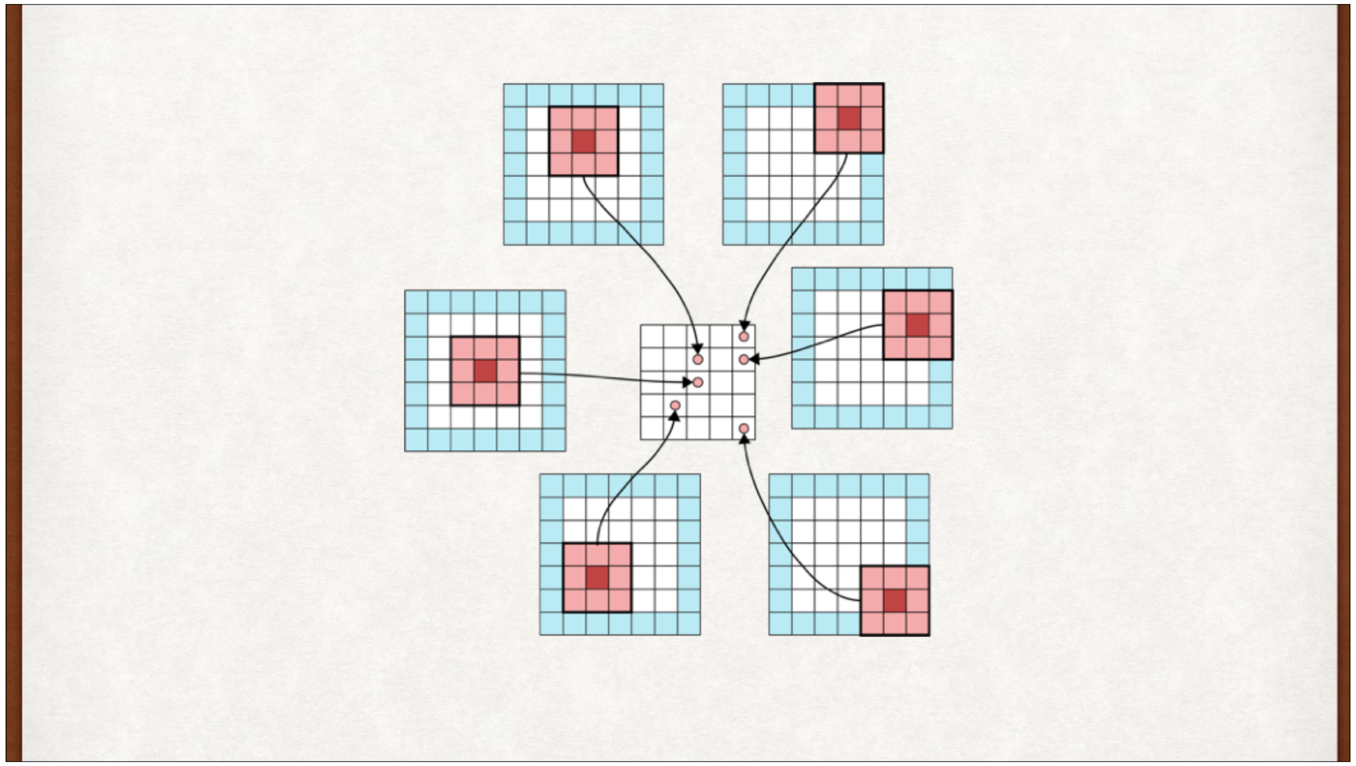
A 5 by 5 input and a 3 by 3 filter. There are only 9 places for the filter to go without falling off the edge, producing a 3 by 3 output.



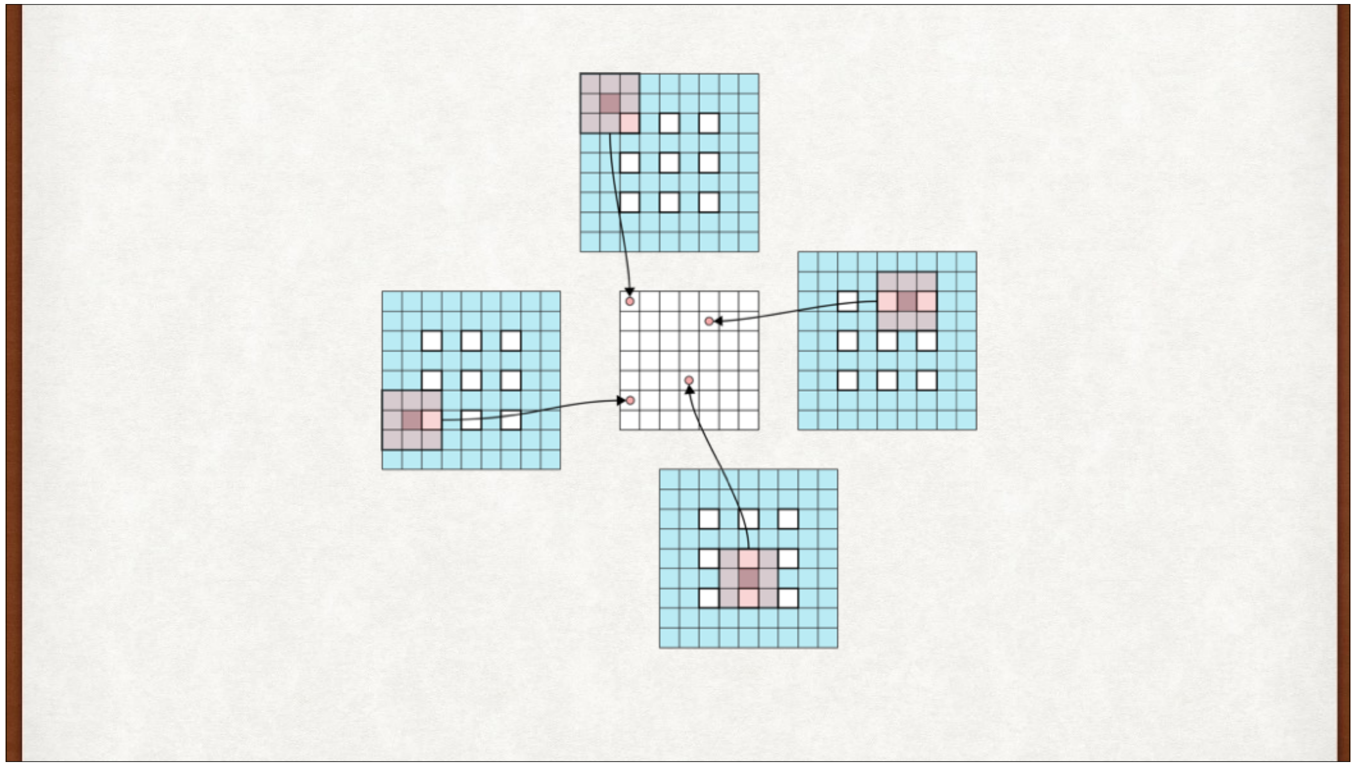
A 7 by 7 input and a 3 by 3 filter. This time there are 25 places where the filter can completely fit, and our output is 5 by 5. It would be nice to make the output have the same size as the input, so the original data doesn’t shrink on every step.



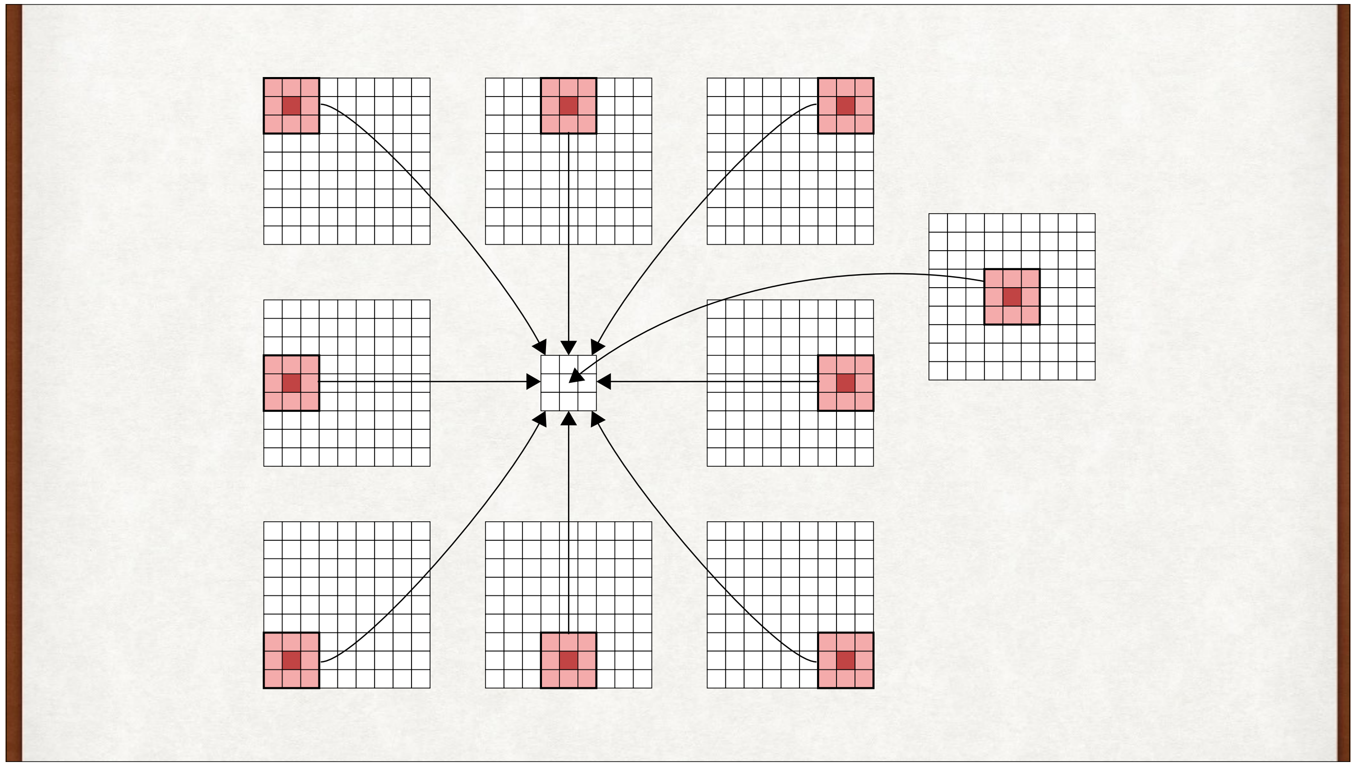
This is why we’ve been shrinking: we haven’t let the filter fall off the edge. Is there a way to make this a valid convolution?



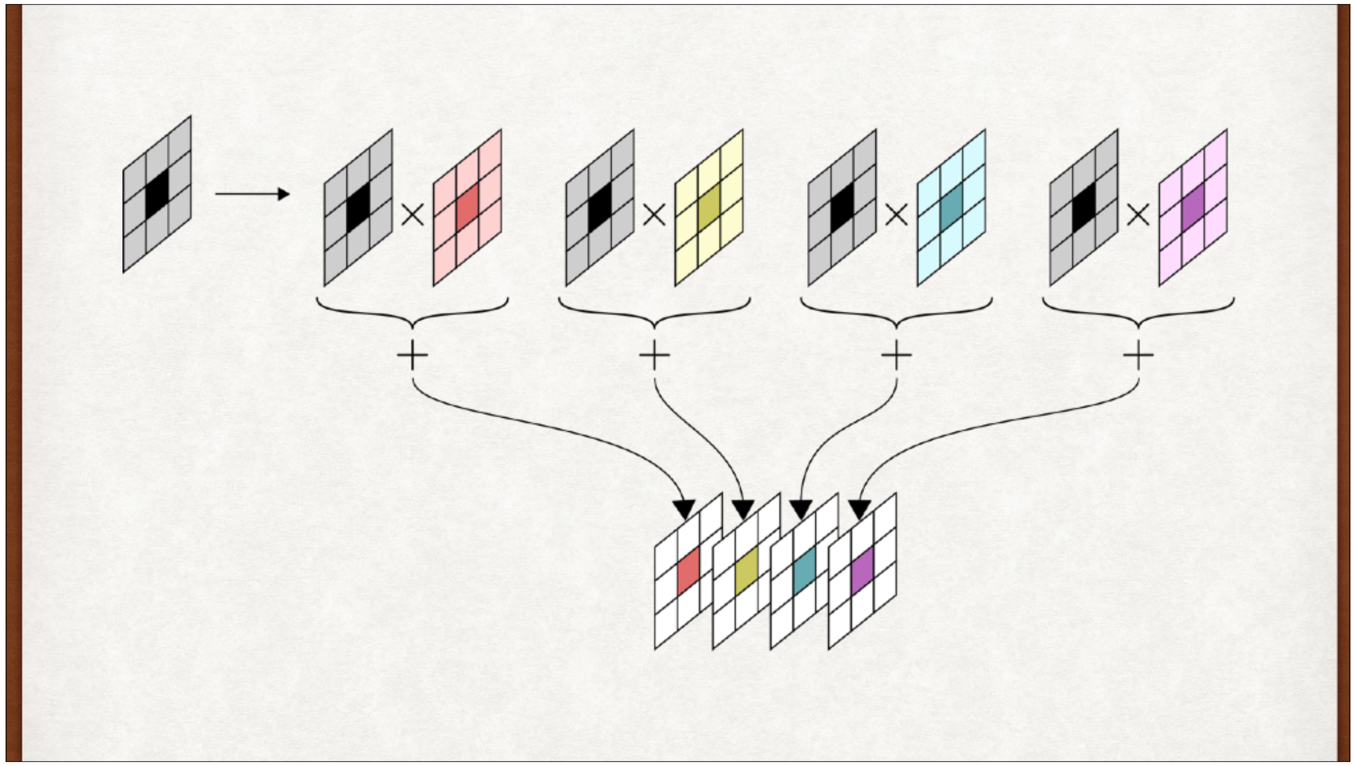
Let’s surround the input with one or more rings of 0’s (in light blue). With one ring, the 3 by 3 filter can be placed anywhere on the original 5 by 5 input, producing a 5 by 5 output. Yes! Now the output is the same size as the input.



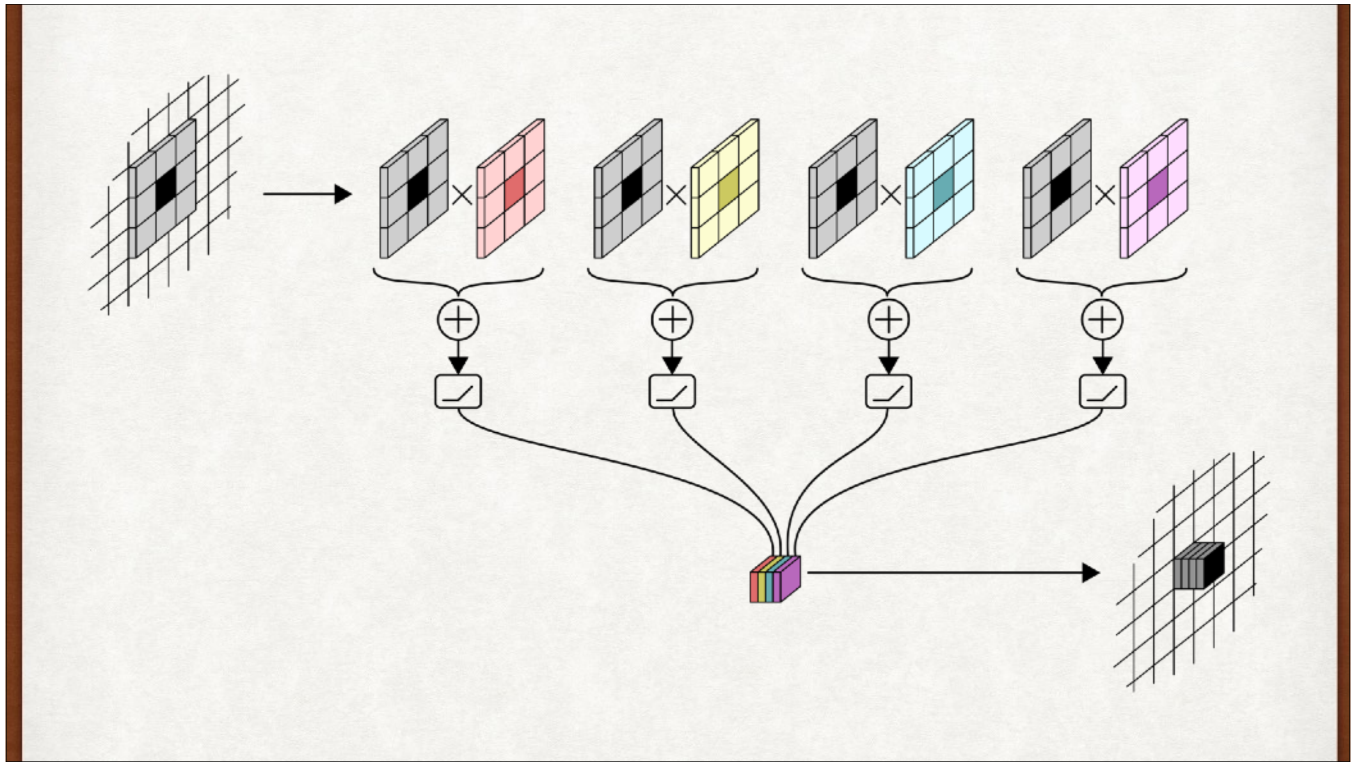
We can even make the output bigger than the input. The input here is 3 by 3. Surround it with two rings of 0’s, and also place a row and column of 0’s between each input element. The result is a field that’s 9 by 9, but mostly zeros. Moving the filter only where it doesn’t fall off the edge gives us a 7 by 7 output. The output is more than twice the size of the input.



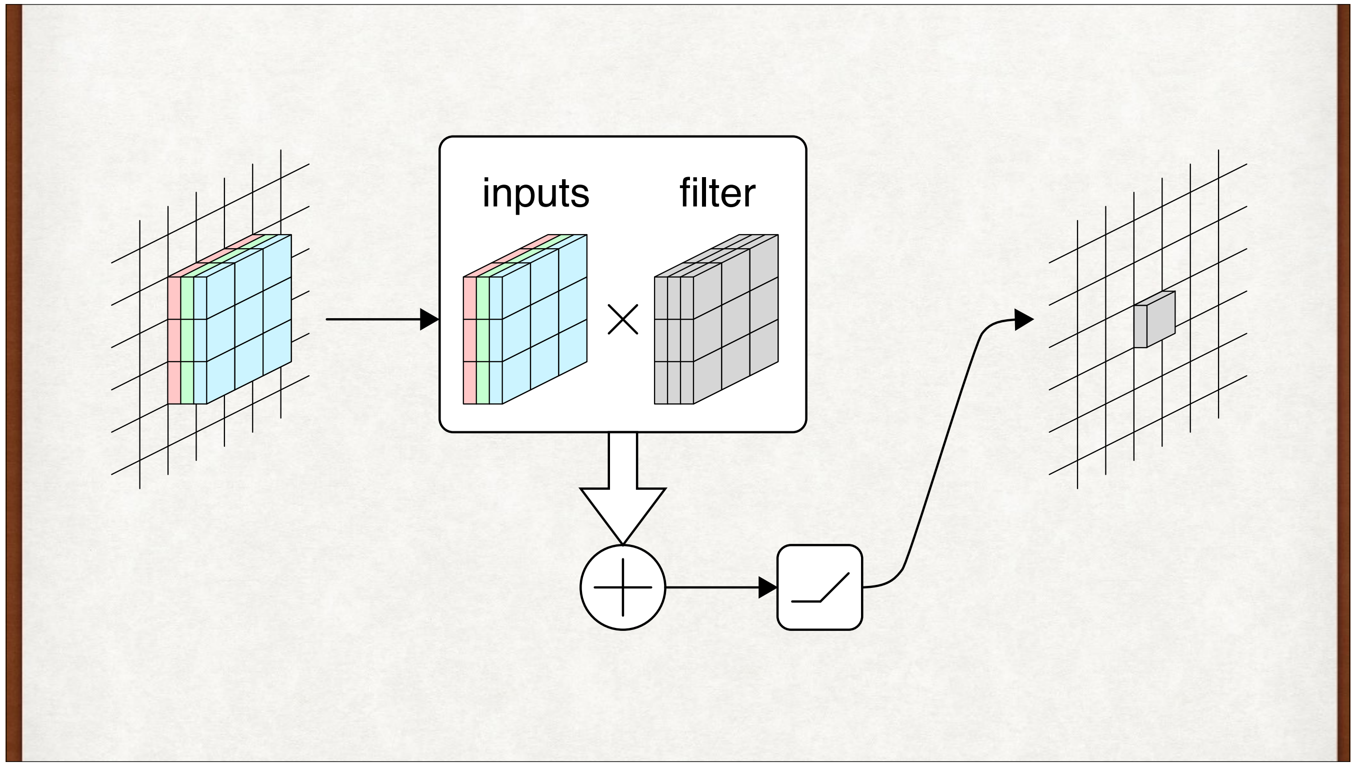
Striding means moving the filter by more than 1 step horizontally and/or vertically. It’s an efficient way to make the output smaller than the output.



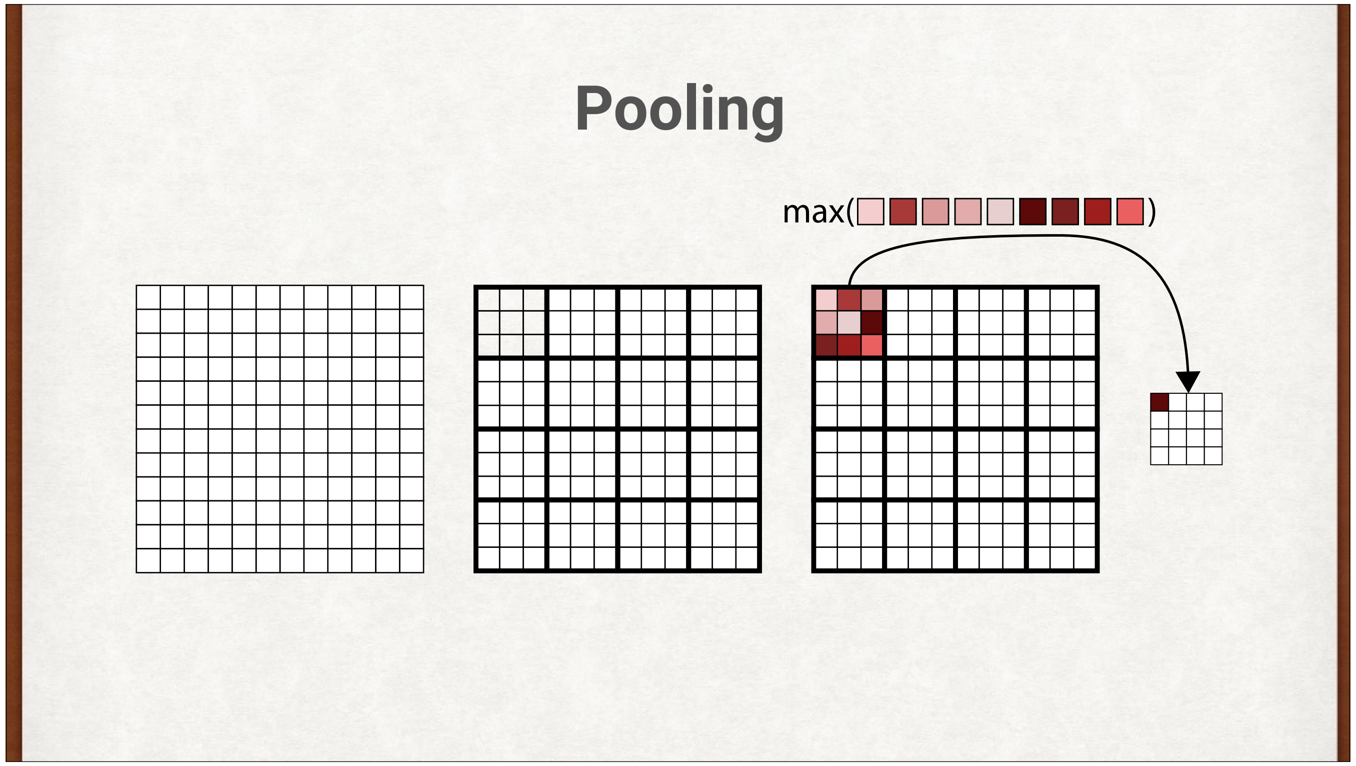
If we have multiple filters, they each produce an output for the same input. This process is simultaneous and independent. Hey, that sounds like something a GPU could do in parallel! It is!



Multiple filters produce multiple channels of output.

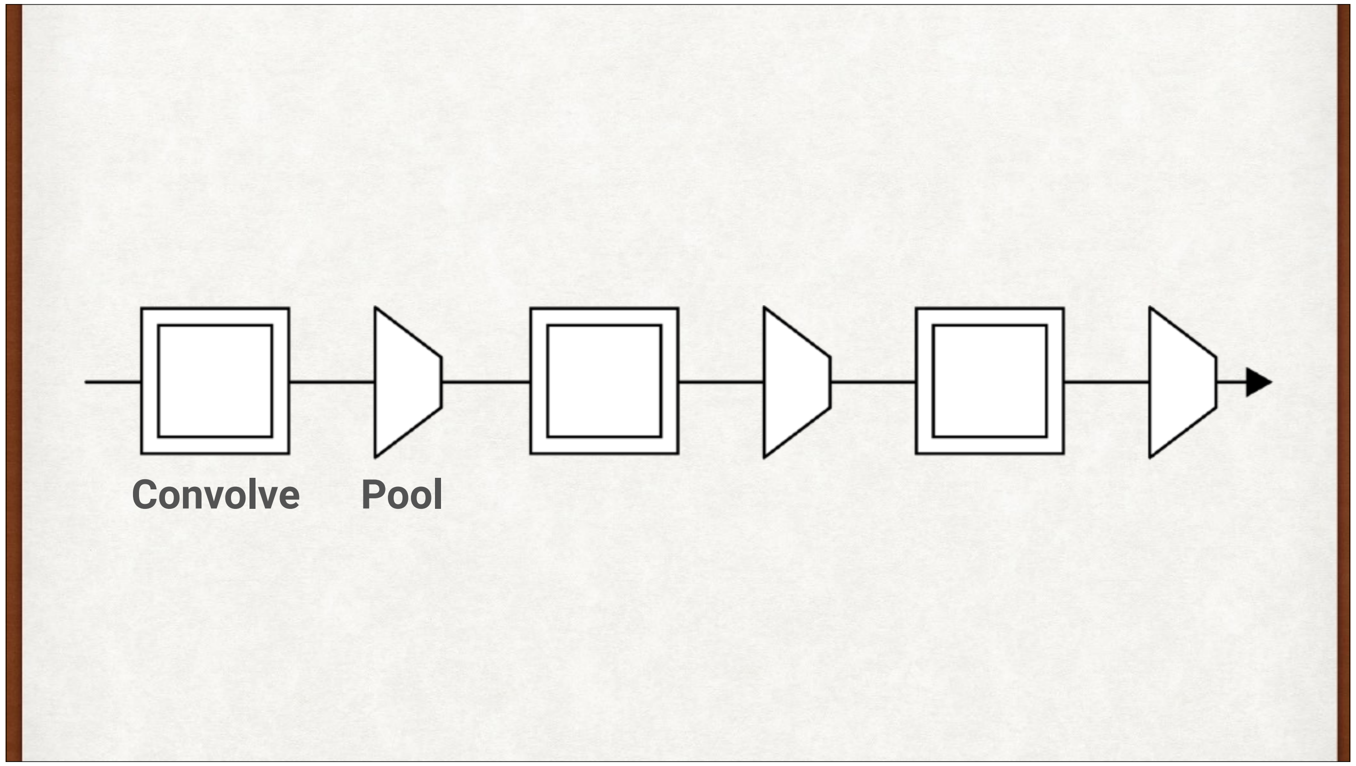


An RGB image over a 3 by 3 region produces a 3x3x3 “core sample.” Thus our filter is 3x3x3 as well. As always, we multiply corresponding values, sum them up, run them through an activation function, and save a single value.



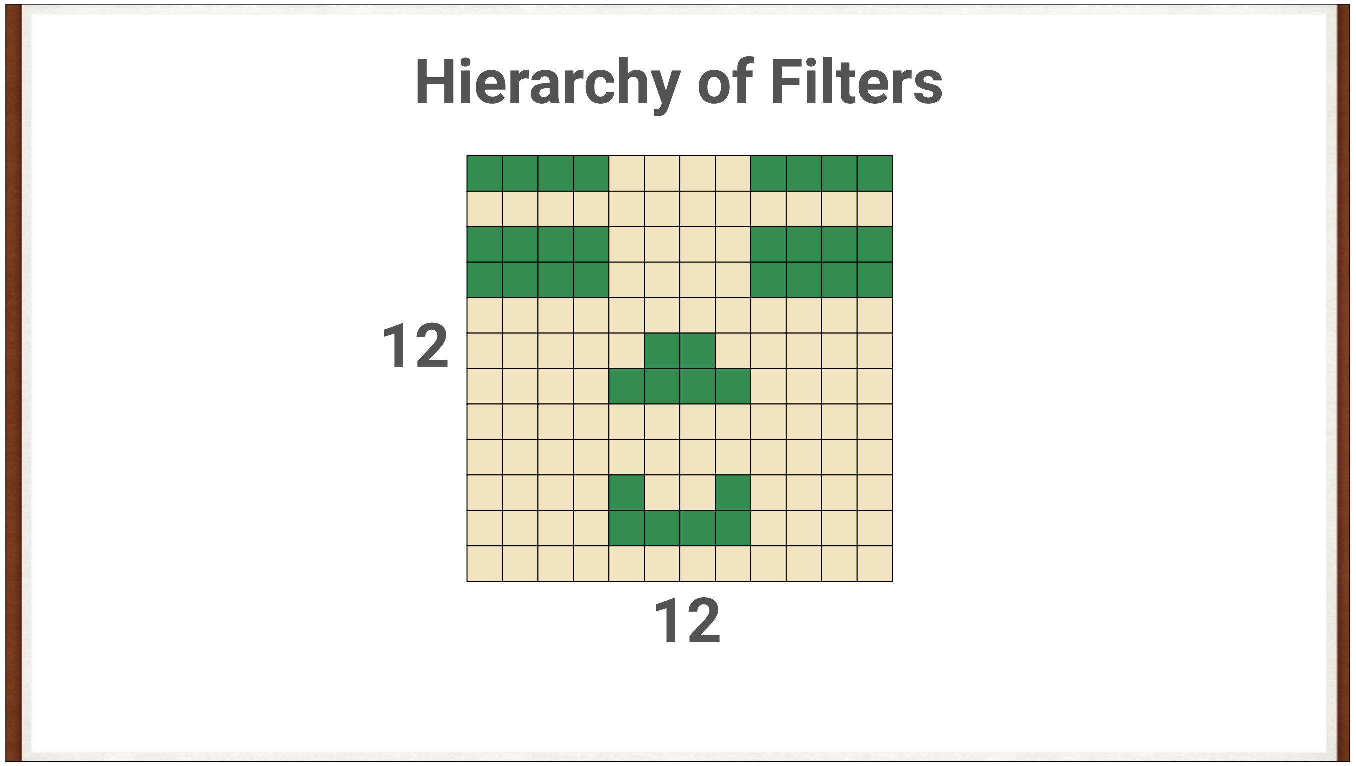
Pooling

Pooling is another way to reduce the size of the input. Here we look at 3-by-3 regions and save just the maximum value. We could also save the average value, or the mode, but the max is most common.



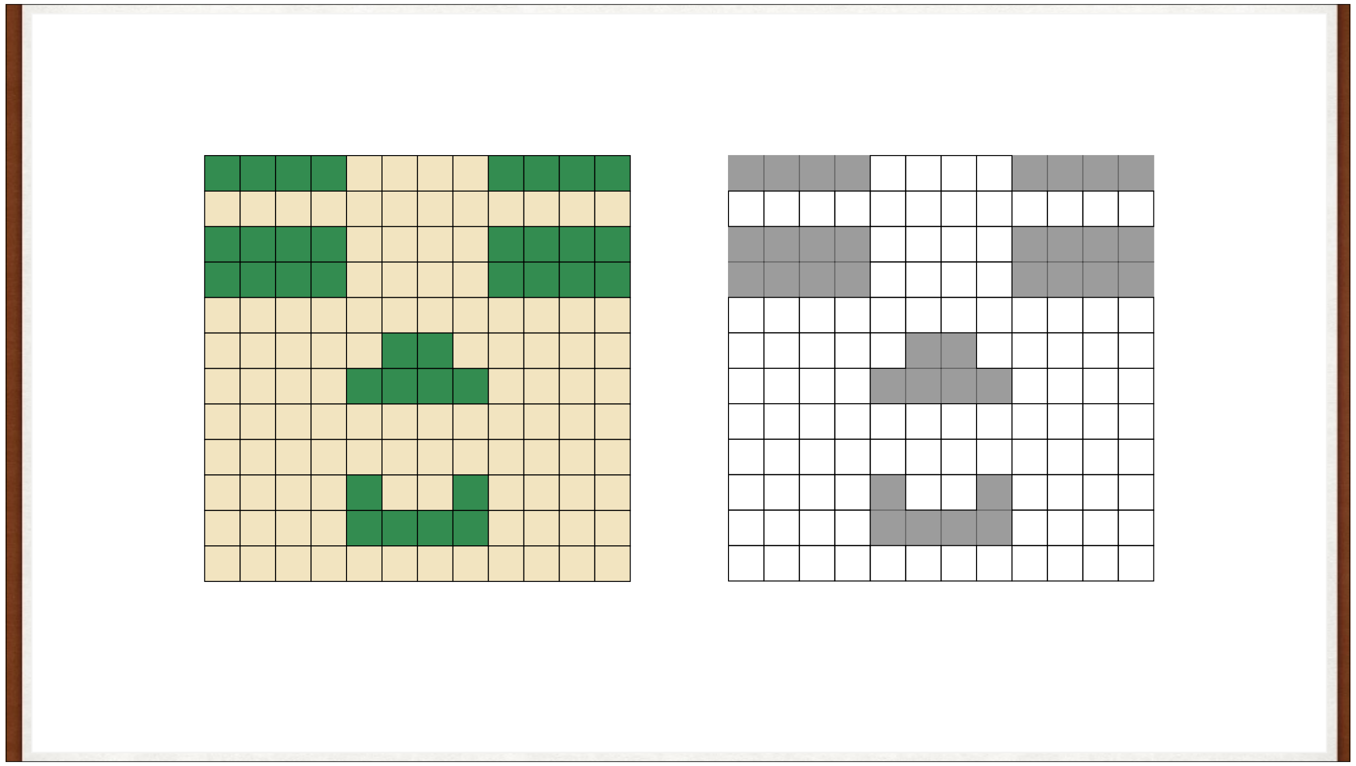
Convolve Pool

My icons for convolution (a box in a box) and pooling (the trapezoid getting smaller left to right).

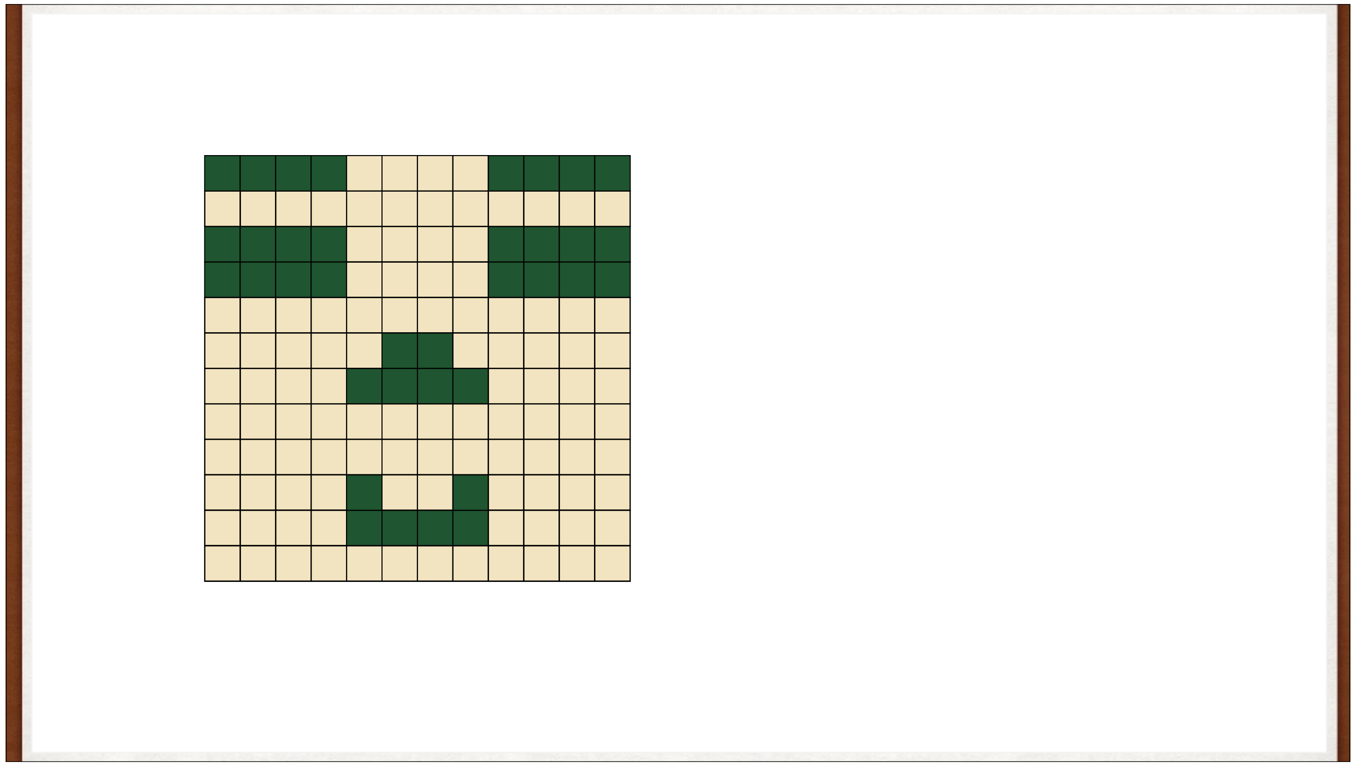


Hierarchy of Filters

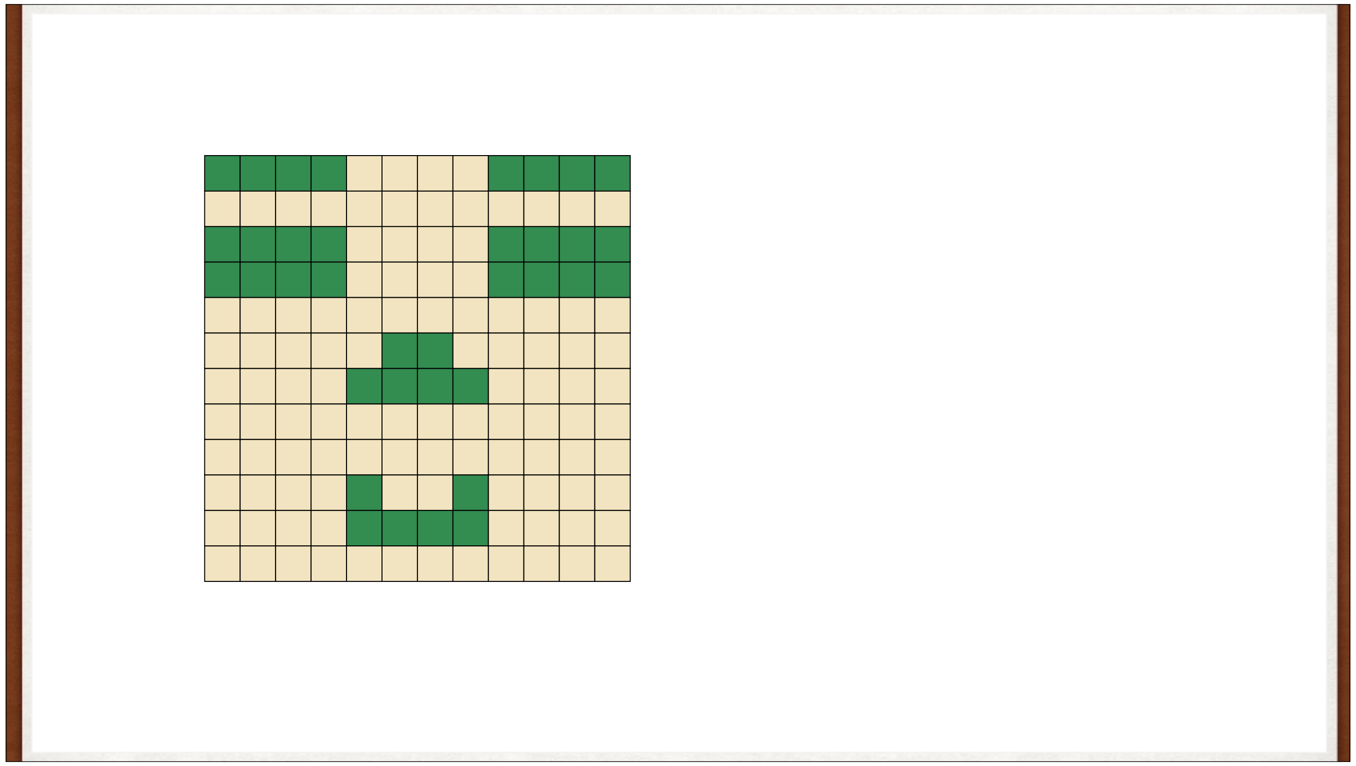
Is there a face in here?



Let’s see if it matches what we think a face looks like

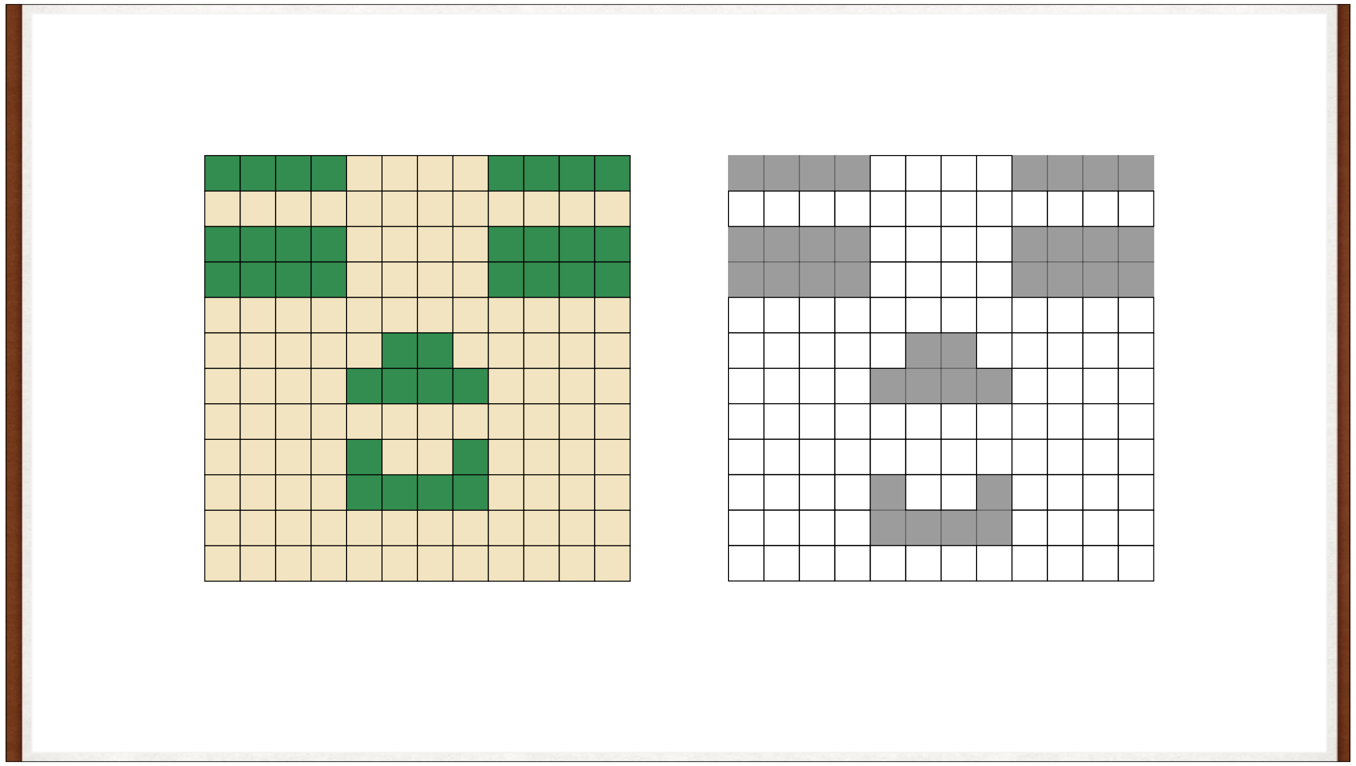


Yes, it was a face.

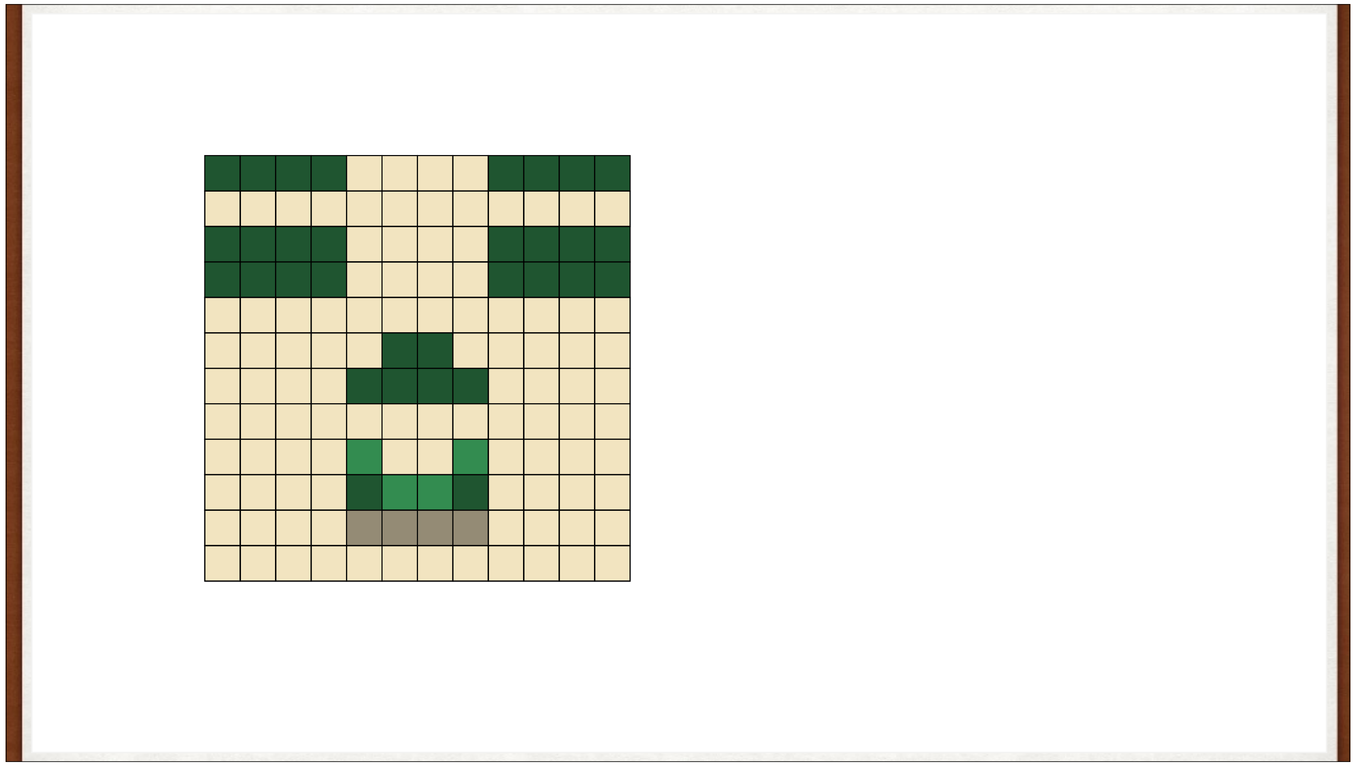


🟩	🟩	🟩	🟩					🟩	🟩	🟩	🟩
🟩	🟩	🟩	🟩					🟩	🟩	🟩	🟩
🟩	🟩	🟩	🟩					🟩	🟩	🟩	🟩
					🟩	🟩					
				🟩	🟩	🟩	🟩				
				🟩			🟩				
				🟩	🟩	🟩	🟩				

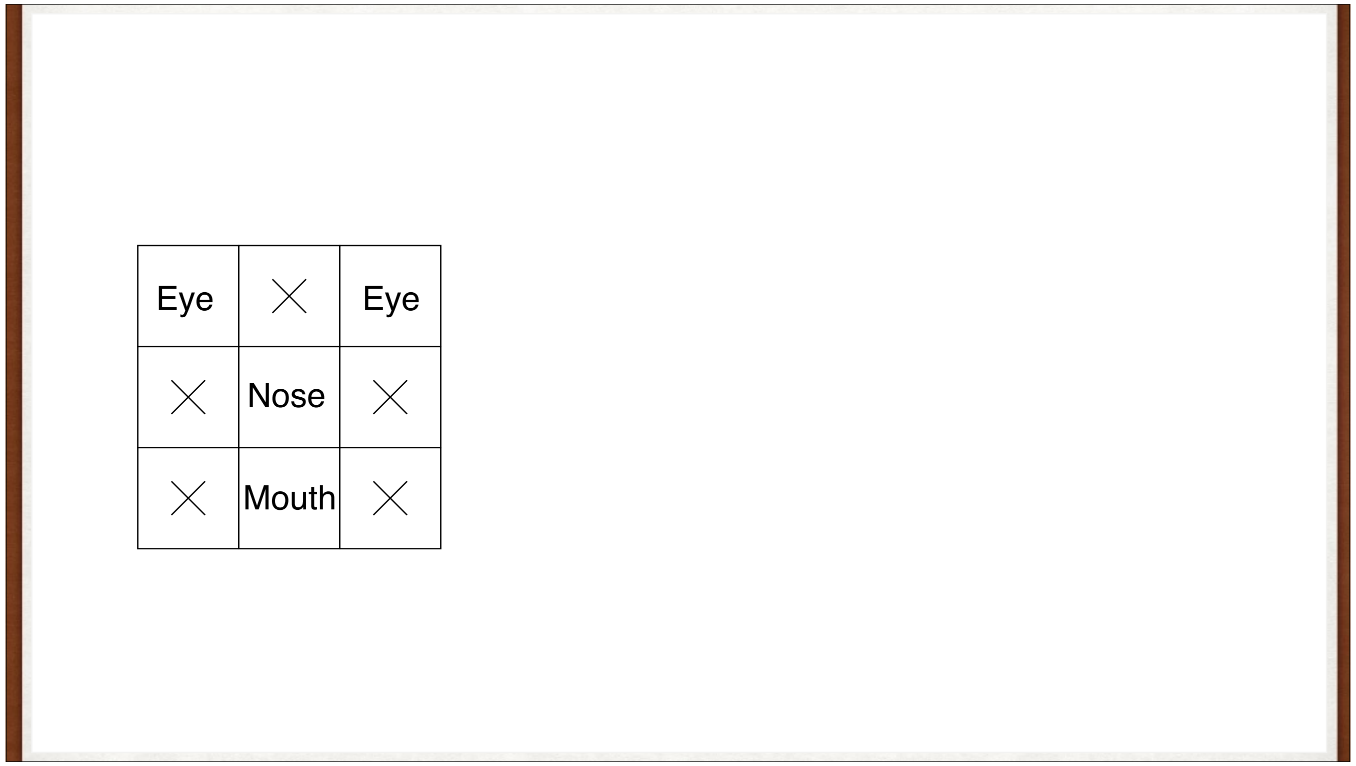
The original face



Let’s raise up the mouth 1 pixel. It’s still a face. So we get out the mask…

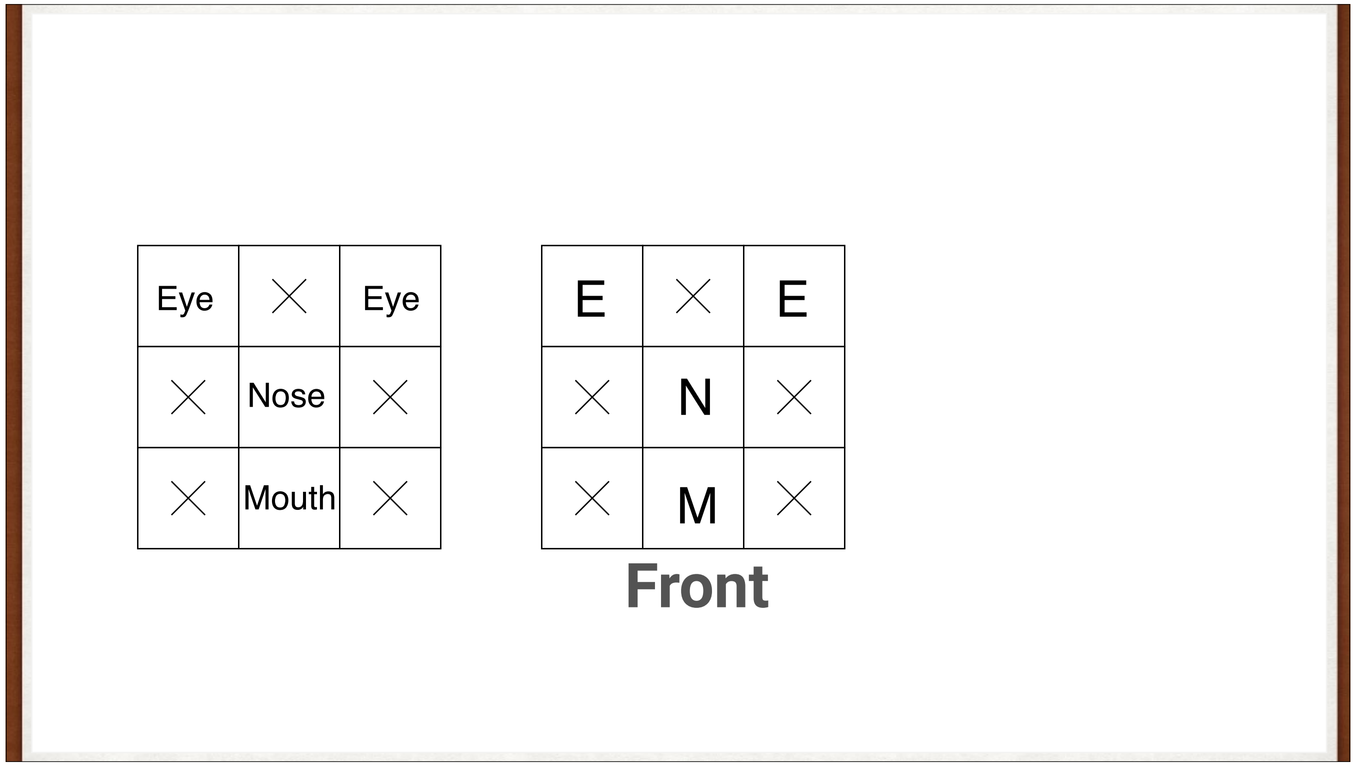


And we don’t get a match. Surely there’s a more robust and efficient way to do this?



Eye	✕	Eye
✕	Nose	✕
✕	Mouth	✕

This is what we probably want to start with. A face has eyes in the upper corners, and a nose and mouth below. We don’t care what’s in the other places. Note that this is just a 3-by-3 diagram, so we can’t directly apply it to our higher-res input. But let’s start with this and work backwards to the input.

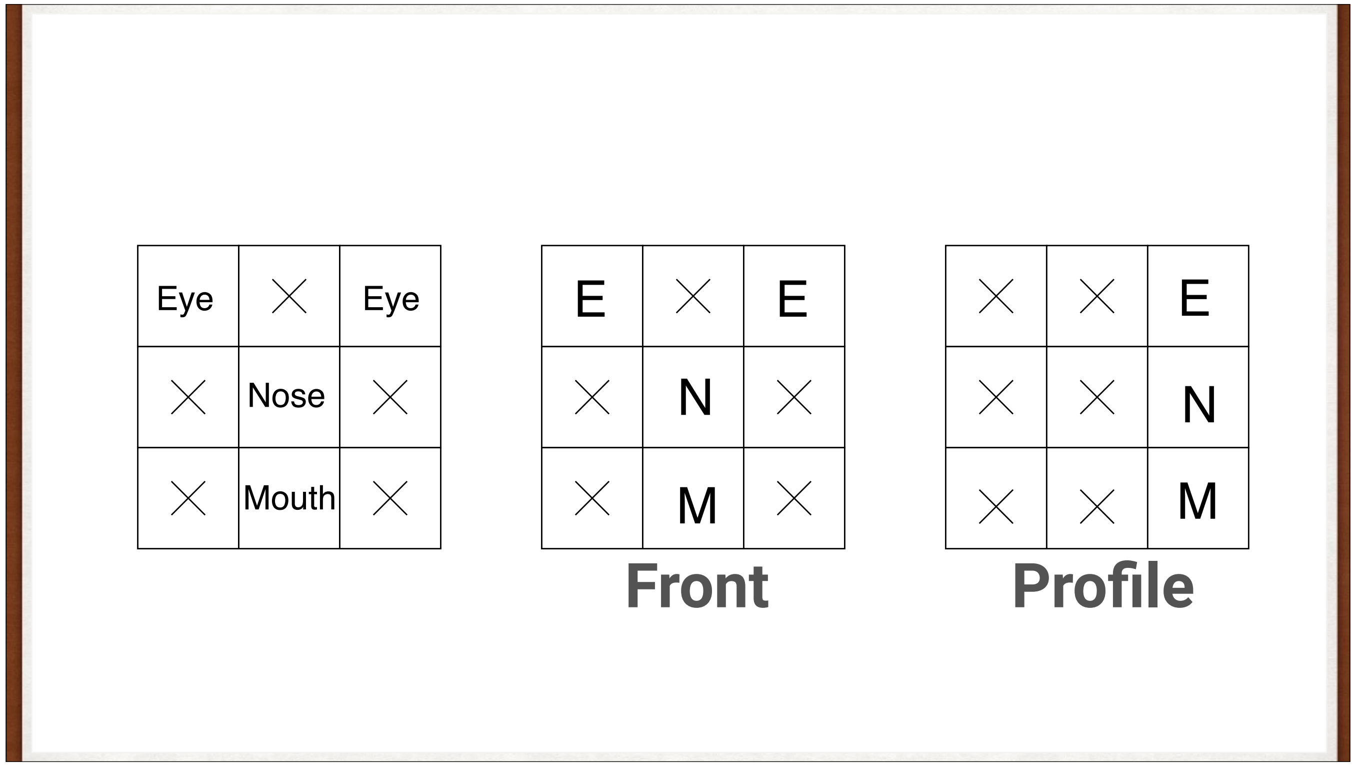


Eye	×	Eye
×	Nose	×
×	Mouth	×

E	×	E
×	N	×
×	M	×

Front

A simpler way to represent what we want, using just letters instead of words



Eye	×	Eye
×	Nose	×
×	Mouth	×

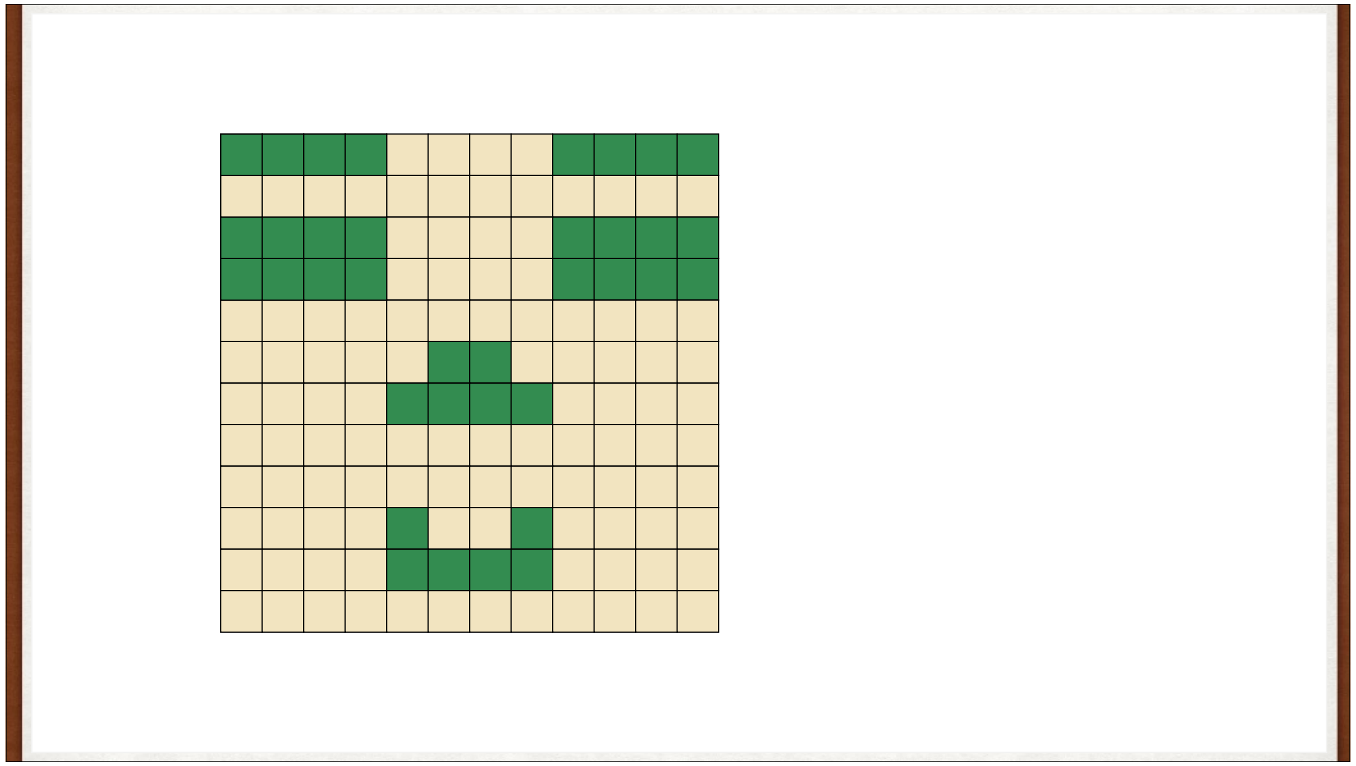
E	×	E
×	N	×
×	M	×

Front

×	×	E
×	×	N
×	×	M

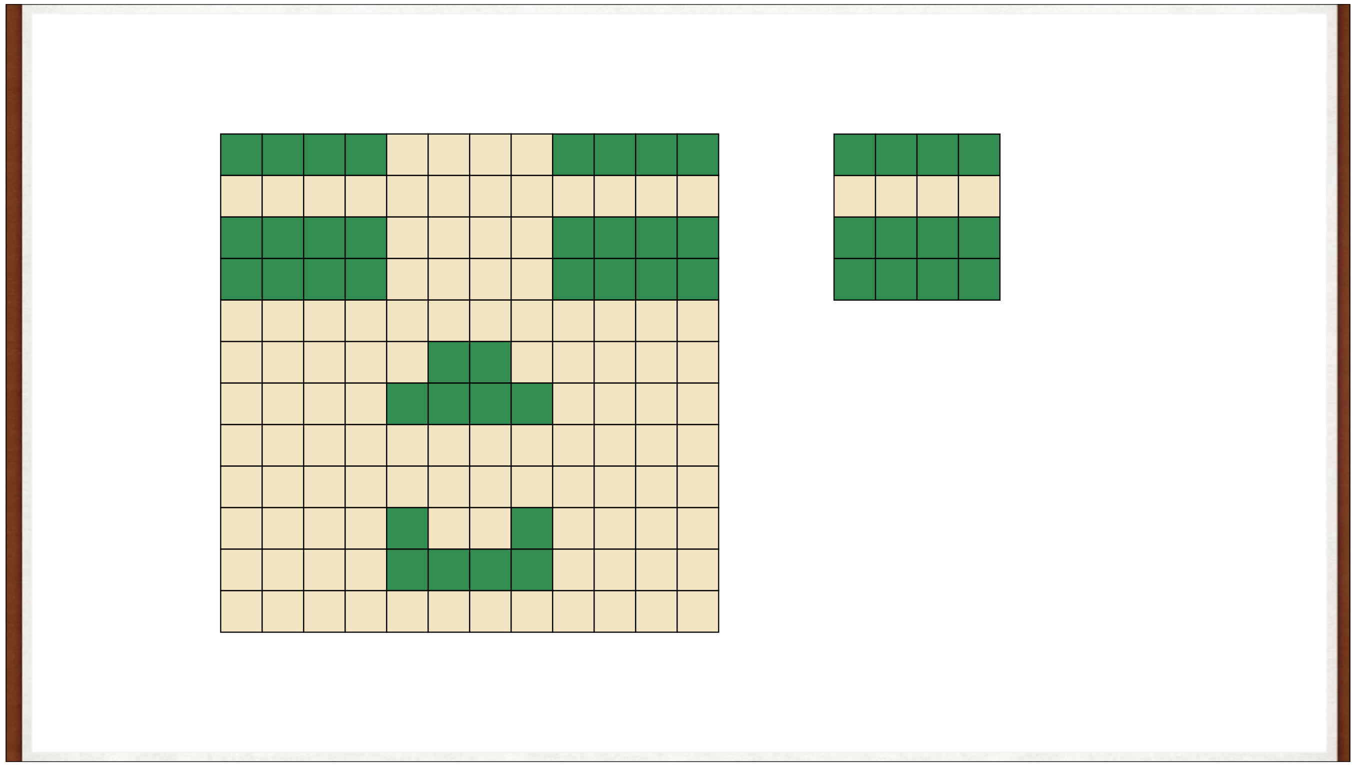
Profile

We can look for profiles as well just by using a different 3-by-3 goal.



🟩	🟩	🟩	🟩					🟩	🟩	🟩	🟩
🟩	🟩	🟩	🟩					🟩	🟩	🟩	🟩
🟩	🟩	🟩	🟩					🟩	🟩	🟩	🟩
					🟩	🟩					
				🟩	🟩	🟩	🟩				
				🟩			🟩				
				🟩	🟩	🟩	🟩				

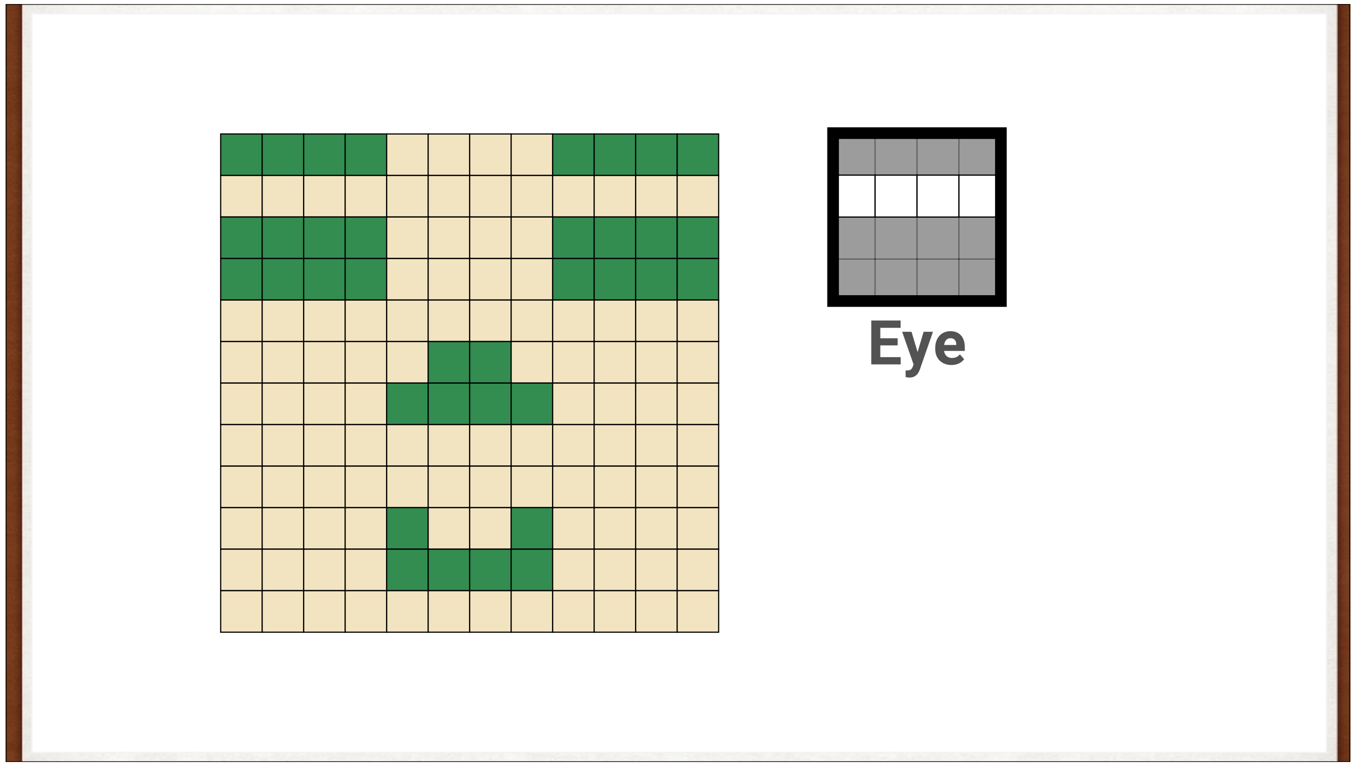
Our goal wants to know where eyes are. So we can make an “eye” filter and convolve with it.



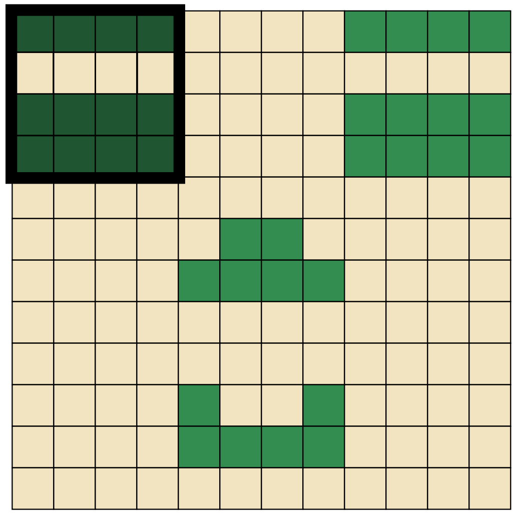
🟩	🟩	🟩	🟩	⬜	⬜	⬜	⬜	🟩	🟩	🟩	🟩
⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜
🟩	🟩	🟩	🟩	⬜	⬜	⬜	⬜	🟩	🟩	🟩	🟩
🟩	🟩	🟩	🟩	⬜	⬜	⬜	⬜	🟩	🟩	🟩	🟩
⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜
⬜	⬜	⬜	⬜	⬜	🟩	🟩	⬜	⬜	⬜	⬜	⬜
⬜	⬜	⬜	⬜	🟩	🟩	🟩	🟩	⬜	⬜	⬜	⬜
⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜
⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜
⬜	⬜	⬜	⬜	🟩	⬜	⬜	🟩	⬜	⬜	⬜	⬜
⬜	⬜	⬜	⬜	🟩	🟩	🟩	🟩	⬜	⬜	⬜	⬜
⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜

🟩	🟩	🟩	🟩
⬜	⬜	⬜	⬜
🟩	🟩	🟩	🟩
🟩	🟩	🟩	🟩

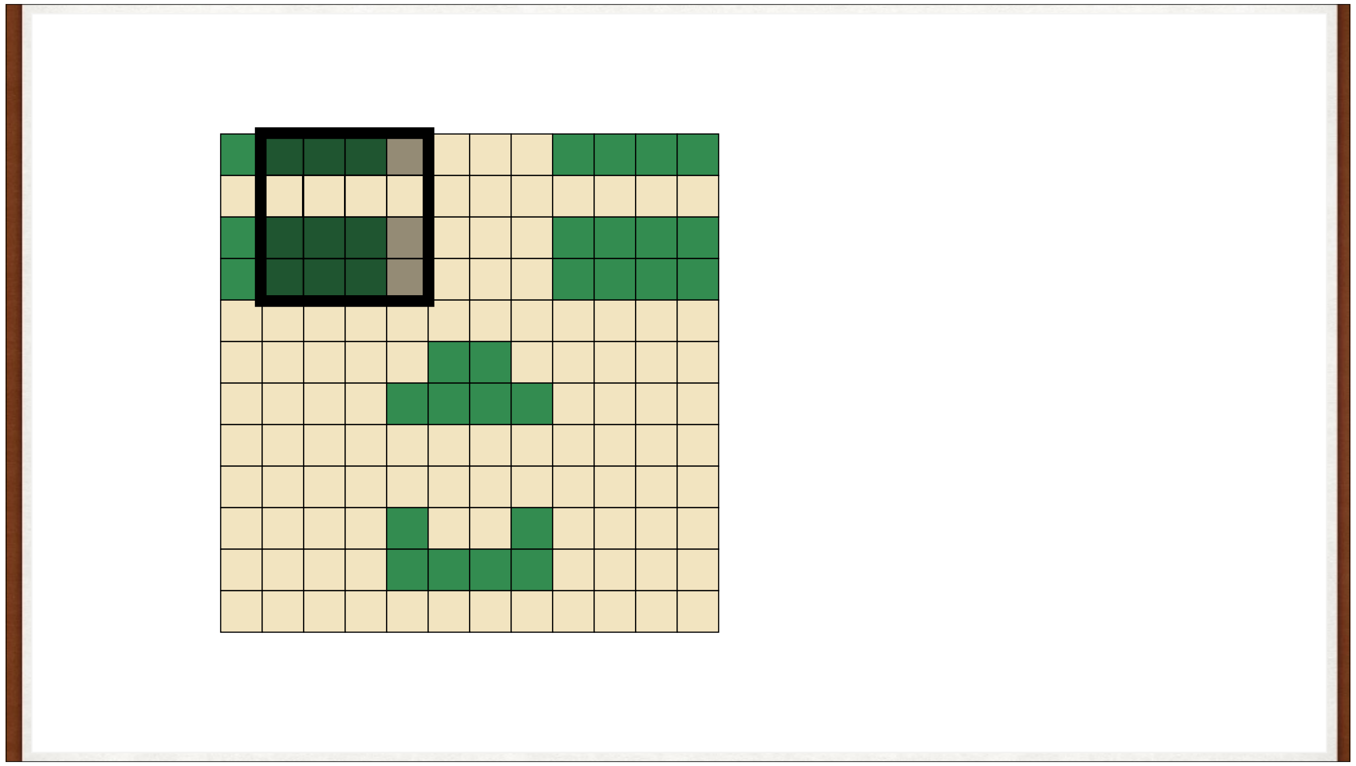
Here’s the eye…



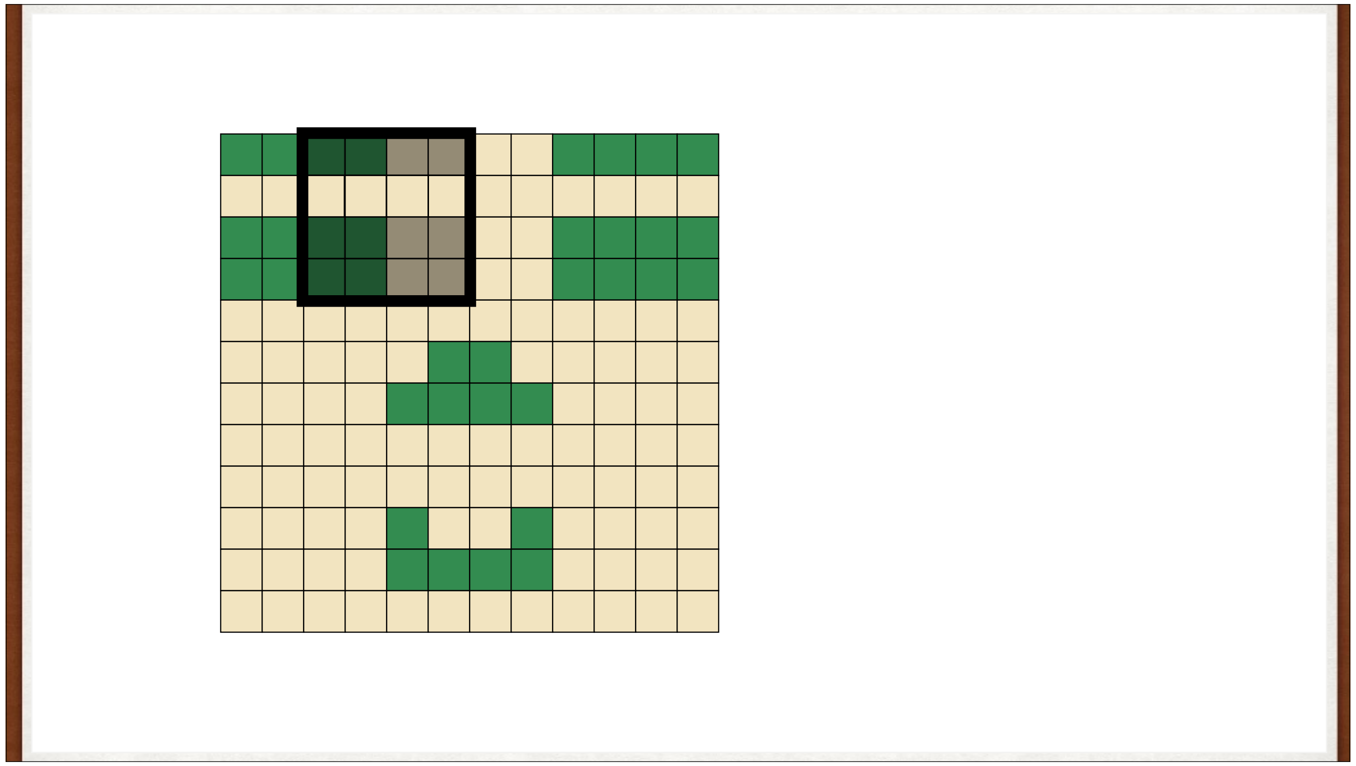
…as a 4x4 filter.



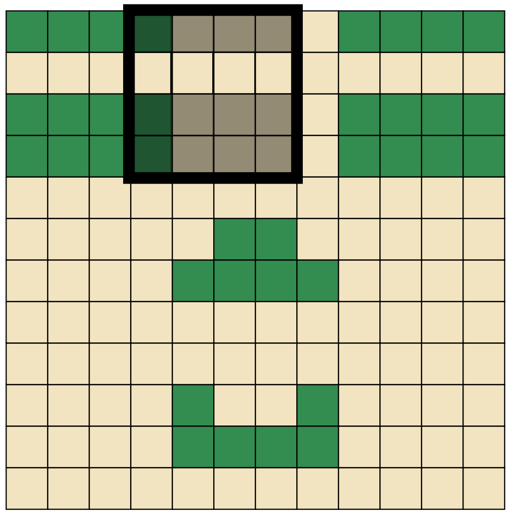
Let’s not worry about padding for now. Upper-left corner is a match!



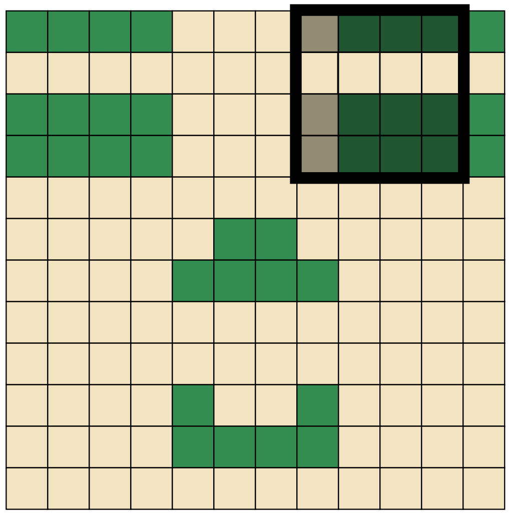
No match.



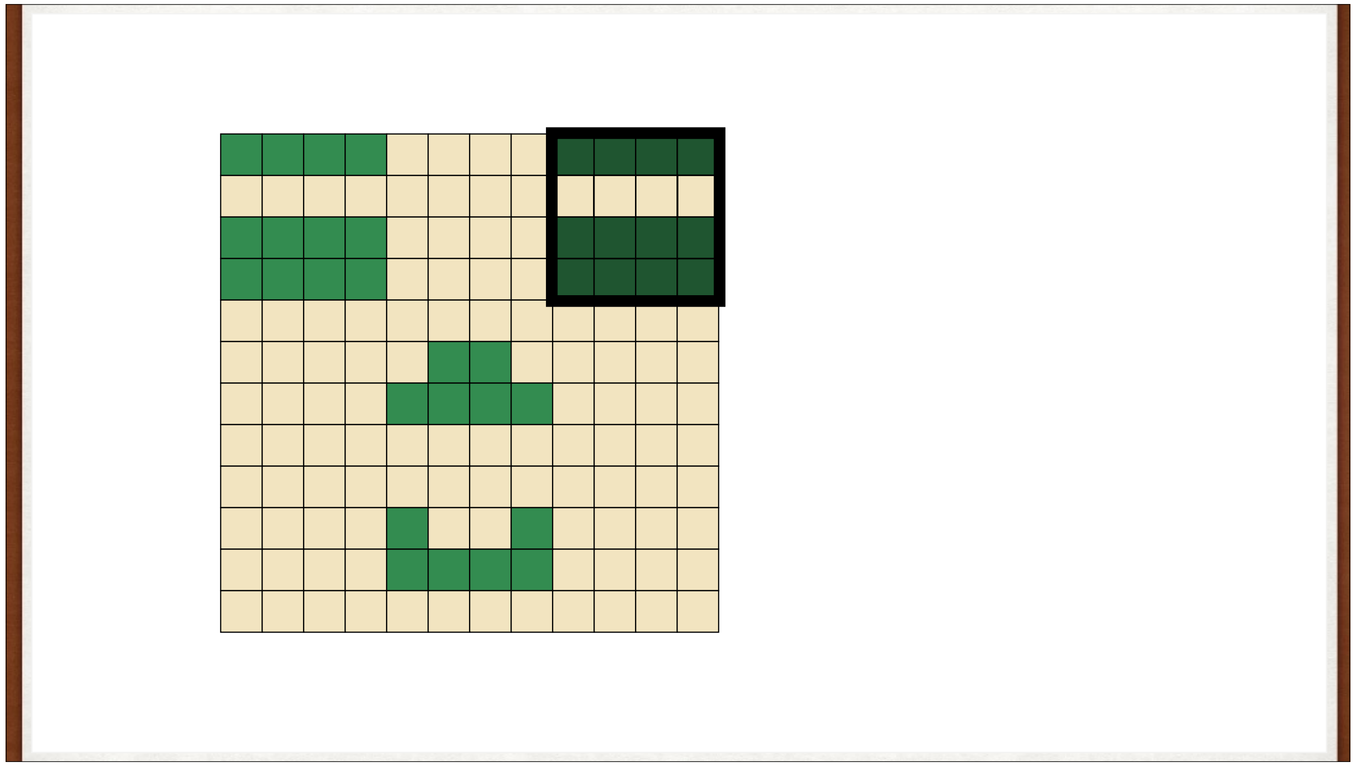
No match.



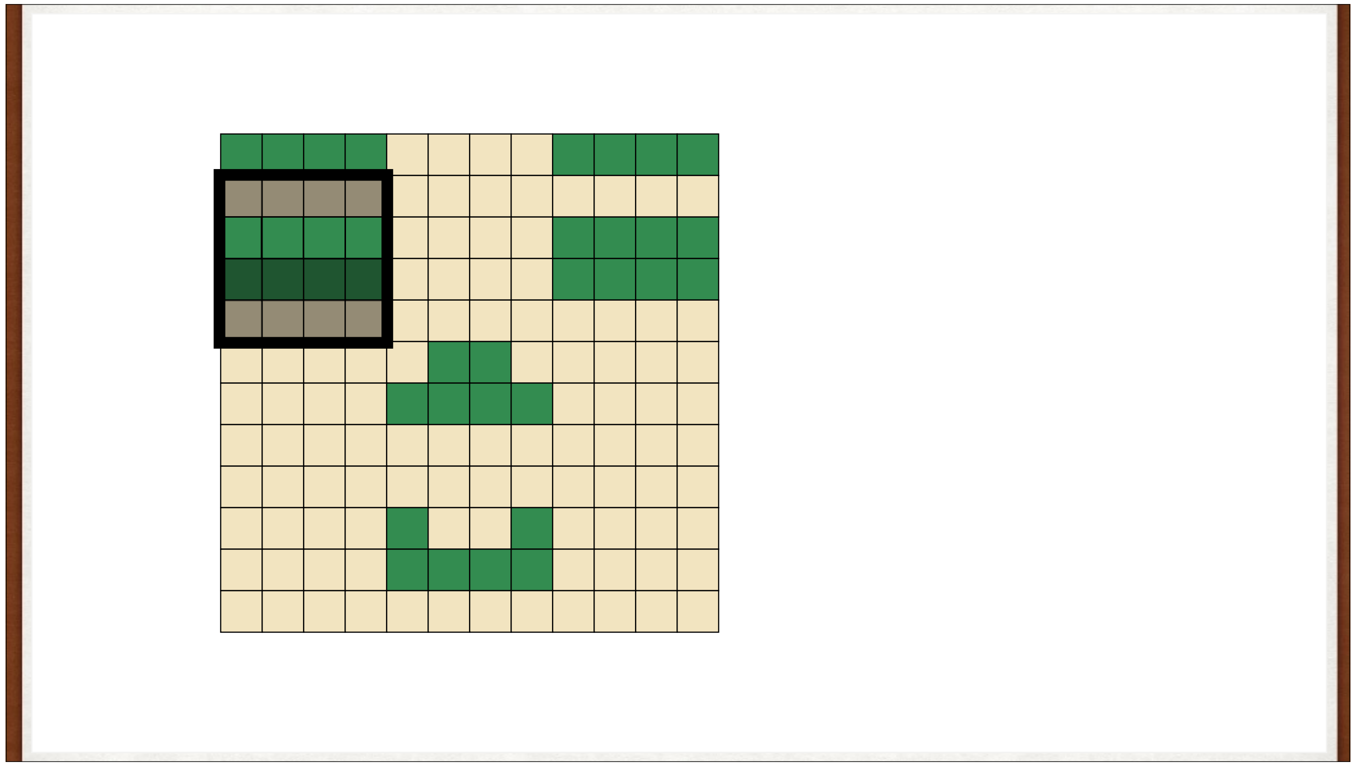
No match.



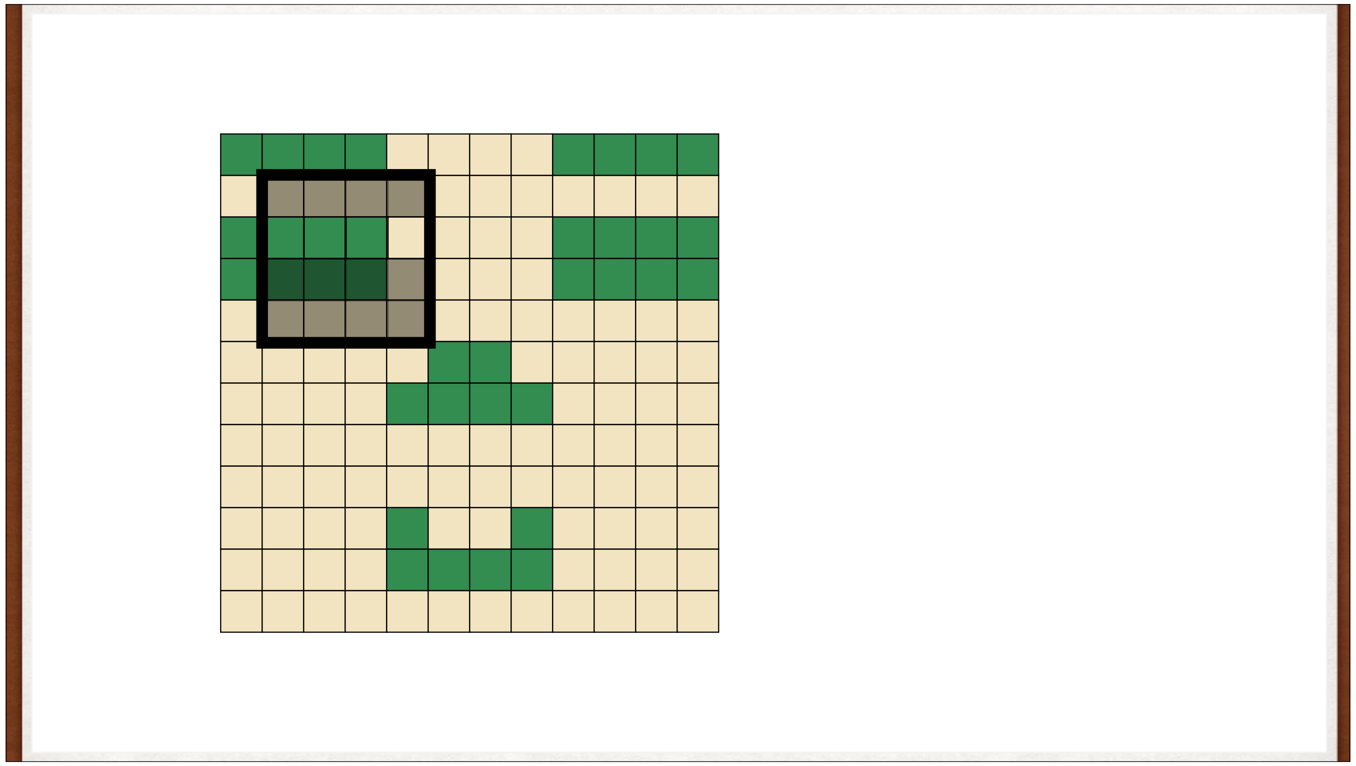
Moving forward… No match.



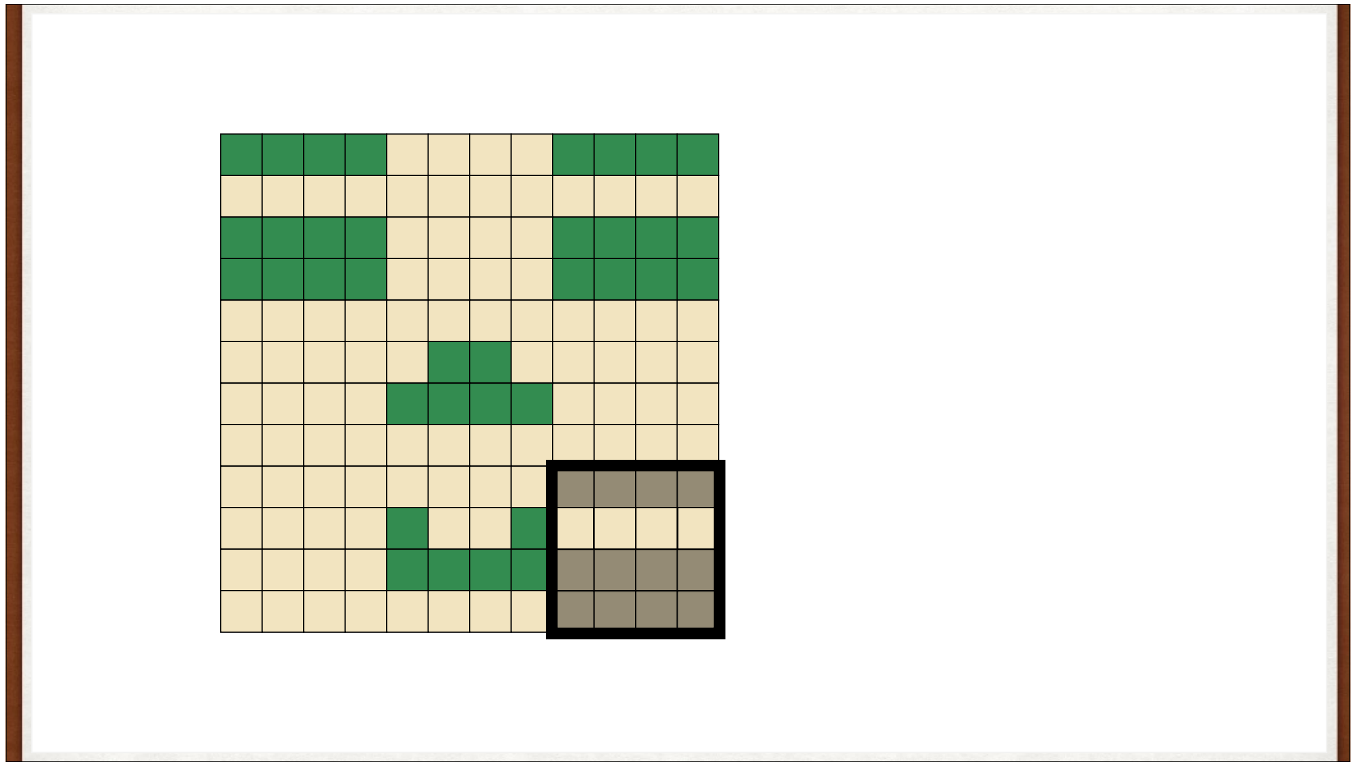
And a match!



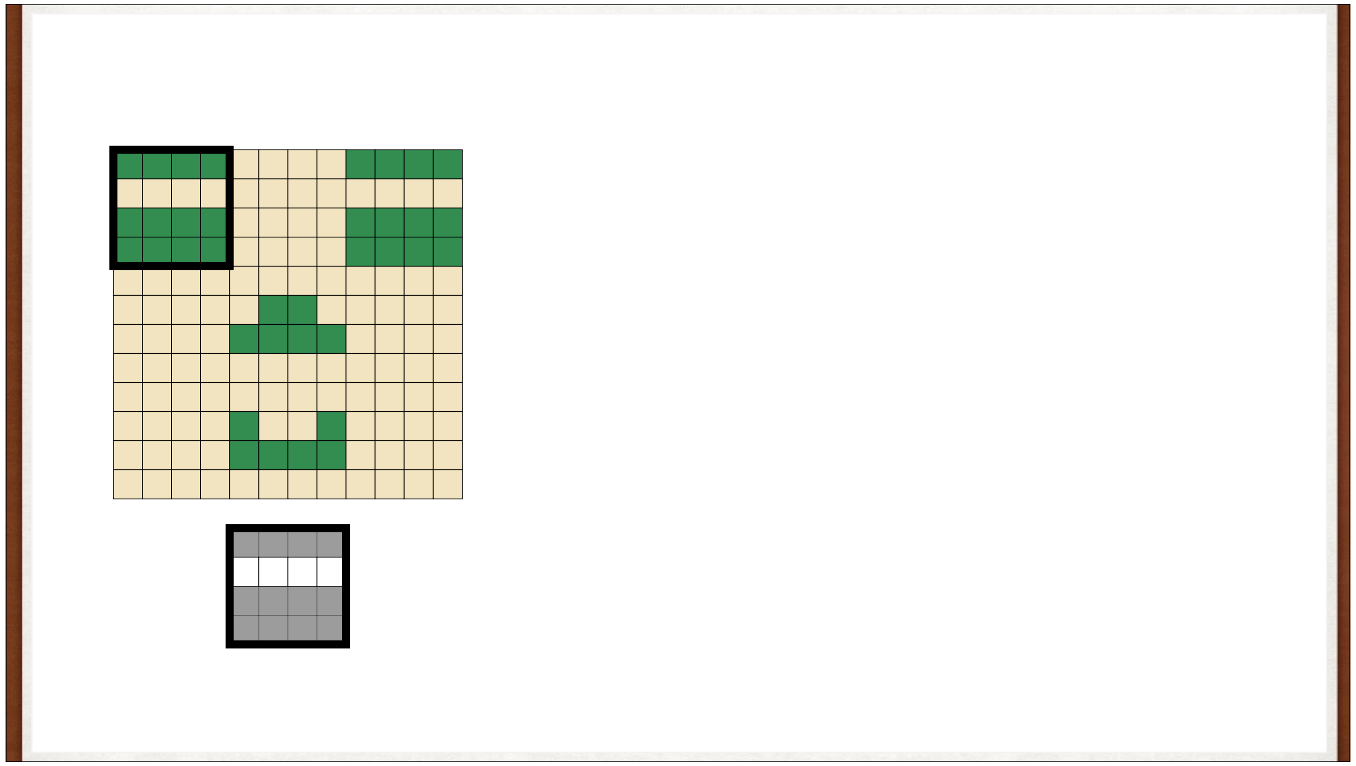
Starting the second row, we have a match…



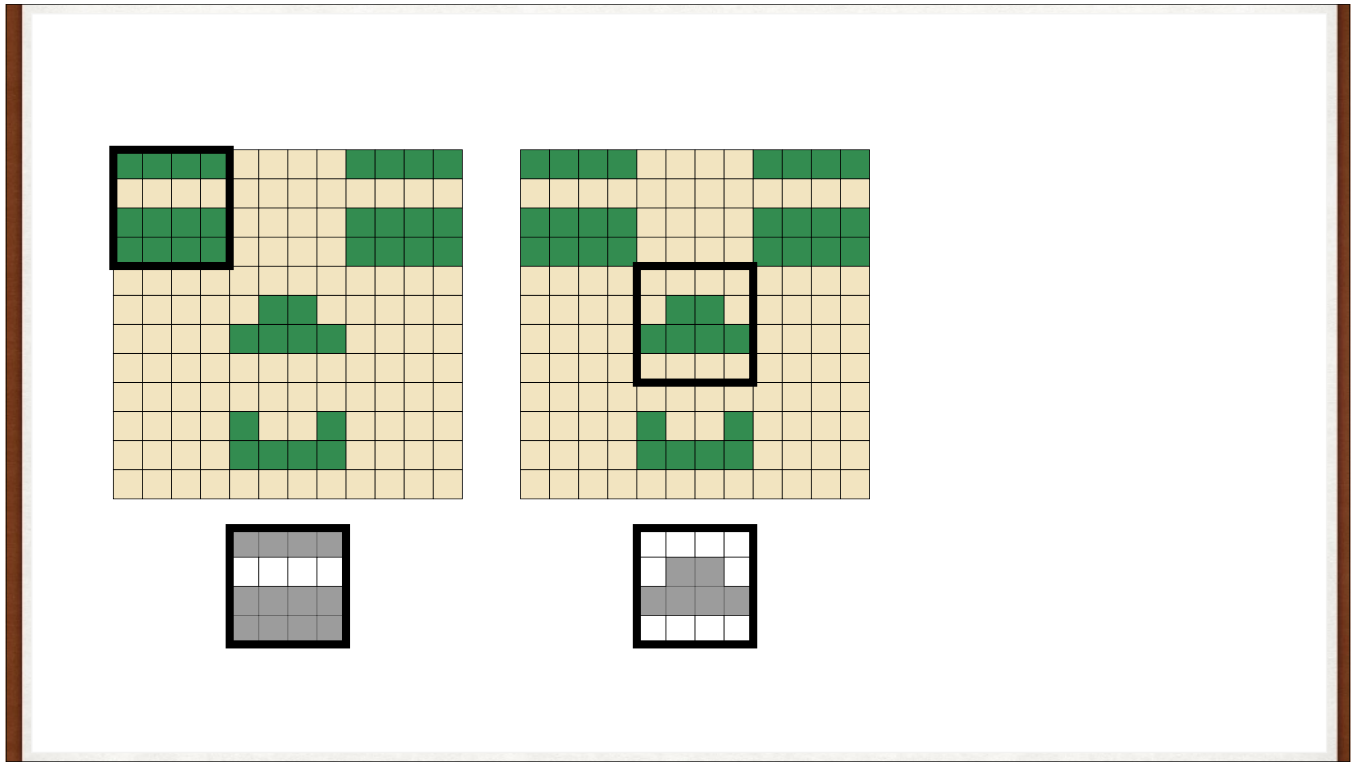
Starting the second row, we have a match…



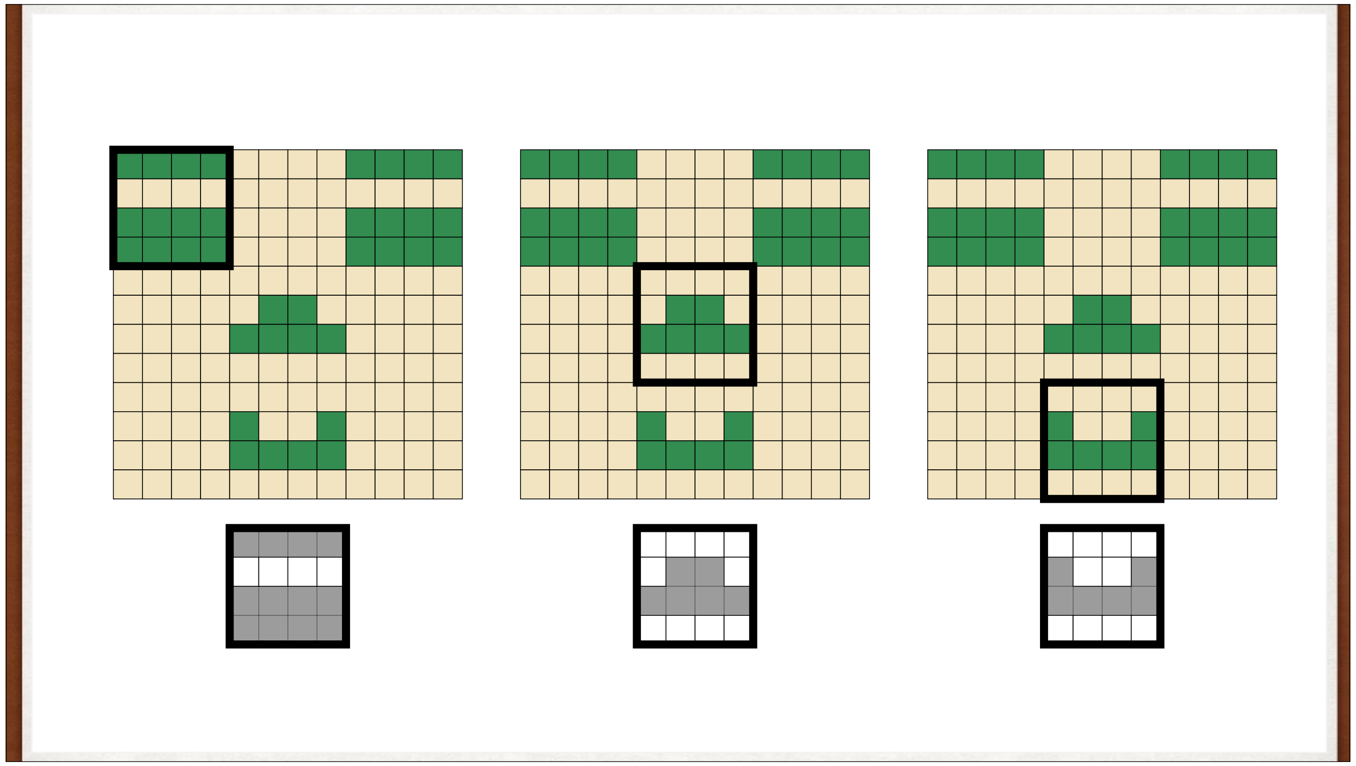
…and so on over the whole thing. That was a lot of computation. Now we'll have to do it all again for the nose, and all again for the mouth. Is there a more efficient way? Let's do what we did before, and break up our "eye" goal into smaller pieces.



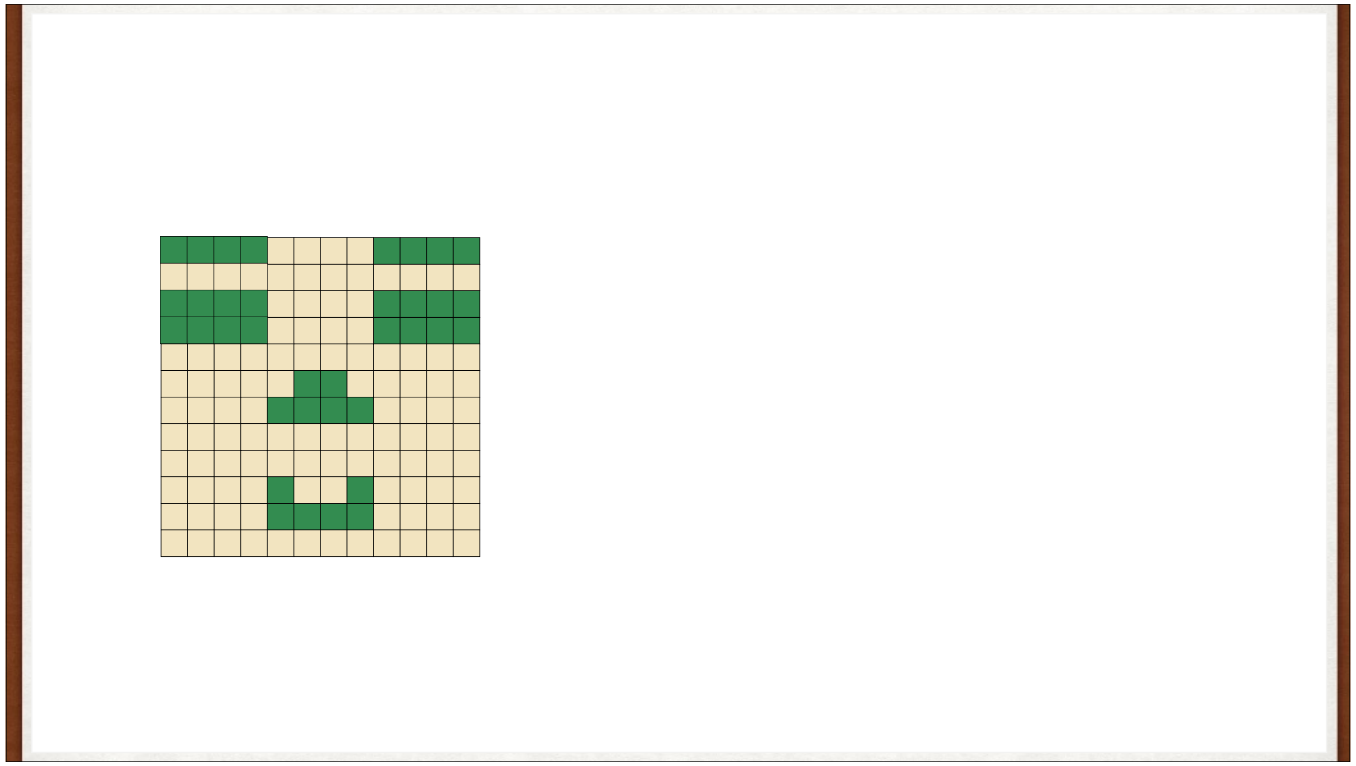
This was our eye filter…



the nose filter…

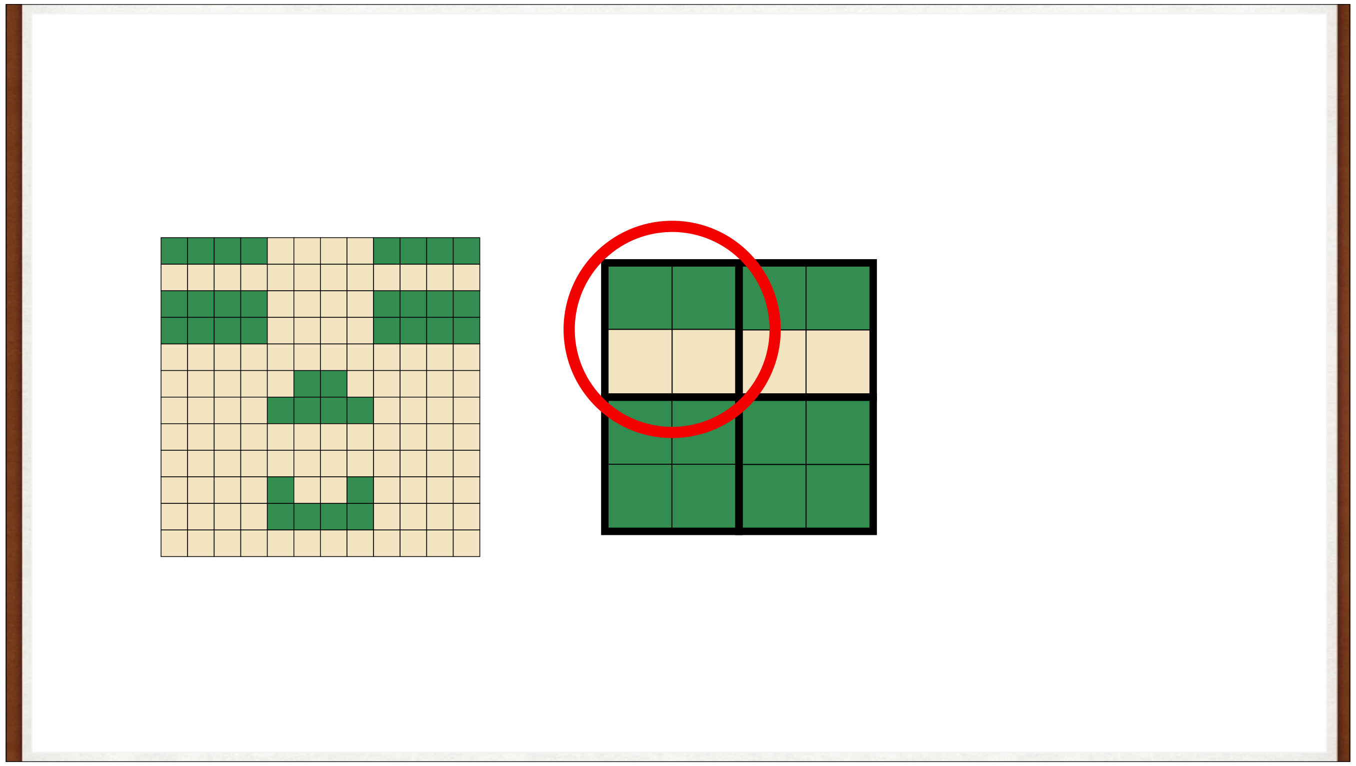


…and the mouth.

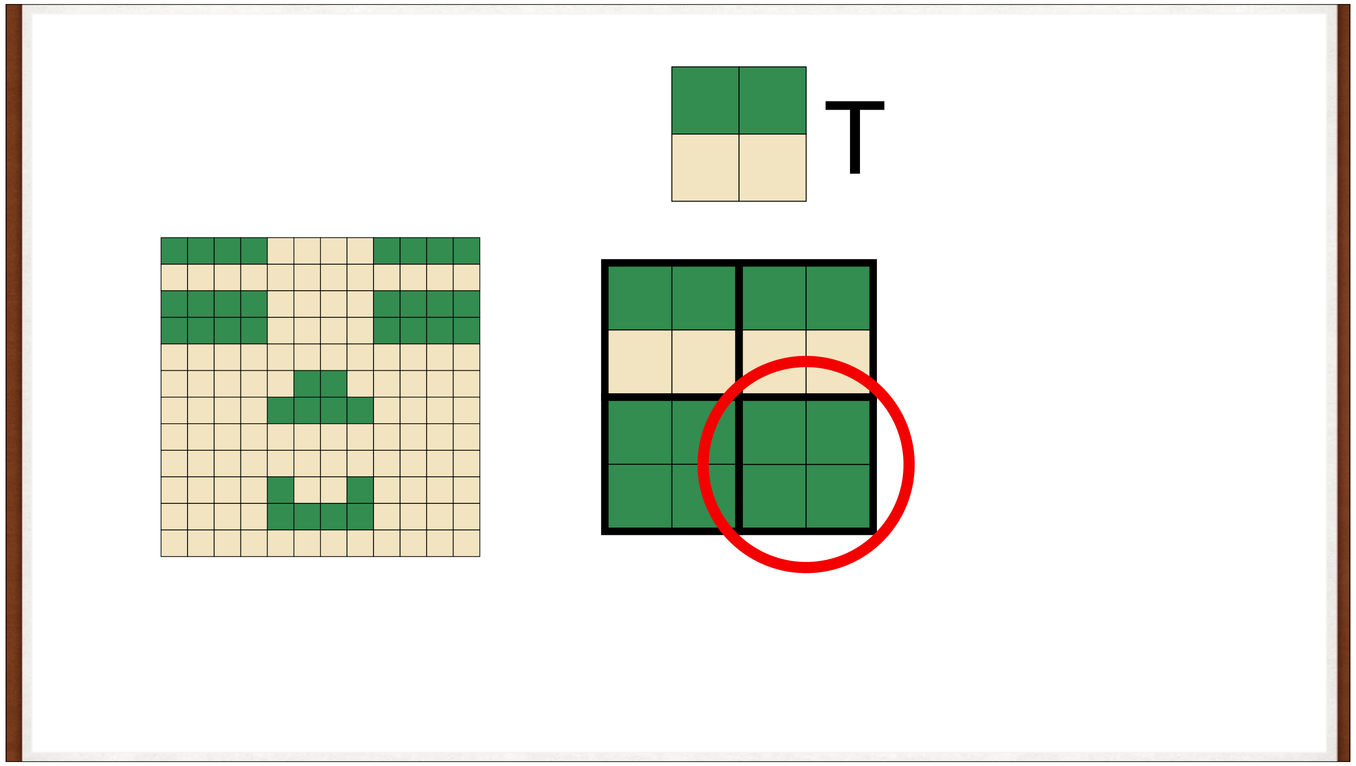


🟩	🟩	🟩	🟩	⬜	⬜	⬜	⬜	🟩	🟩	🟩	🟩
⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜
🟩	🟩	🟩	🟩	⬜	⬜	⬜	⬜	🟩	🟩	🟩	🟩
🟩	🟩	🟩	🟩	⬜	⬜	⬜	⬜	🟩	🟩	🟩	🟩
⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜
⬜	⬜	⬜	⬜	⬜	🟩	🟩	⬜	⬜	⬜	⬜	⬜
⬜	⬜	⬜	⬜	🟩	🟩	🟩	🟩	⬜	⬜	⬜	⬜
⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜
⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜
⬜	⬜	⬜	⬜	🟩	⬜	⬜	🟩	⬜	⬜	⬜	⬜
⬜	⬜	⬜	⬜	🟩	🟩	🟩	🟩	⬜	⬜	⬜	⬜
⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜	⬜

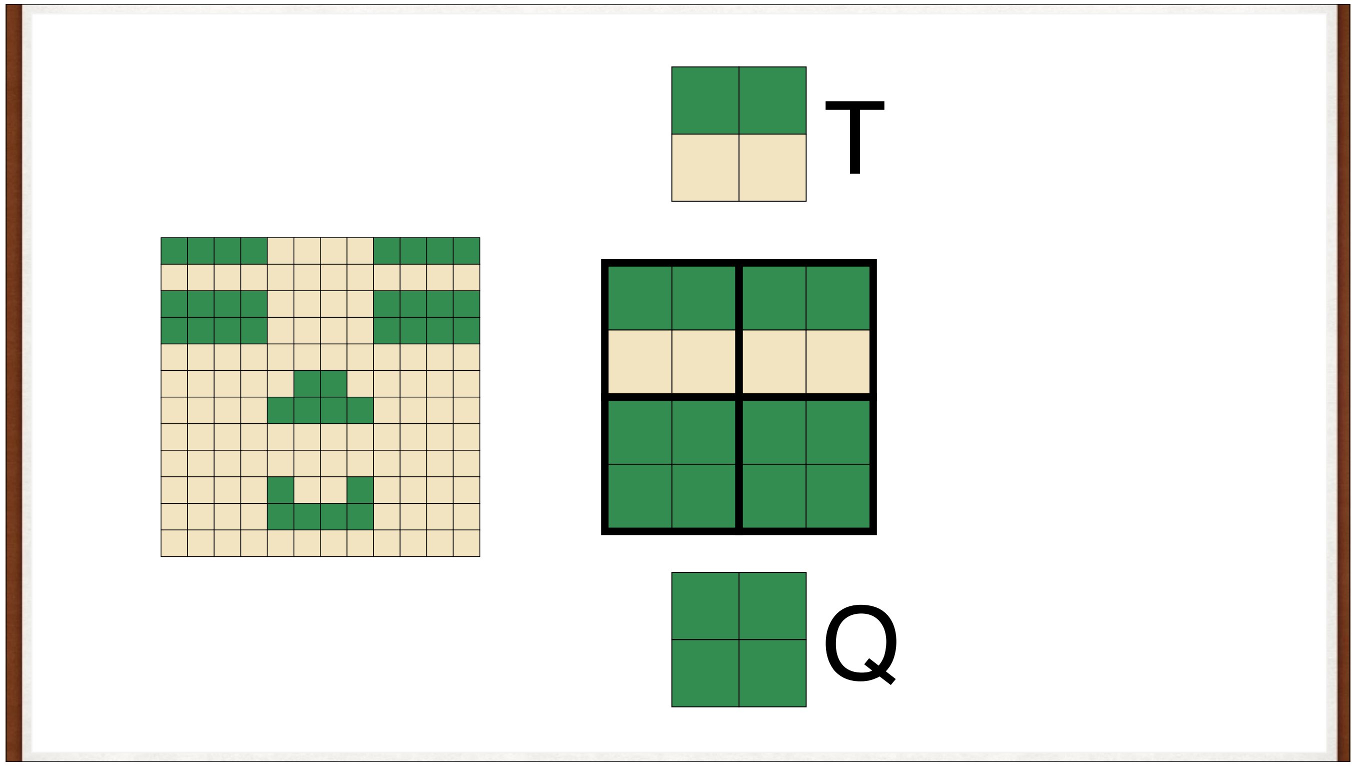
Let’s break up the eye…



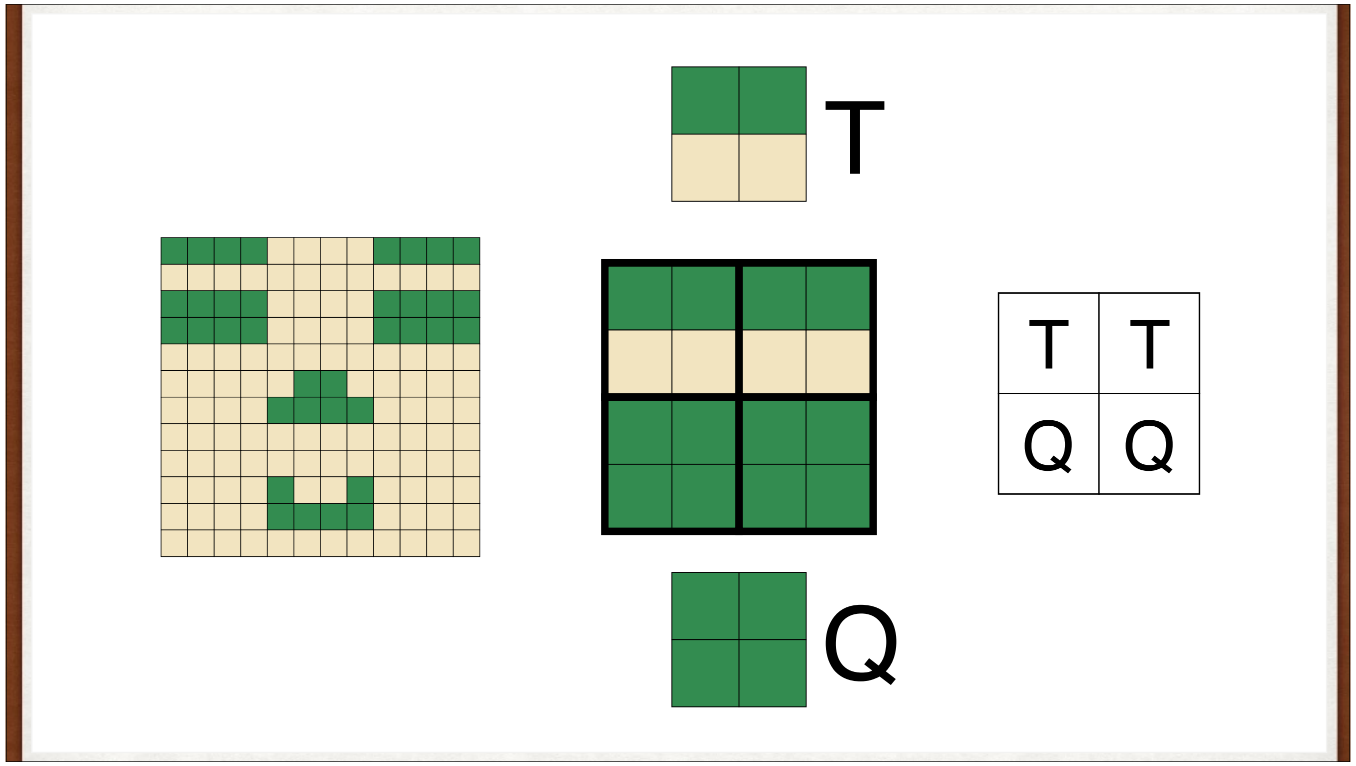
into 4 cells that are 2x2 each. Either upper cell…



can be matched with a filter called T, for Top. Either of the bottom cells…

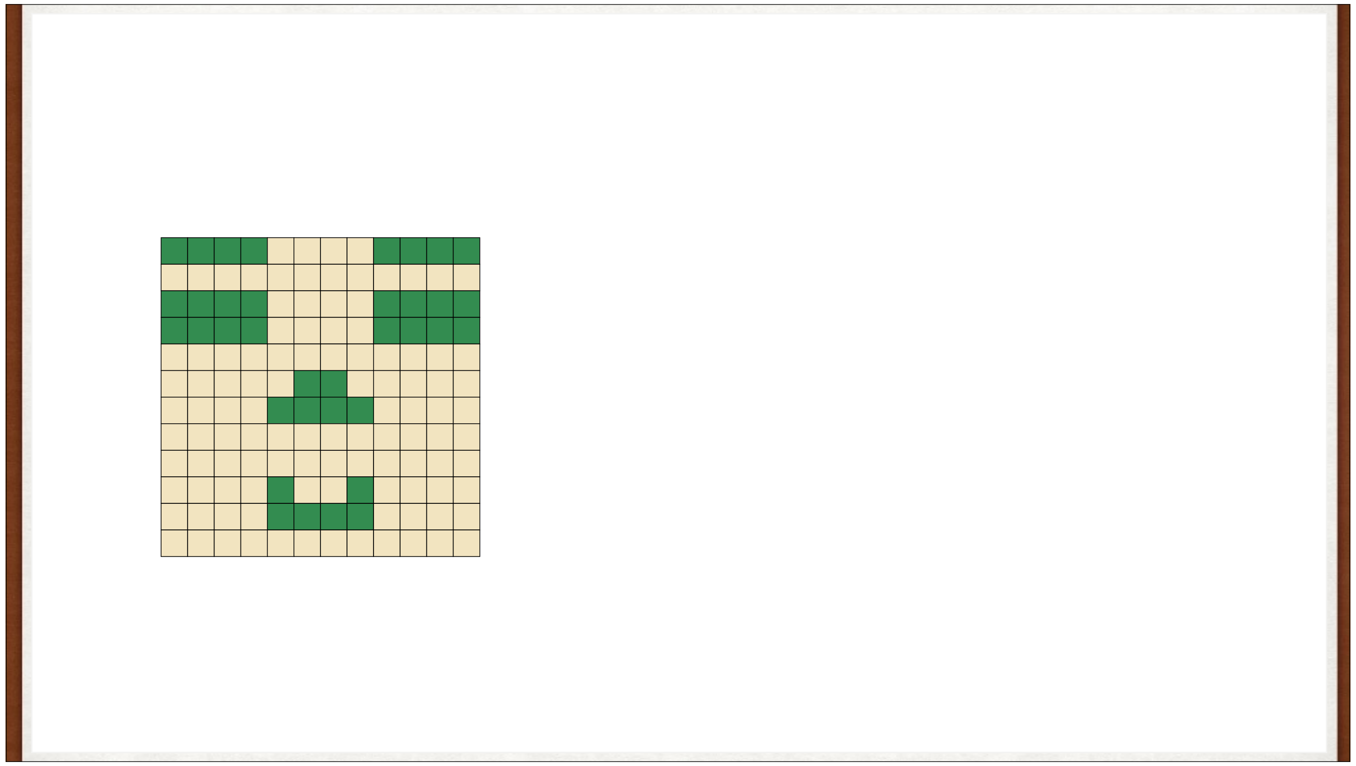


…can be matched with a filter Q, for quad.

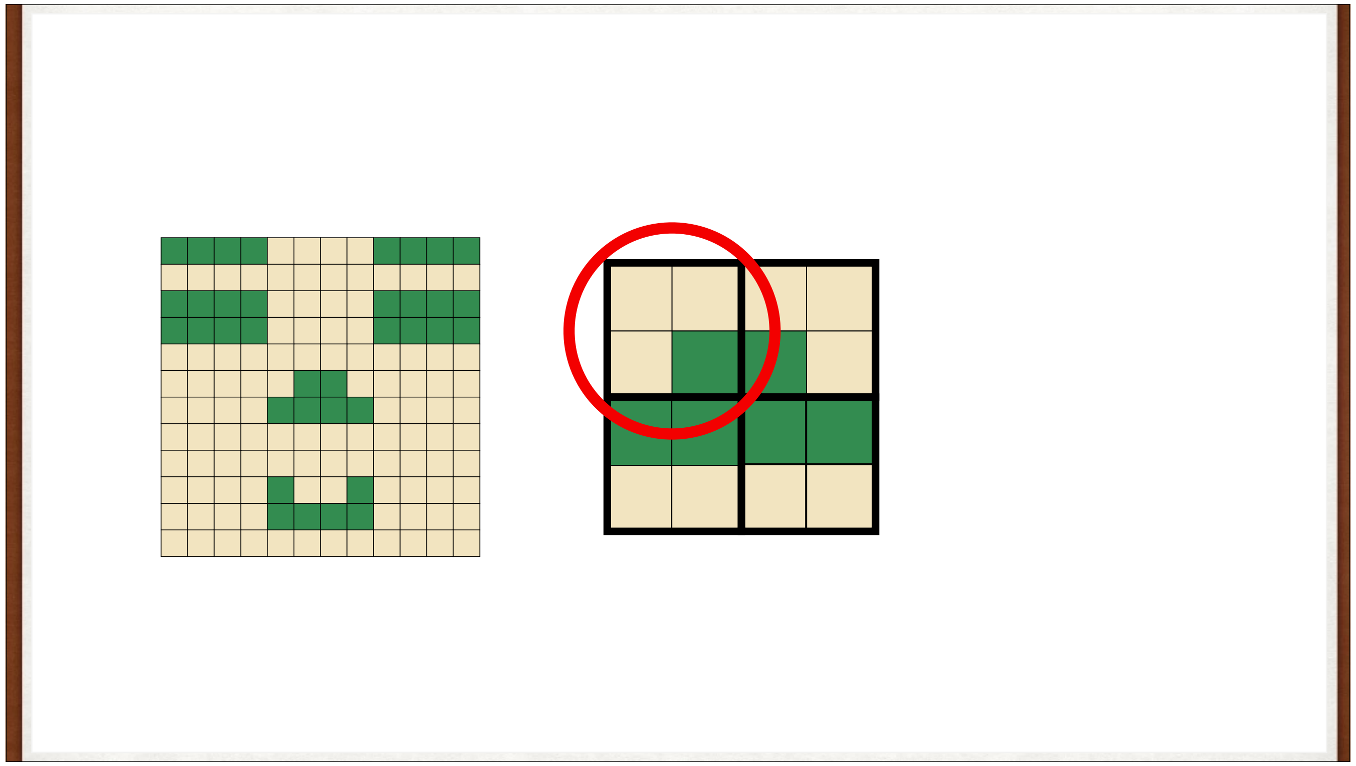


T	T
Q	Q

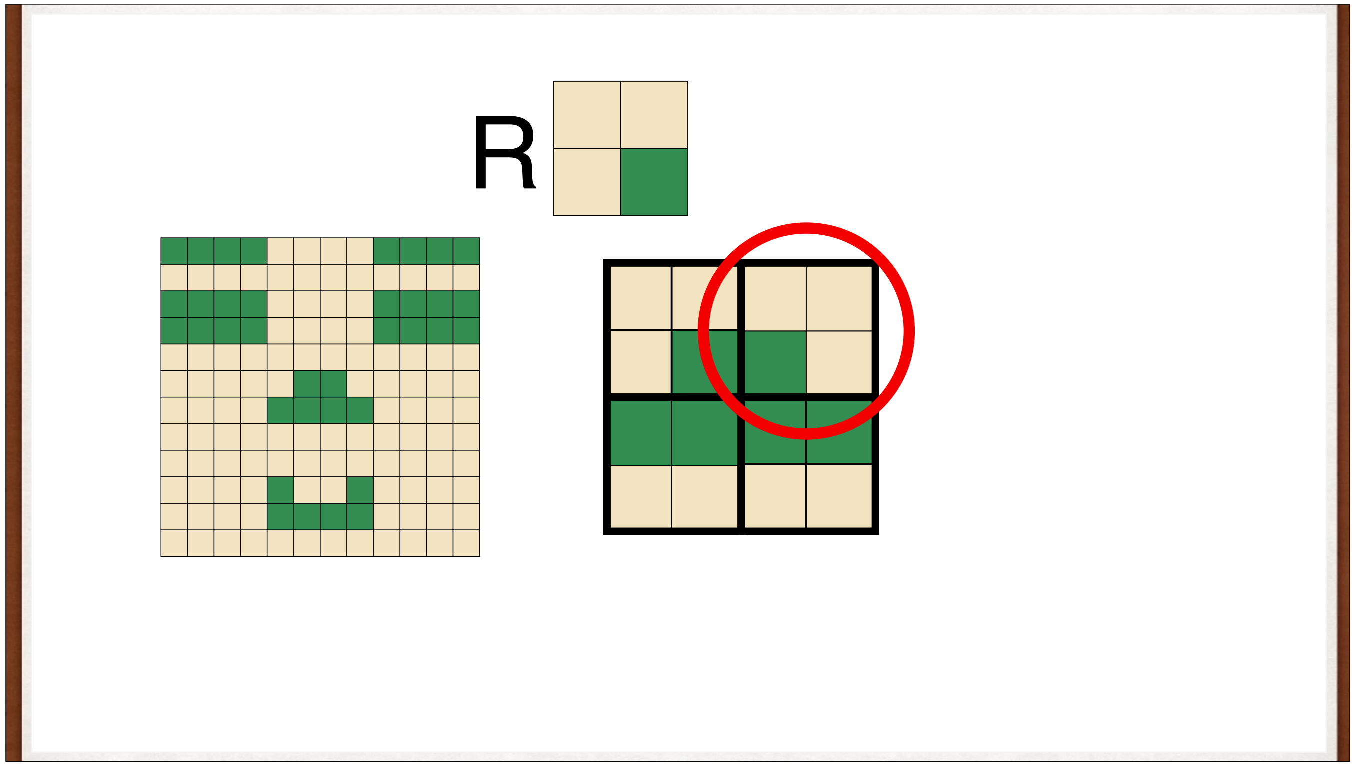
So the eye filter can be satisfied if we find the T and Q filters form this pattern.



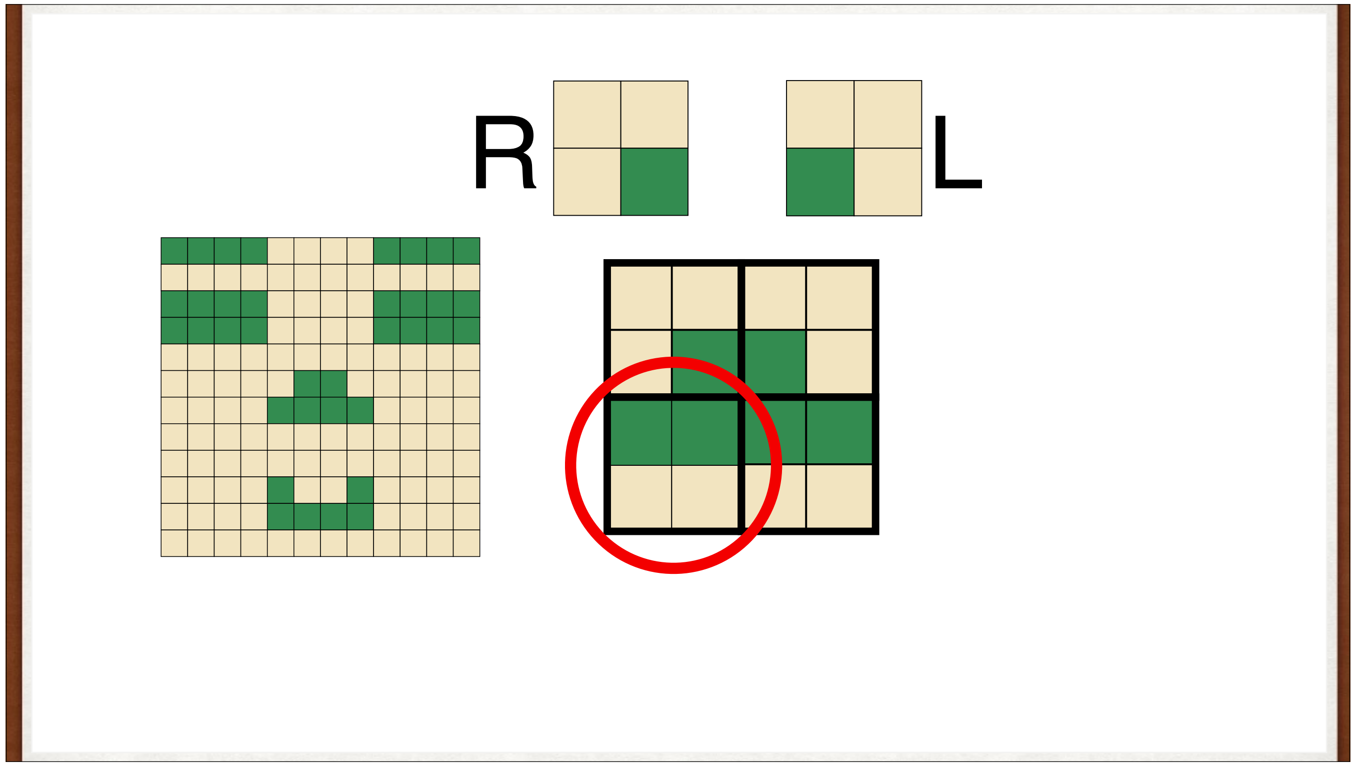
Next let’s check the nose.



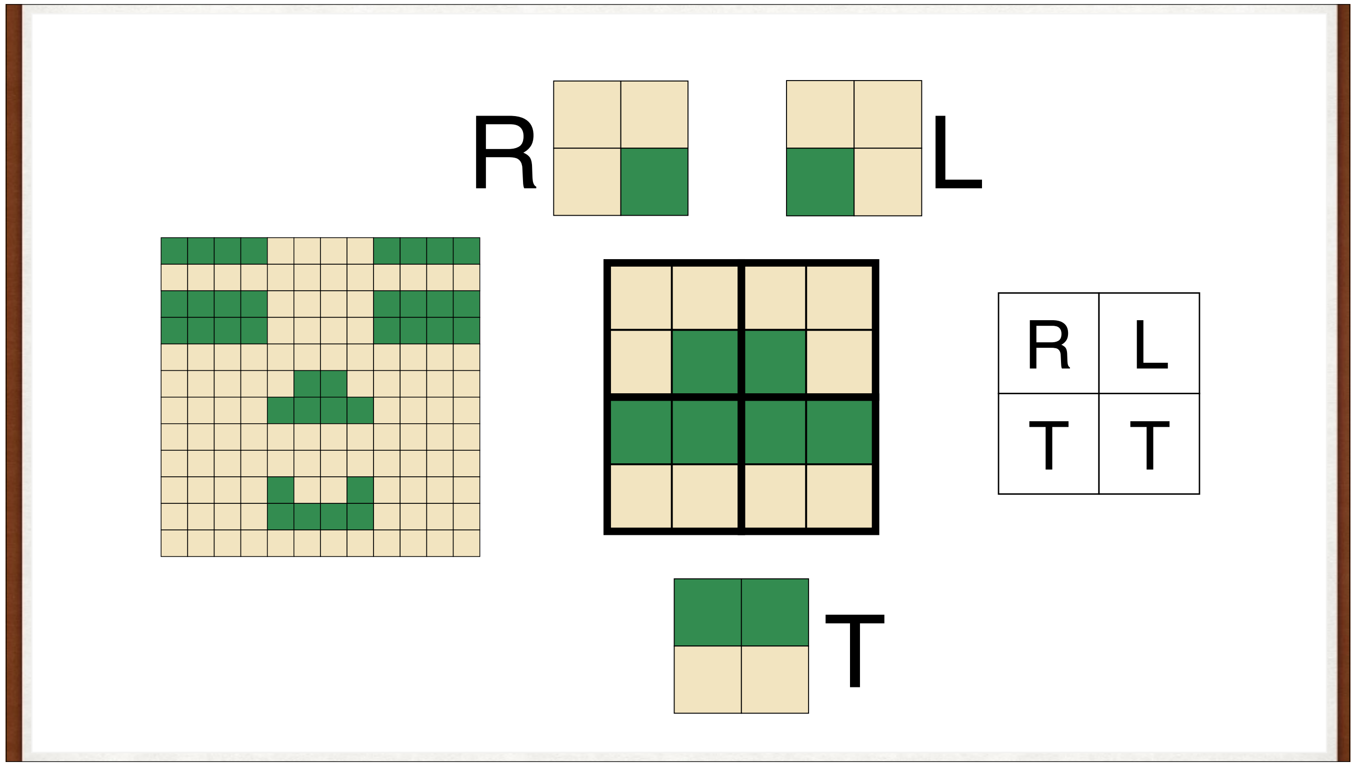
The upper left cell has just one “on” pixel



in the lower right, so let’s call it R. The cell in the upper-right has one “on” pixel in the lower left,

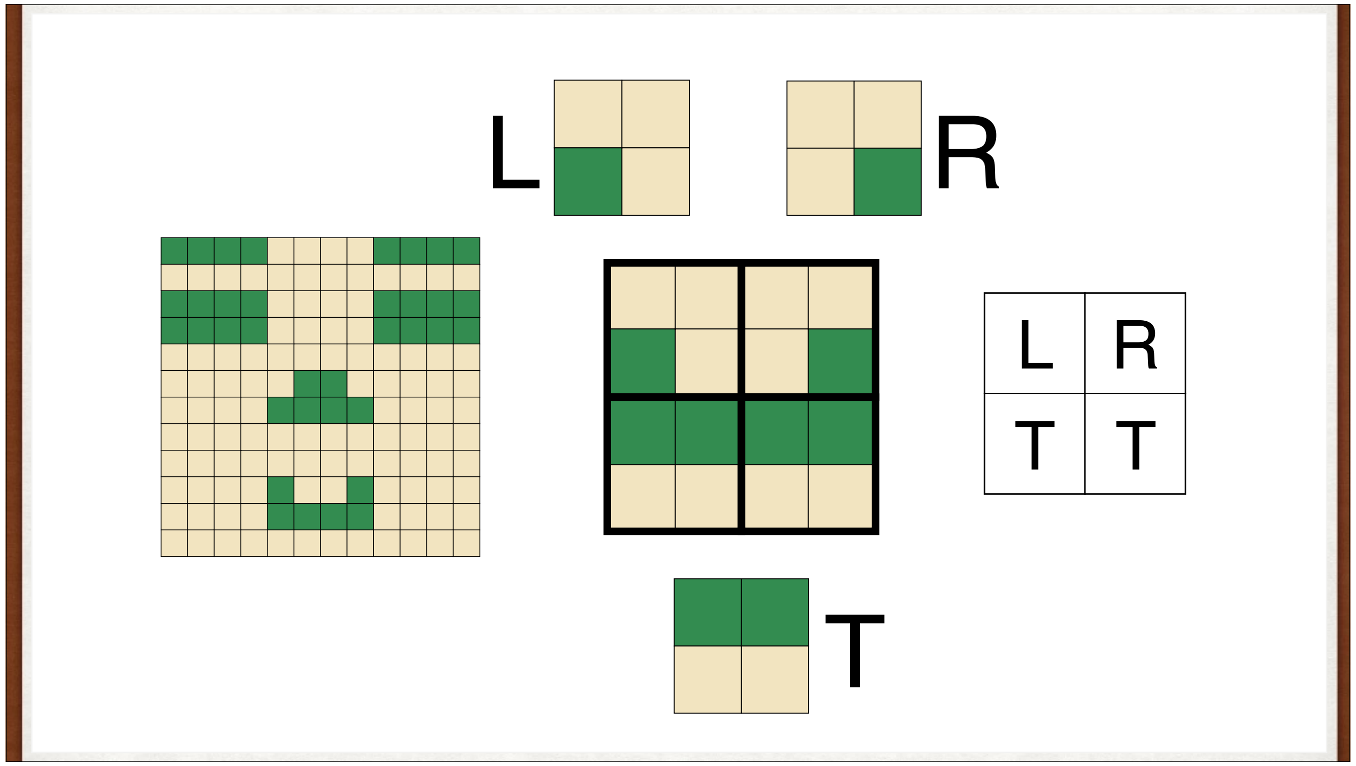


So we can call it L. And either cell on the bottom



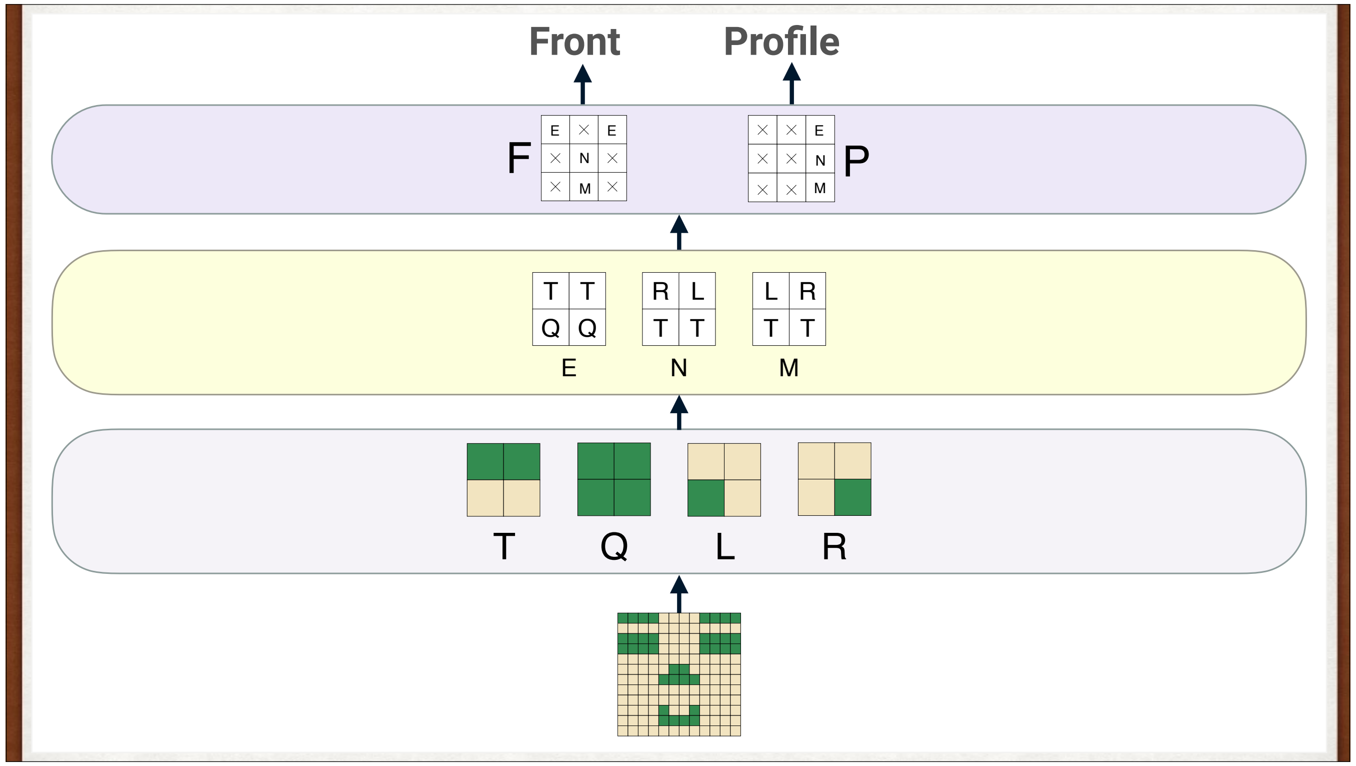
R	L
T	T

is just T from before. So the nose needs only find this pattern of matched 2x2 filters.

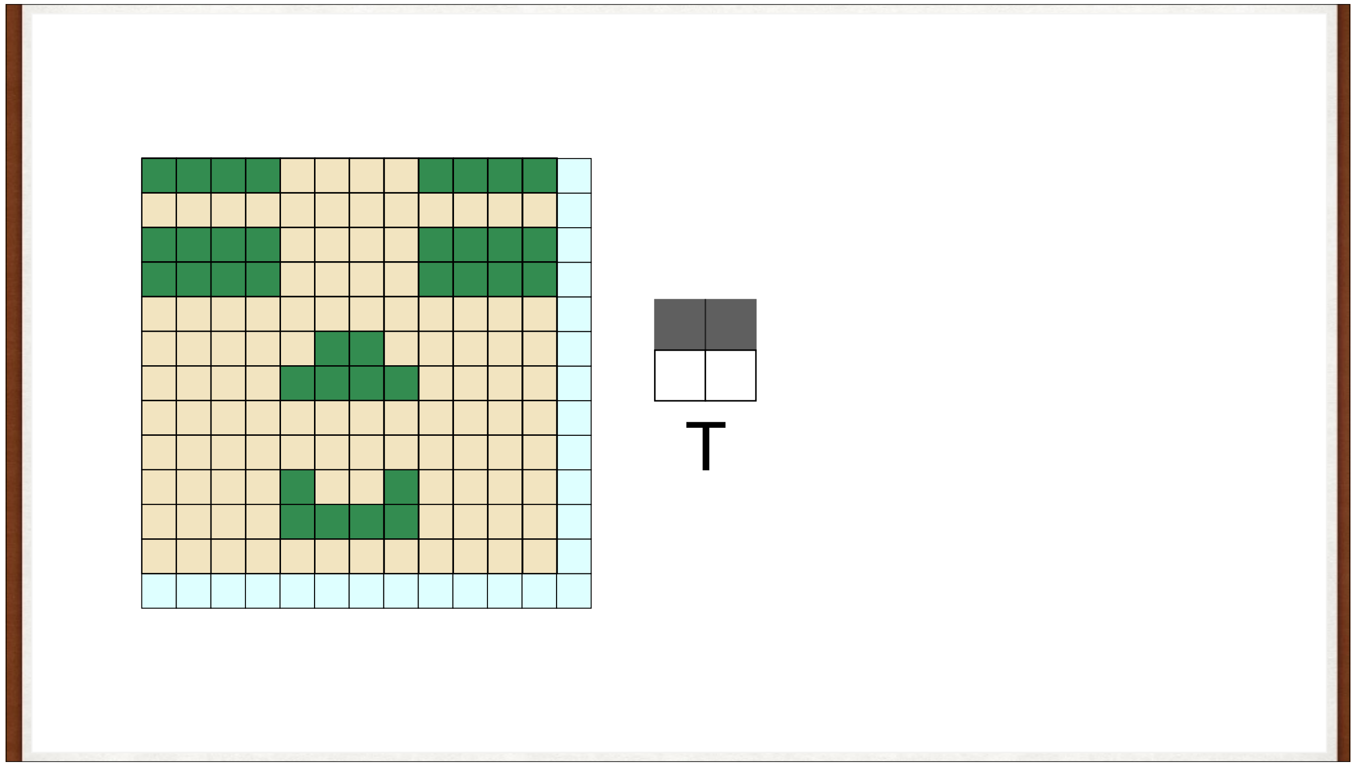


L	R
T	T

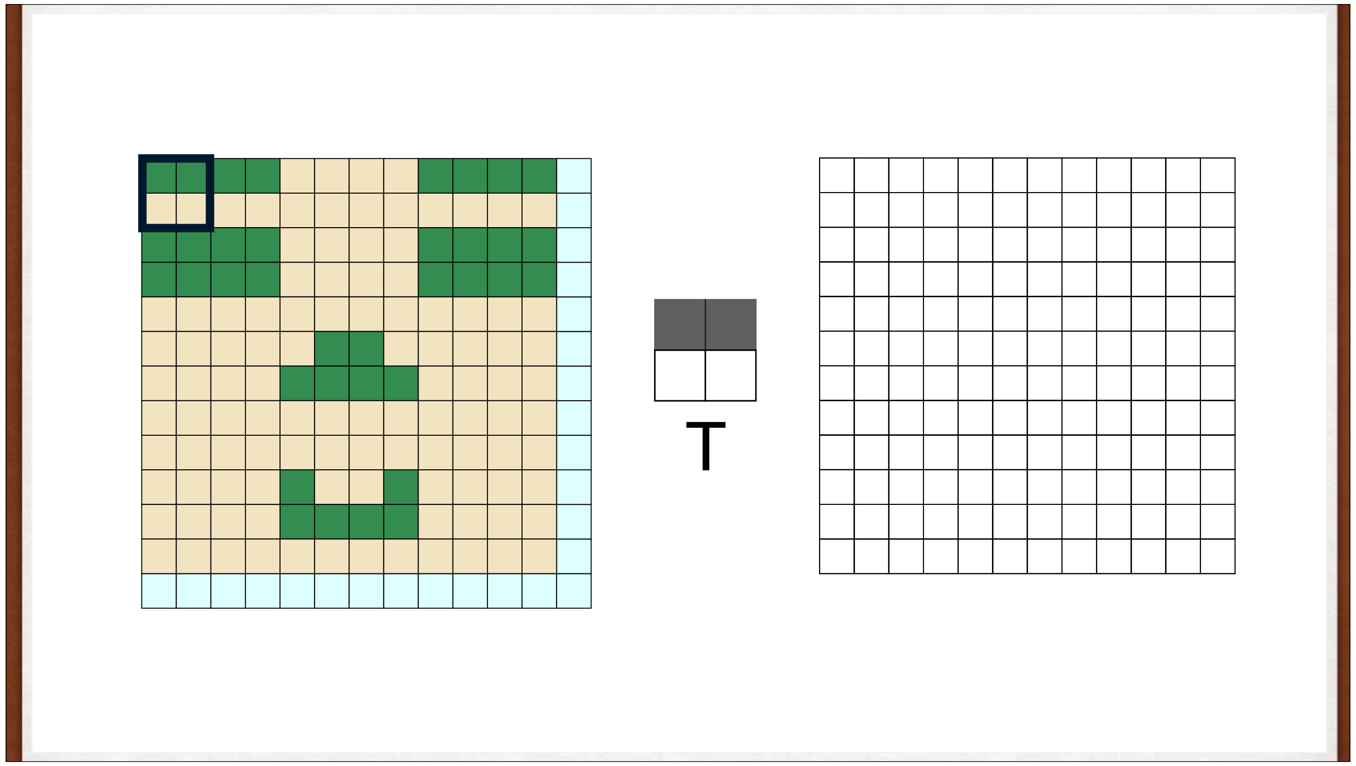
The mouth is just like the nose, only L and R are reversed.



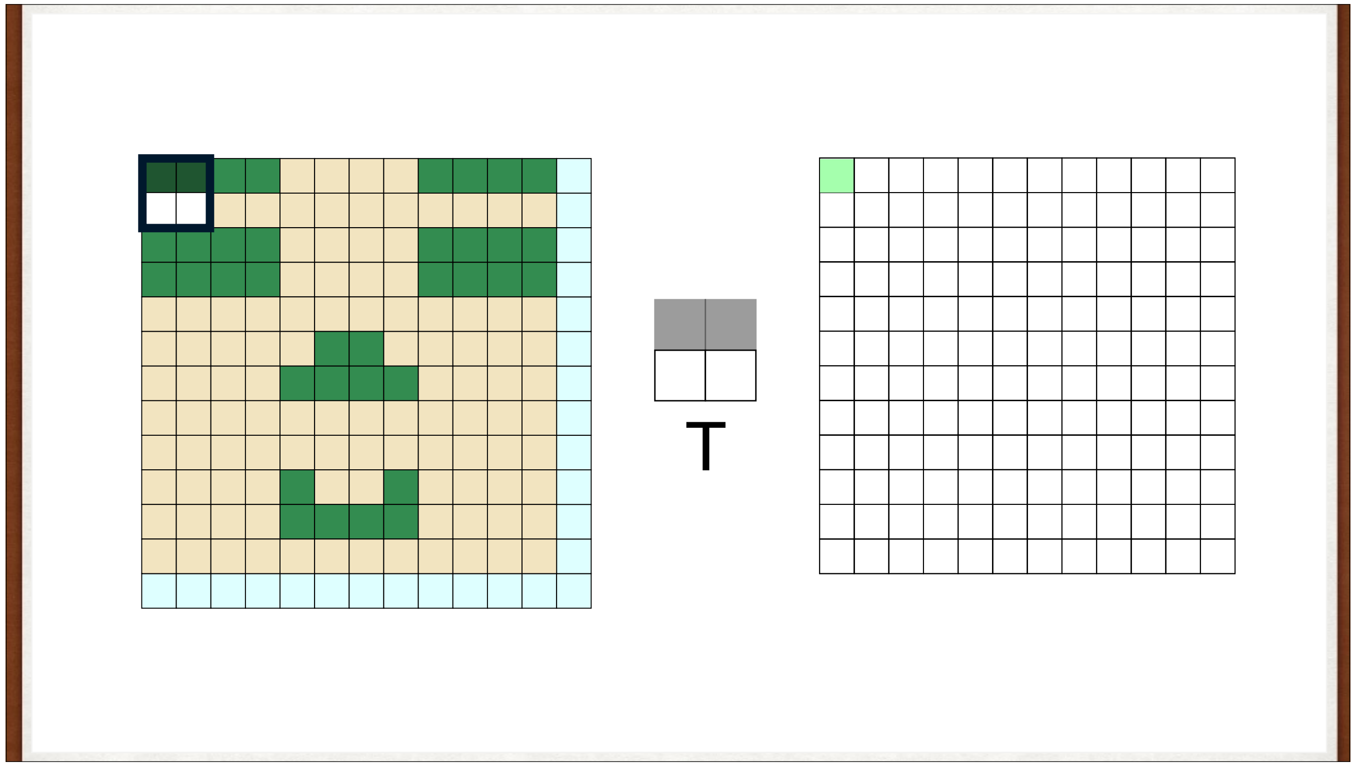
The beauty of the hierarchy. A sequence of banks (or layers) of filters, each finding patterns from the predecessor.



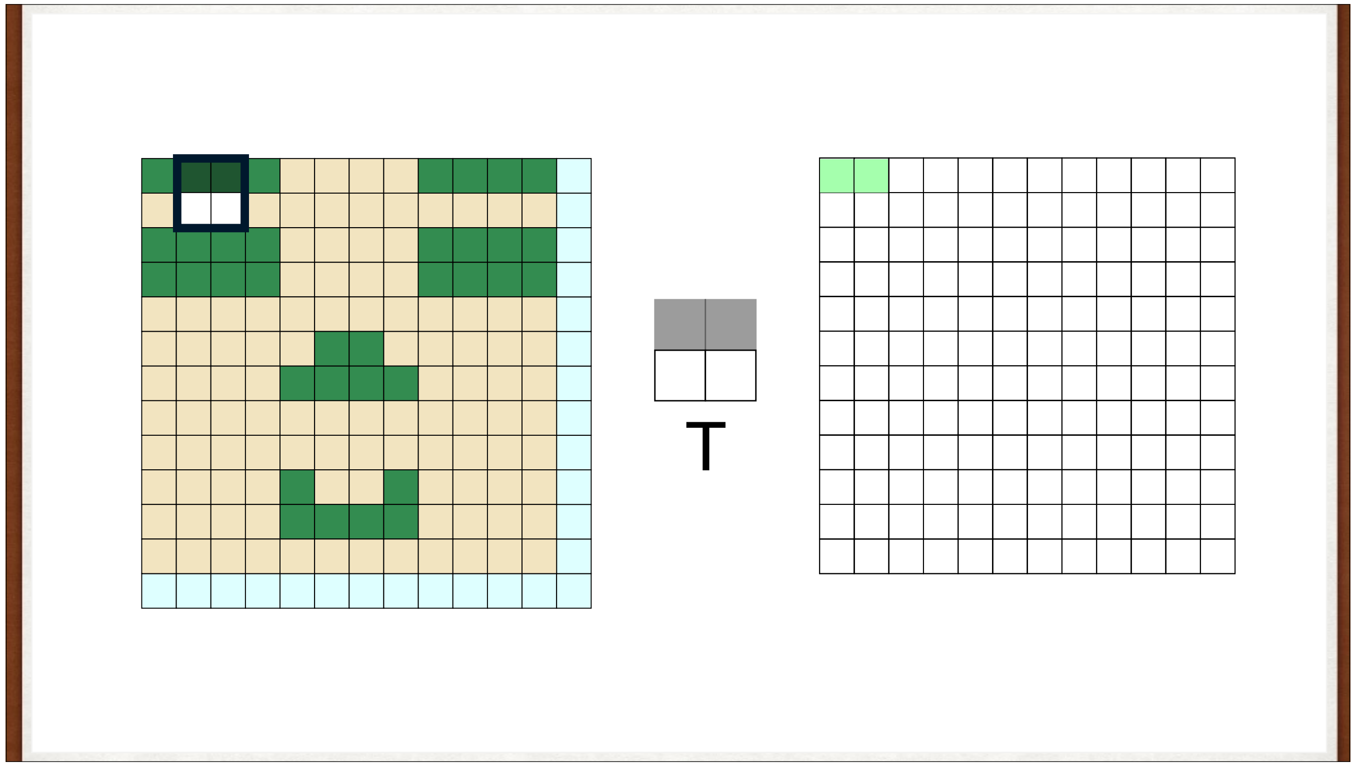
Let’s run the low-level T filter on the input image.



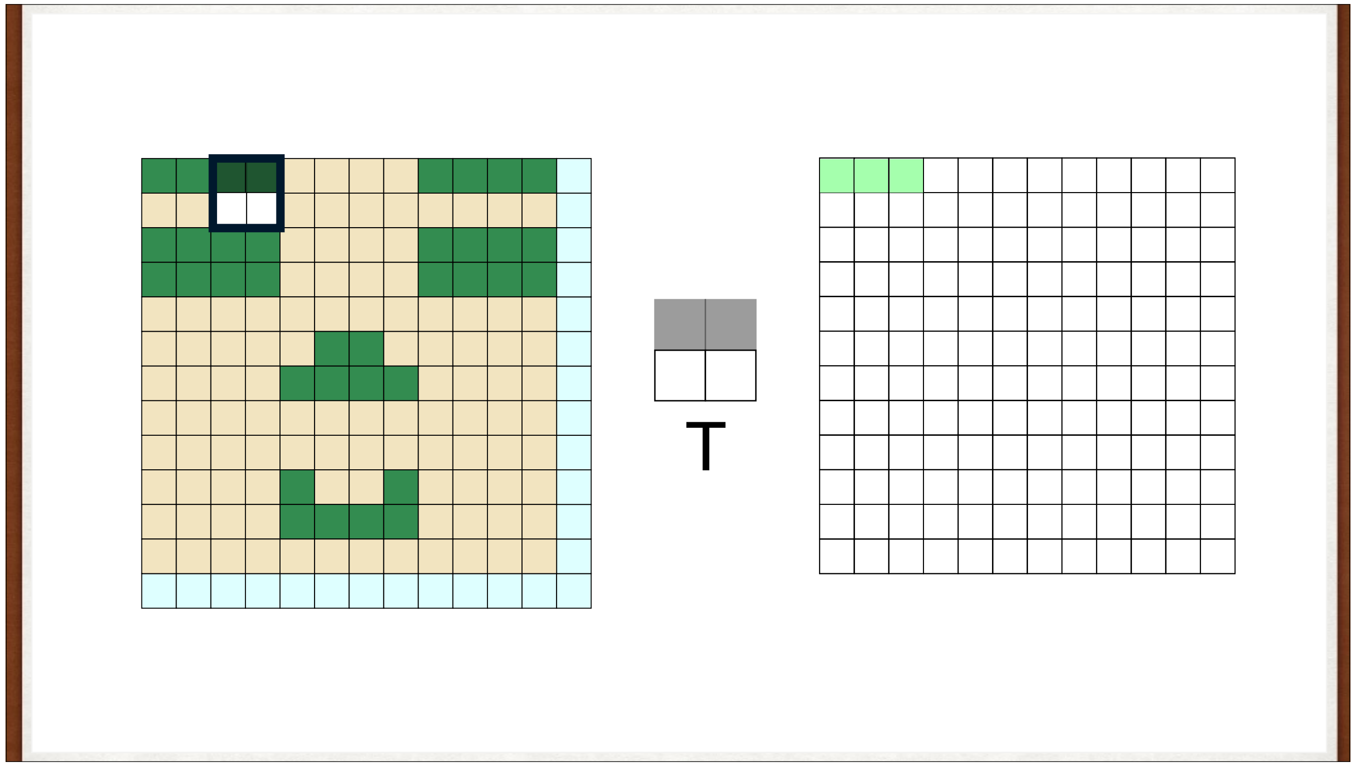
Starting at the upper-left corner…



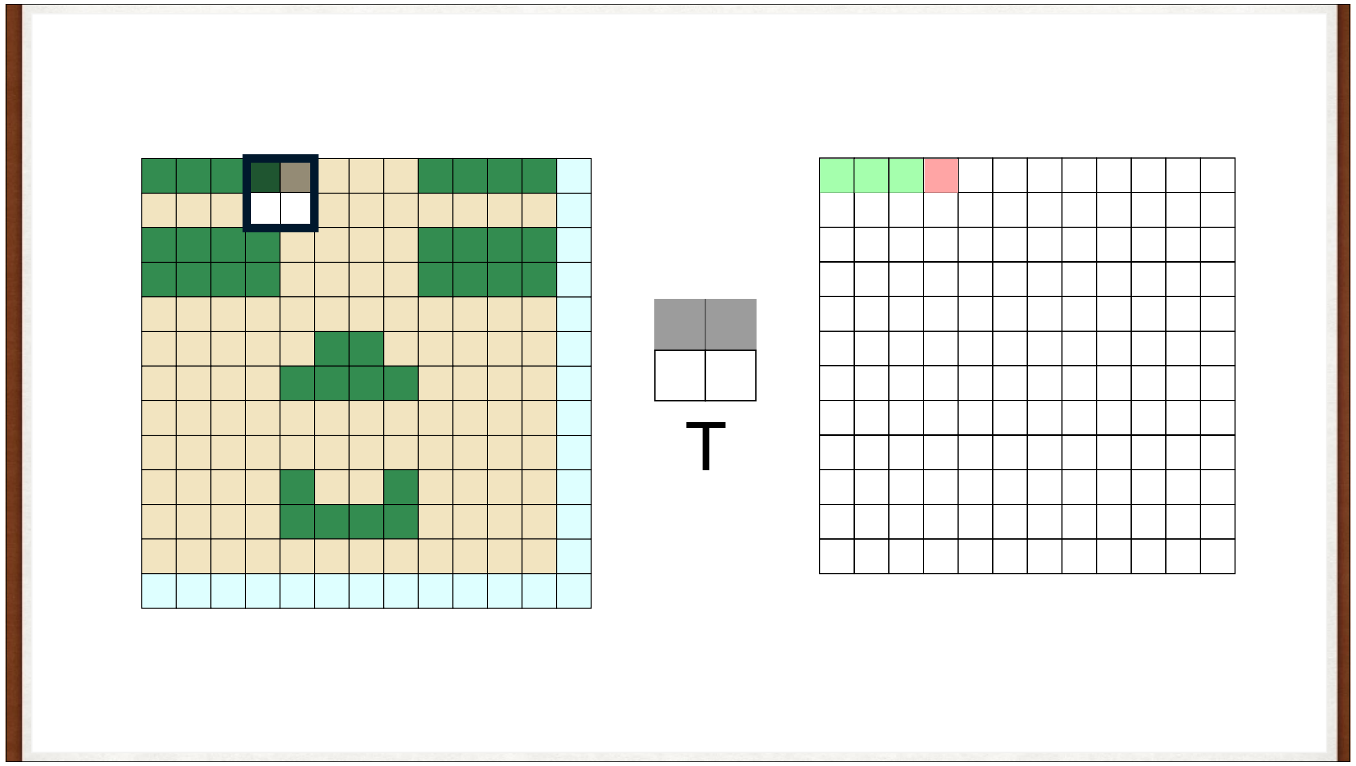
We find a match.



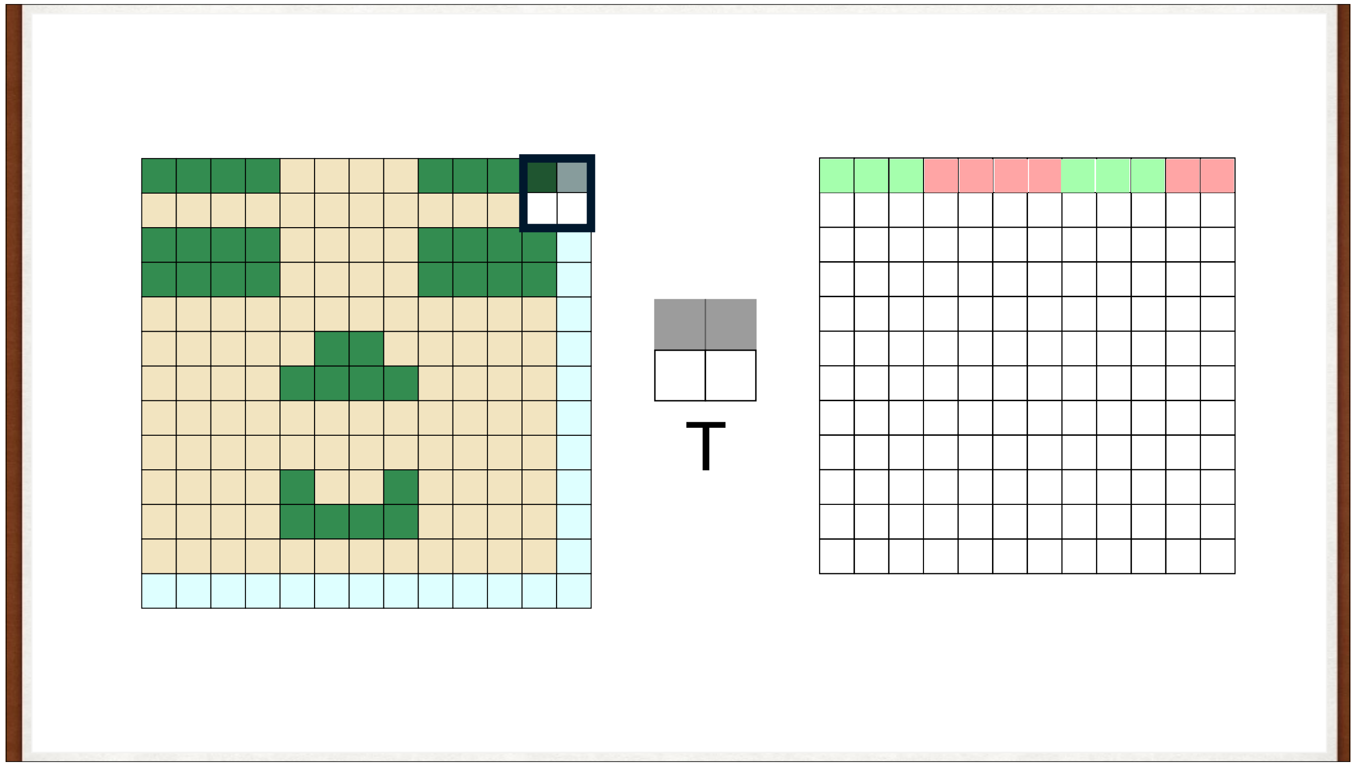
Another match.



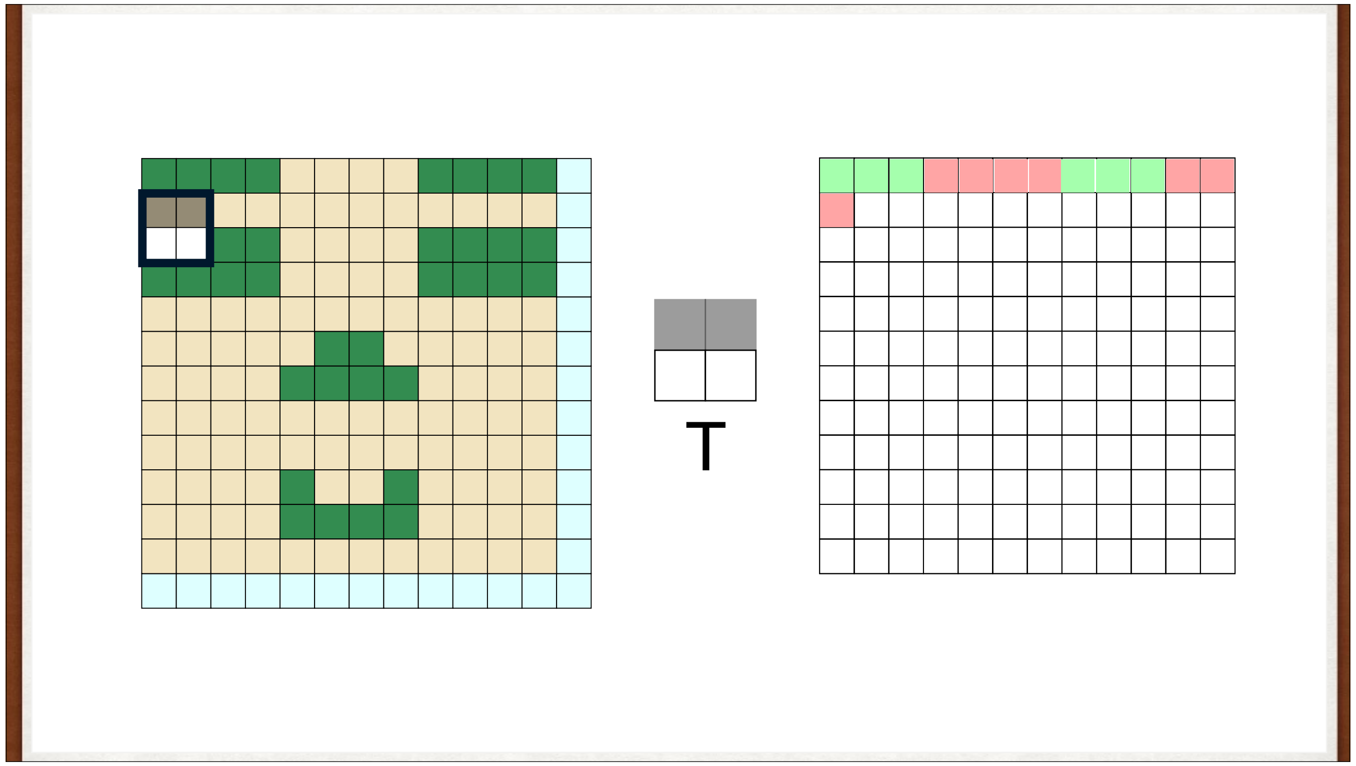
Another match.



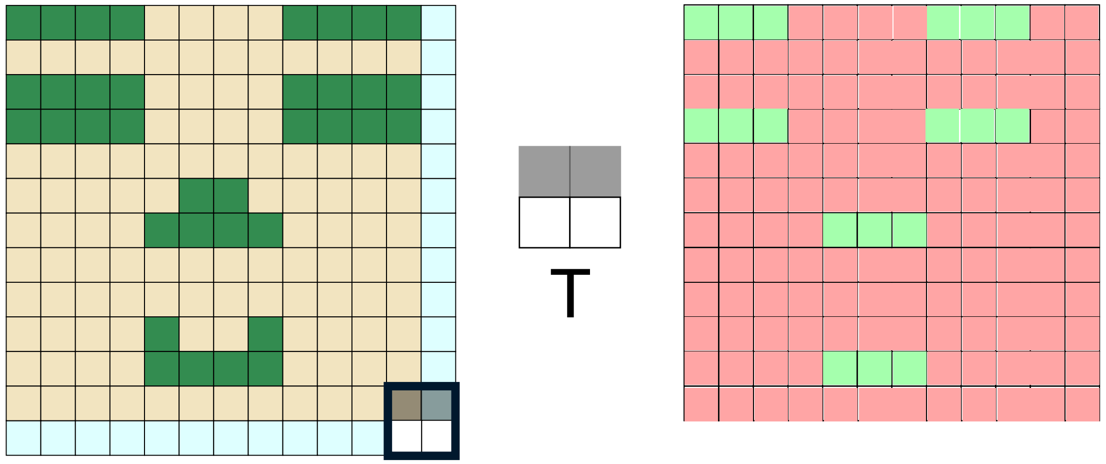
No match here.



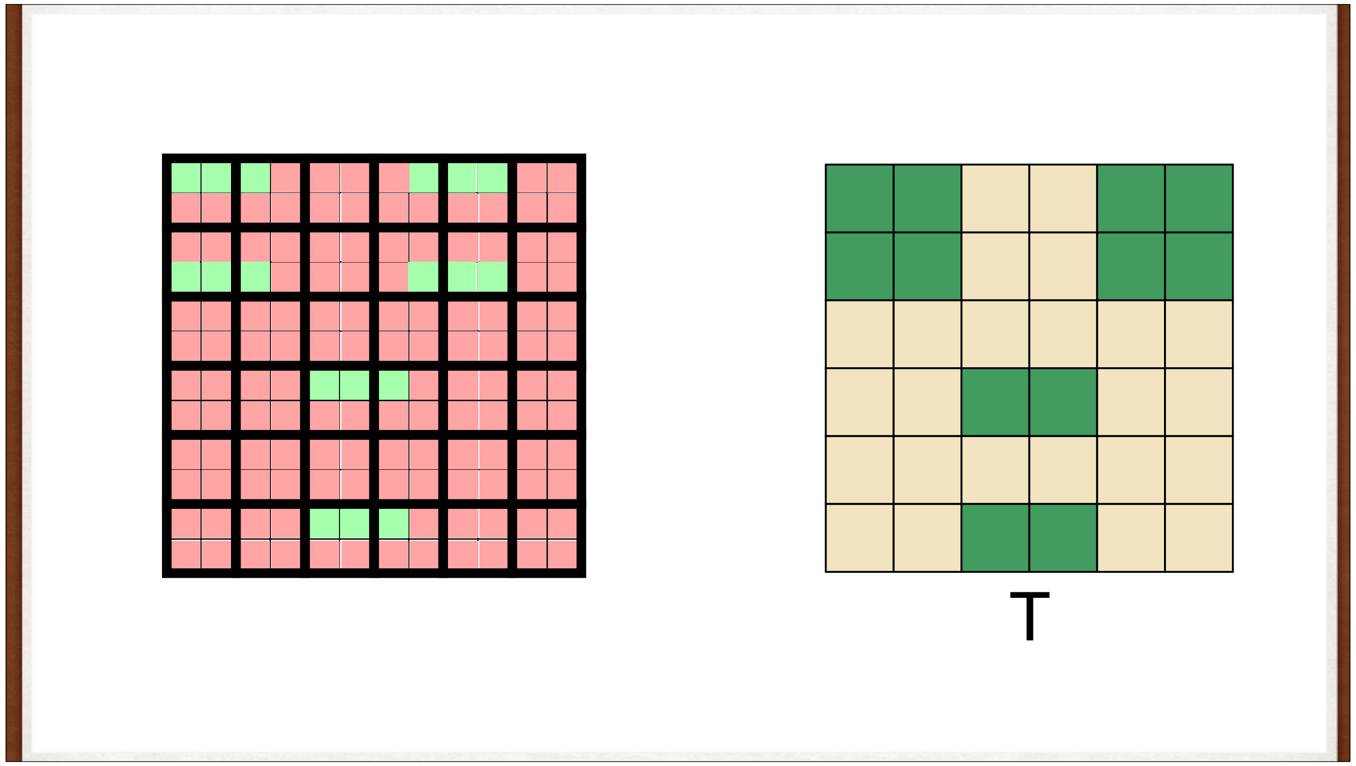
The first row.



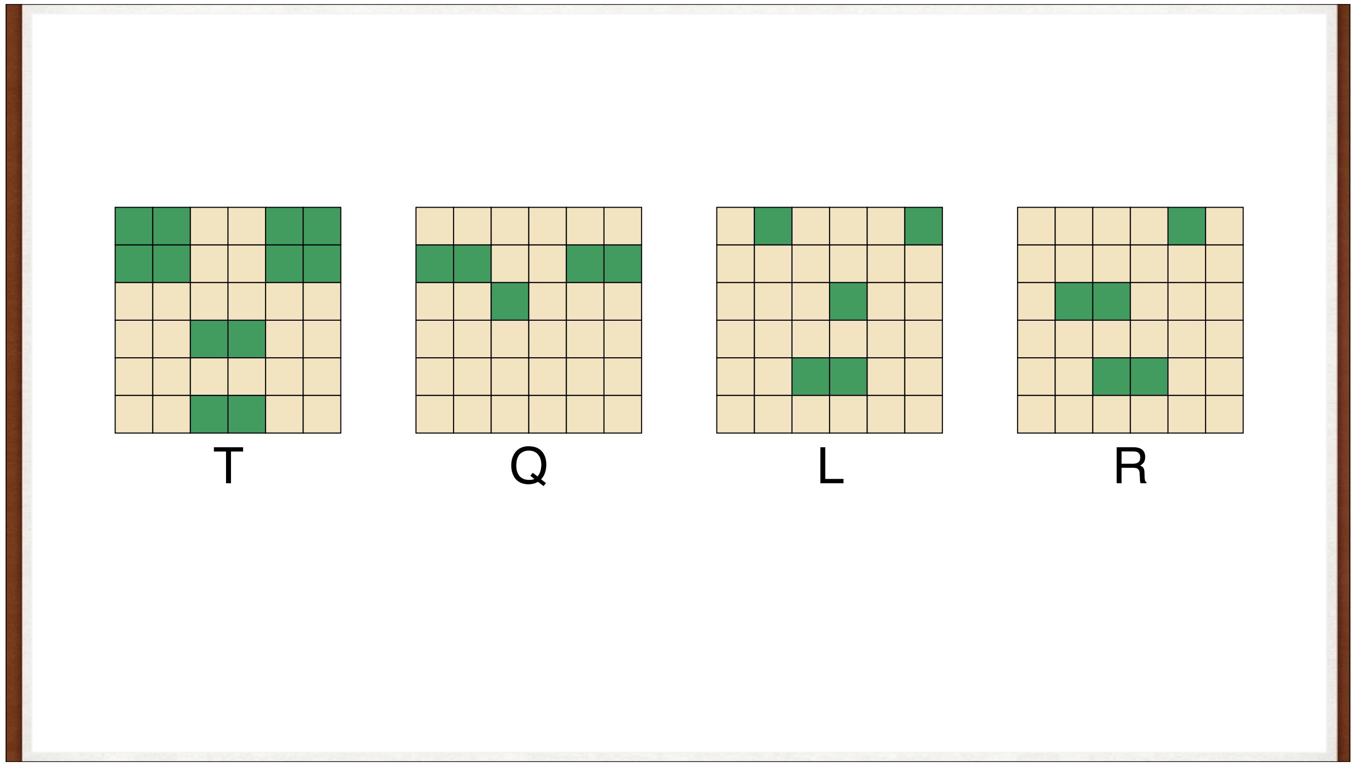
Starting the second row, no match.



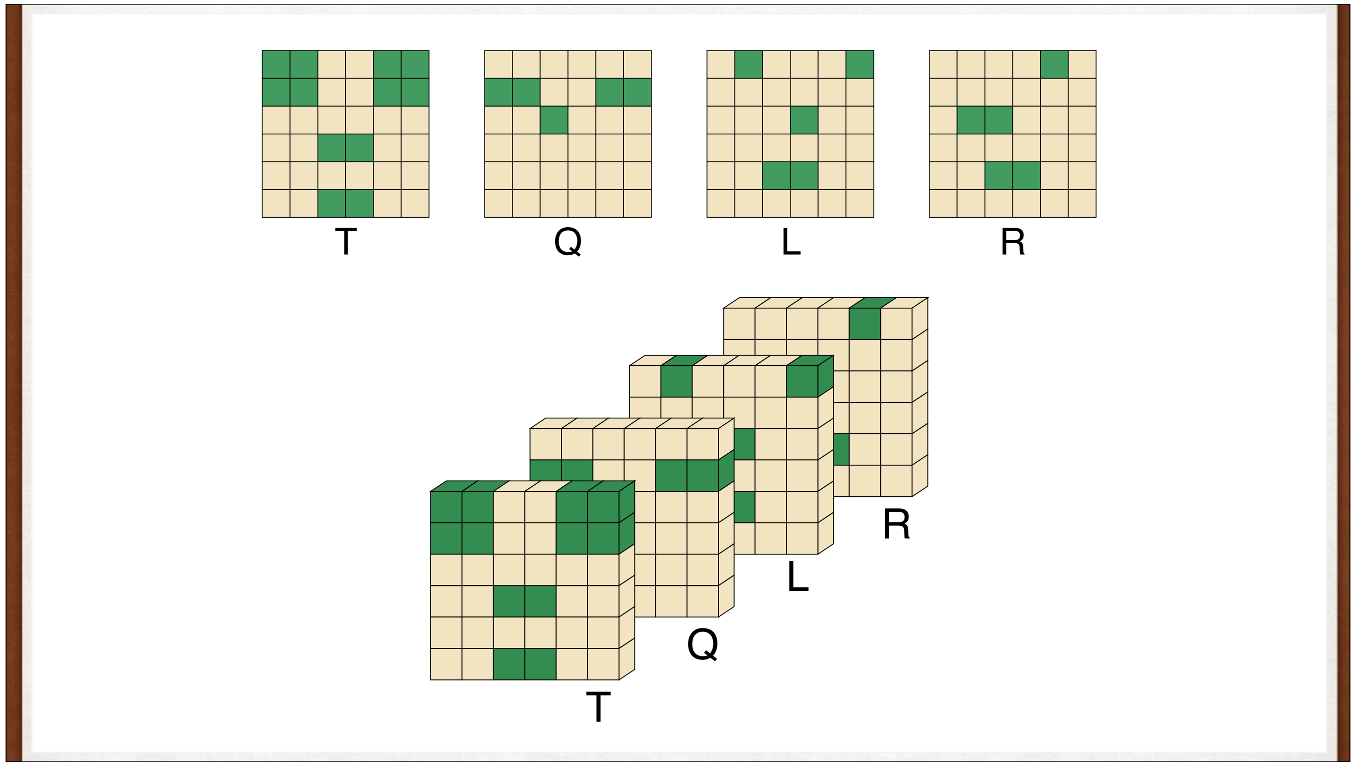
The final results for every input pixel the NW corner of T matches the image.



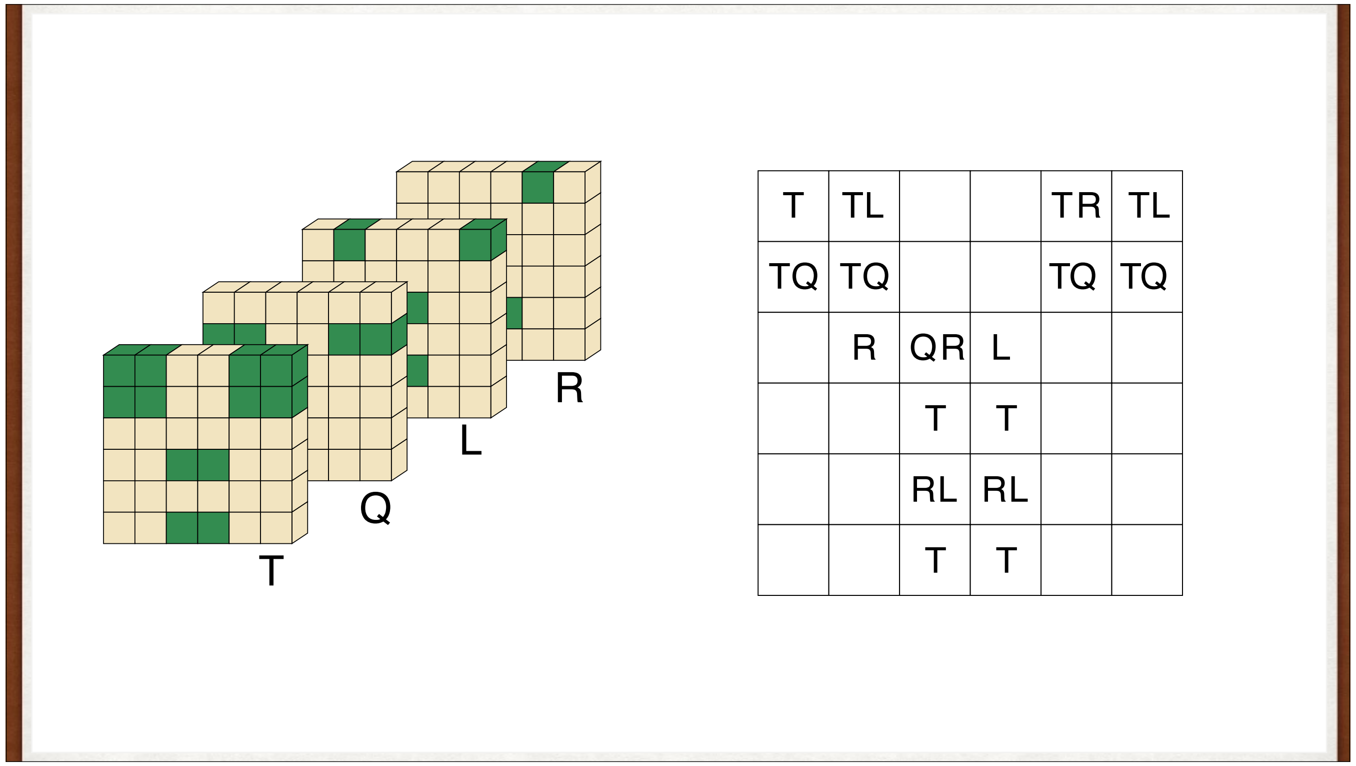
Let’s pool that using 2x2 blocks to get a 6x6 output for the next layer of filters. The next layer doesn’t care about the pixel-level placement of the T matches. This gives us some spatial invariance in the features we’re looking for (like the mouth moving up or down a pixel).



The four filters we designed and the results of pooling their responses to the input.

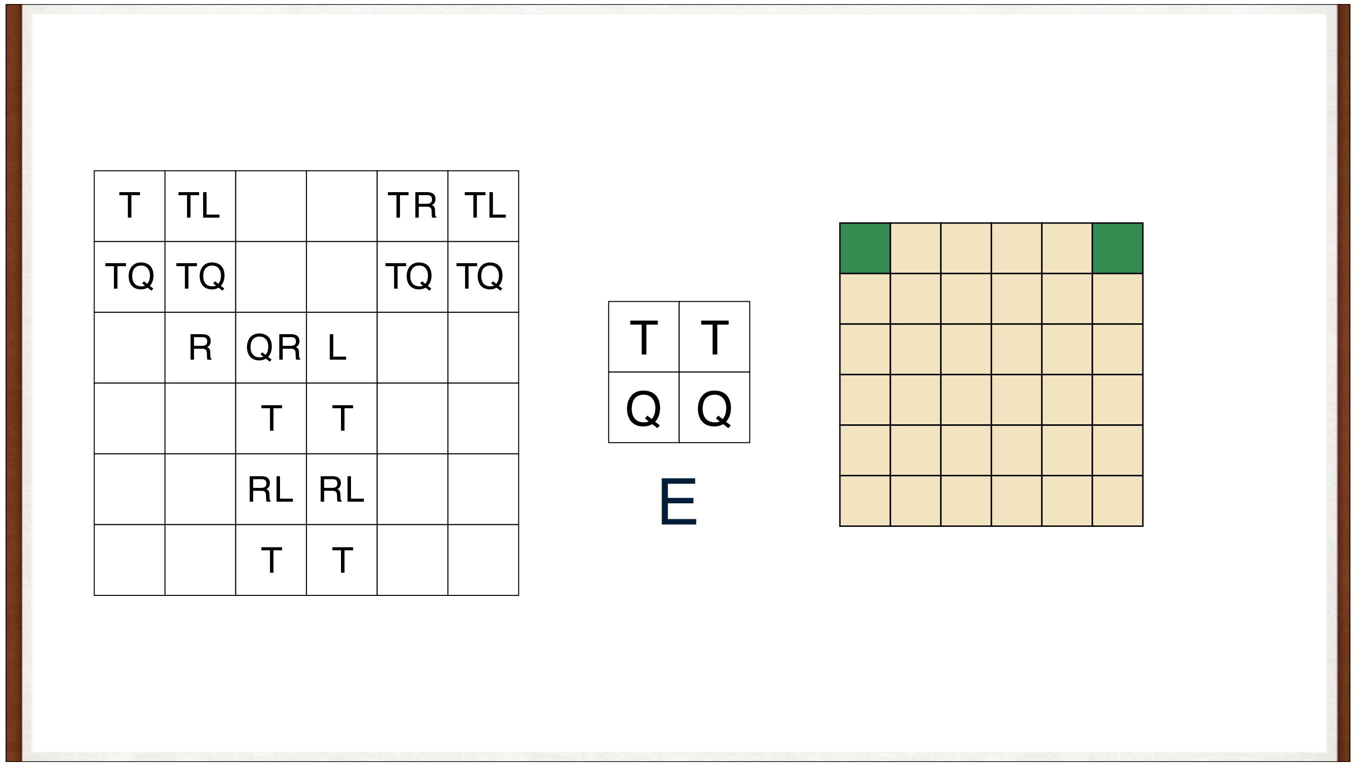


We can arrange these outputs in a tensor that’s 6x6x4, but that’s hard to picture, so instead…



T	TL			TR	TL
TQ	TQ			TQ	TQ
	R	QR	L		
		T	T		
		RL	RL		
		T	T		

We’ll just label the elements with all the filters that are on at that location. This only makes sense because everything is binary.



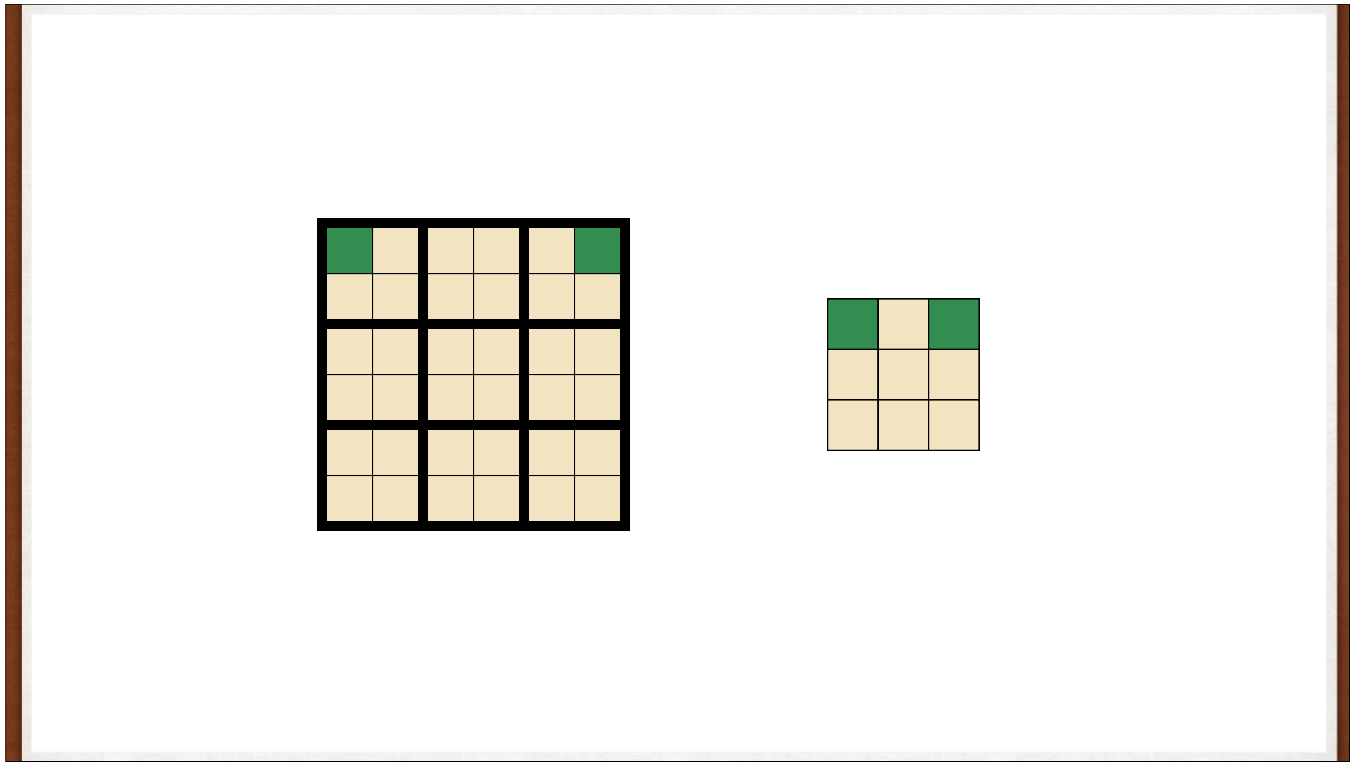
T	TL			TR	TL
TQ	TQ			TQ	TQ
	R	QR	L		
		T	T		
		RL	RL		
		T	T		

T	T
Q	Q

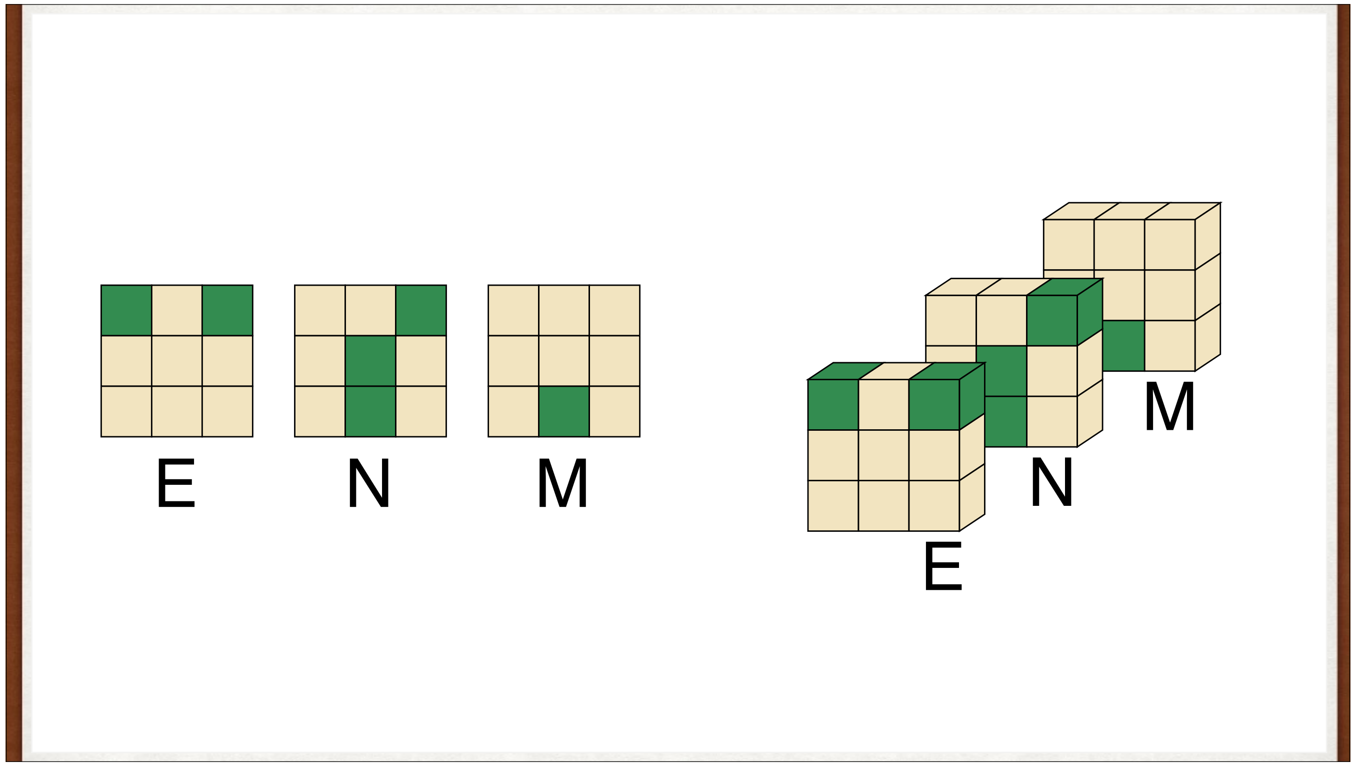
E

🟩					🟩

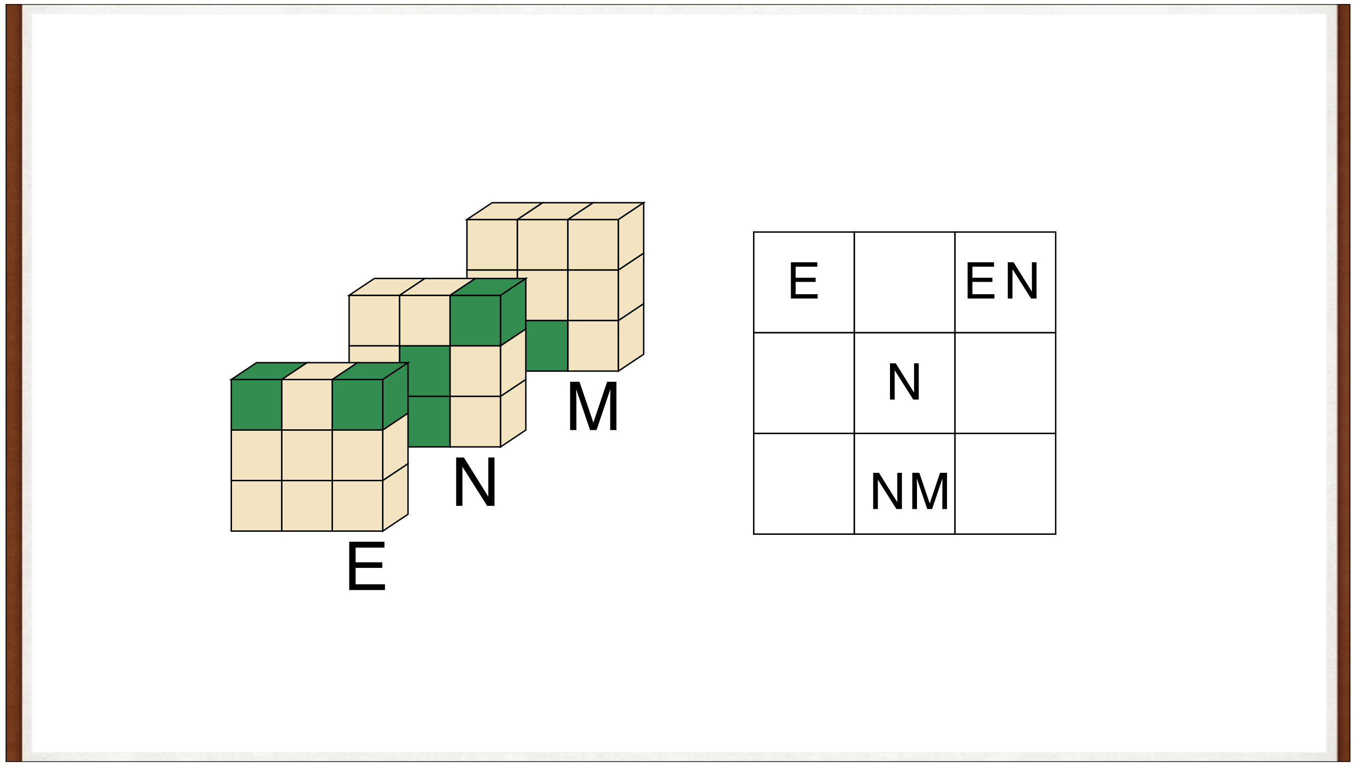
Looking at the pixel-level filters for combinations that make up an eye.



Again, we’ll filter this down in 2x2 blocks, giving us now a 3x3 result.

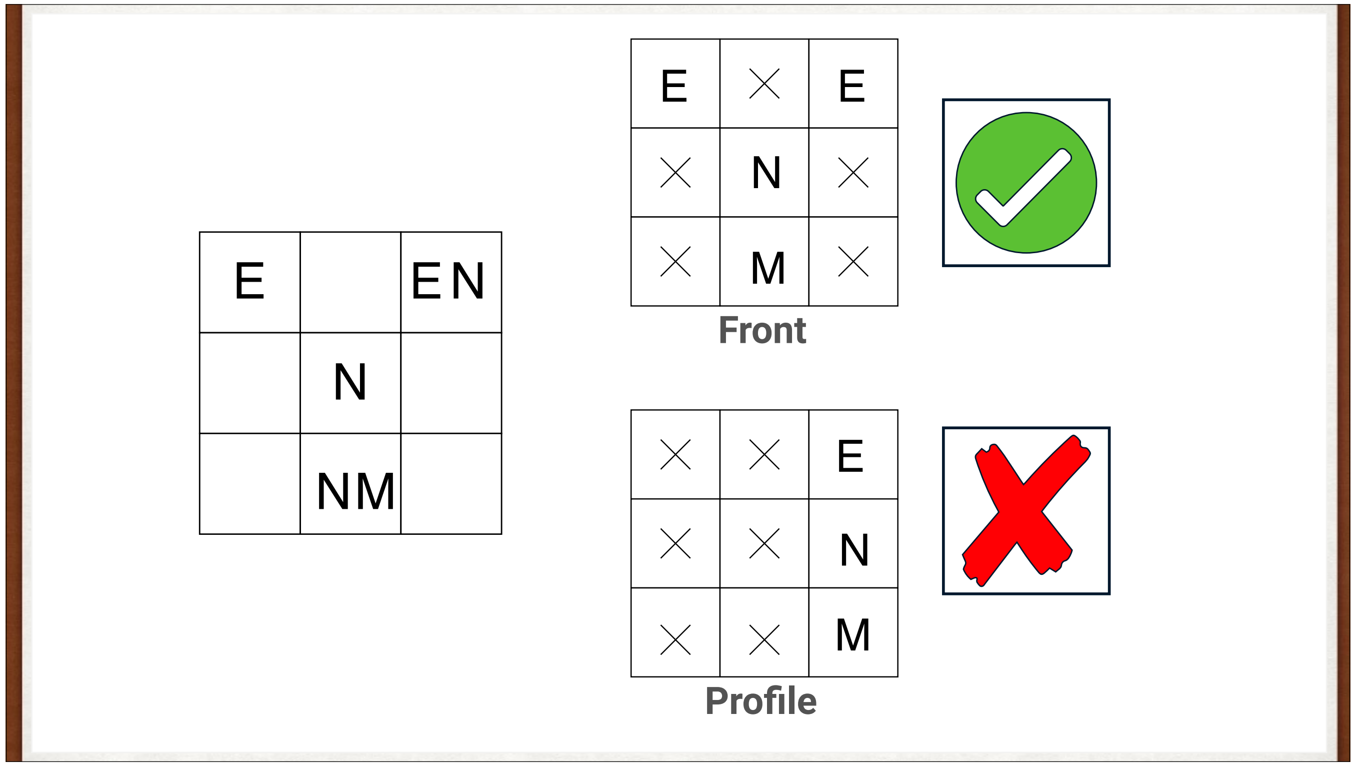


The responses to the eye, nose, and mouth filters. These make a tensor that’s 3x3x3.



E		EN
	N	
	NM	

The output tensor in a more convenient form.



E		EN
	N	
	NM	

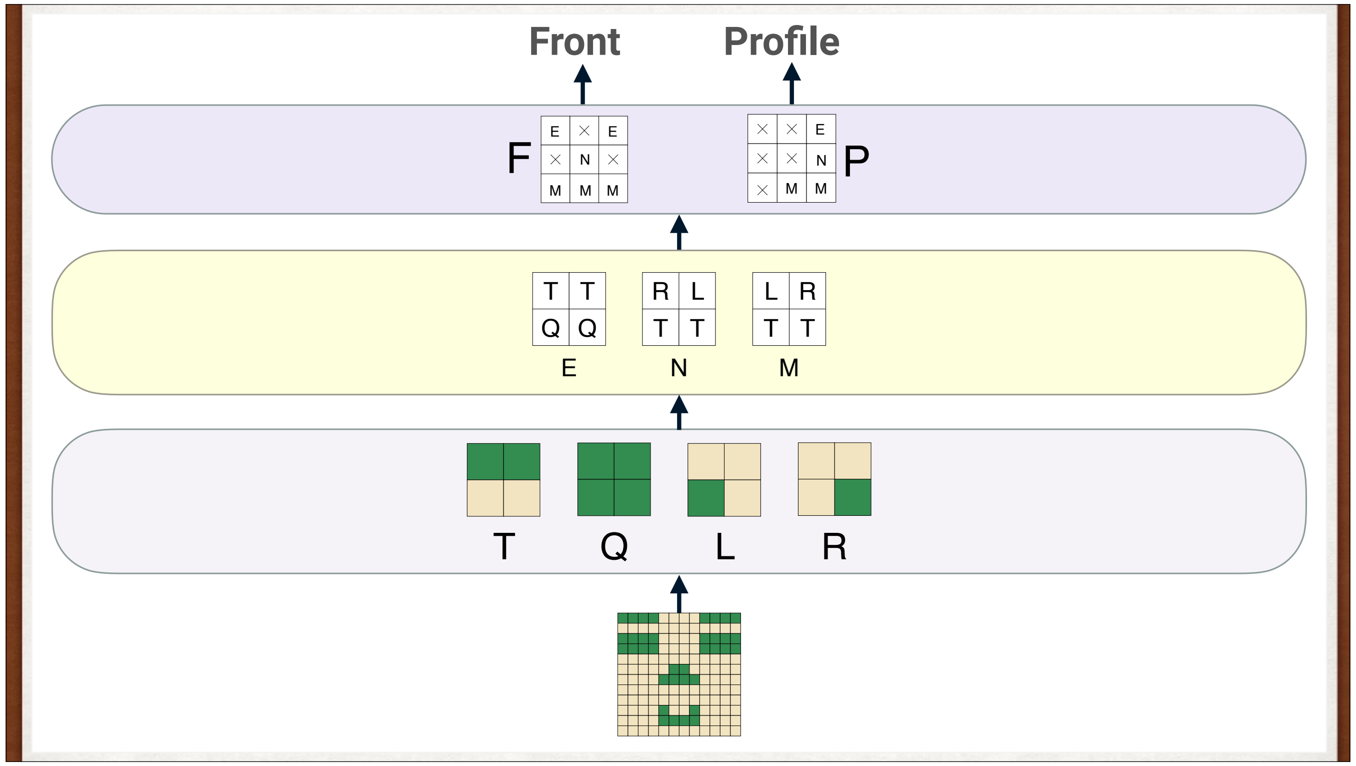
E	×	E
×	N	×
×	M	×

Front ✓

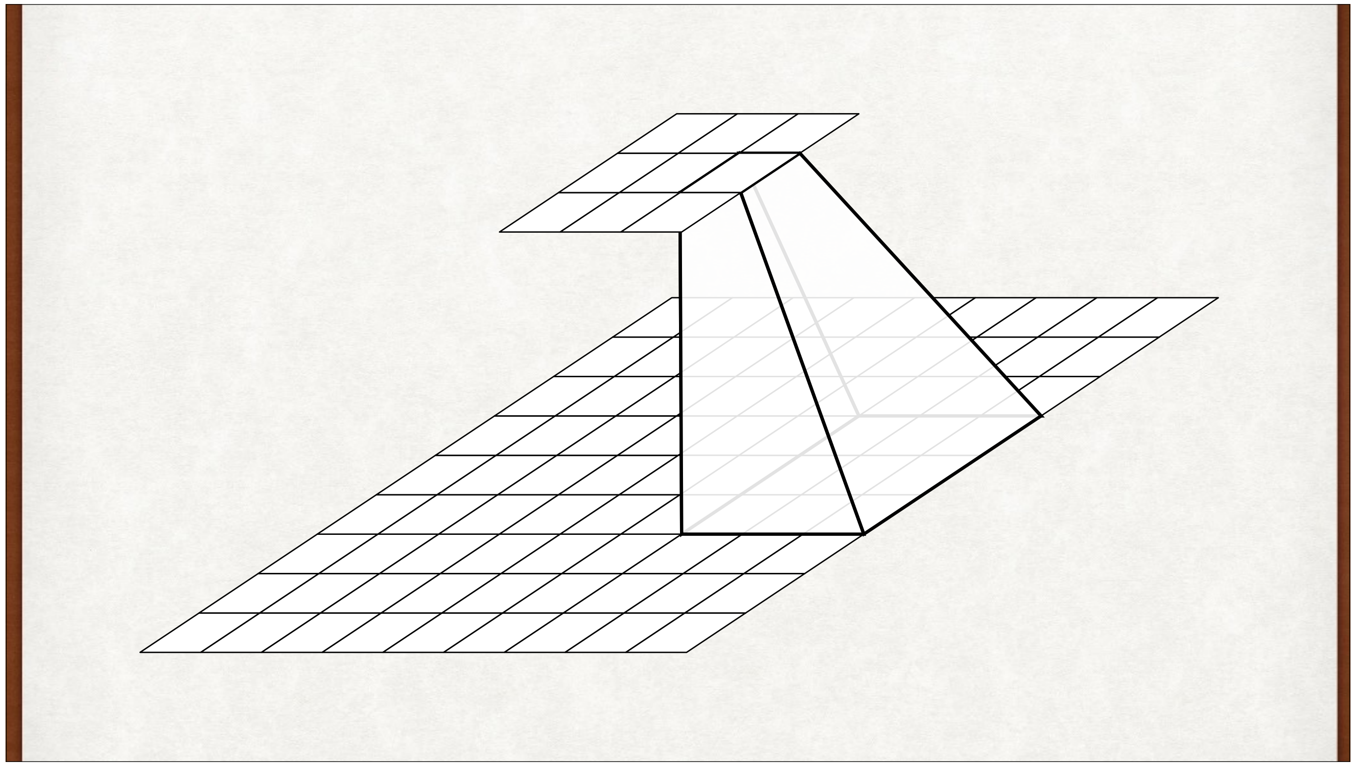
×	×	E
×	×	N
×	×	M

Profile ✗

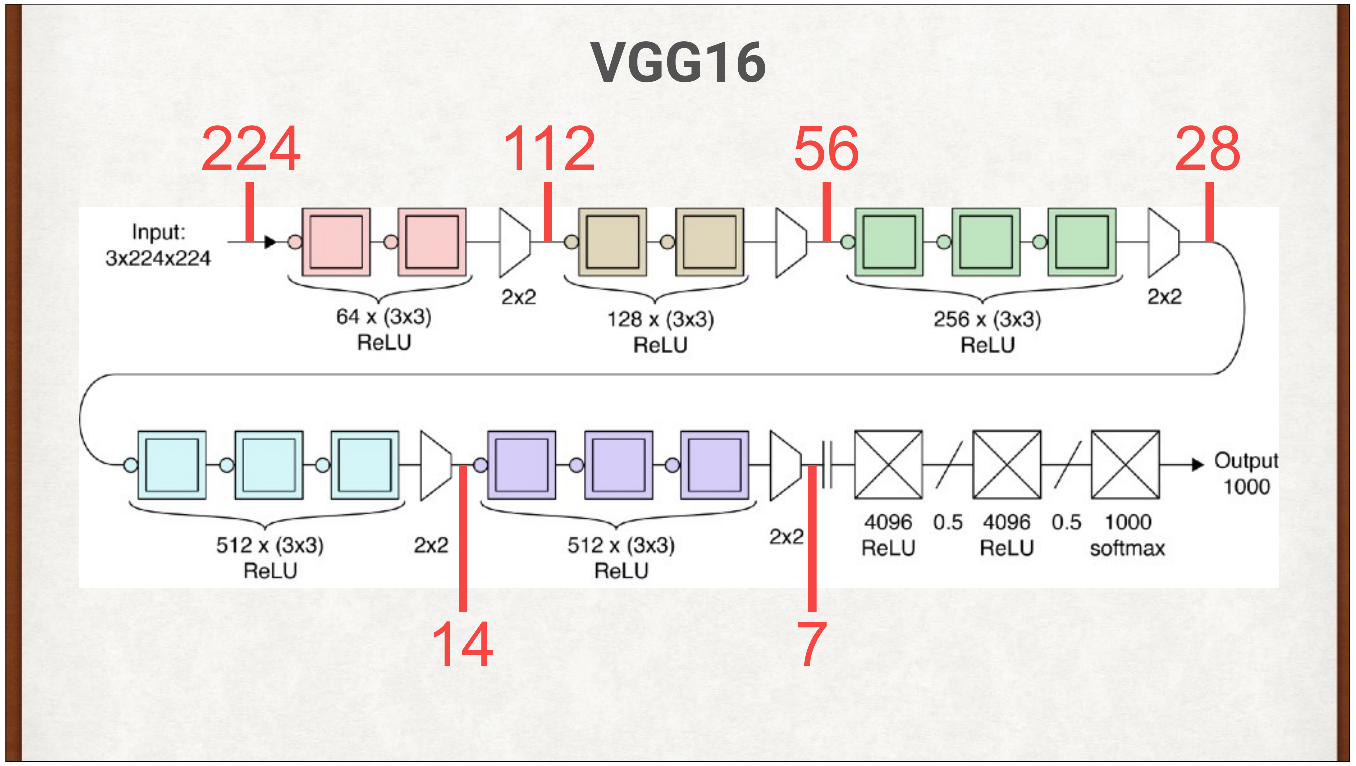
Comparing the output tensor (left) with the filters we’re looking for. The front matches! The profile does not. Recall x means “don’t care”.



The hierarchy. Convolution plus hierarchical filters (managed by pooling) is a powerful combination.

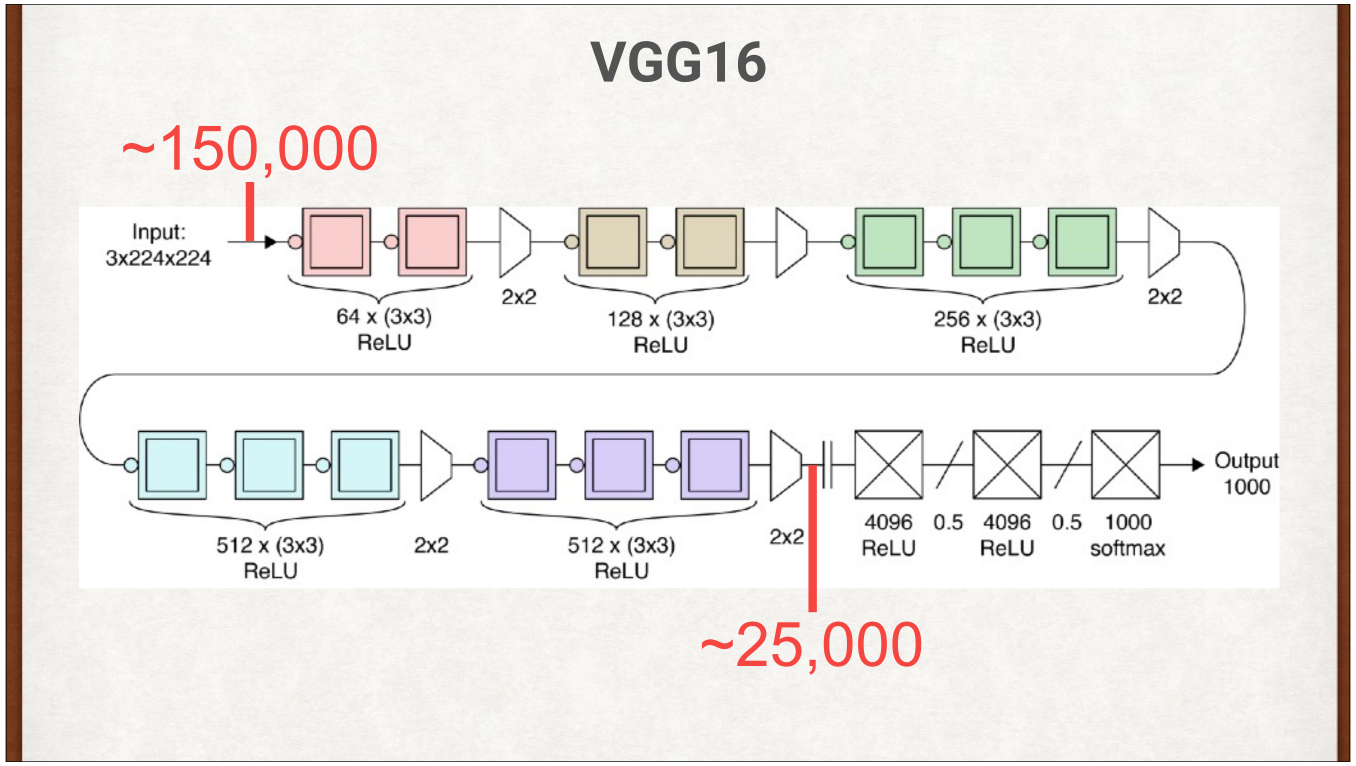


The hierarchy as a pyramid.



VGG16

VGG16 was trained on Imagenet Large Scale Visual Recognition Challenge (ILSVRC). 1000 categories. 1.2 million images.

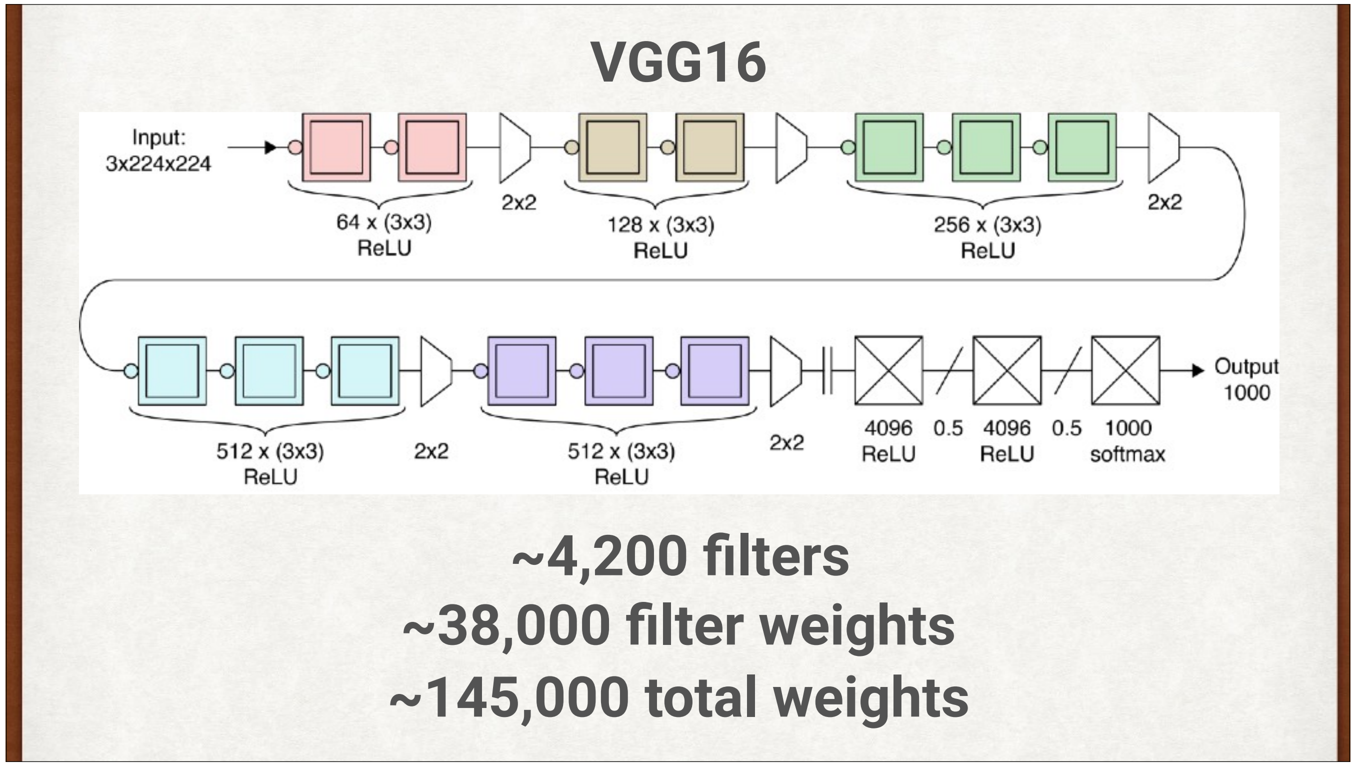


VGG16

~150,000

~25,000

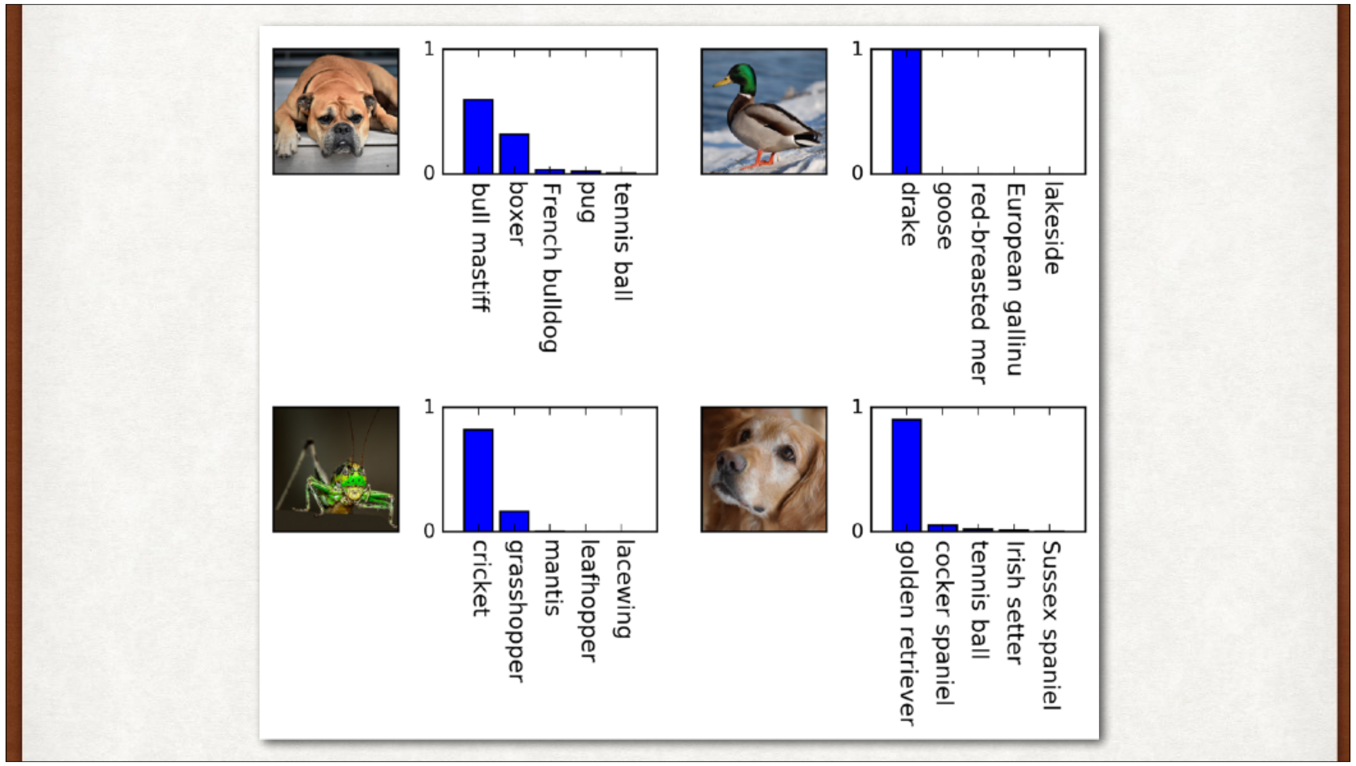
150k values in, a tensor of 25k goes into final fully-connected stage.



VGG16

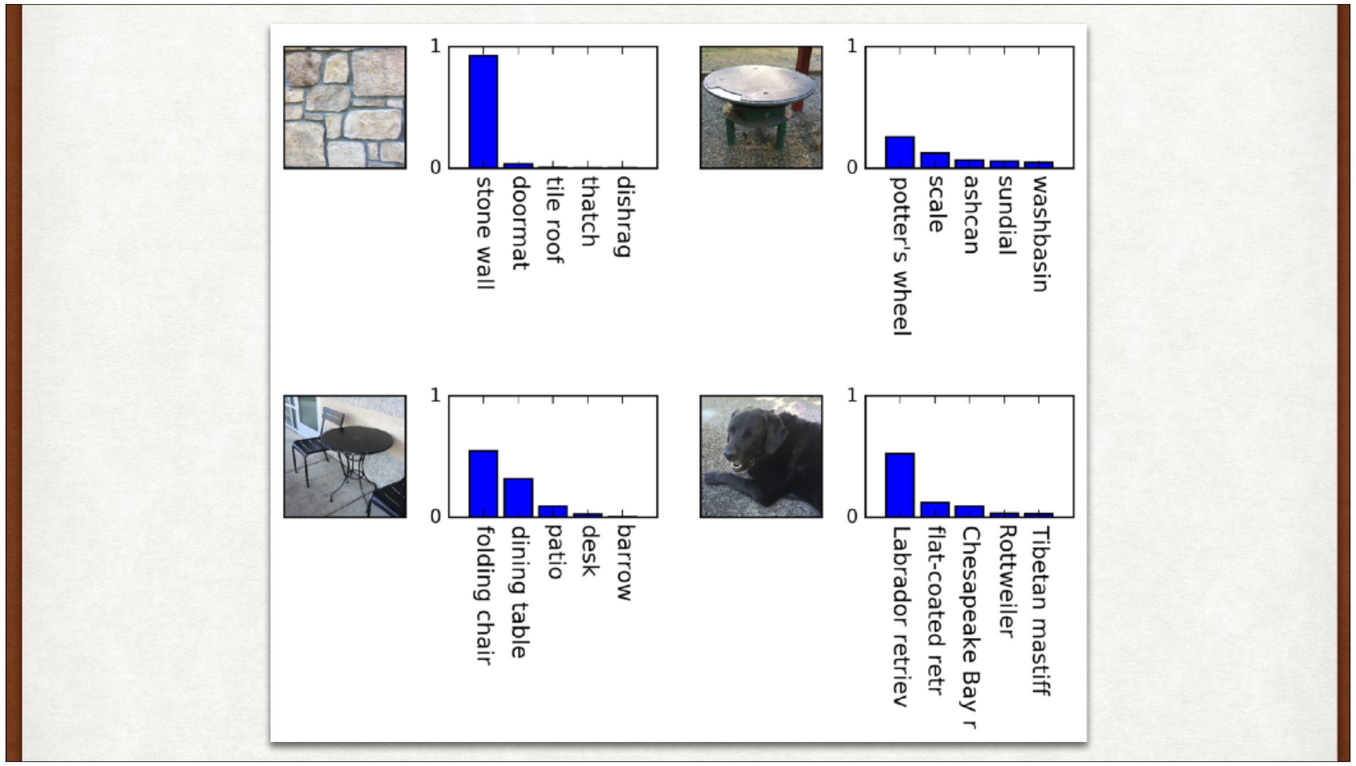
~4,200 filters
~38,000 filter weights
~145,000 total weights

Lots of weights to train. We’d never work them out by hand.

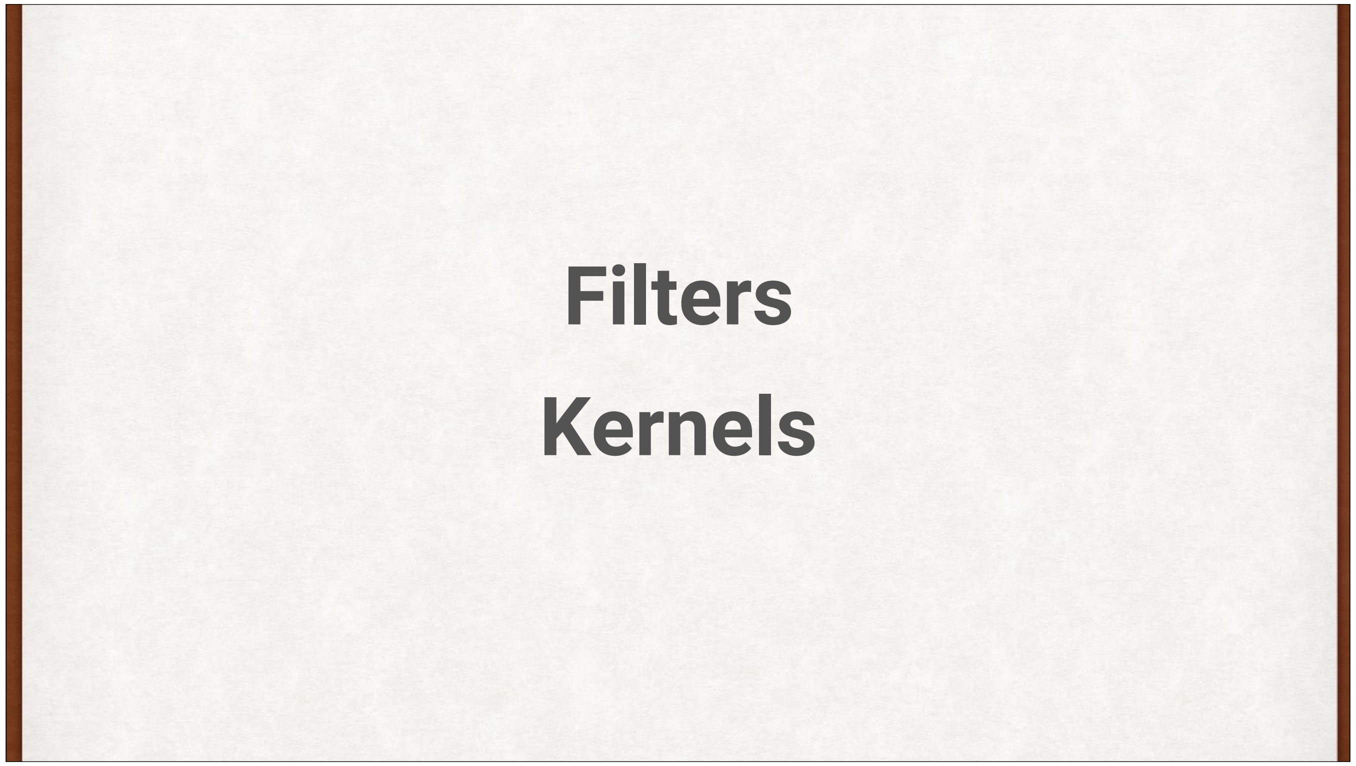


VGG16 classifications for online photographs, so they’re completely new to the system. It did great.

https://pixabay.com/en/dog-2437110
https://pixabay.com/en/duck-268105
https://pixabay.com/en/grasshopper-1533091
https://pixabay.com/en/golden-retriever-2419453



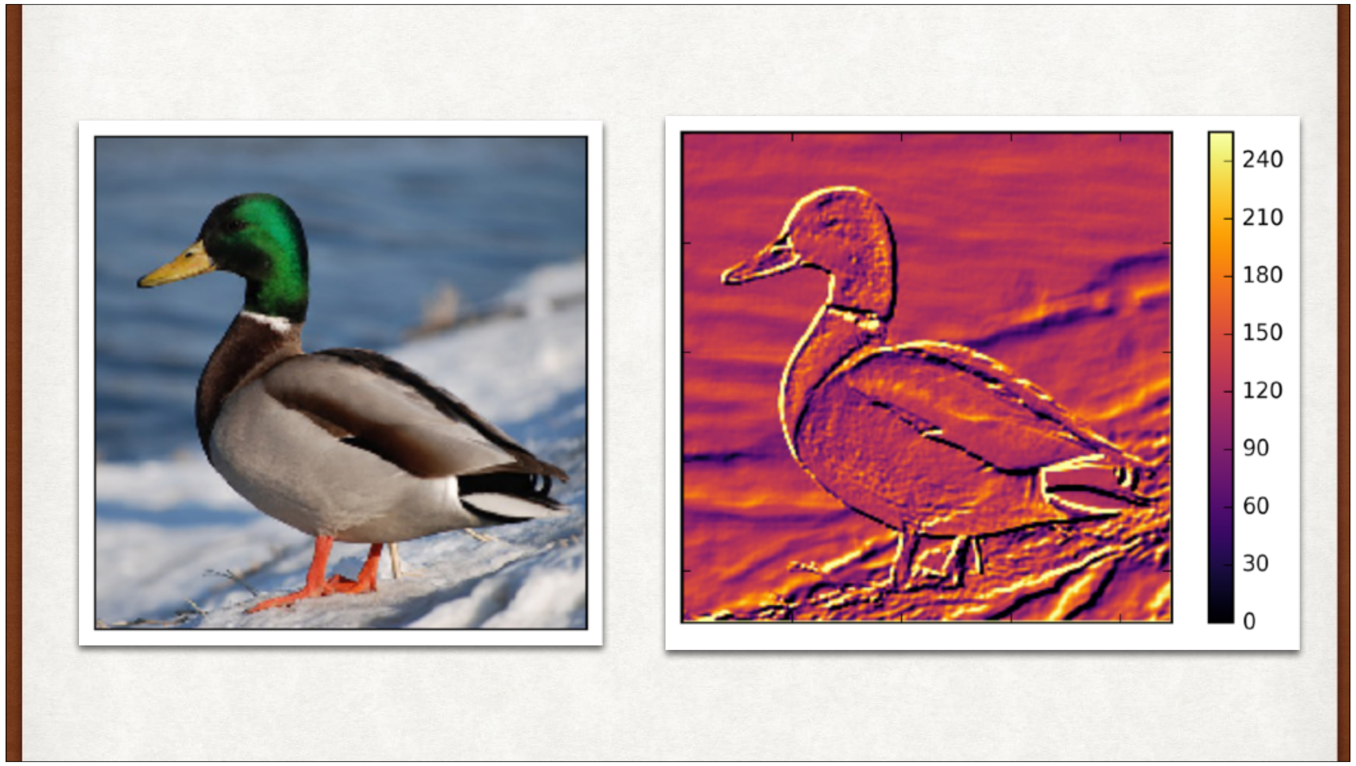
VGG16 classifications for my own photographs, so they're completely new to the system. It did great.



Filters

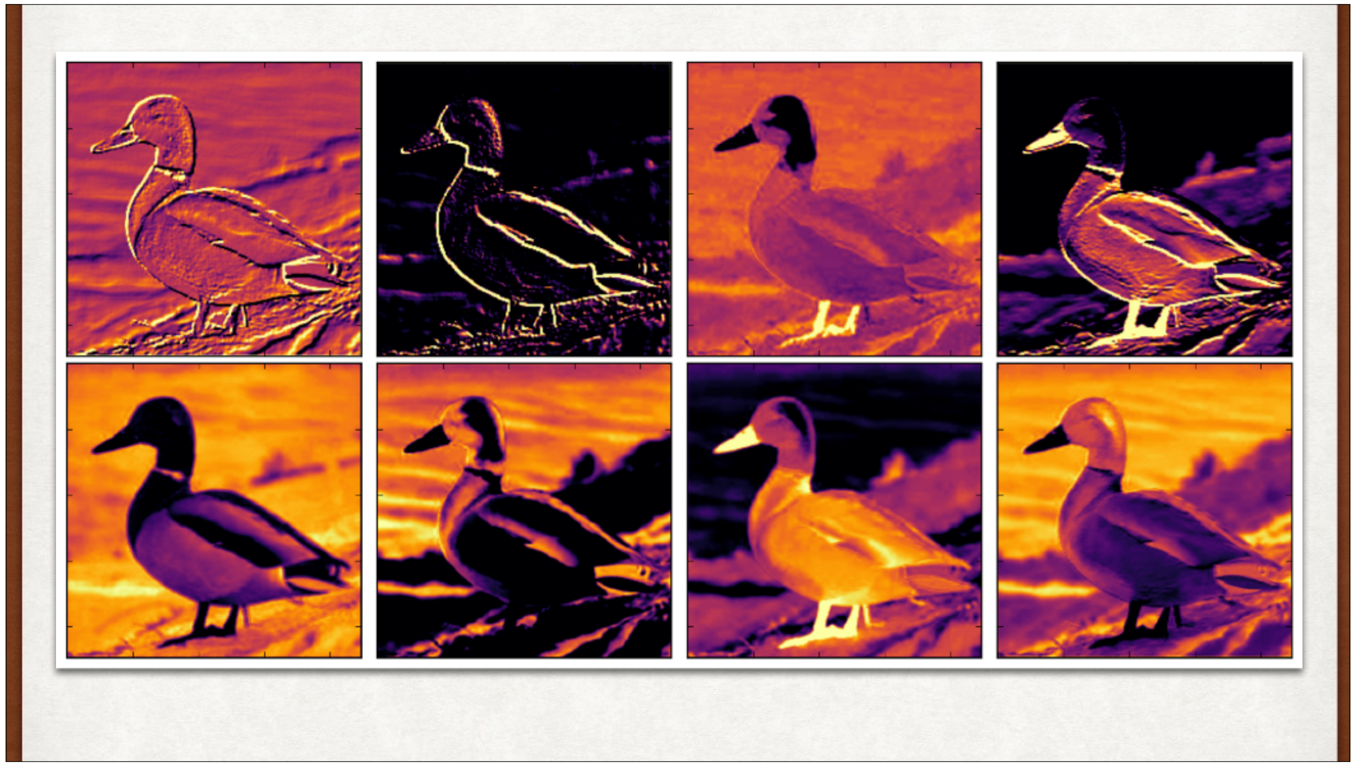
Kernels

Let’s look at filters (or kerns) that VGG16 learned for ImageNet classification.

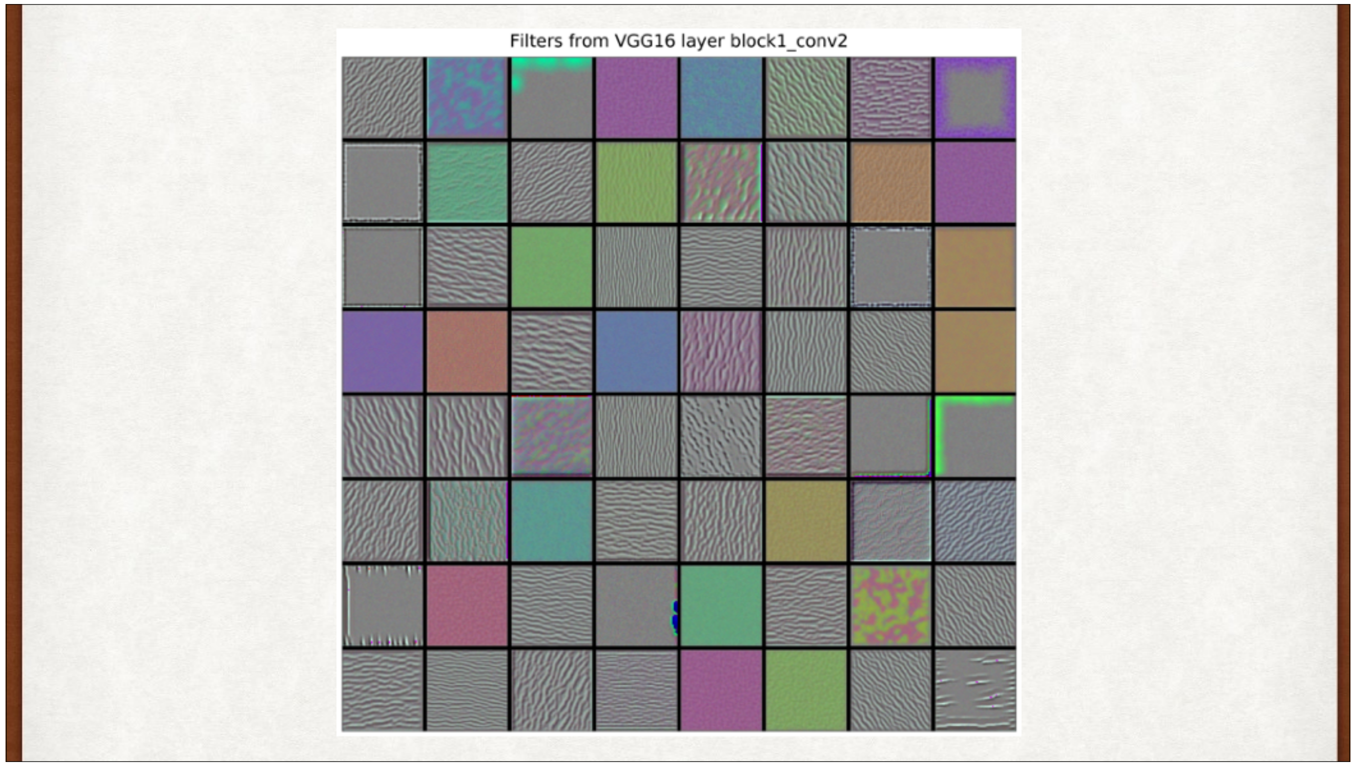


Left, an input image of a duck. Right, the response of filter 0 in the first layer of VGG16. This filter appears to respond to strong edges. Where there’s no clear edge, we get middling values.

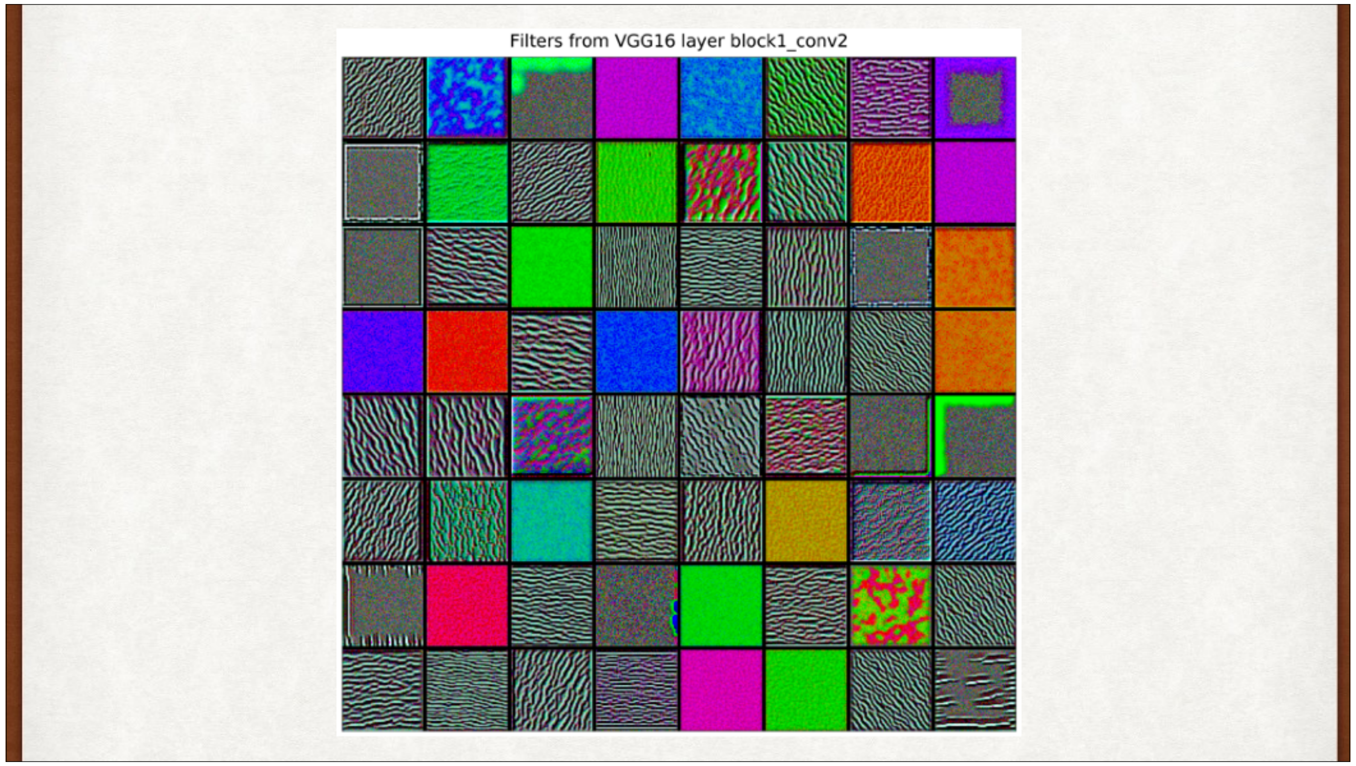
https://pixabay.com/en/duck-268105



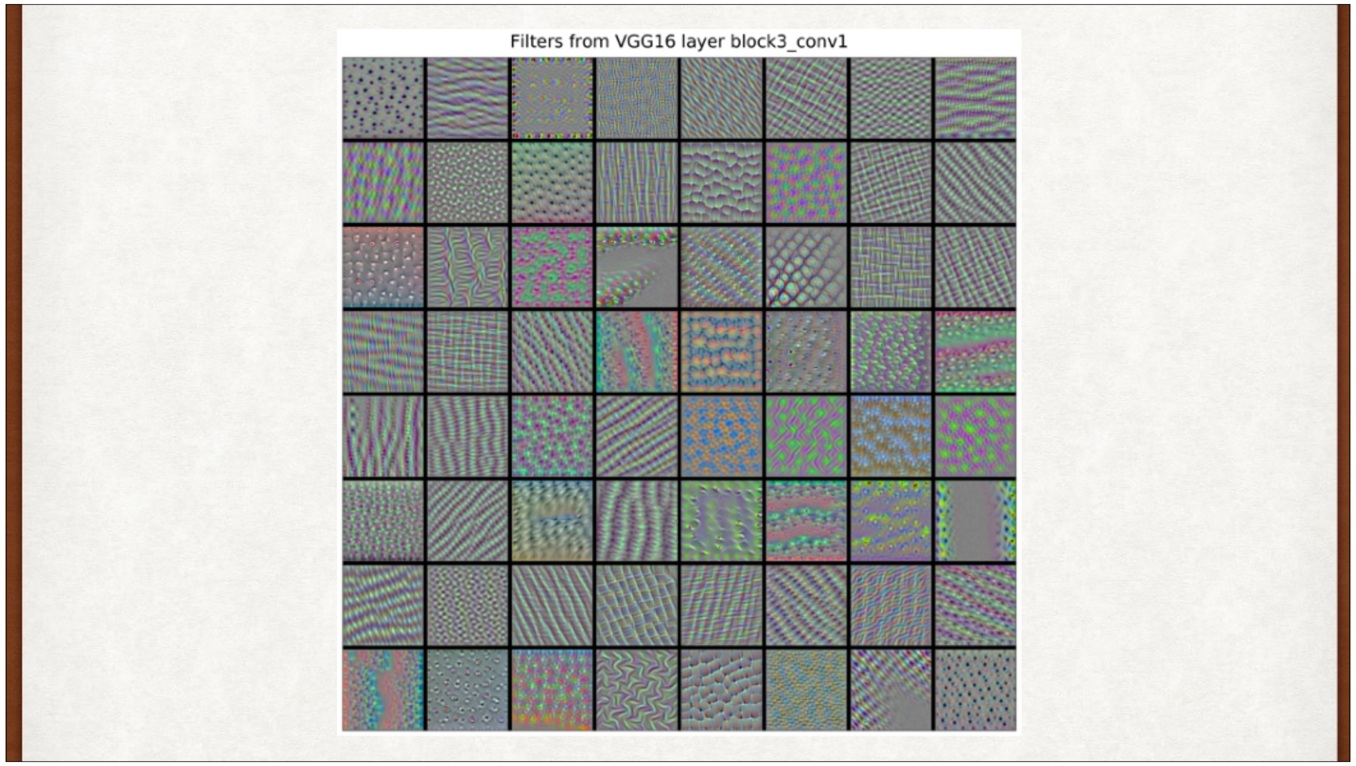
The responses of 8 hand-picked filters on the first layer of VGG16 to the duck. These different filters have learned to “look” for different types of patterns in the image.



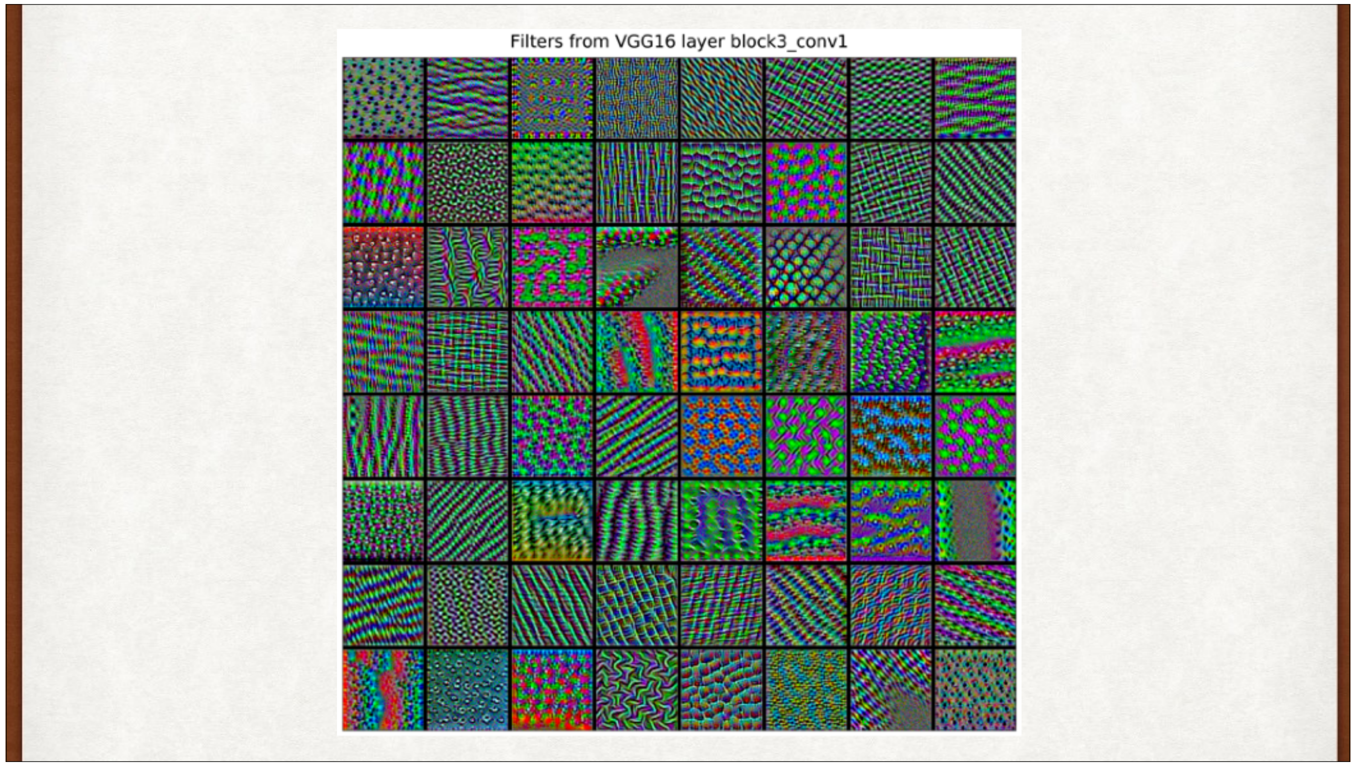
Visualizing layer patterns using noise as an input. We send random noise into the system, and measure how strongly each layer responds by adding up all of its output values. We use that as the “error”, and then modify the noisy input pixel values to make that “error” as large as possible - that is, we’re stimulating the layer as much as we can. These images are the final values of that starting noise, showing the kind of pattern that the filter is “looking for.”



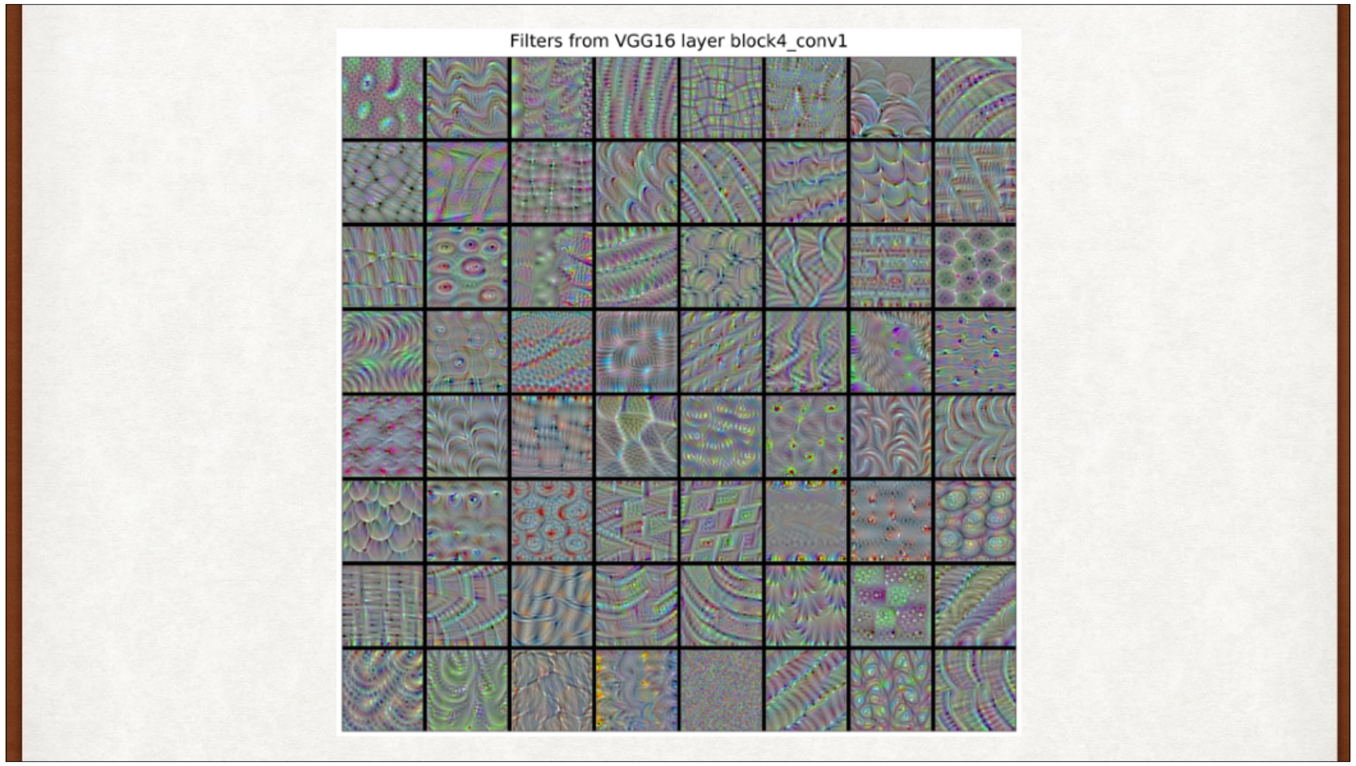
Visualizing layer patterns using noise as an input. We send random noise into the system, and measure how strongly each layer responds by adding up all of its output values. We use that as the “error”, and then modify the noisy input pixel values to make that “error” as large as possible - that is, we’re stimulating the layer as much as we can. These images are the final values of that starting noise, showing the kind of pattern that the filter is “looking for.”



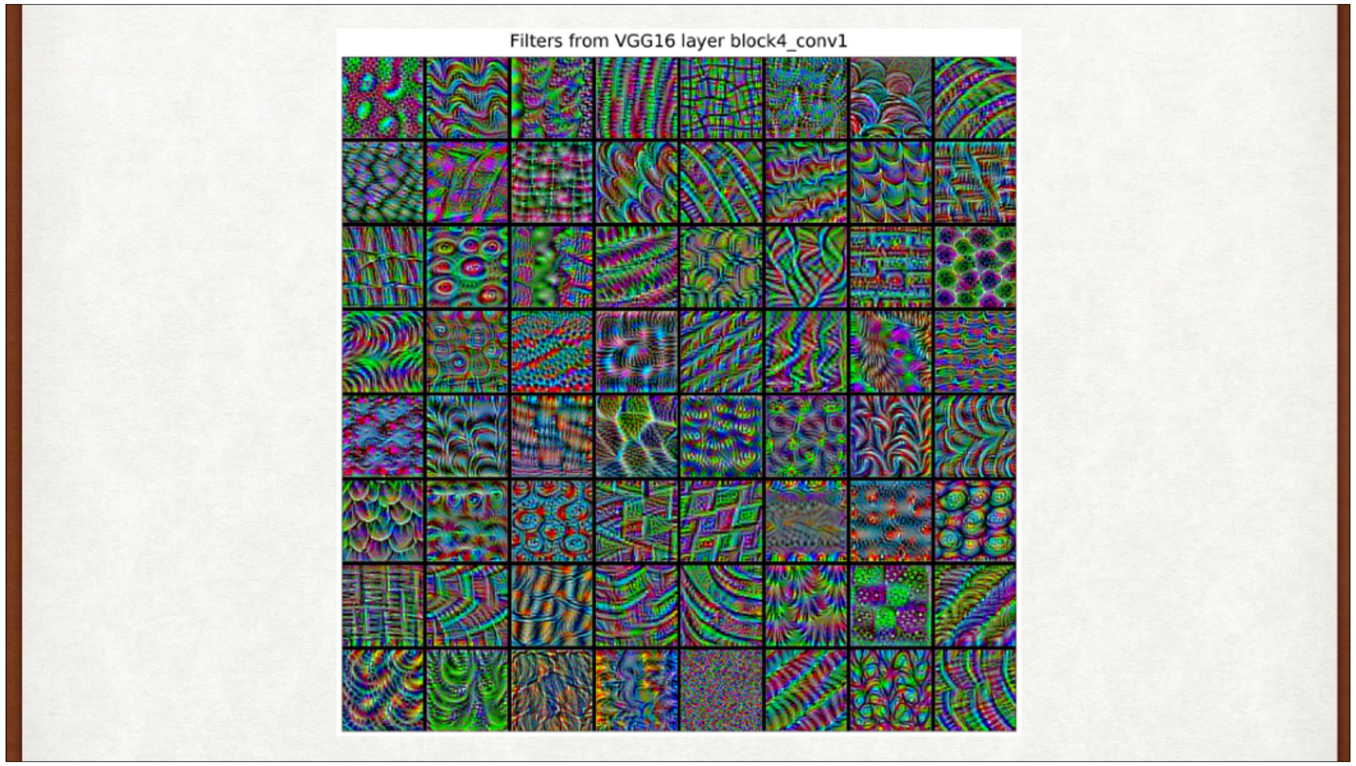
Visualizing filters from the third block of VGG16.



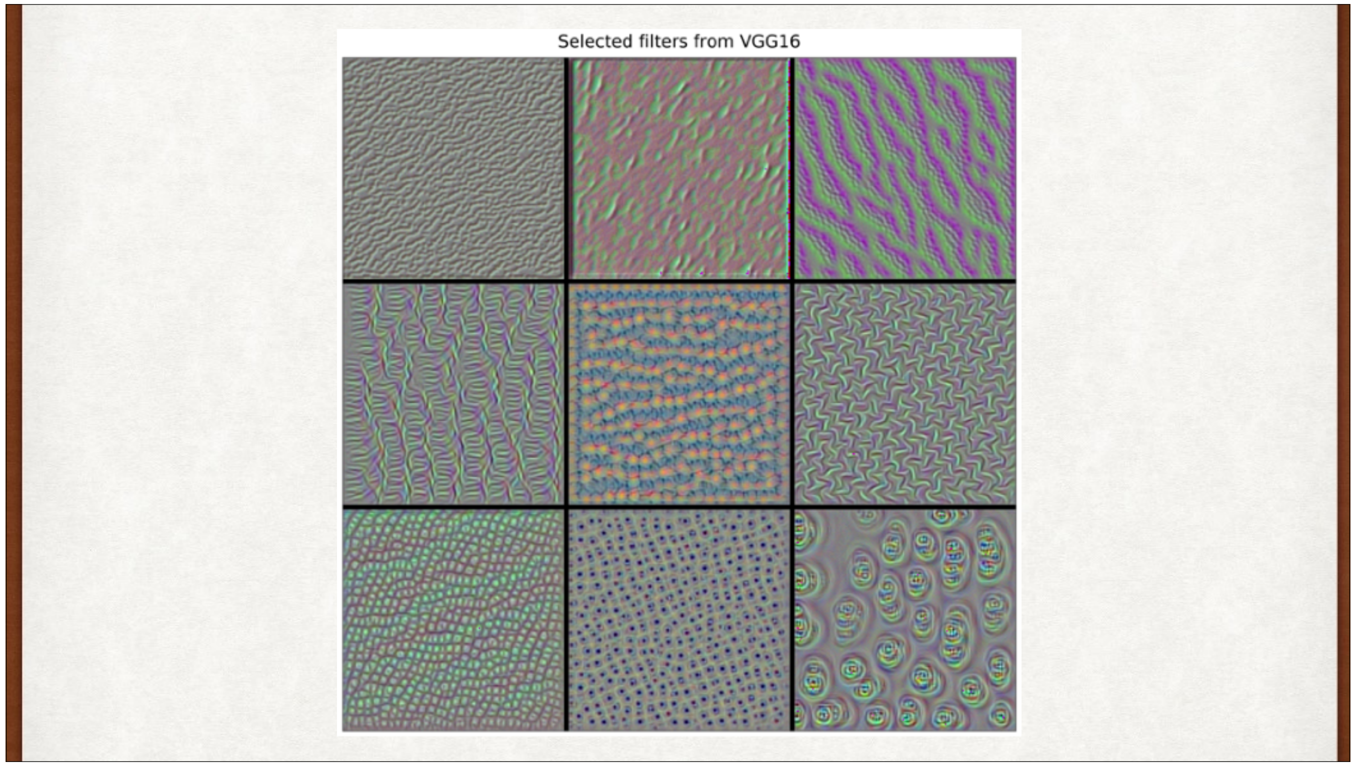
Visualizing filters from the third block of VGG16.



Visualizing filters from the fourth block of VGG16.

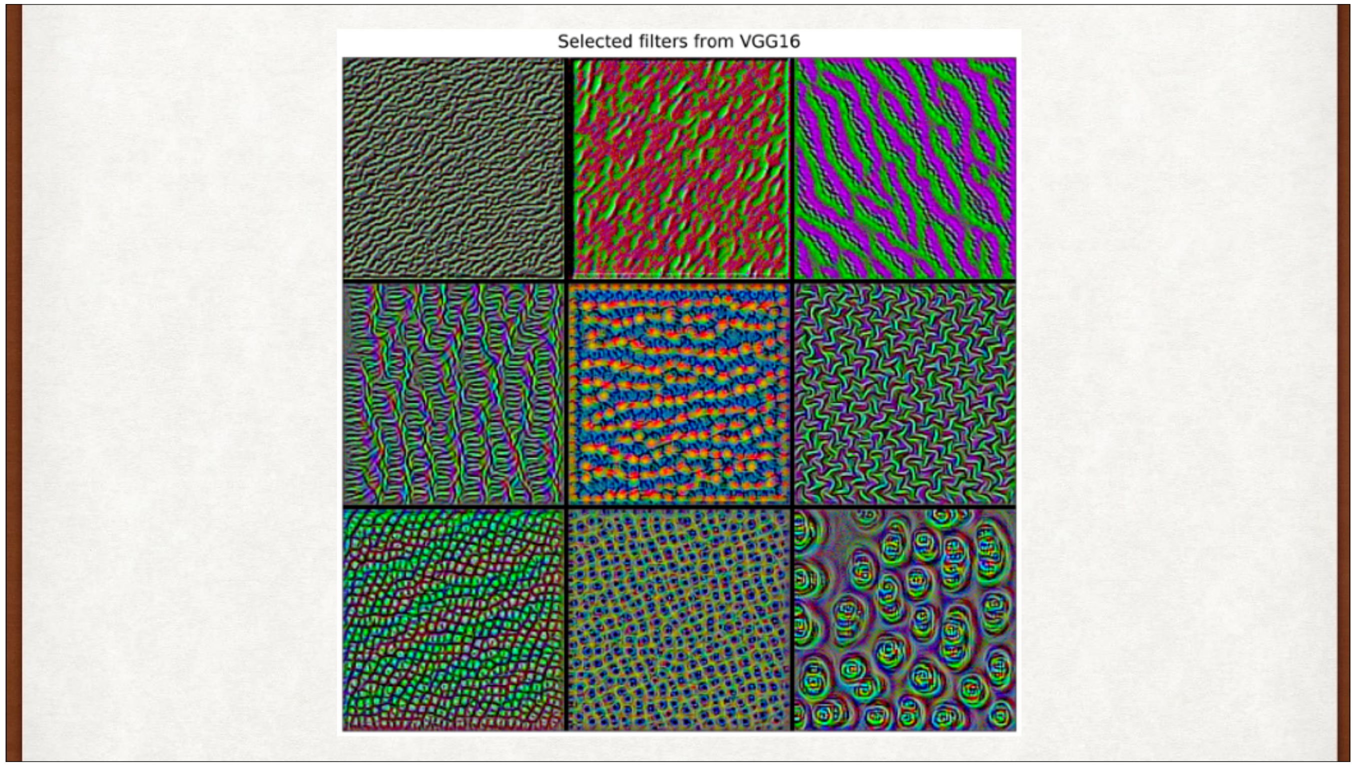


Visualizing filters from the fourth block of VGG16.



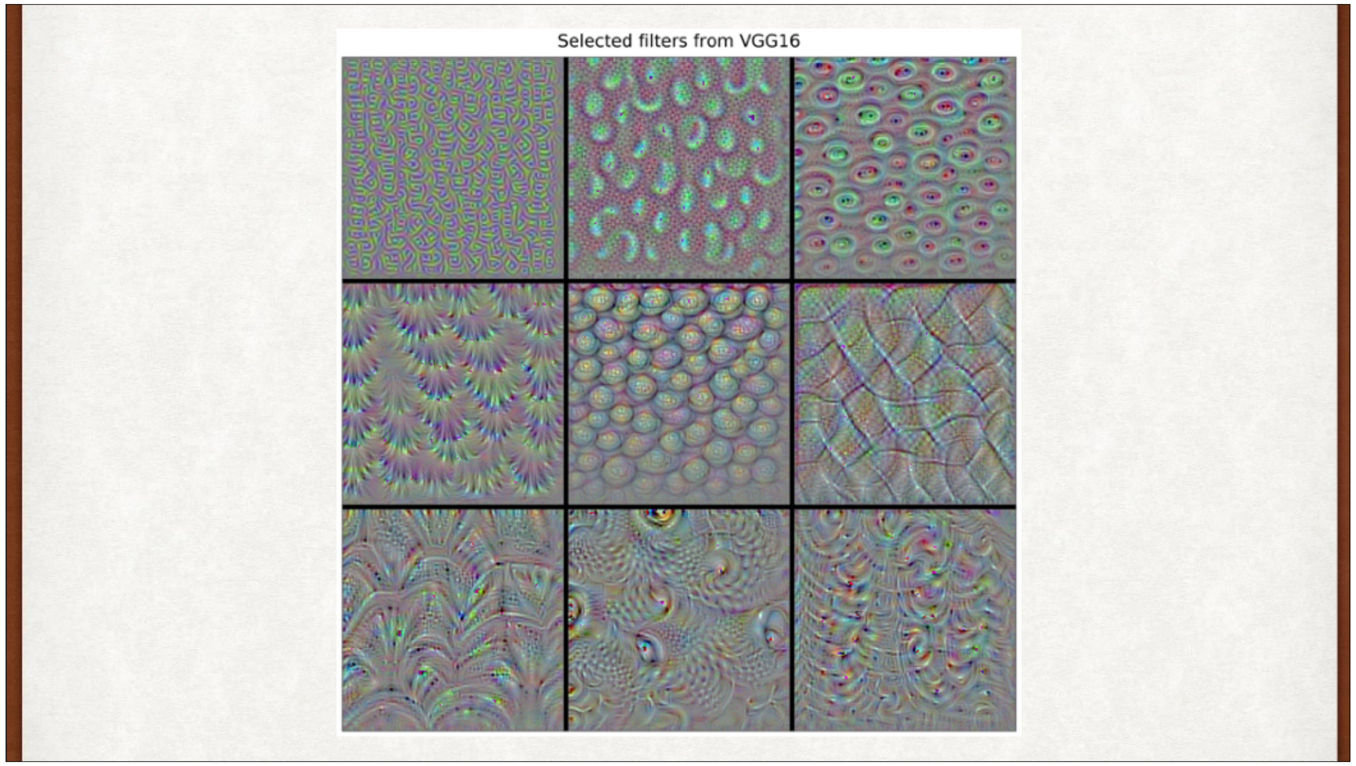
Selected filters from VGG16

Some filter responses I chose by hand from the complete set because they looked cool.

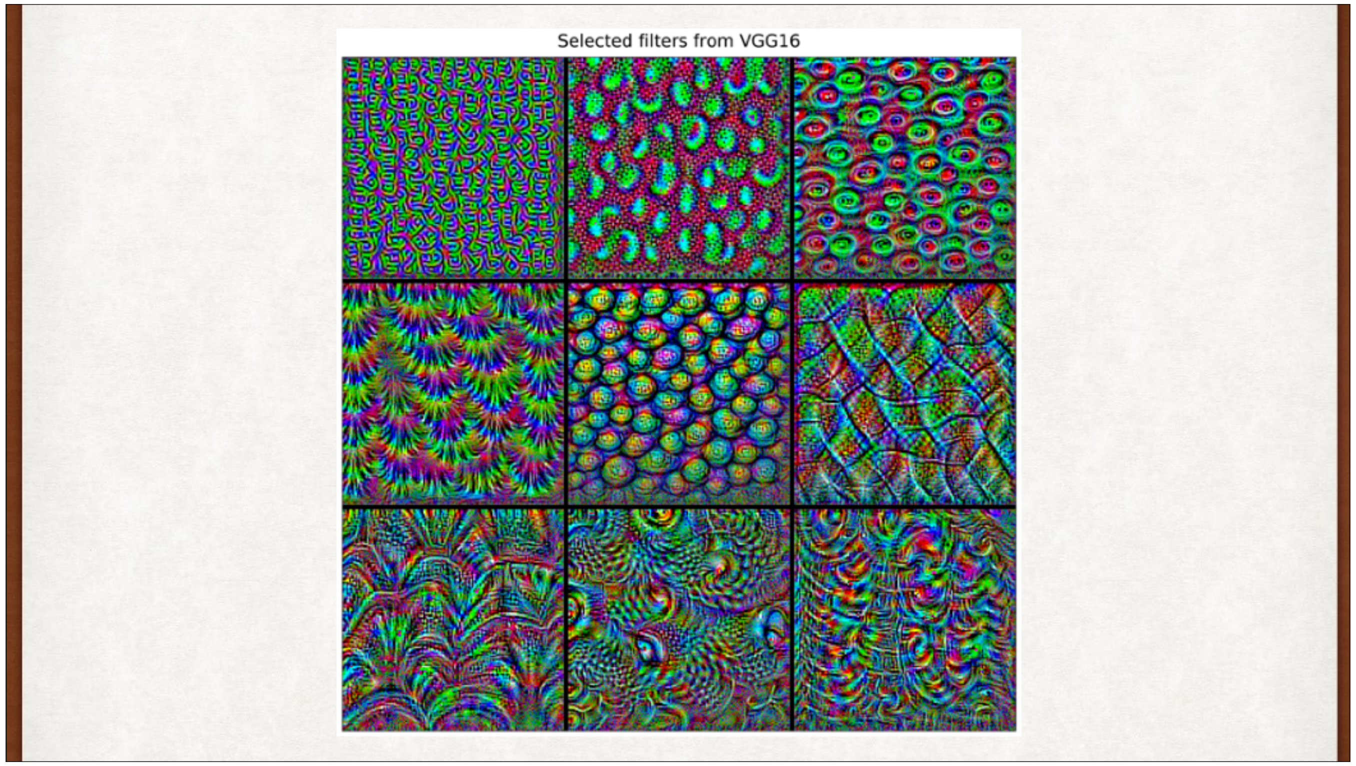


Selected filters from VGG16

Some filter responses I chose by hand from the complete set because they looked cool.

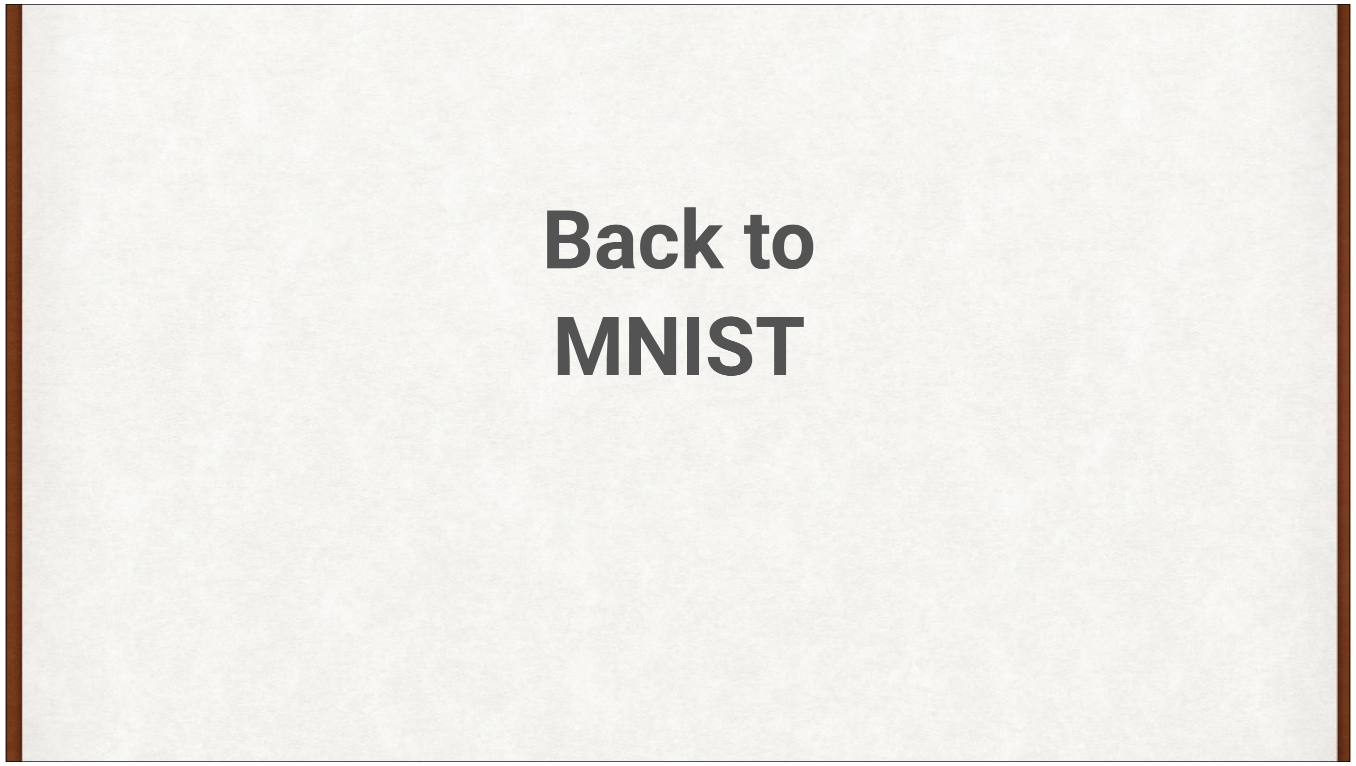


Some more cool filter responses I picked by hand.



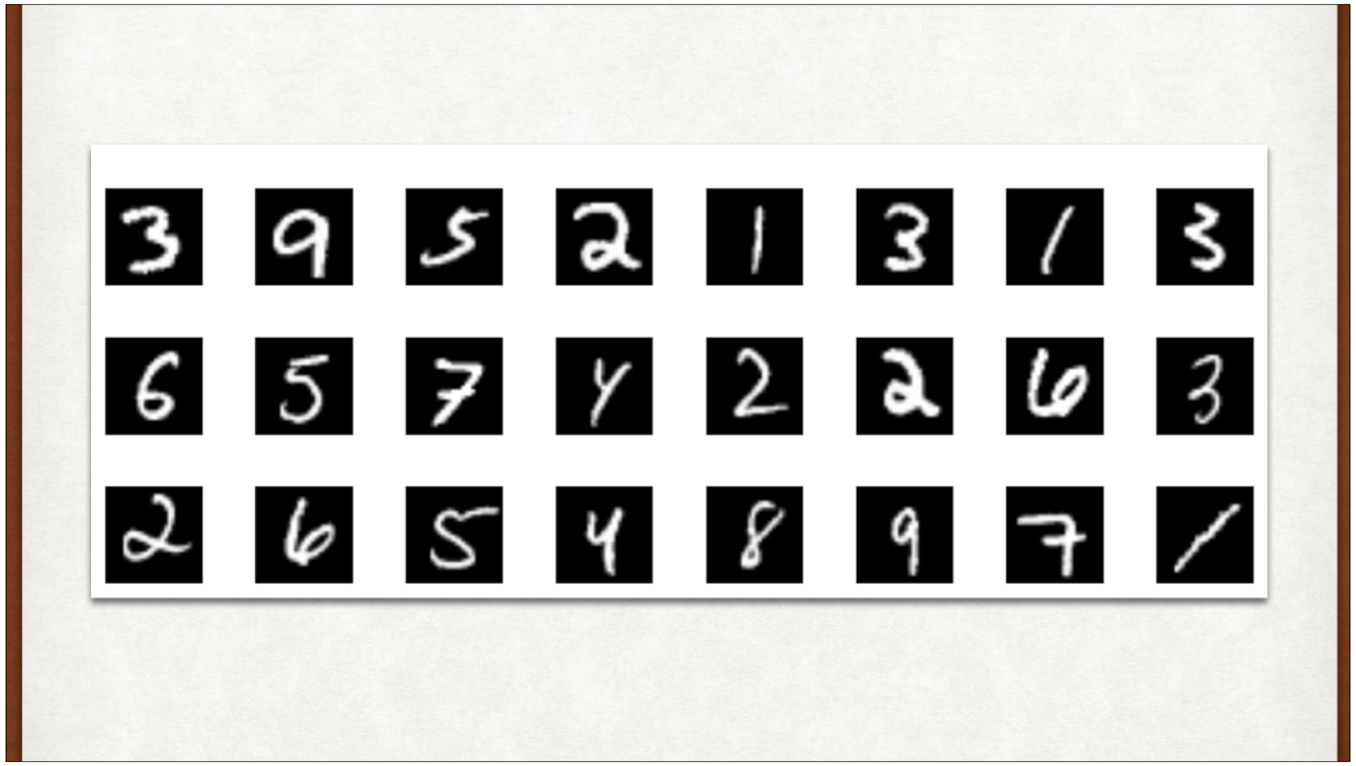
Selected filters from VGG16

Some more cool filter responses I picked by hand.

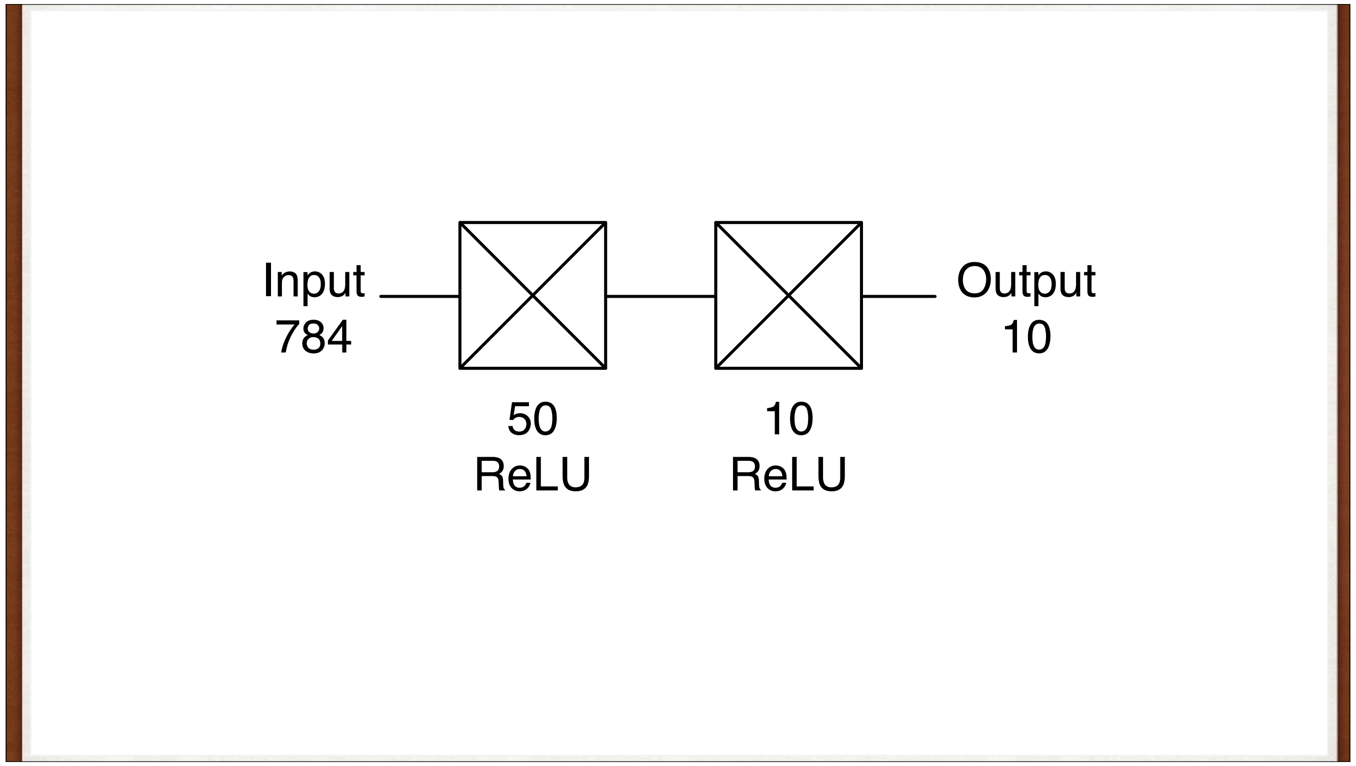


Back to MNIST

Back to our original problem. Let’s use convolution to identify handwritten digits.



Back to our original problem. Let’s use convolution to identify handwritten digits.



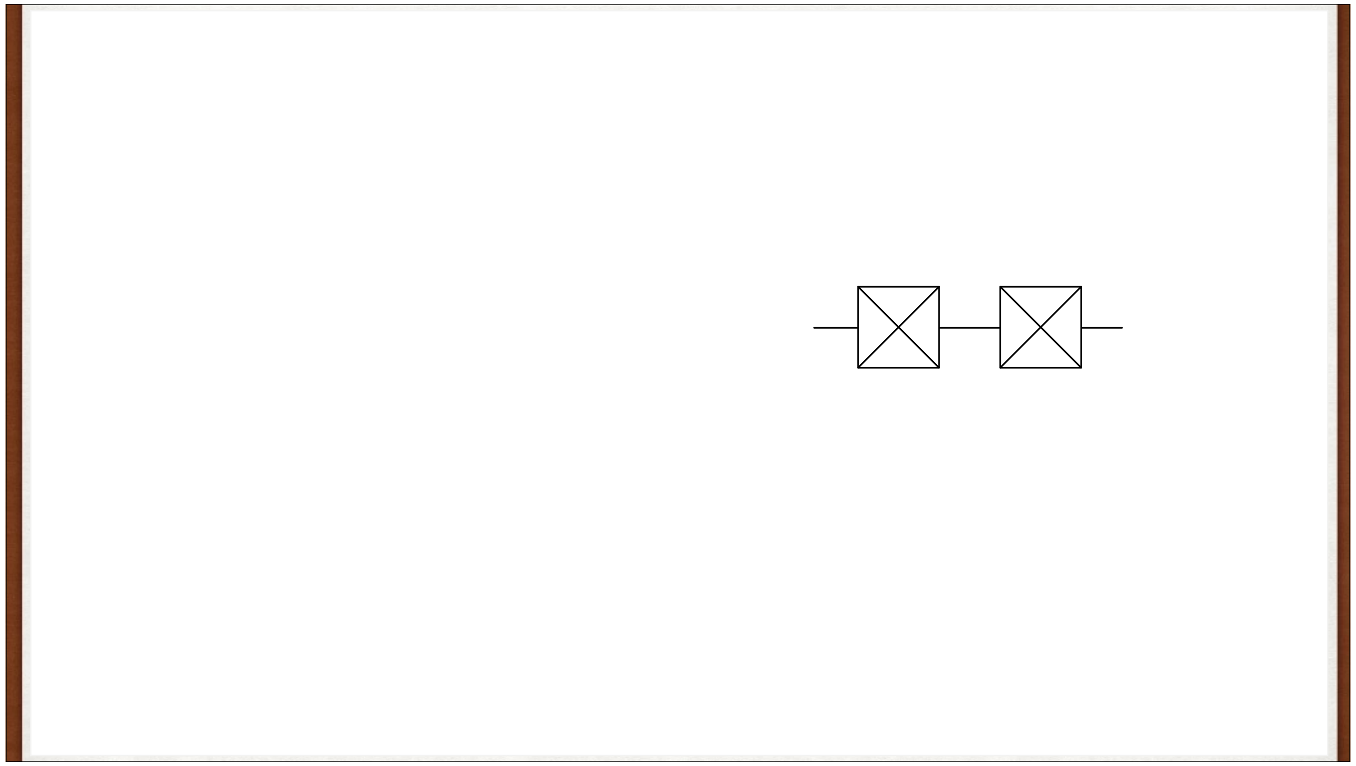
Input
784

50
ReLU

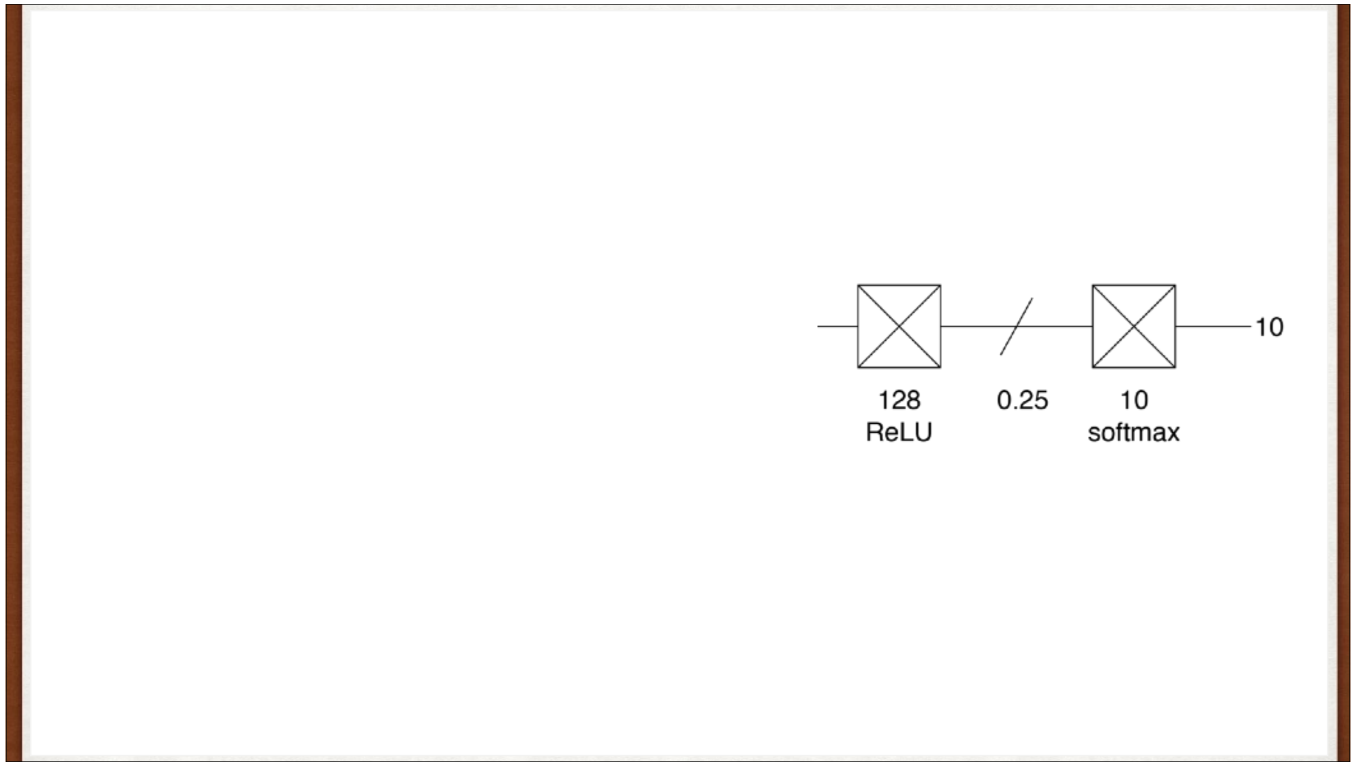
10
ReLU

Output
10

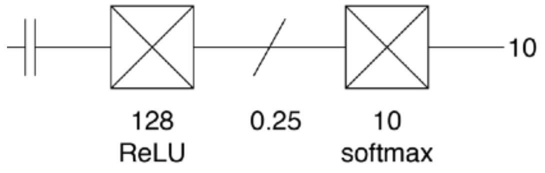
The network we saw at the start of the talk, in schematic form.



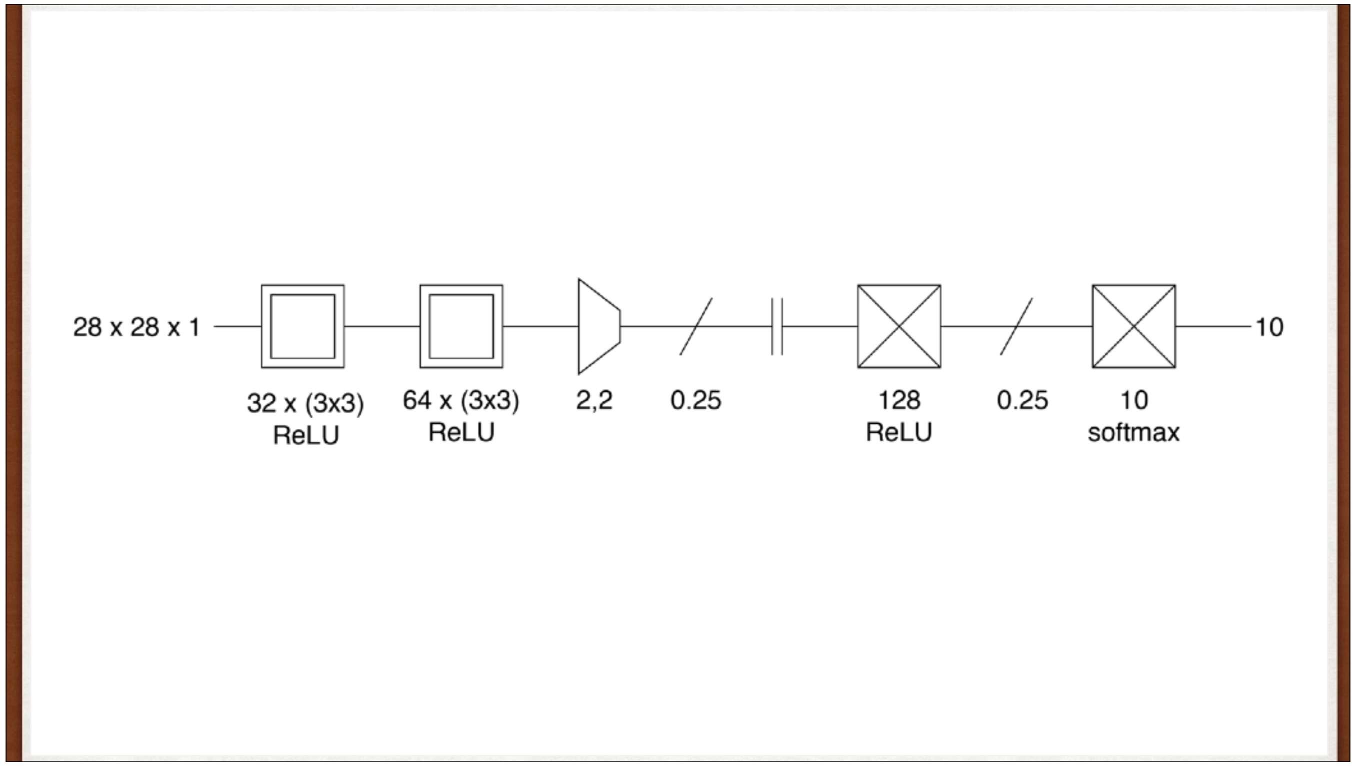
Let’s take the fully connected layers…



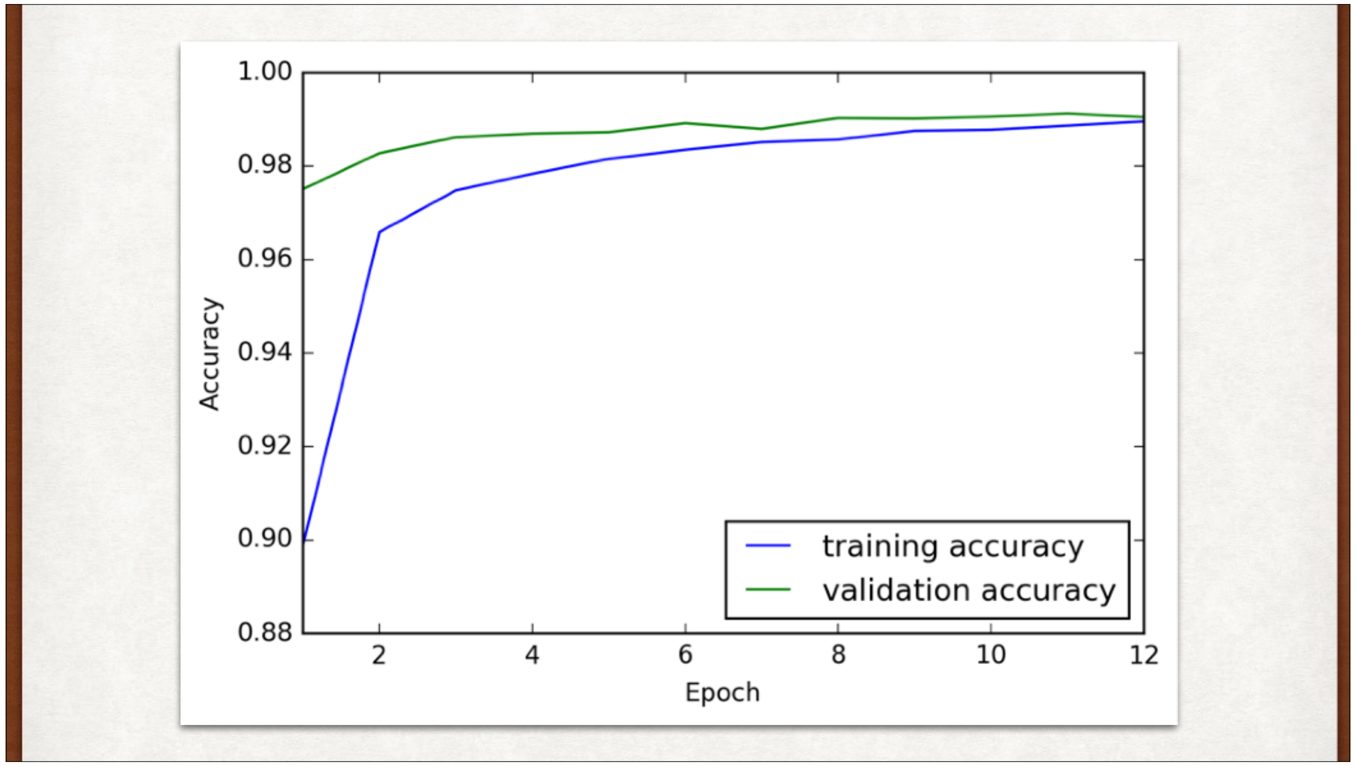
…put ReLu on the first, setting that to 128 neurons, and softmax on the second, with a dropout of 0.25 on the first layer.



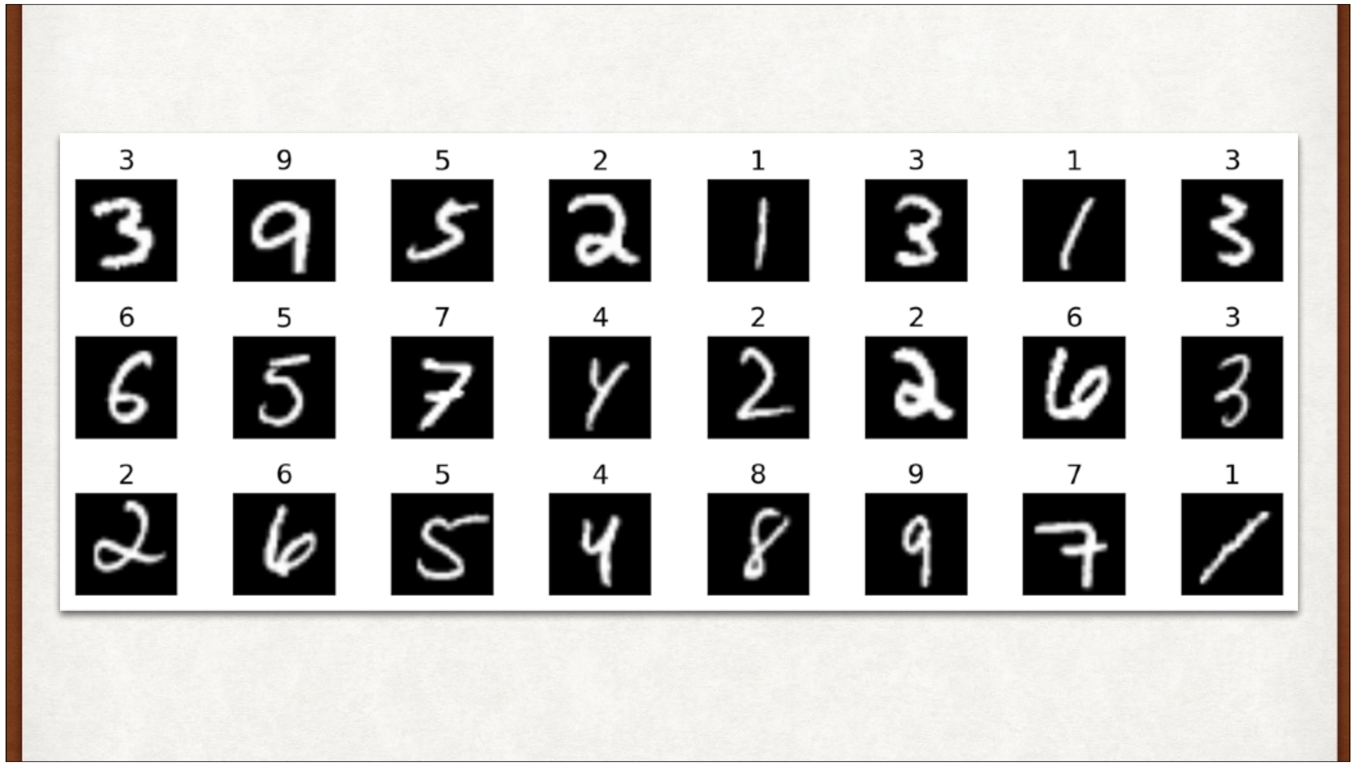
This is going to take data from convolution, so we flatten it first.



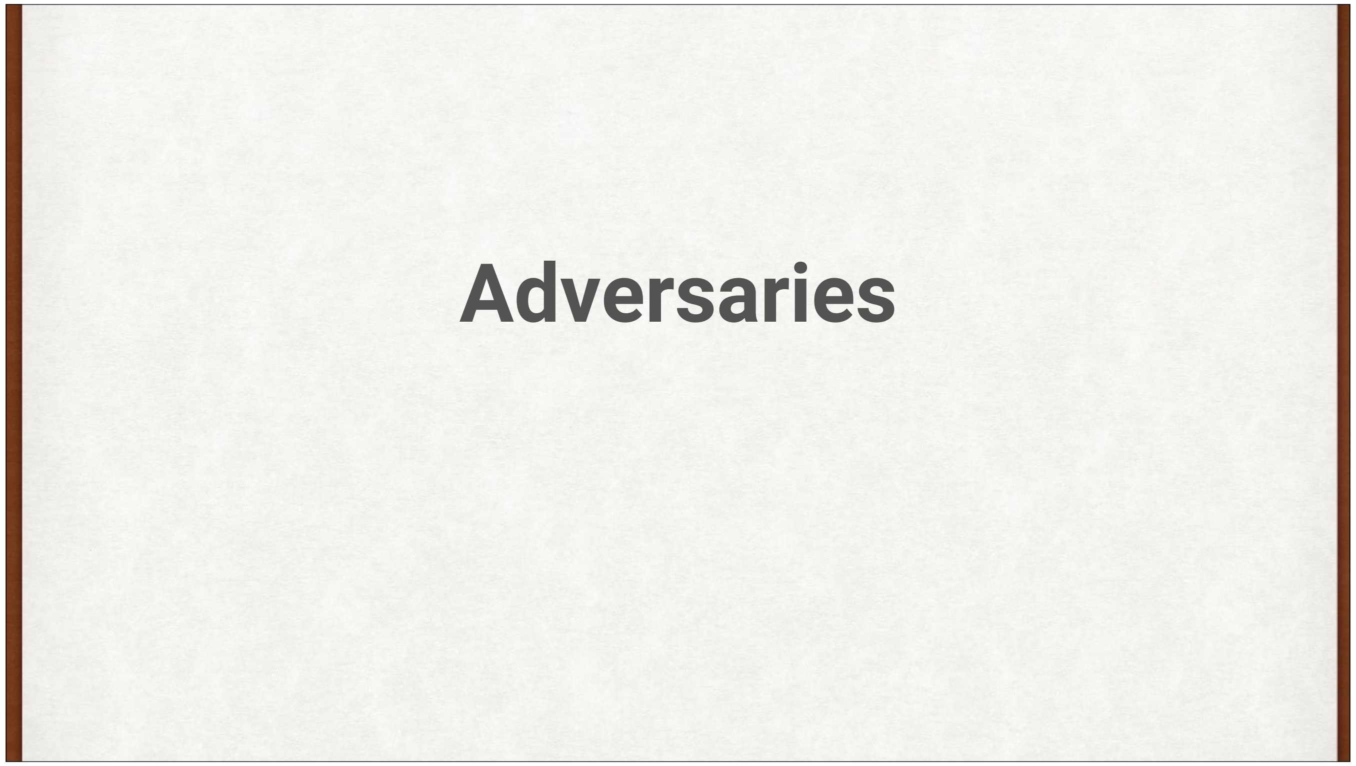
Here’s a basic convolutional neural net (CNN) in schematic form. Two layers of convolution are followed by a downsampler (max pool) to reduce the input to 14 by 14. We use dropout with a rate of 0.25 for regularization. Then we flatten the data, go through a fully-connected layer of 128 units (followed by dropout), and then a 10-neuron fully-connected layer with softmax. This could be made even more bare bones by taking out the downsampler and the dropout layers, or using striding in the second convolution instead of the max-pooling layer.



Performance of our CNN. Just about 99% accuracy on the MNIST test data after 12 epochs. Pretty great!

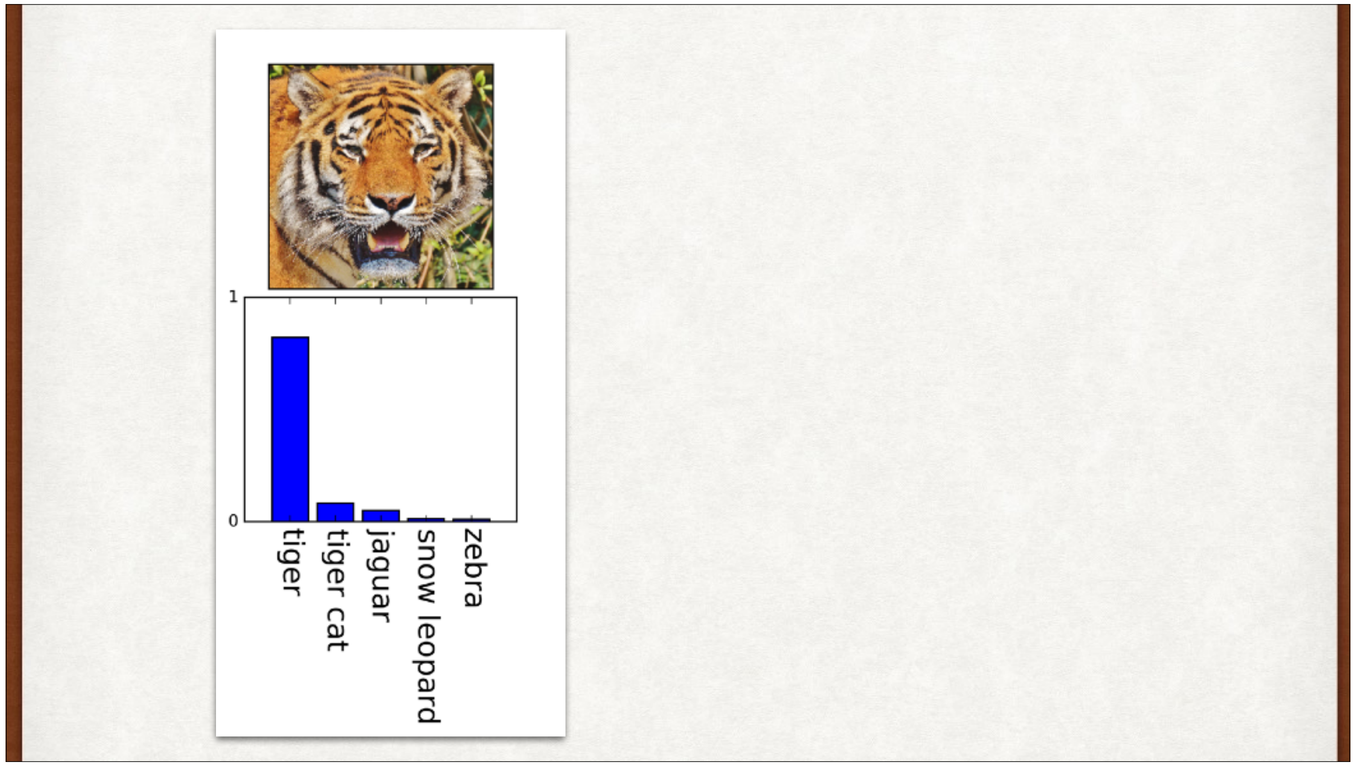


MNIST results from our CNN. All correct!



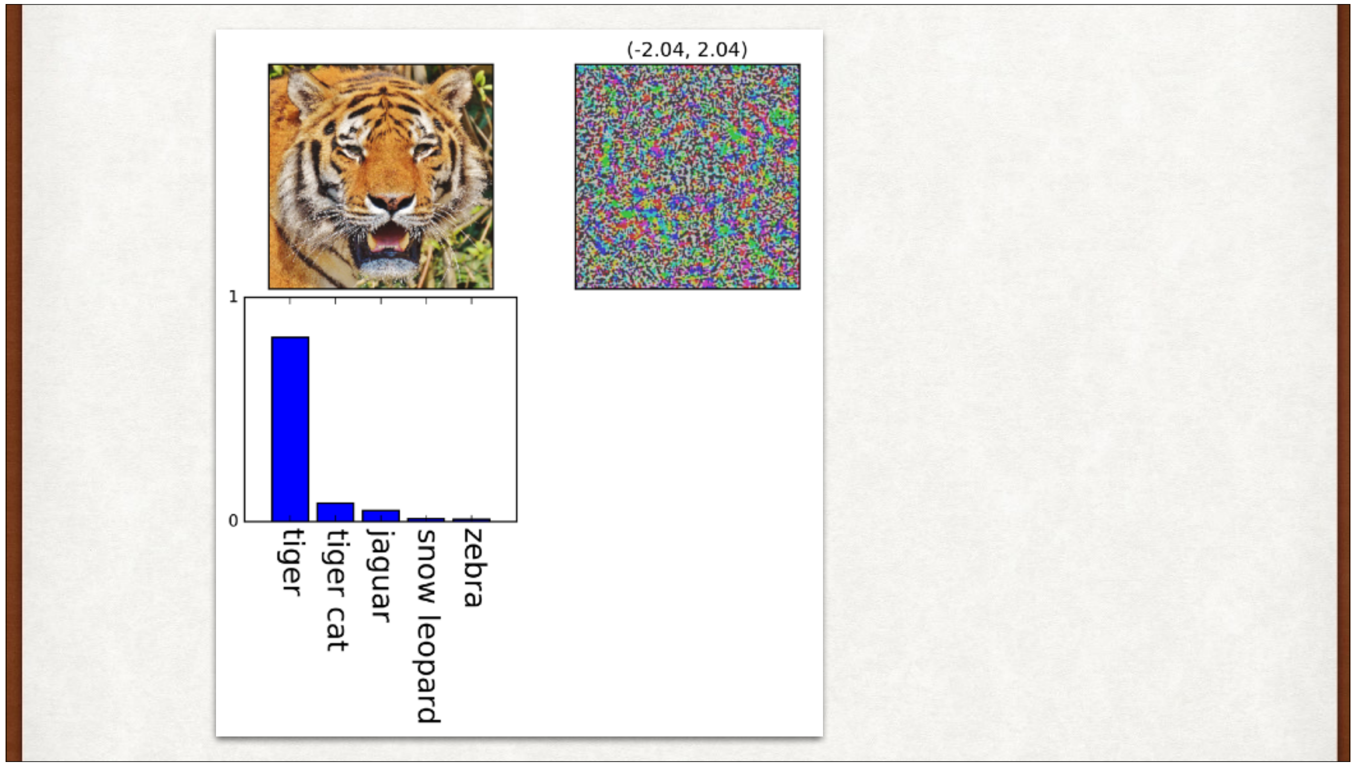
Adversaries

A CNN can be tricked by adding just a little bit of carefully-craft perturbations, called an adversary, to an image. To our eye, the image is unchanged. But the CNN thinks it’s a different object.

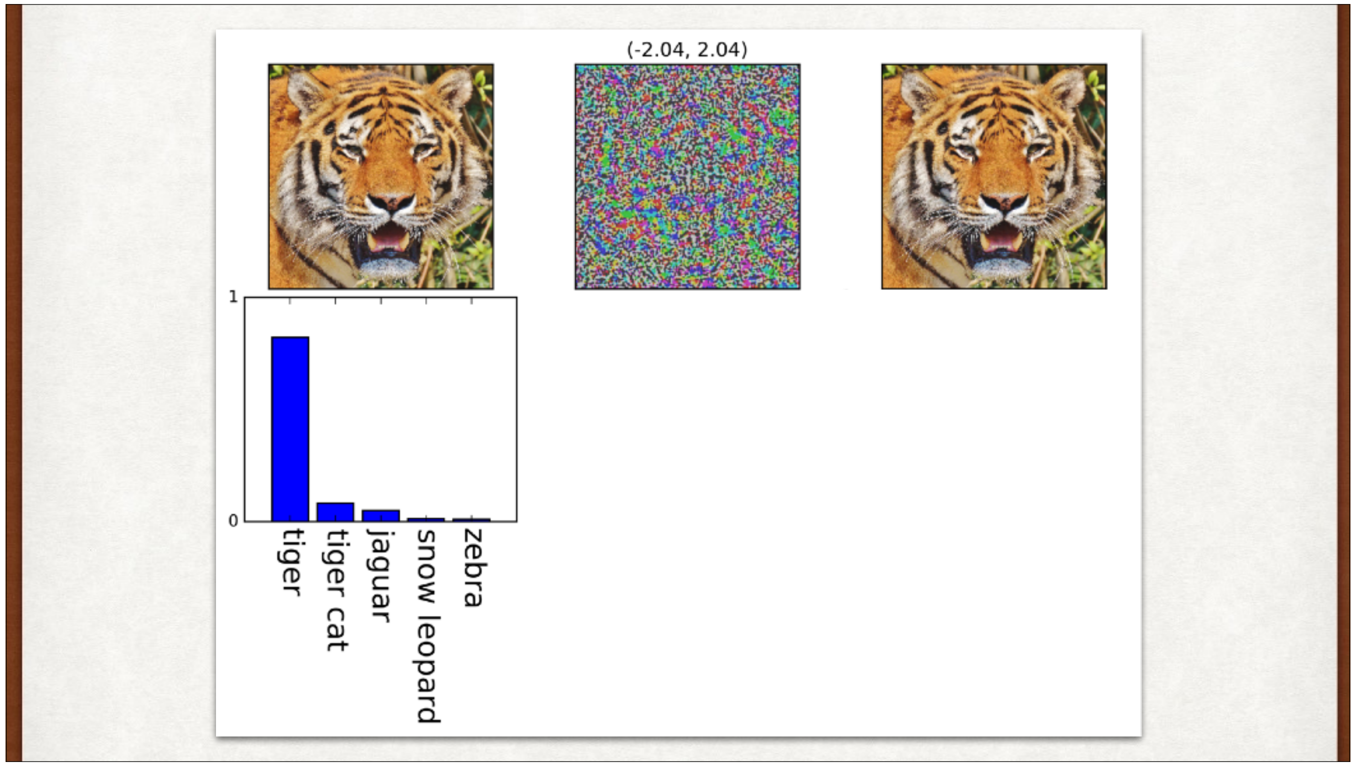


A tiger (left) is correctly identified, but add some carefully constructed noise (middle) in the range of about (-2,2), and VGG16’s predictions (right) are… weird.

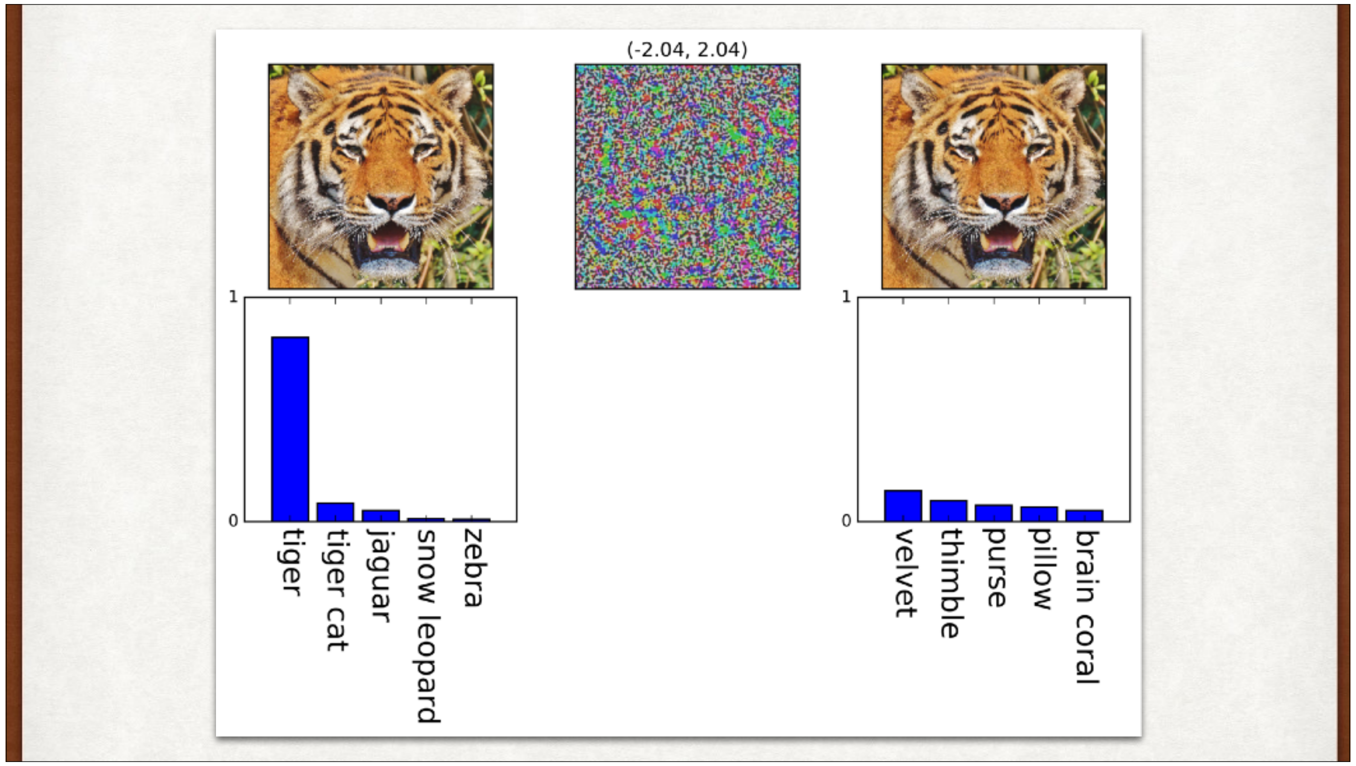
https://pixabay.com/photos/tiger-siberian-tiger-tiger-head-3424791/



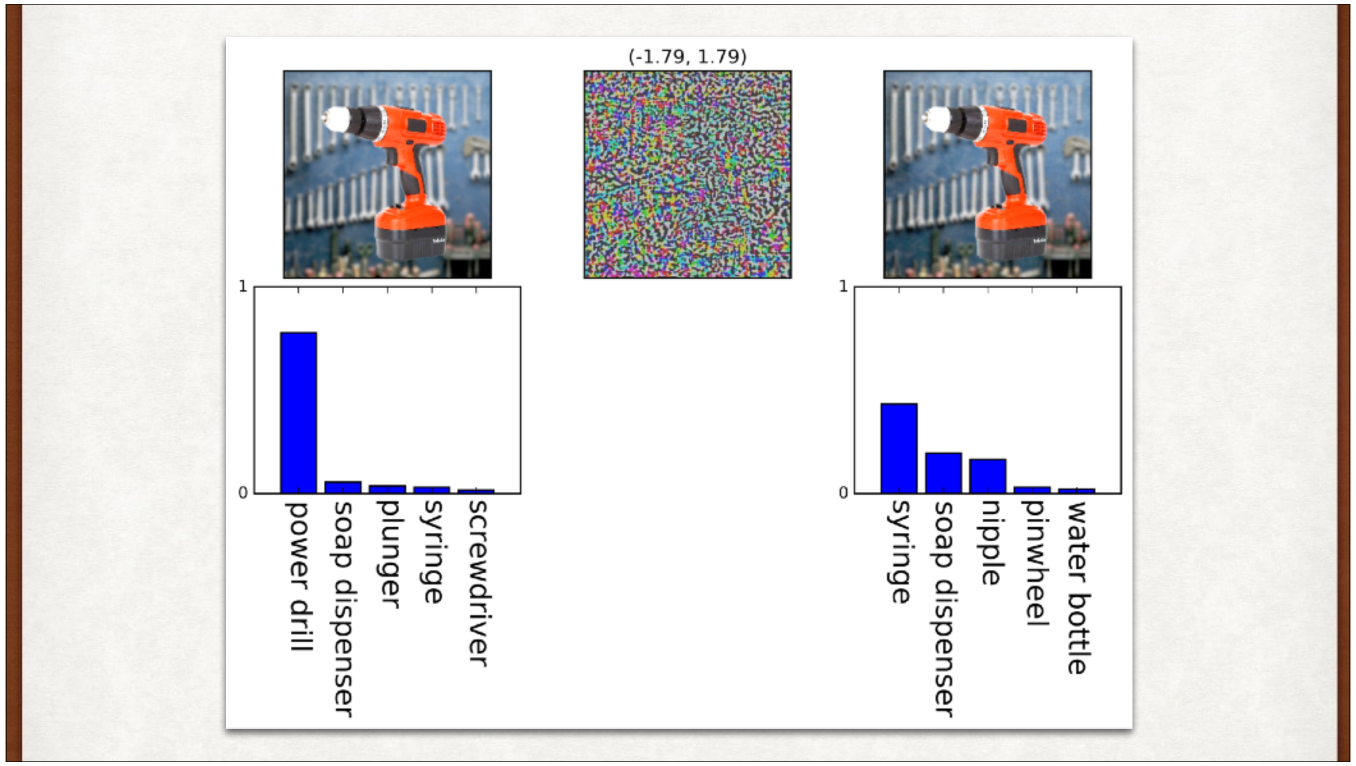
A tiger (left) is correctly identified, but add some carefully constructed noise (middle) in the range of about (-2,2), and VGG16's predictions (right) are… weird.



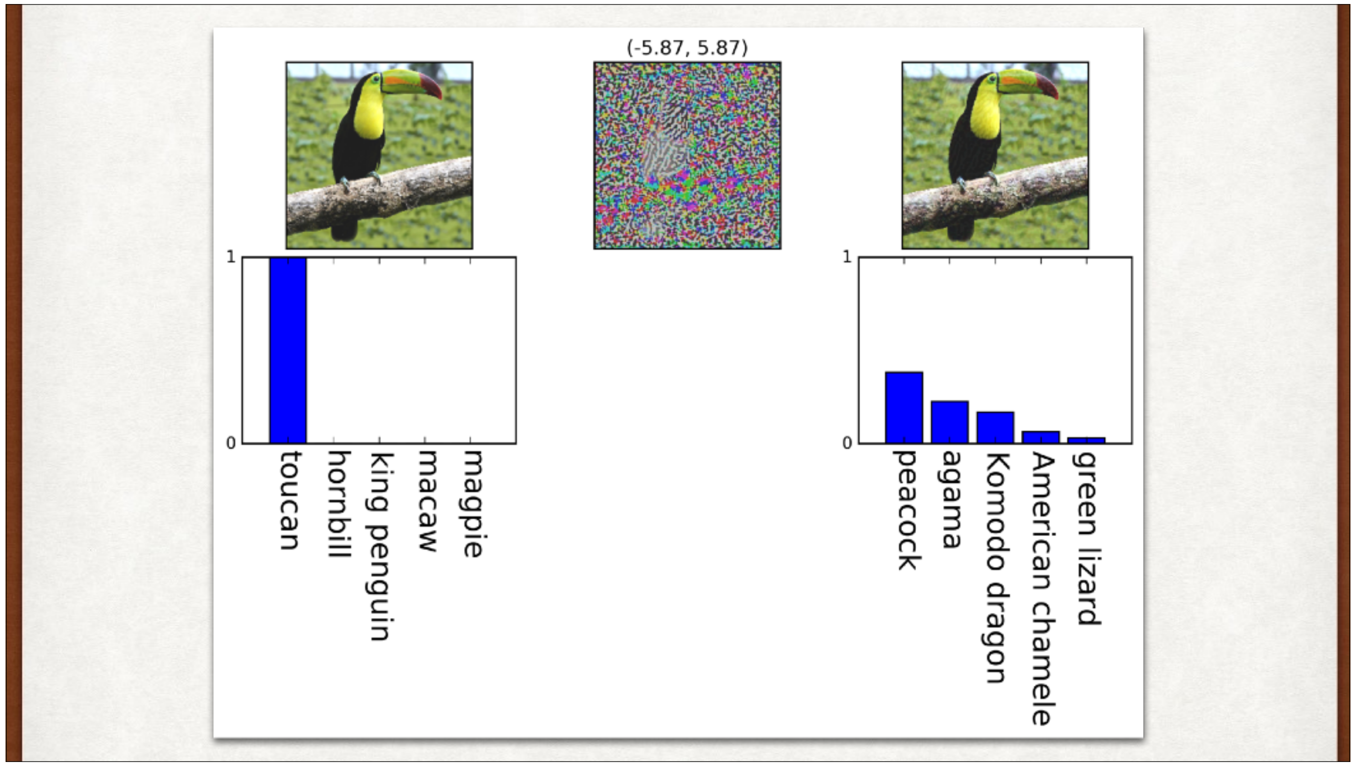
A tiger (left) is correctly identified, but add some carefully constructed noise (middle) in the range of about (-2,2), and VGG16's predictions (right) are… weird.



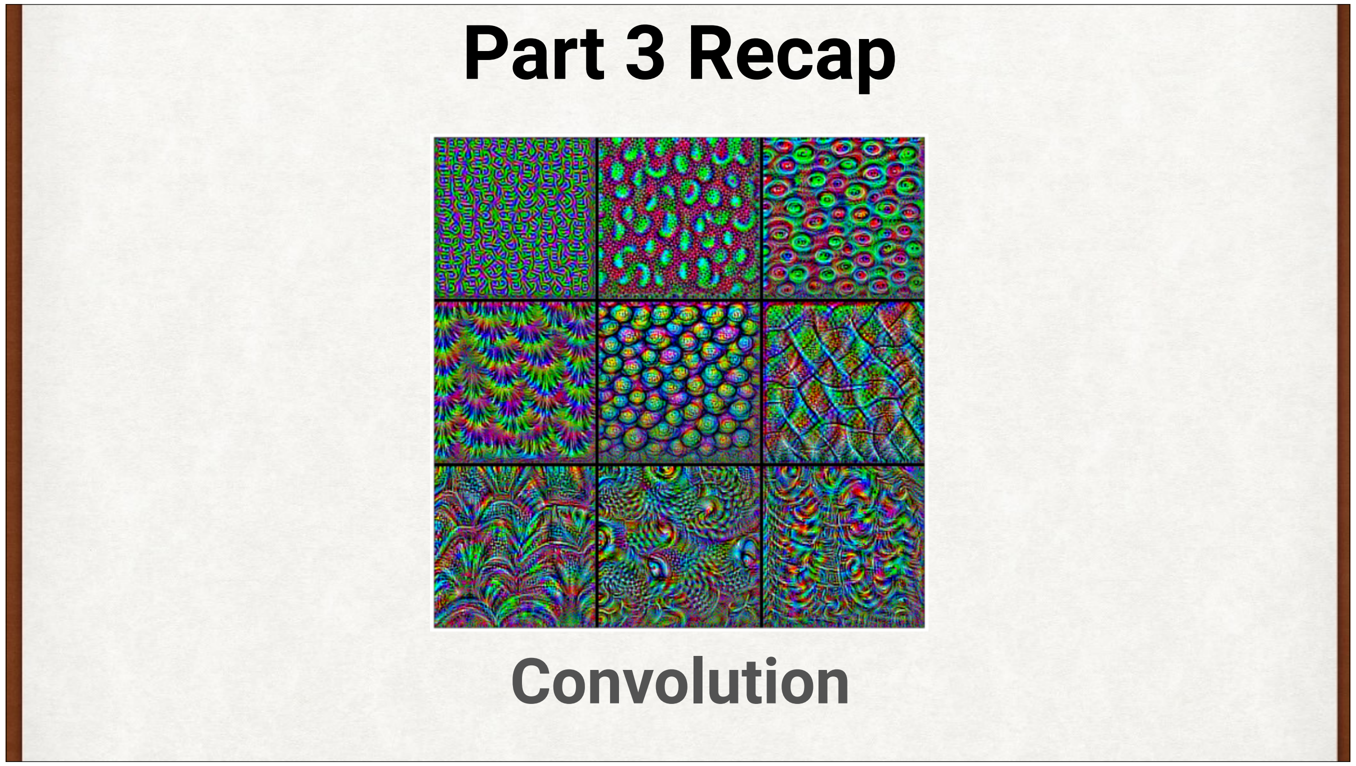
A tiger (left) is correctly identified, but add some carefully constructed noise (middle) in the range of about (-2,2), and VGG16's predictions (right) are… weird.



A power drill, plus perturbation, is a syringe?



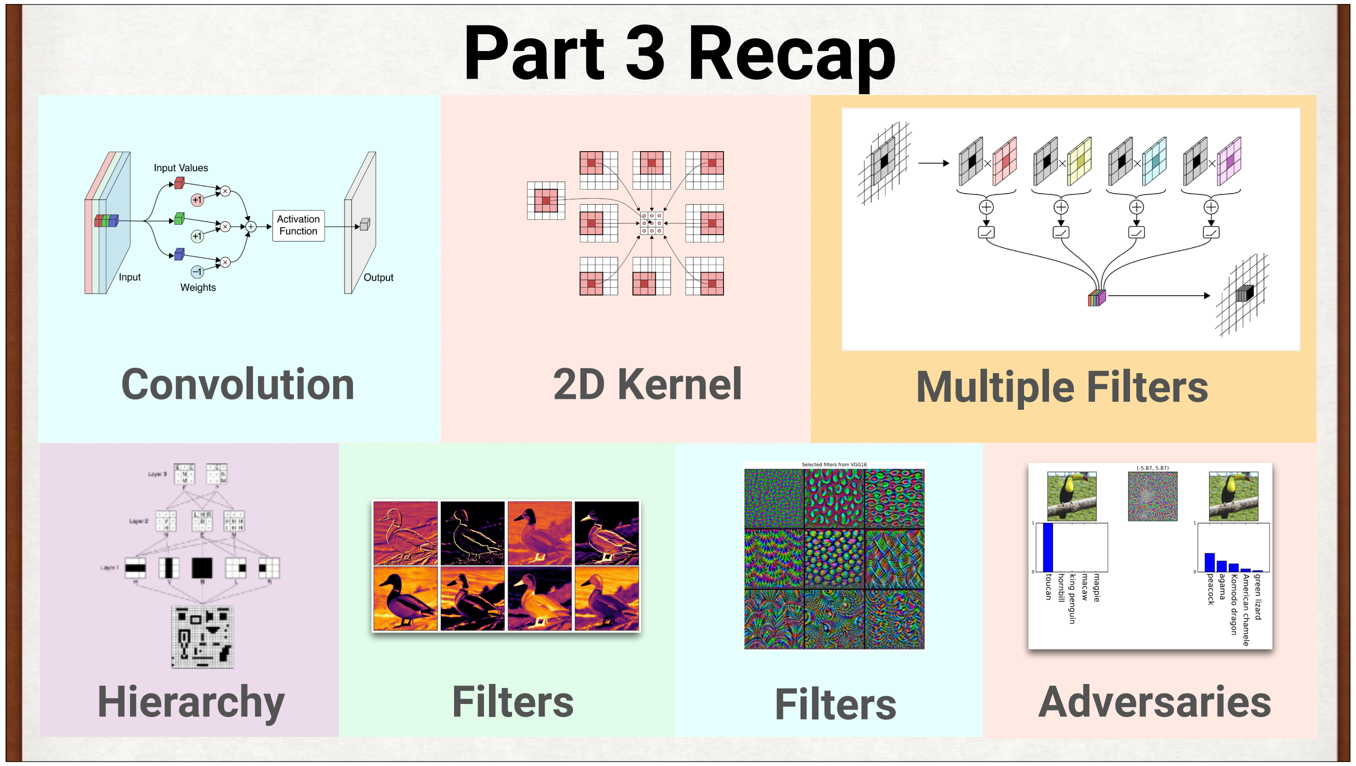
This is absolutely a toucan, but then probably a peacock, or one of several different lizards.



Part 3 Recap

Convolution

That was convolution!



Part 3 Recap

Convolution

2D Kernel

Multiple Filters

Hierarchy

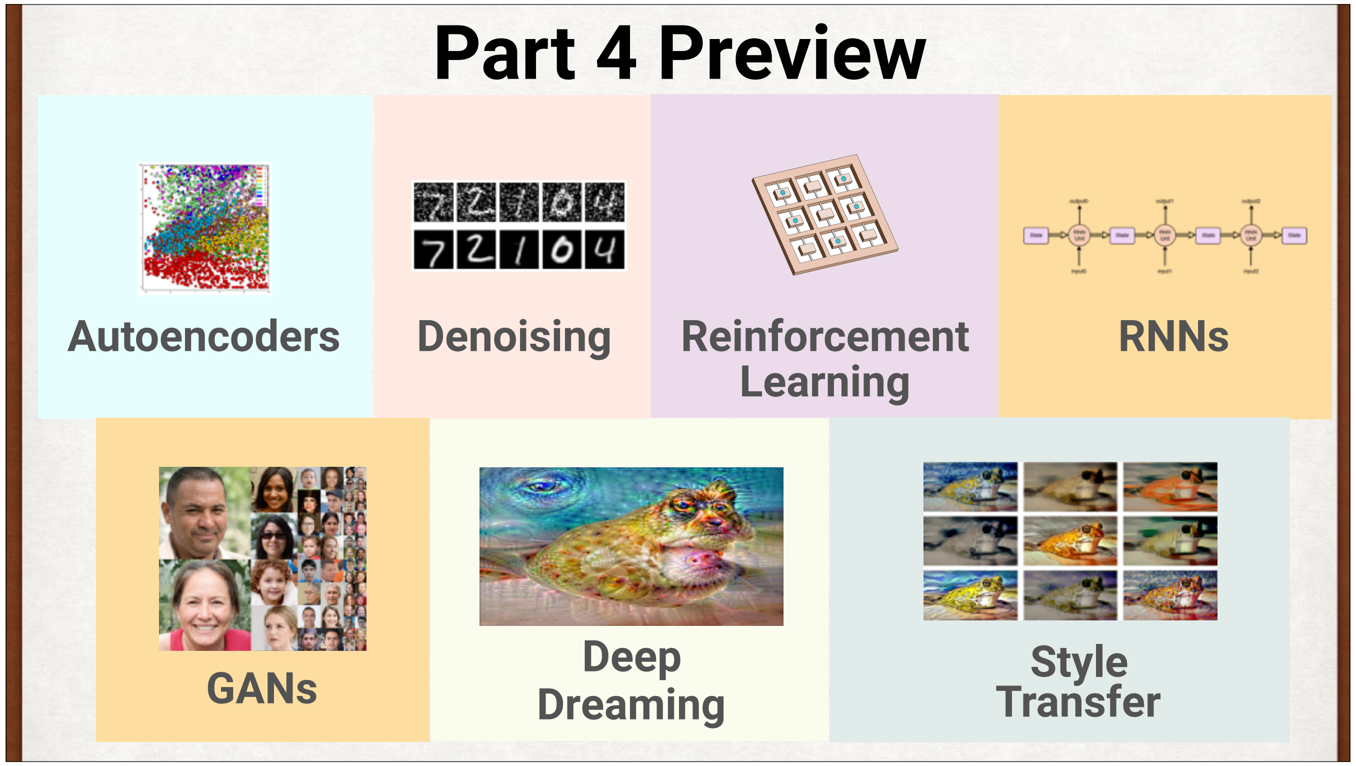
Filters

Filters

Adversaries

Recap.

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



Part 4 Preview

Autoencoders

Denoising

Reinforcement Learning

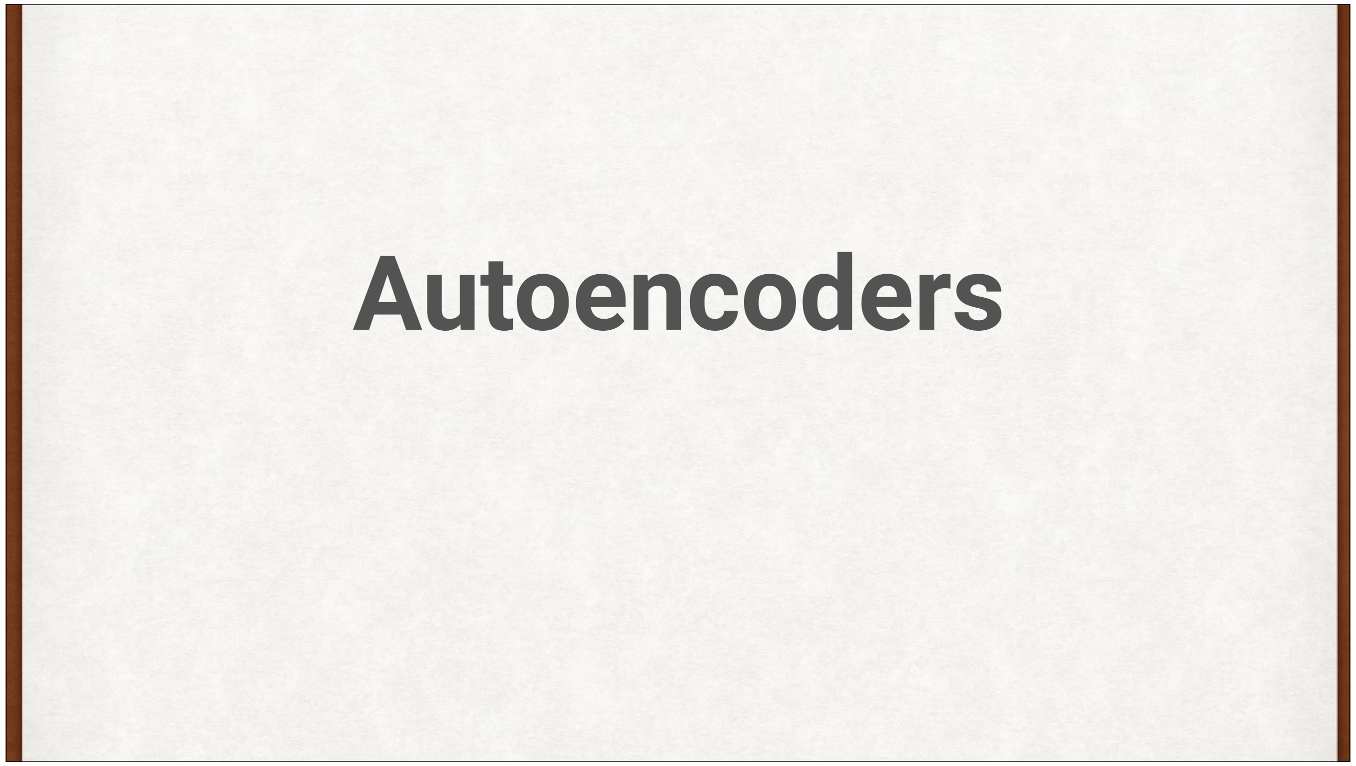
RNNs

GANs

Deep Dreaming

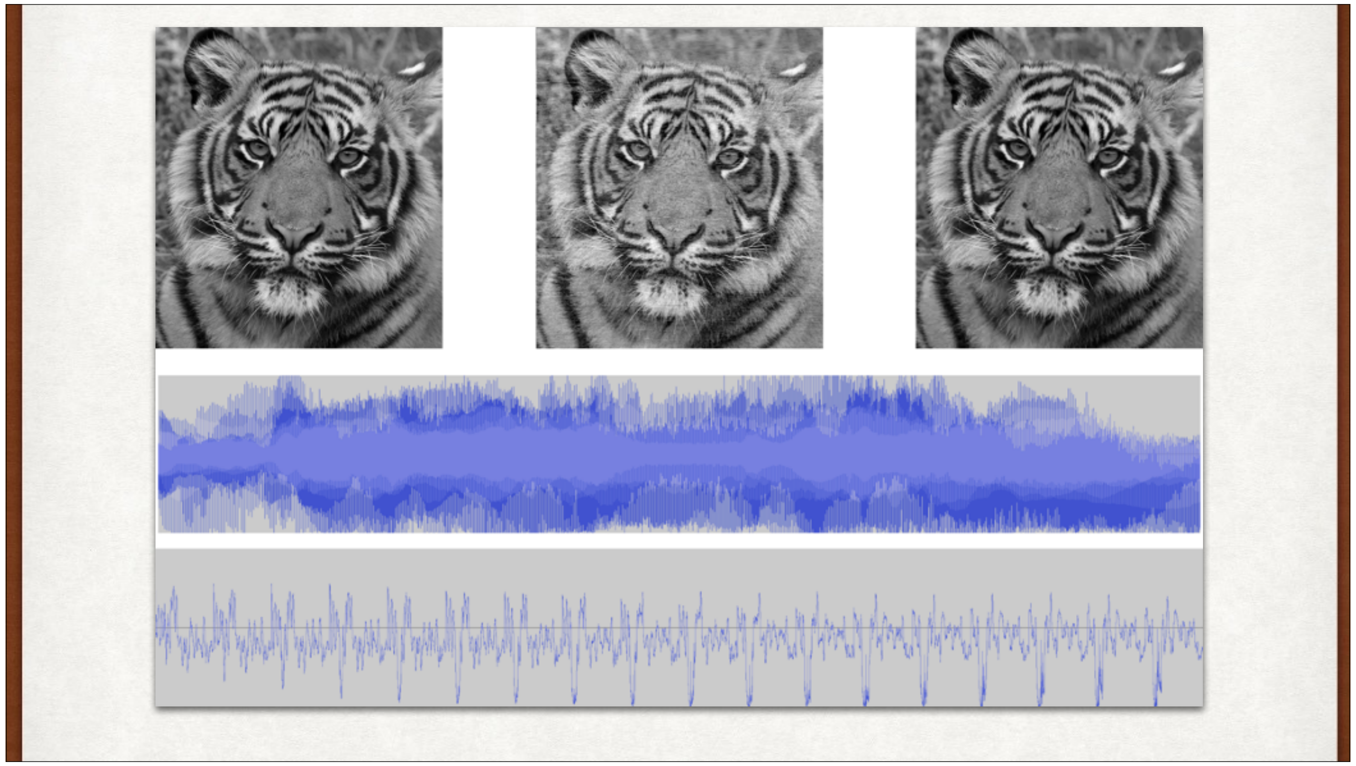
Style Transfer

Part 4’s topics.

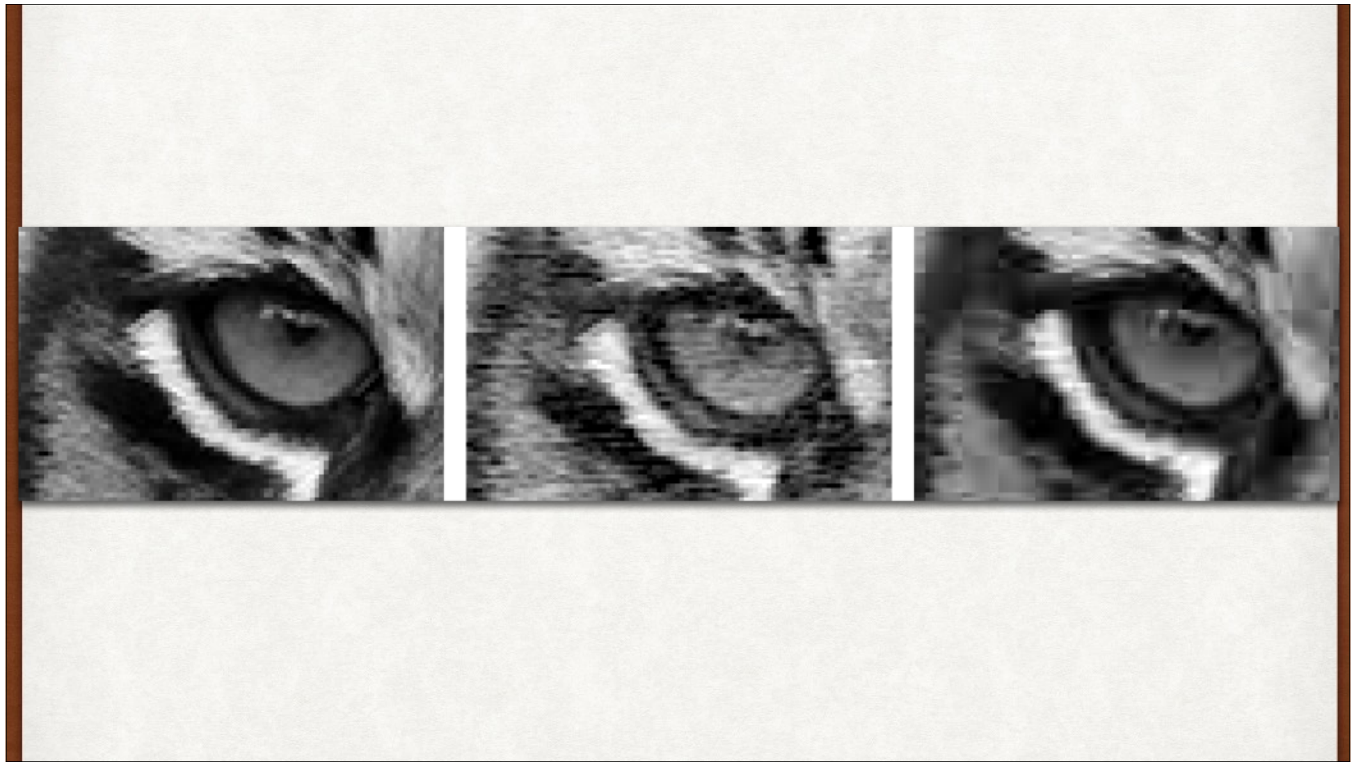


Autoencoders

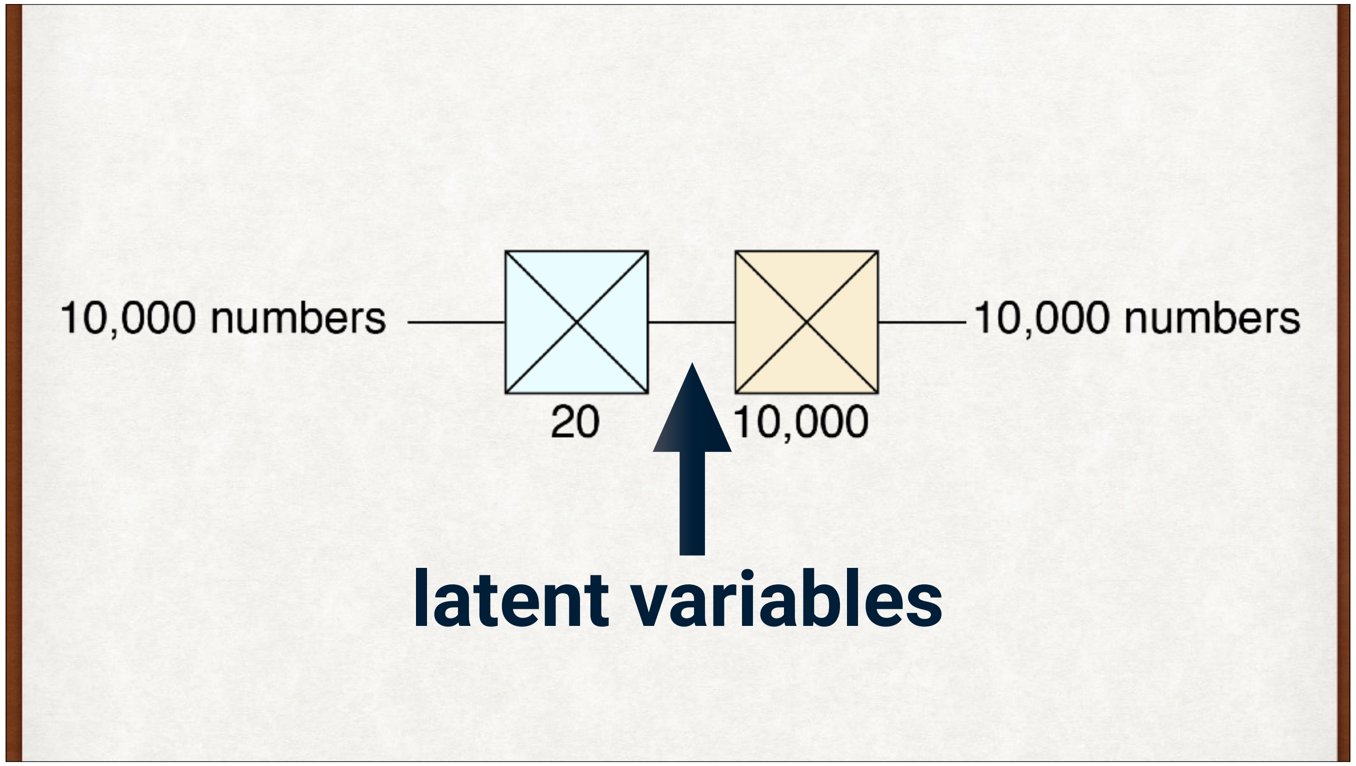
An autoencoder learns how to “automatically” compress, or “encode,” an input into a smaller size that can later be decoded into a version of the original that is acceptable for a given use.



Compression of an image. Upper-left: original tiger. Upper-middle: tiger data compressed with MP3 and then decompressed. Upper-right: compressed by JPG. Below, the original tiger data in WAV format. Bottom: Close-up of the start of the waveform. The dips correspond to the black tip of the ear in the upper left.

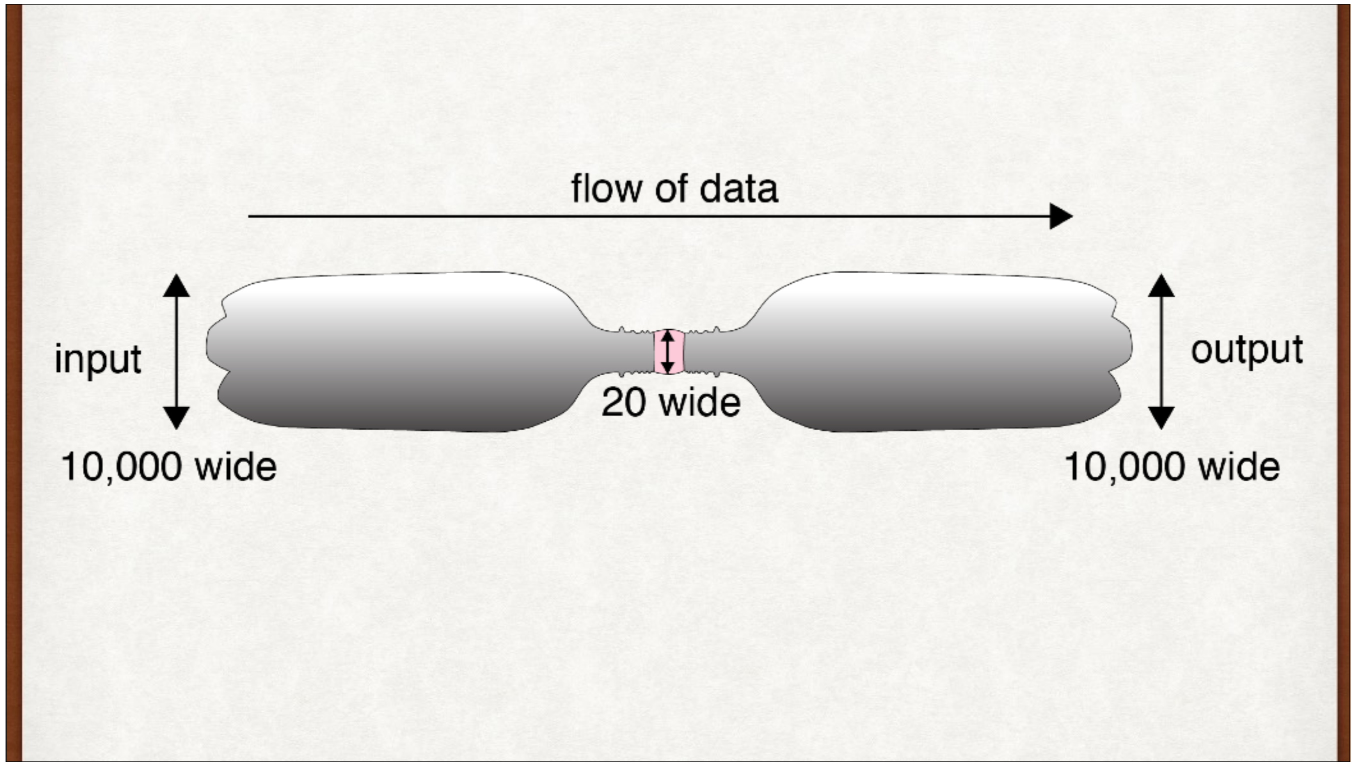


Closeups of the tiger’s eye. Original, MP3, and JPG. Though MP3 was designed for sounds (see the horizontal striping), it does remarkably well on this image.

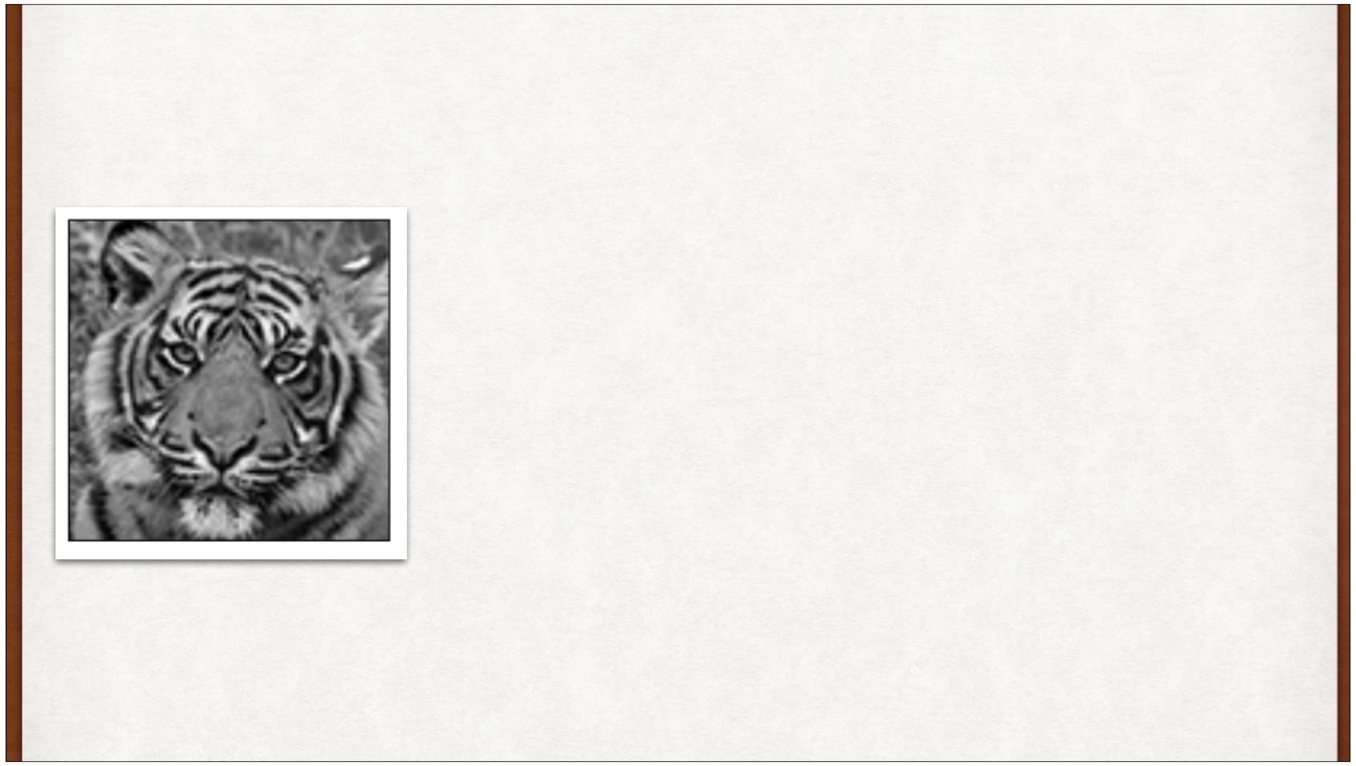


latent variables

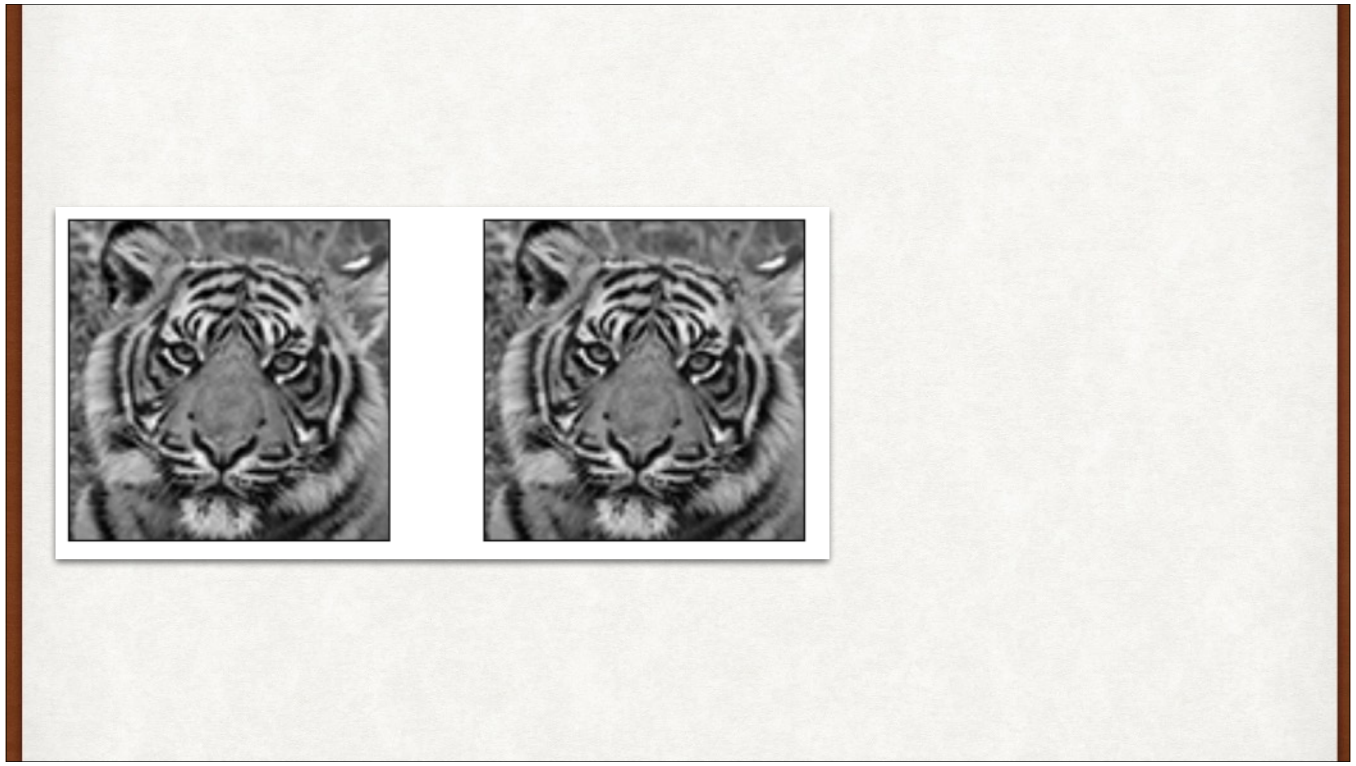
A tiny autoencoder. The compression stage, in blue, is a single fully-connected layer. The decompression stage, in beige, is another fully-connected layer. There are 20 “latent variables” in the middle, so the network, after training, will do its best to compress the 10,000 element input into just 20 numbers.



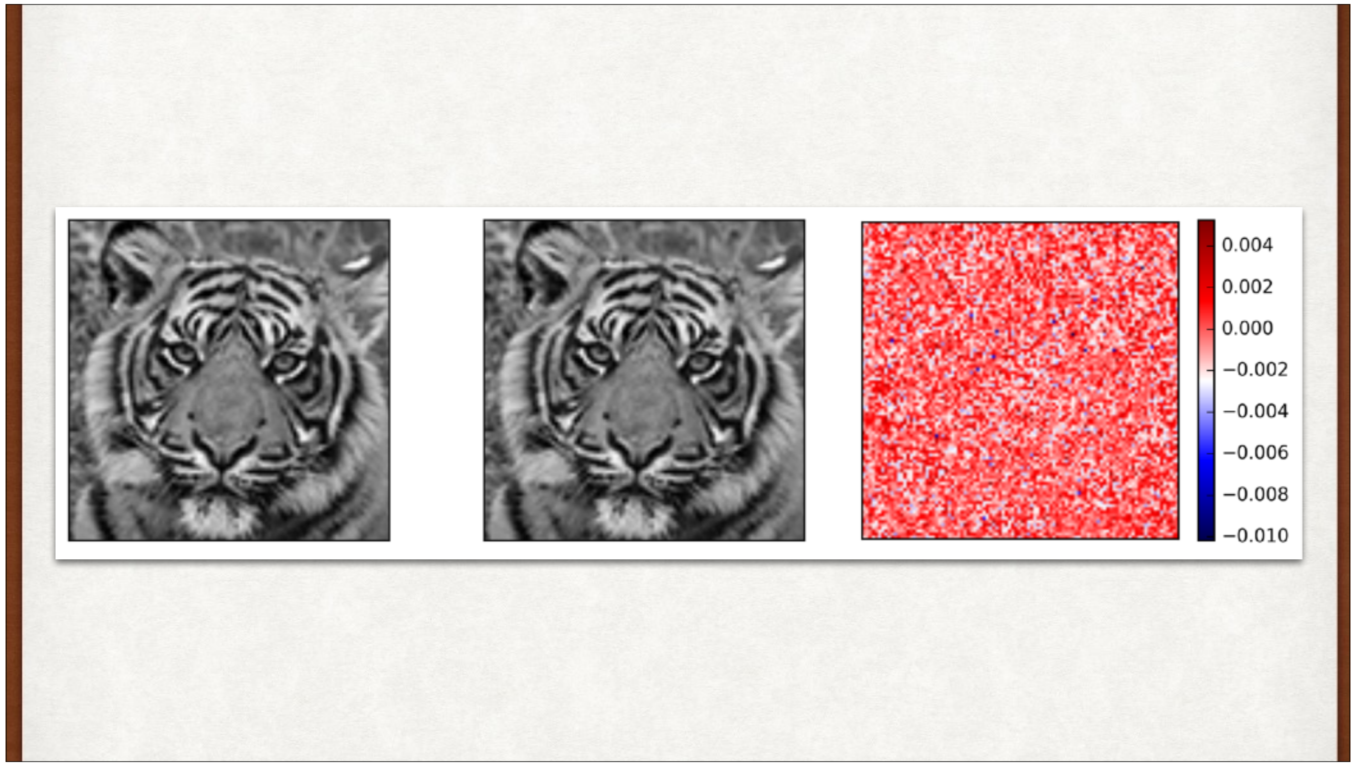
Many autoencoders have a “bottleneck” where the data is compressed to a minimum.



We trained the autoencoder on the picture of the tiger. Here are the results. Left: the input, 100x100. Middle: the autoencoder's output. Right: The pixel-by-pixel differences between input and output. The autoencoder seems to have found an amazing way to compress images super well with only 20/10,000 or 0.002% of the data.

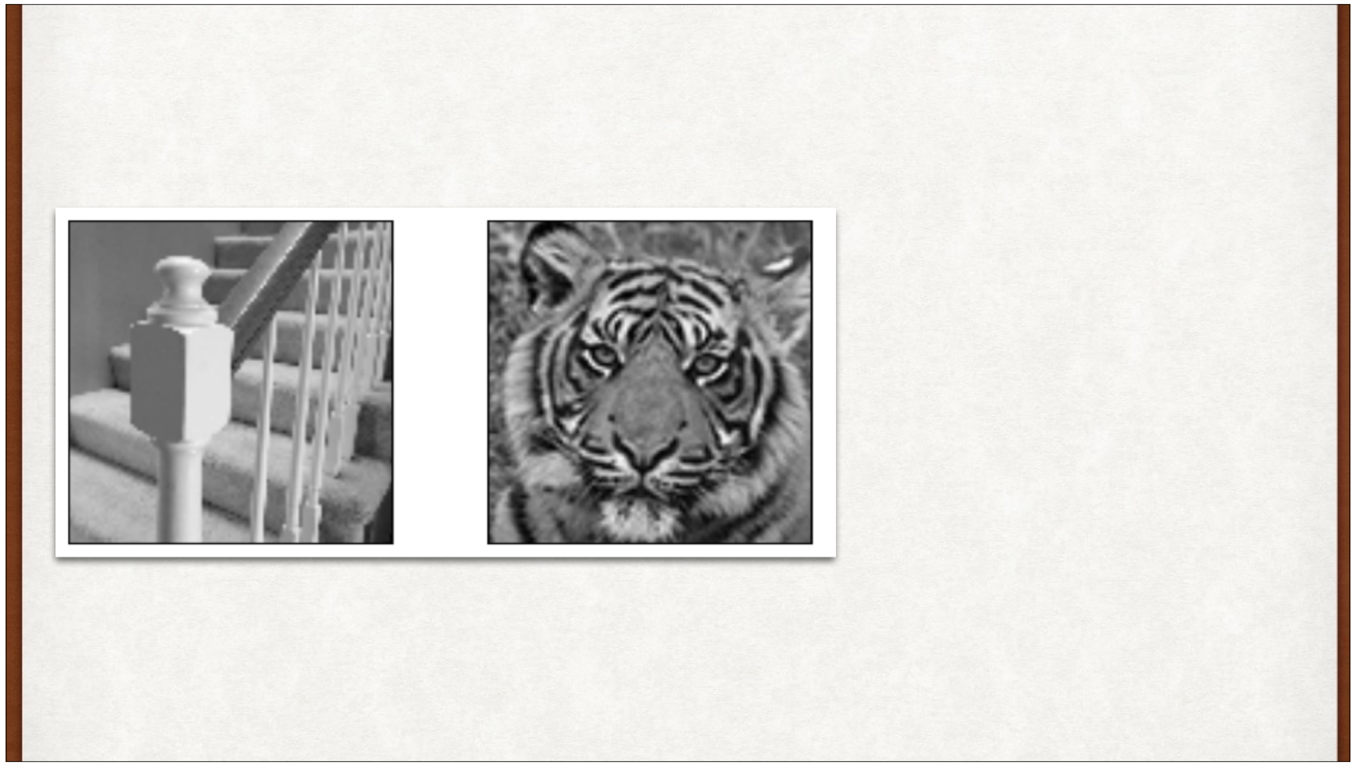


We trained the autoencoder on the picture of the tiger. Here are the results. Left: the input, 100x100. Middle: the autoencoder’s output. Right: The pixel-by-pixel differences between input and output. The autoencoder seems to have found an amazing way to compress images super well with only 20/10,000 or 0.002% of the data.

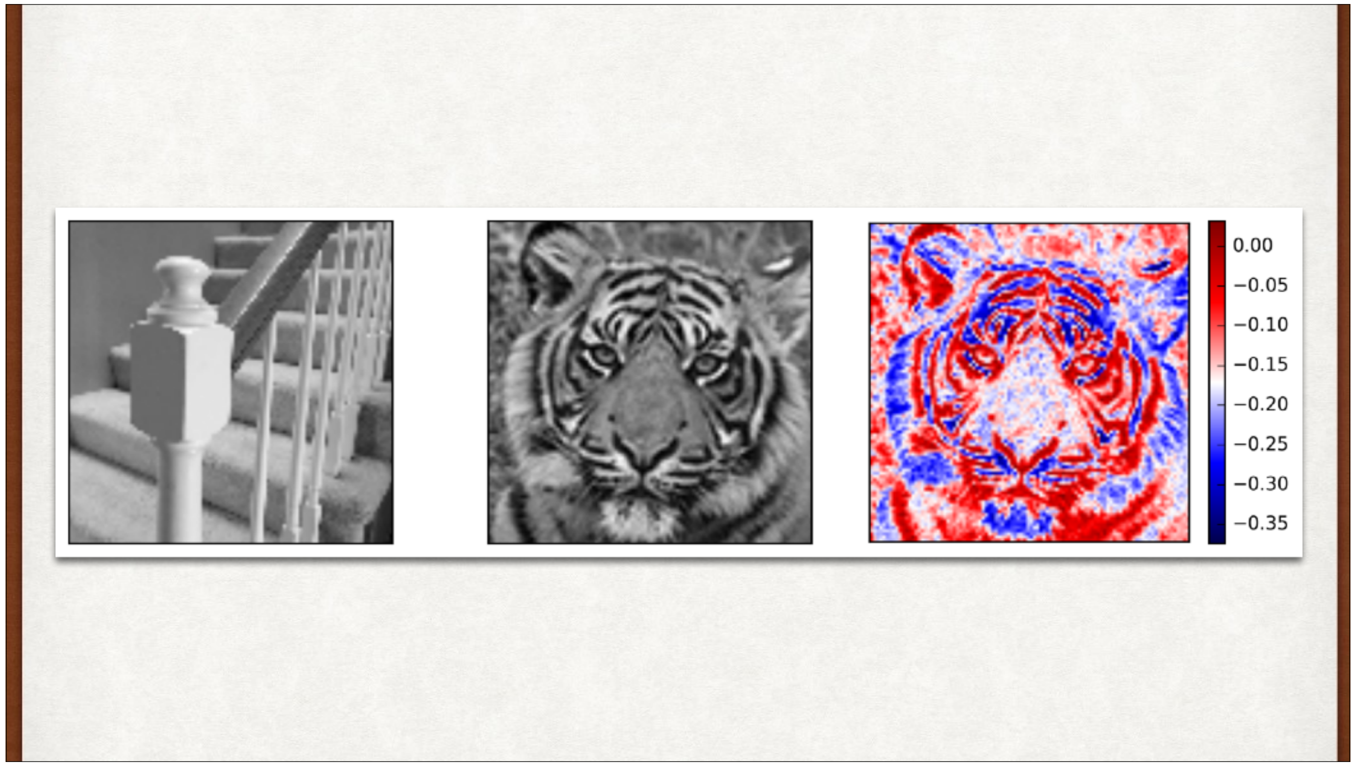


We trained the autoencoder on the picture of the tiger. Here are the results. Left: the input, 100x100. Middle: the autoencoder’s output. Right: The pixel-by-pixel differences between input and output. The autoencoder seems to have found an amazing way to compress images super well with only 20/10,000 or 0.002% of the data.

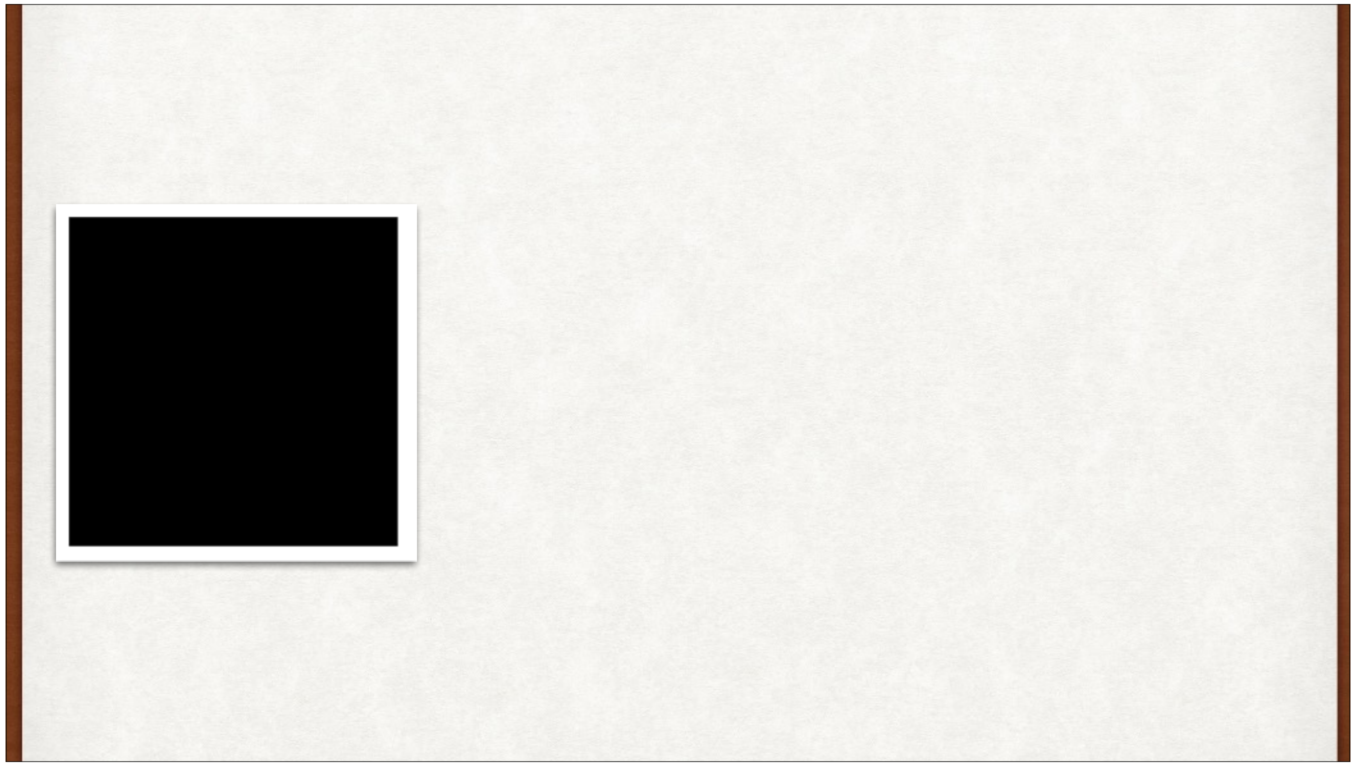
Let’s try feeding the autoencoder a picture of a bannister. Left: input. Middle: the output is the tiger! What? Right: pixel differences.



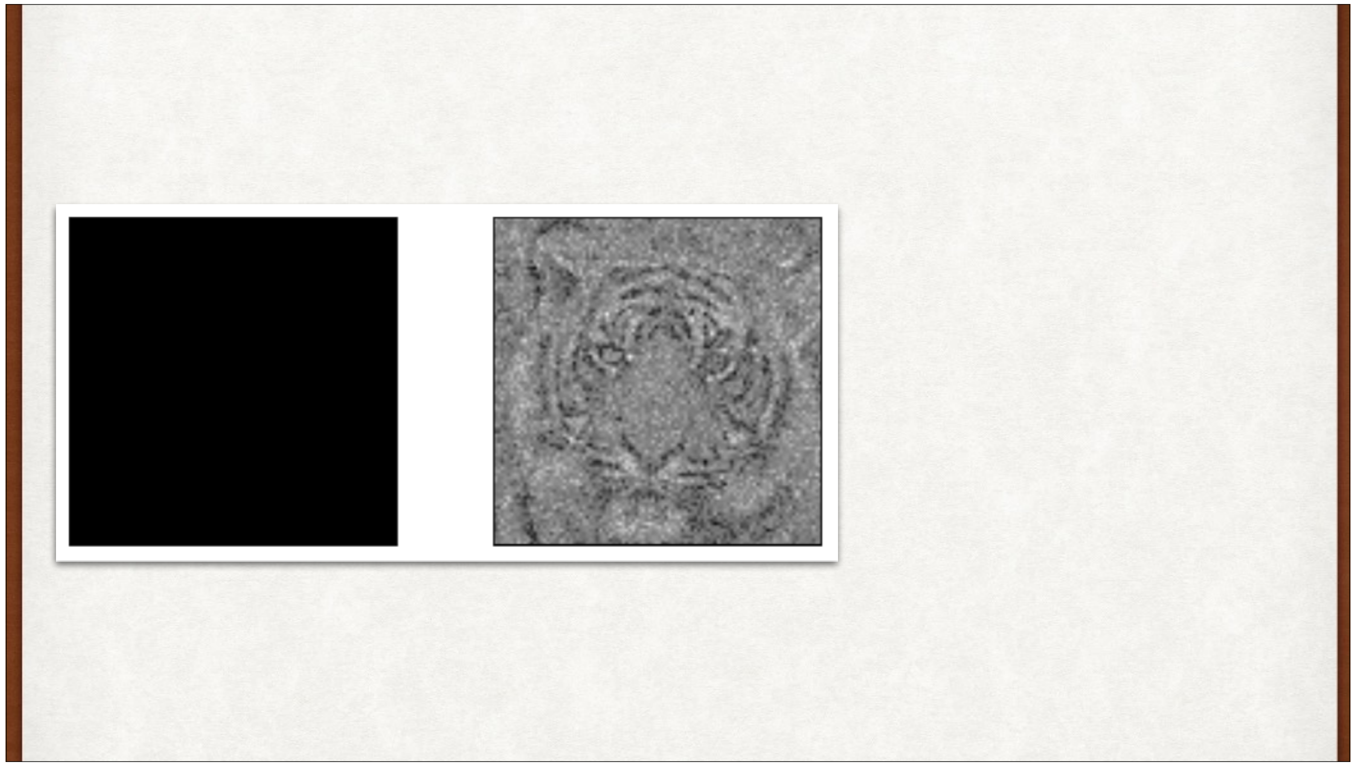
Let’s try feeding the autoencoder a picture of a bannister. Left: input. Middle: the output is the tiger! What? Right: pixel differences.



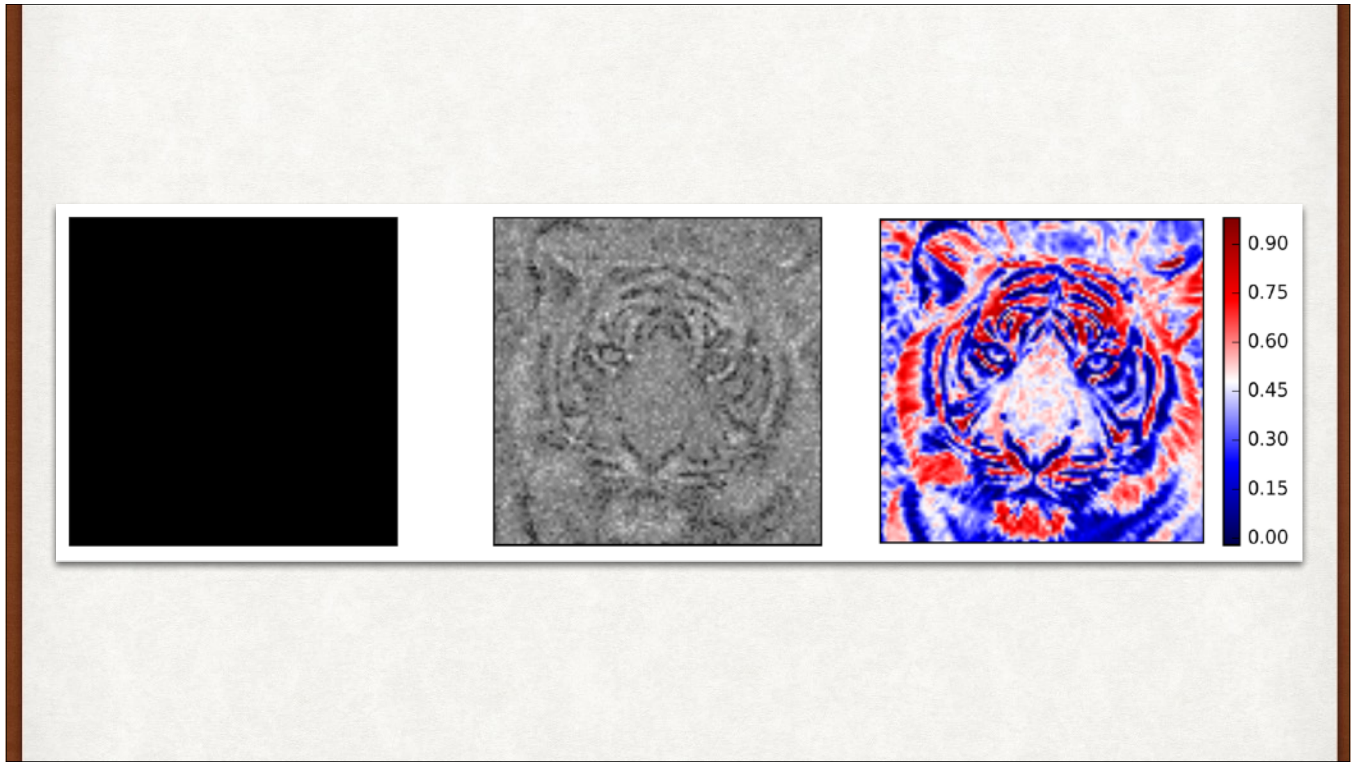
Let’s try feeding the autoencoder a picture of a bannister. Left: input. Middle: the output is the tiger! What? Right: pixel differences.



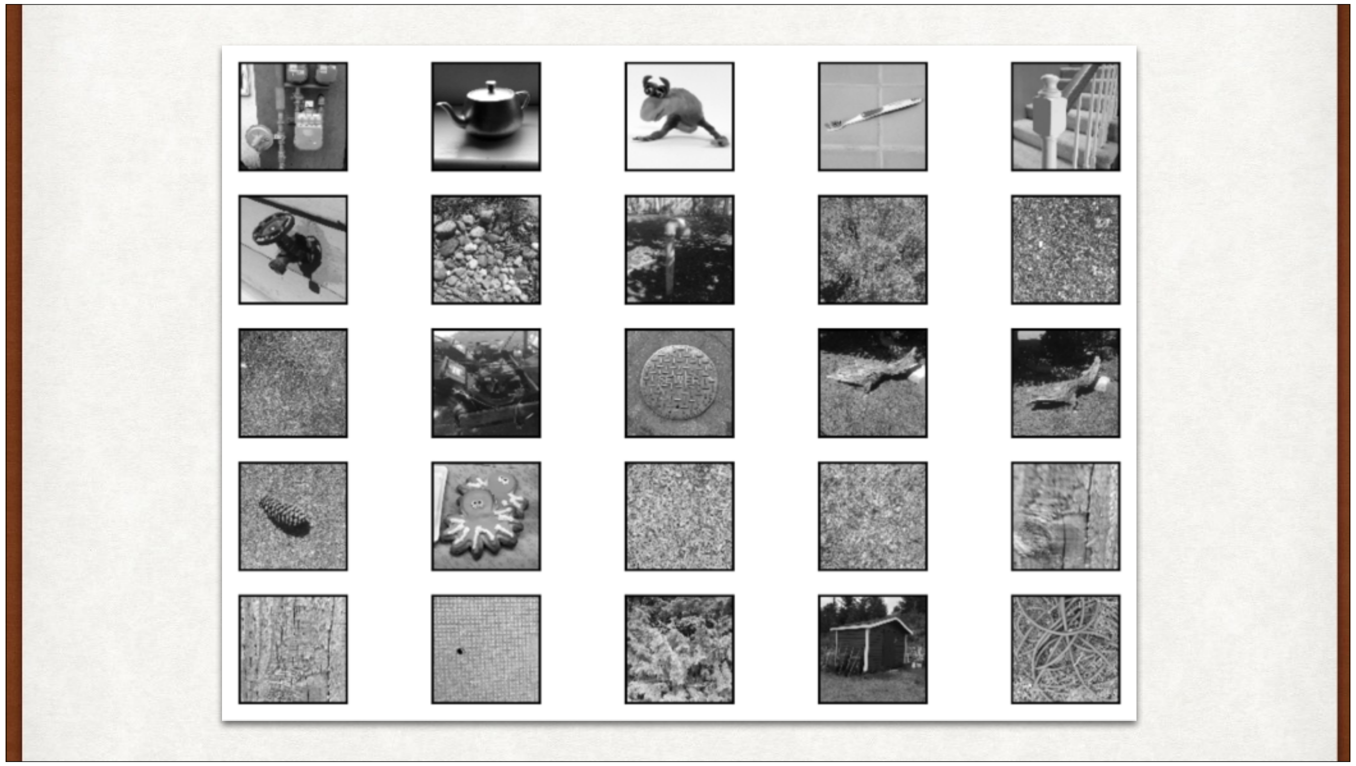
A black input (left) still produces the tiger (middle). Differences on the far right. So this network isn't compressing anything, really, it's just memorized how to produce a tiger image. The bias values in the neurons have “remembered” values that cause the system to produce a tiger.



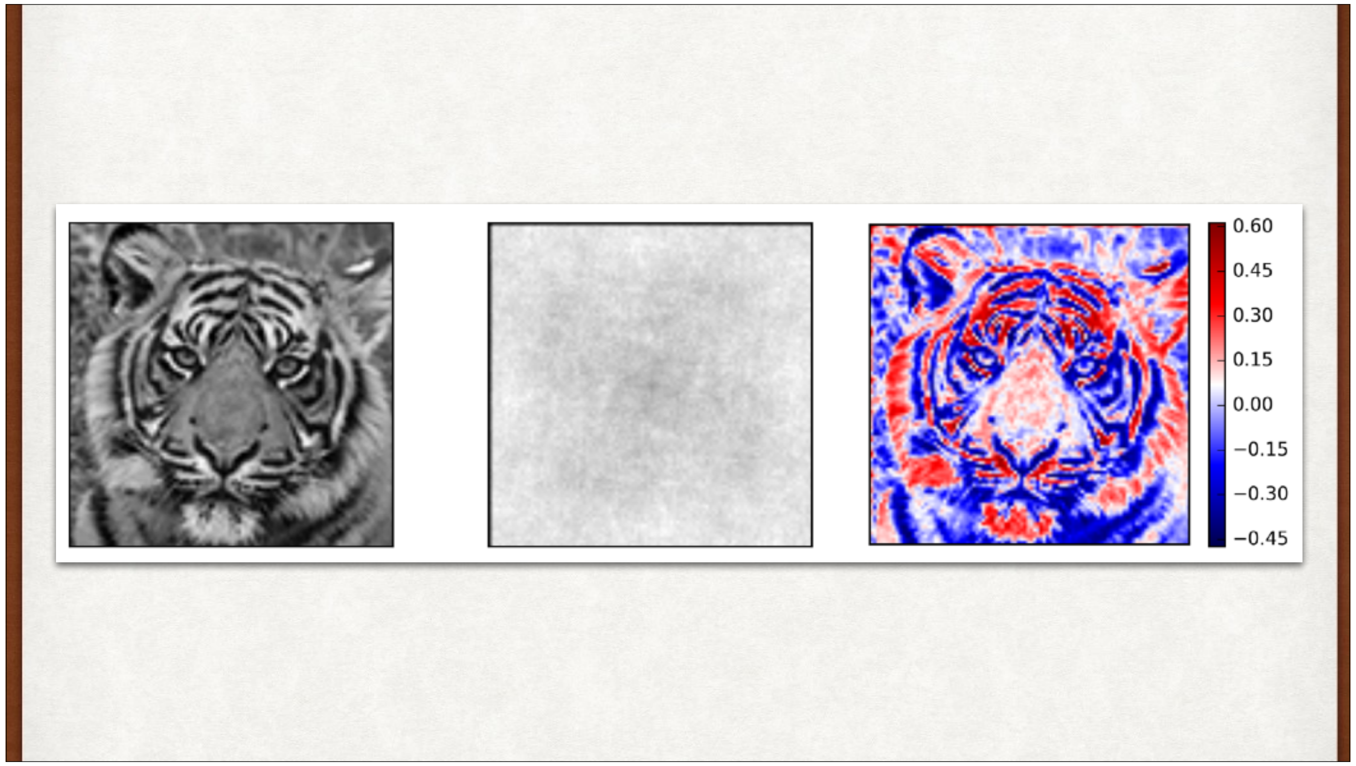
A black input (left) still produces the tiger (middle). Differences on the far right. So this network isn’t compressing anything, really, it’s just memorized how to produce a tiger image. The bias values in the neurons have “remembered” values that cause the system to produce a tiger.



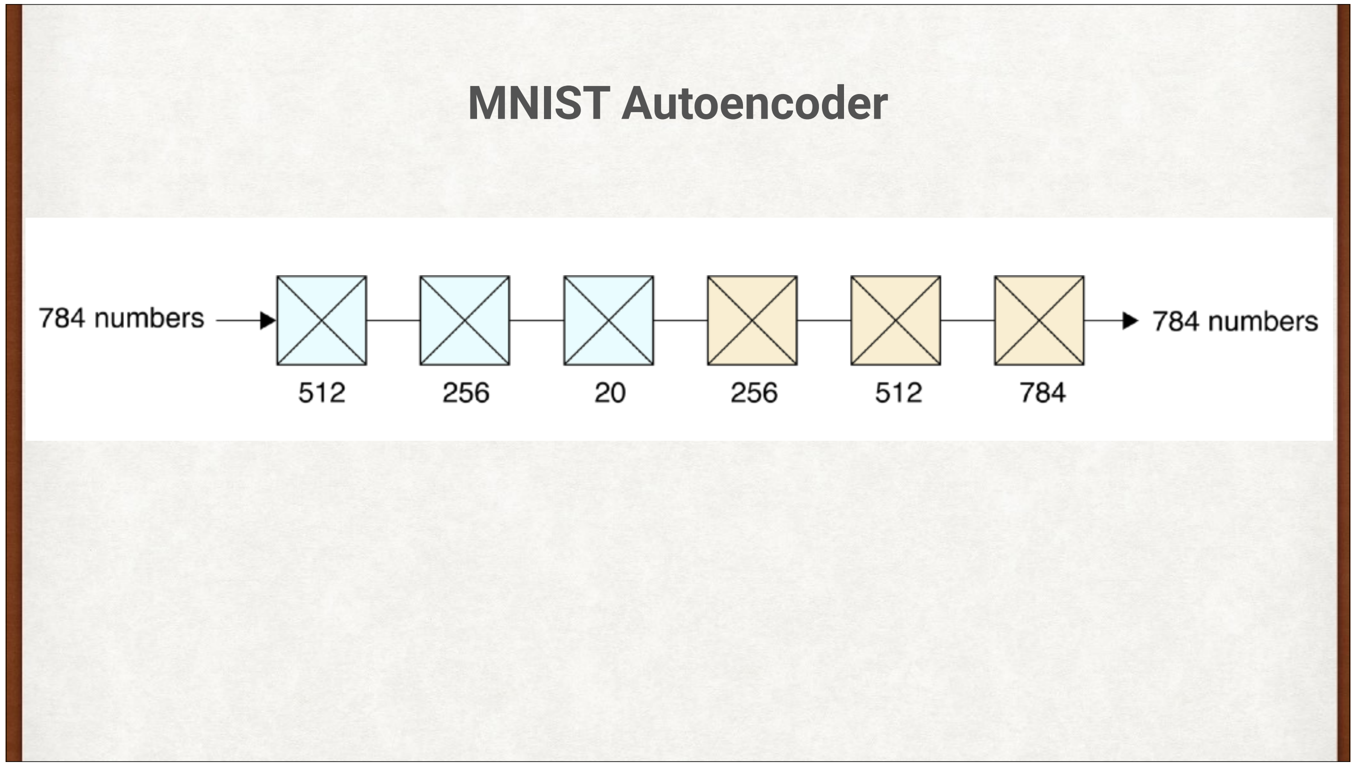
A black input (left) still produces the tiger (middle). Differences on the far right. So this network isn’t compressing anything, really, it’s just memorized how to produce a tiger image. The bias values in the neurons have “remembered” values that cause the system to produce a tiger.



A more diverse training set. We trained with each image in each of 4 rotations for 100 images.

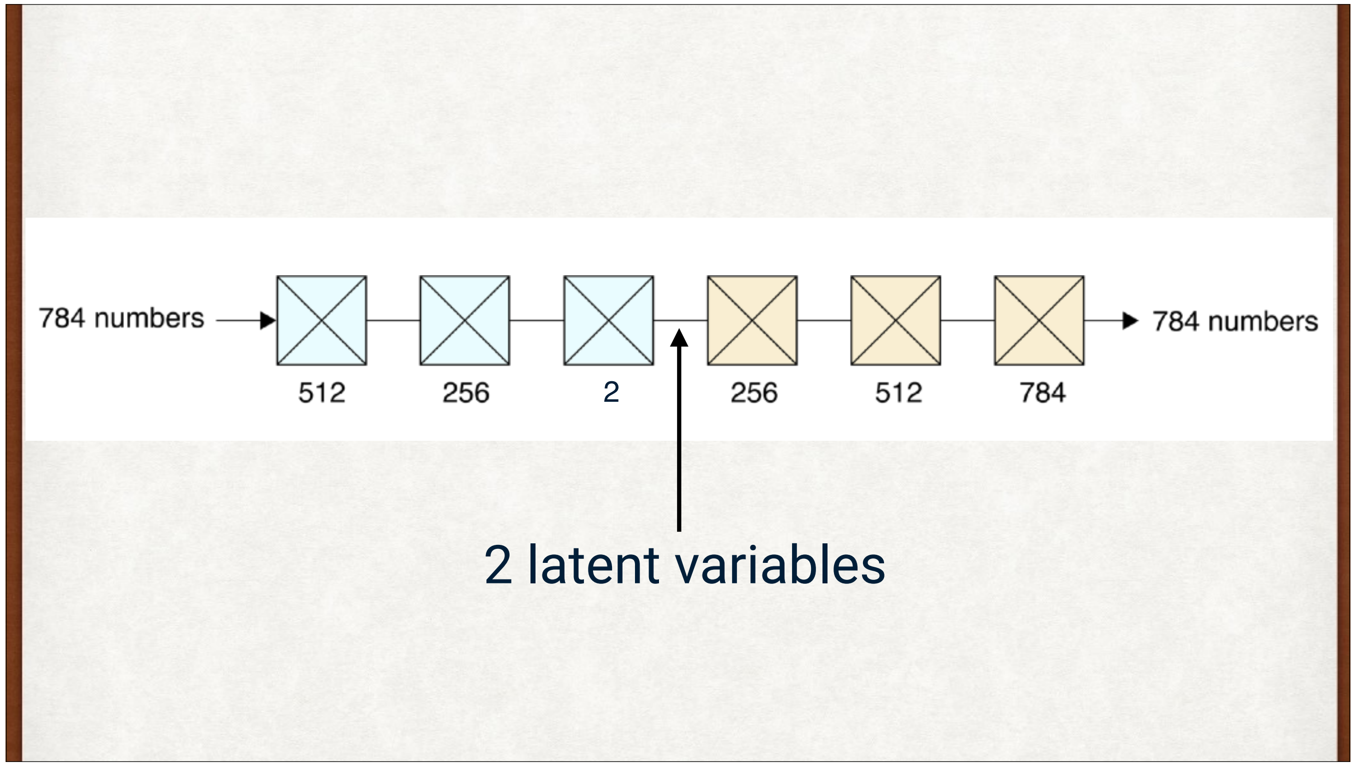


Not so amazing after all! When trained on diverse images, turning a whole picture into just 20 numbers isn’t going to let us get back much of anything. Left: input. Middle: output. Right: pixel differences.



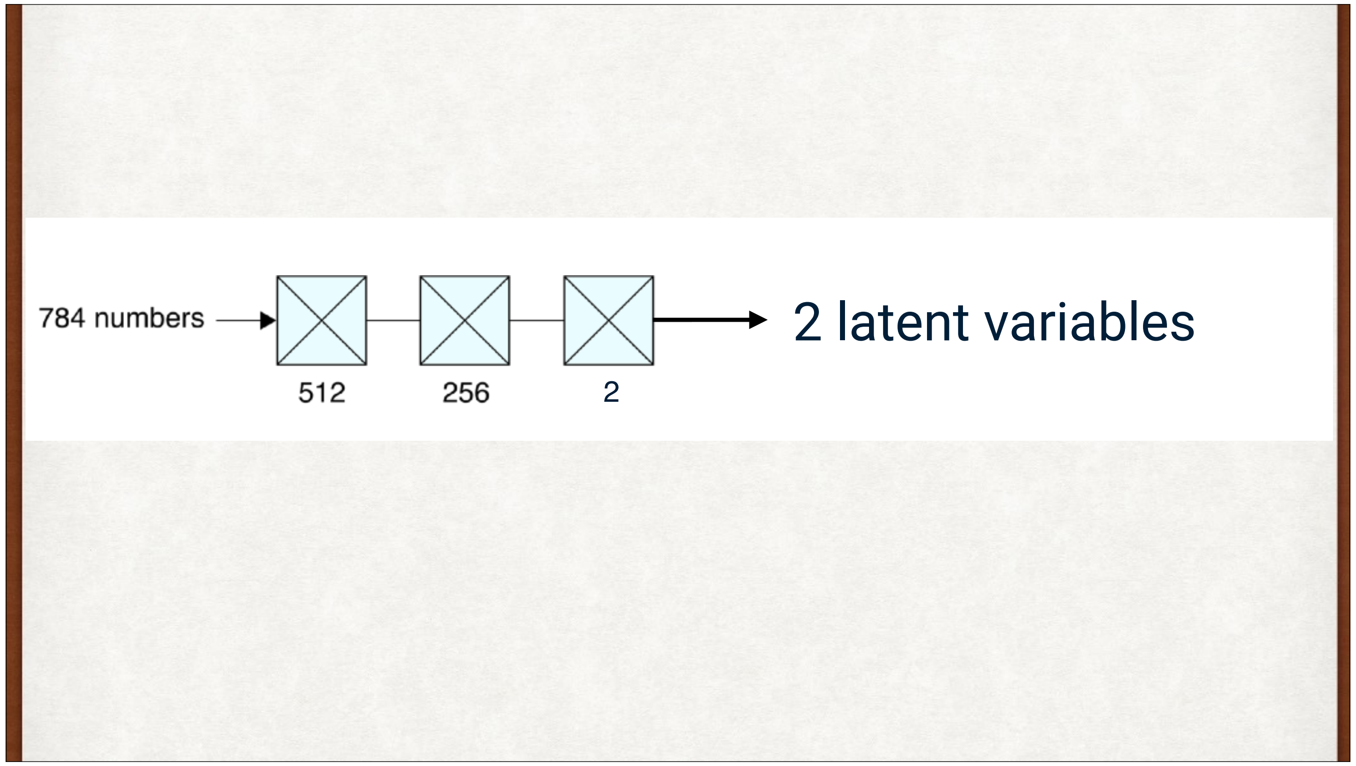
MNIST Autoencoder

An autoencoder for MNIST data, with more fully-connected layers.



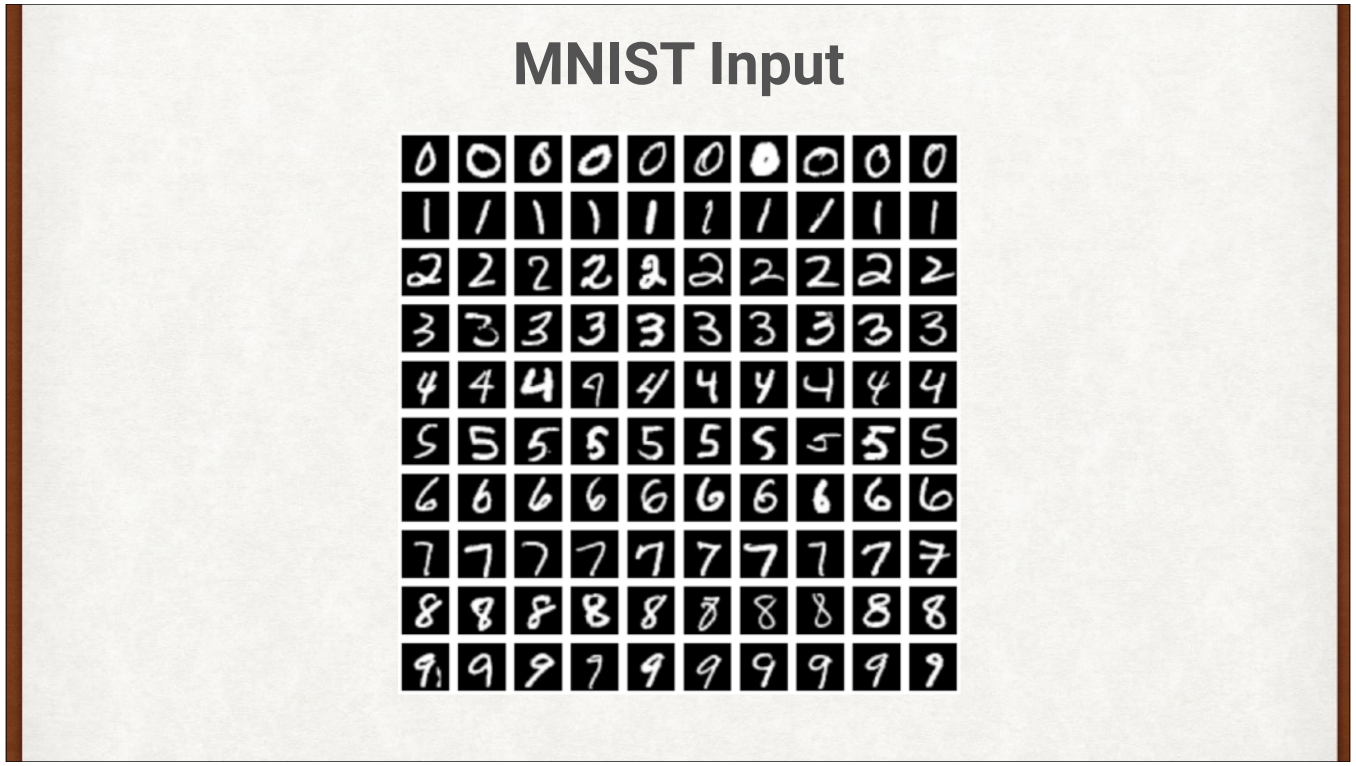
2 latent variables

We’ll feed in the picture, and get back 2 latents. We can plot those.



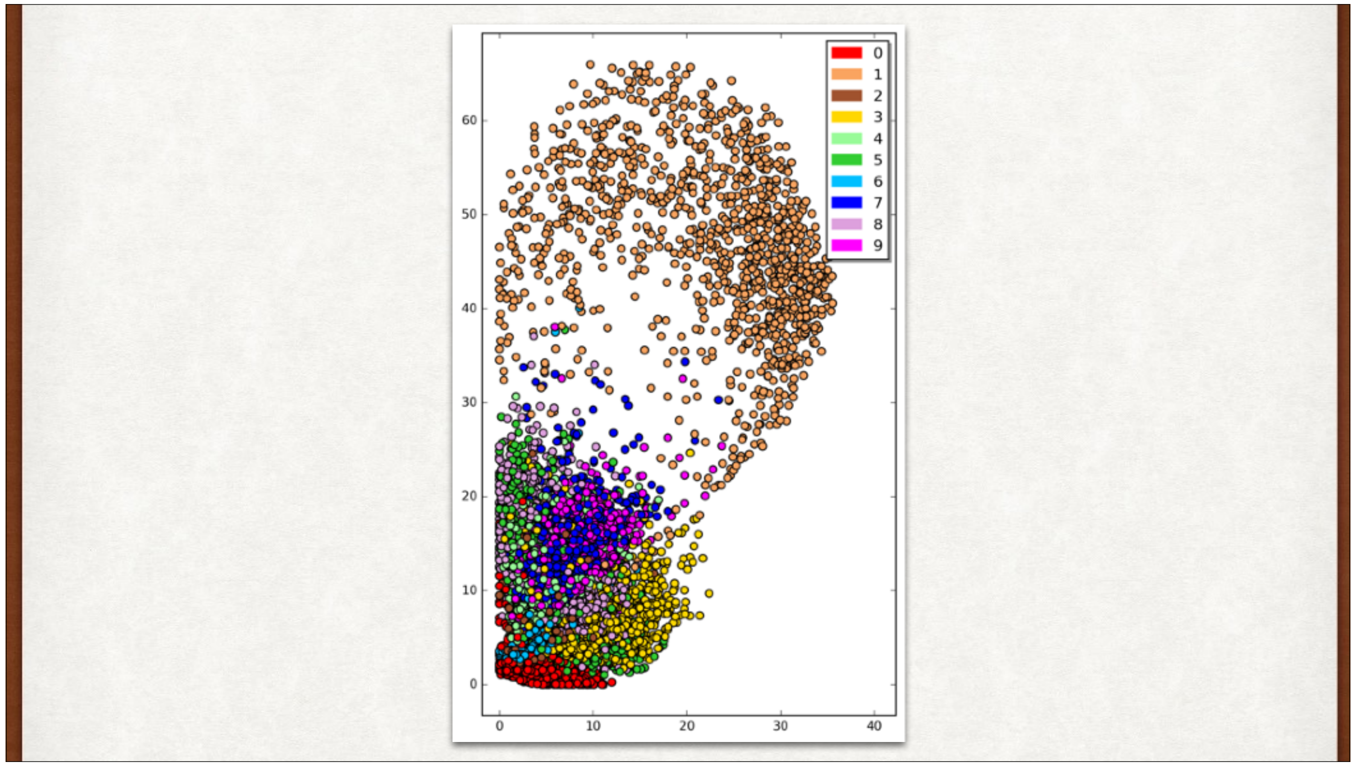
2 latent variables

Let’s reduce this down to just 2 latent variables.

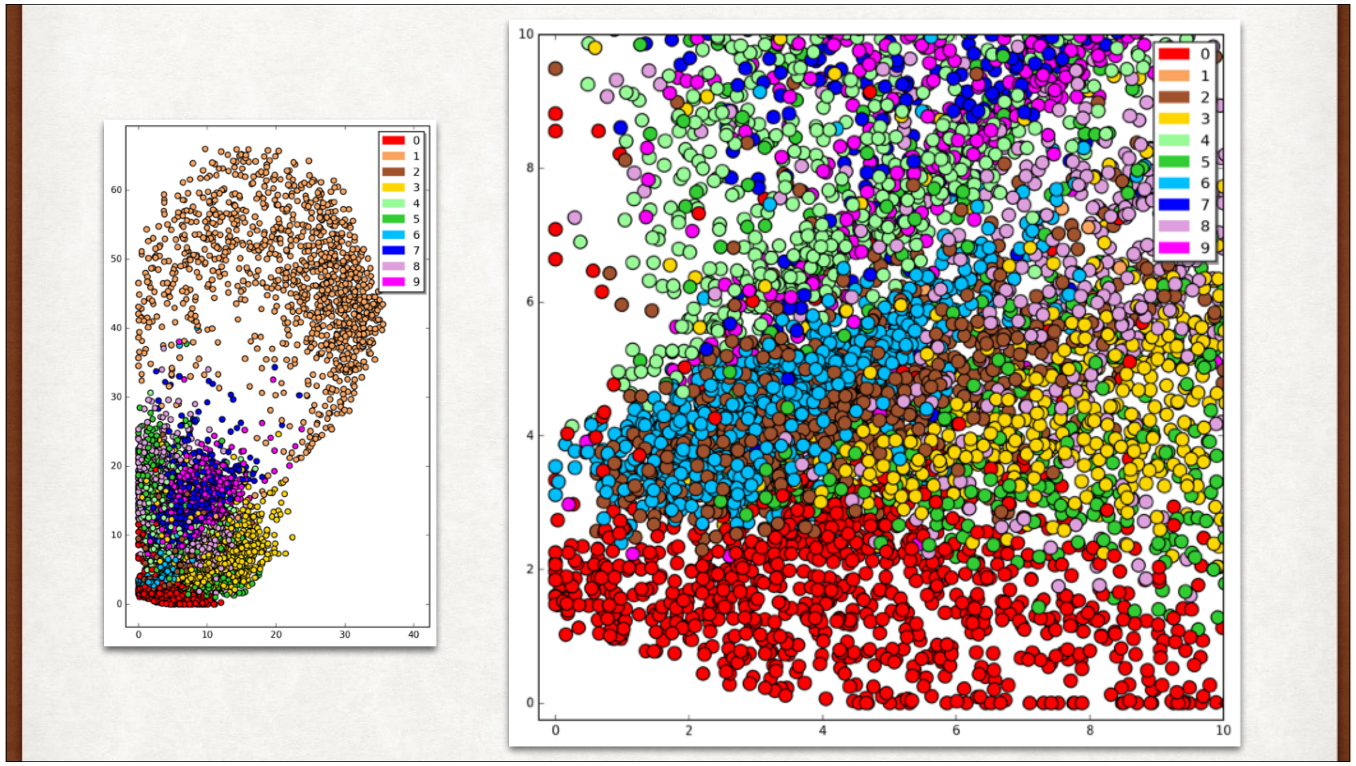


MNIST Input

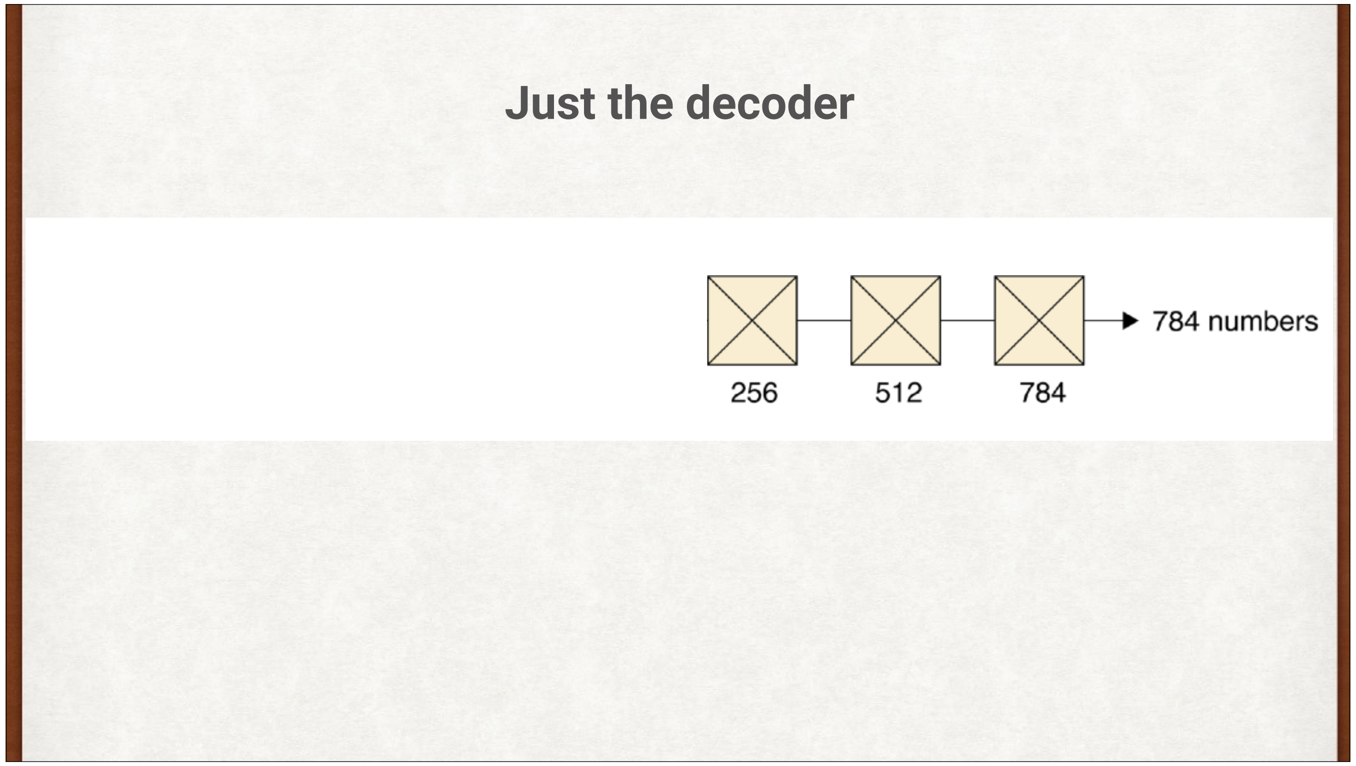
Now let’s try the MNIST data, which is way easier than the tiger.



If we use just 2 latent variables (which is not enough to do a good job on an input of 784 elements), we can visualize the results. Notice how much structure there is in the values assigned to pictures of different digits.

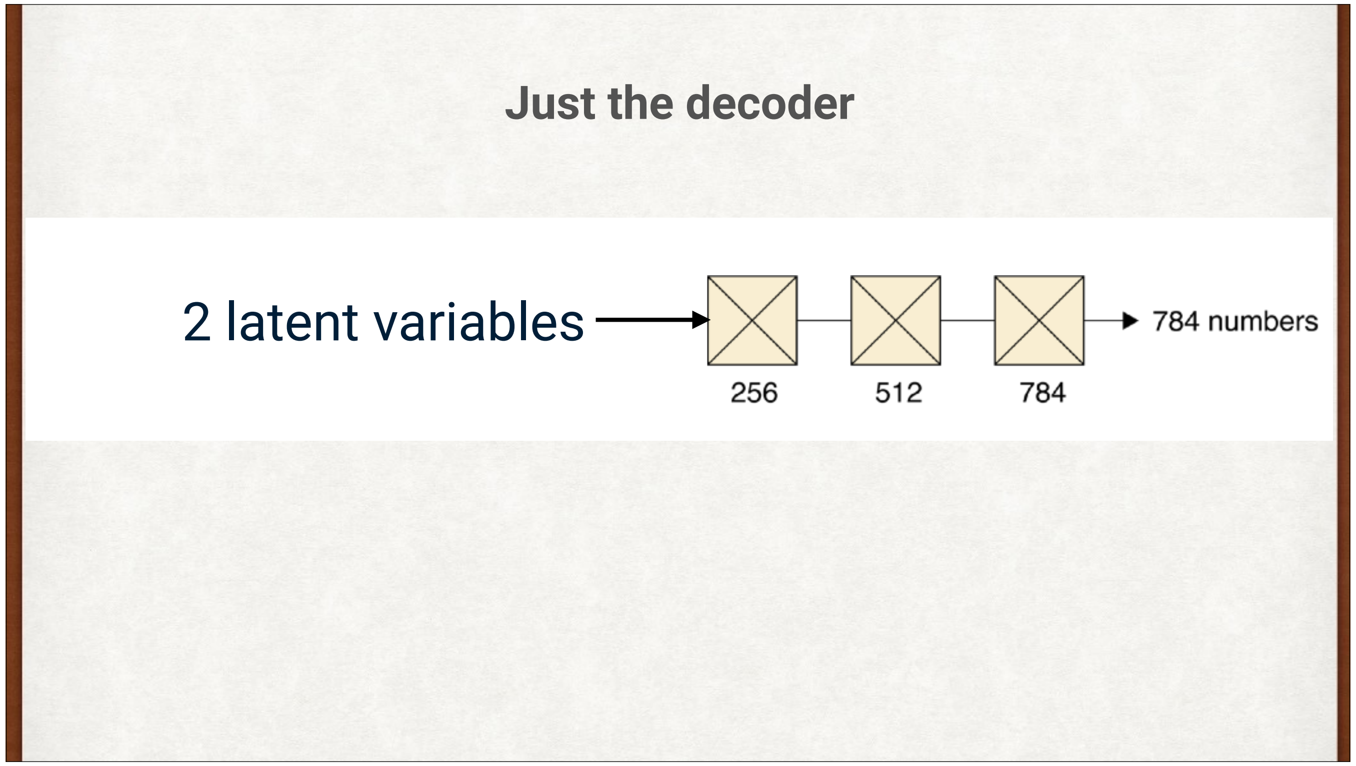


A close-up of the bottom left.



Just the decoder

Let’s try feeding it pairs of latent variables.



Just the decoder

2 latent variables

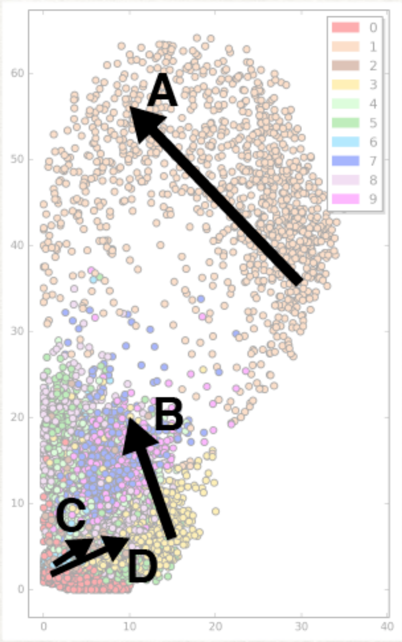
256

512

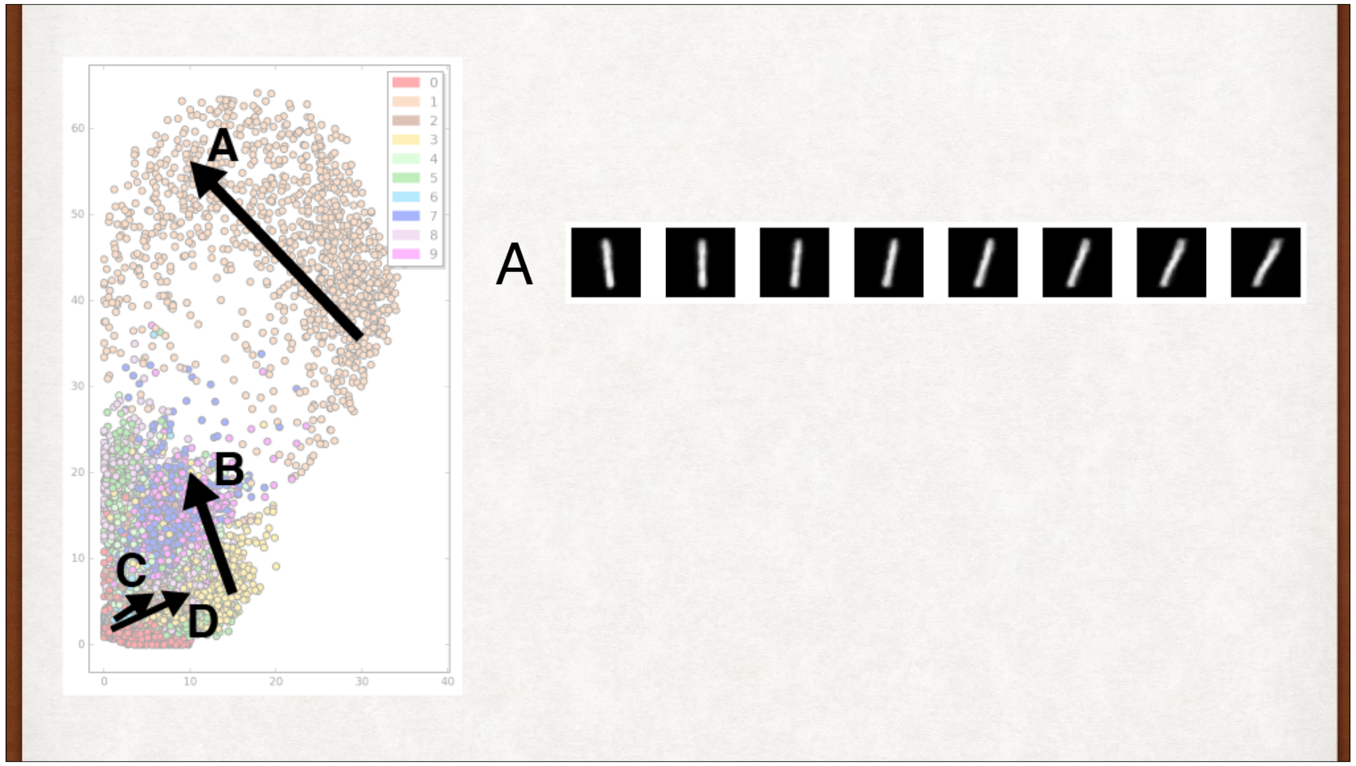
784

784 numbers

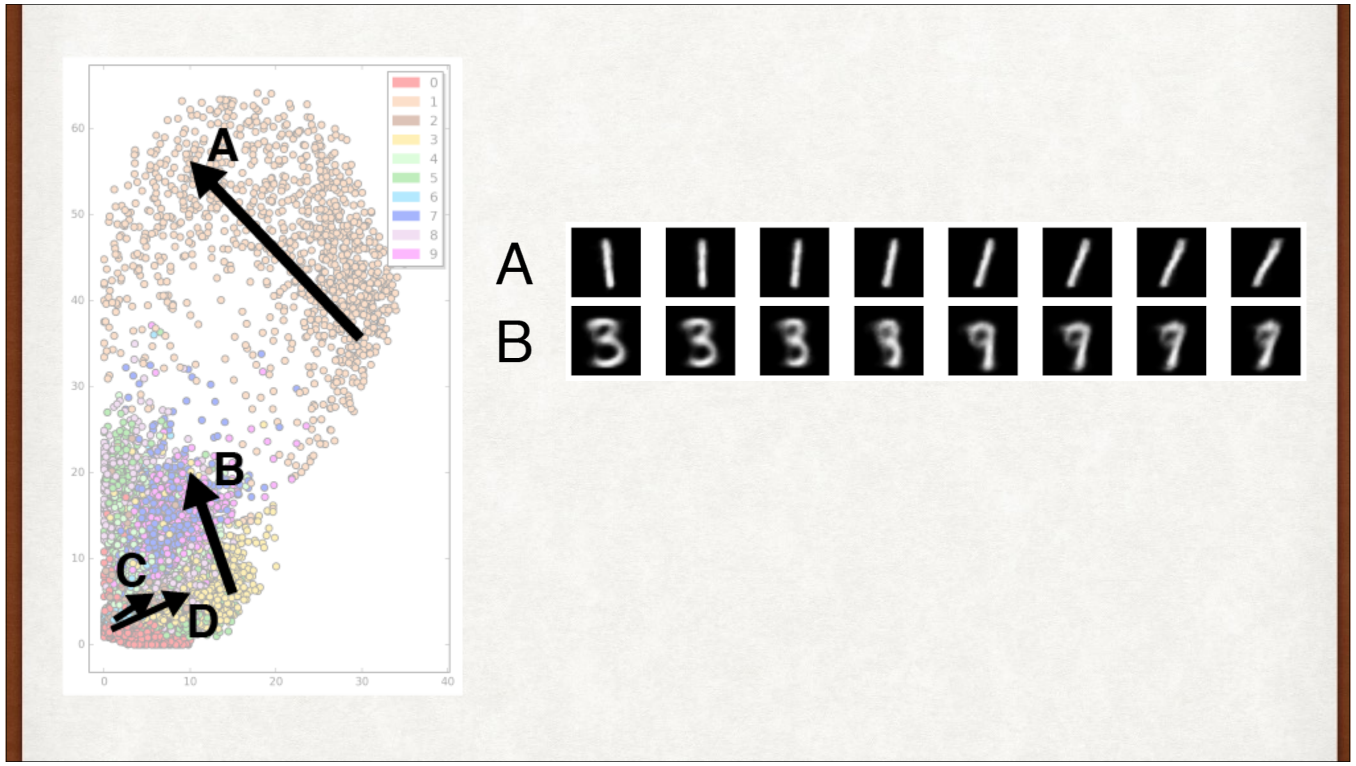
Feed them in.



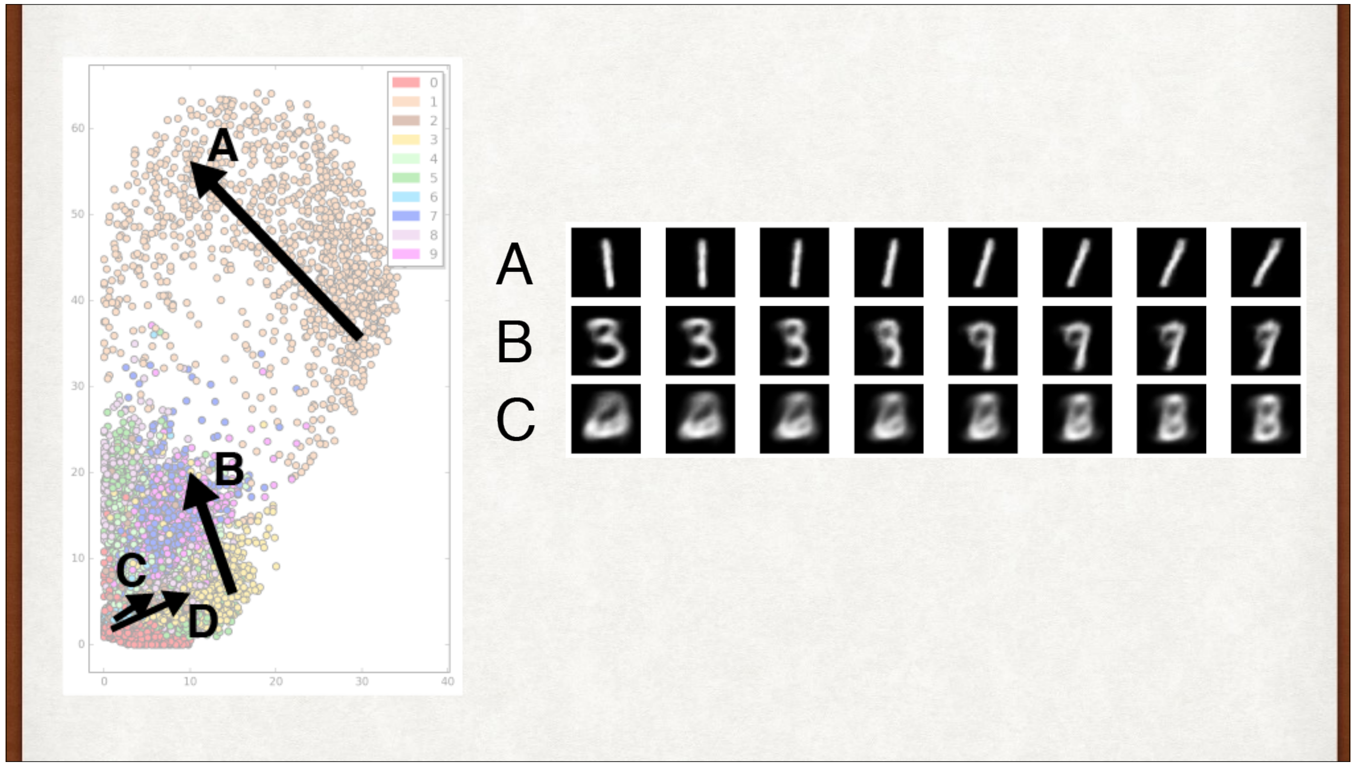
Blending in latent space. Each row shows equally-spaced steps along the corresponding arrow. Interpolated “digits” aren’t great, but they’re not random nonsense.



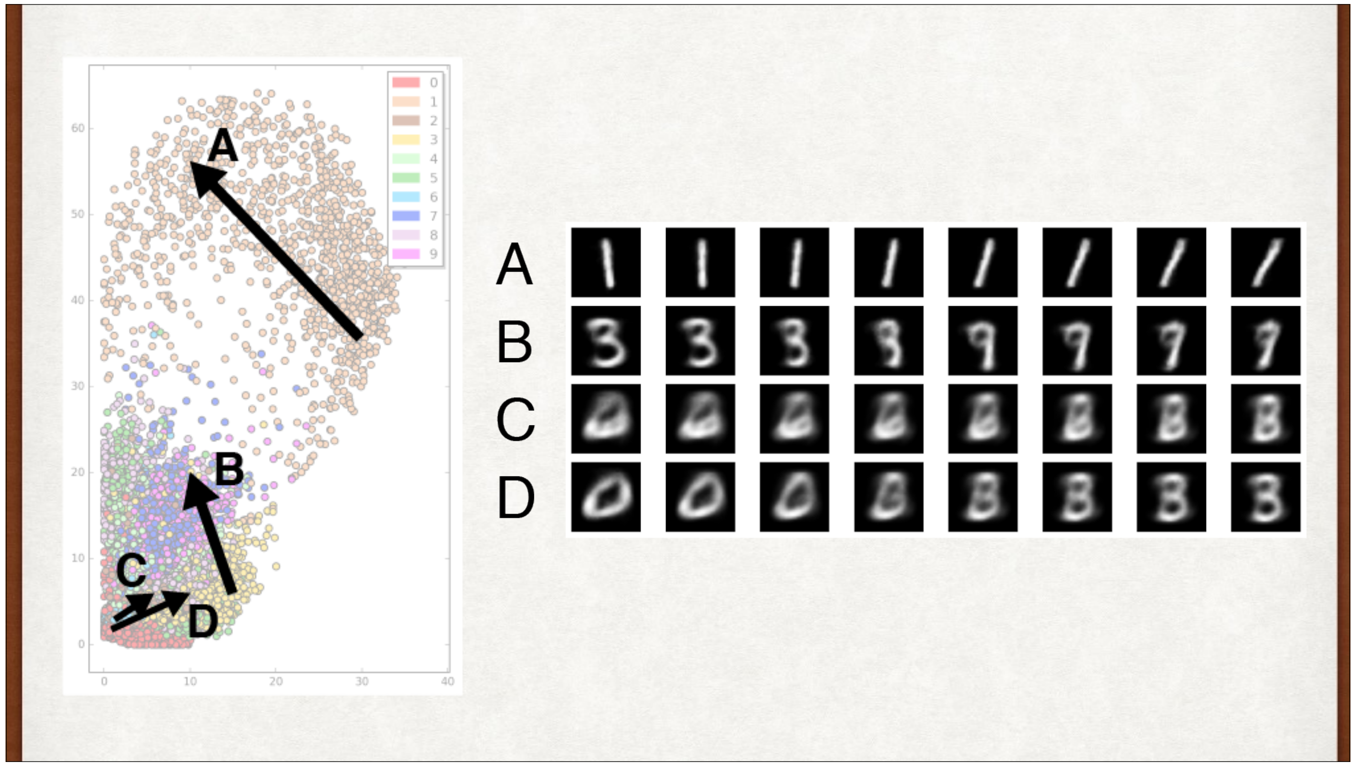
Blending in latent space. Each row shows equally-spaced steps along the corresponding arrow. Interpolated “digits” aren’t great, but they’re not random nonsense.



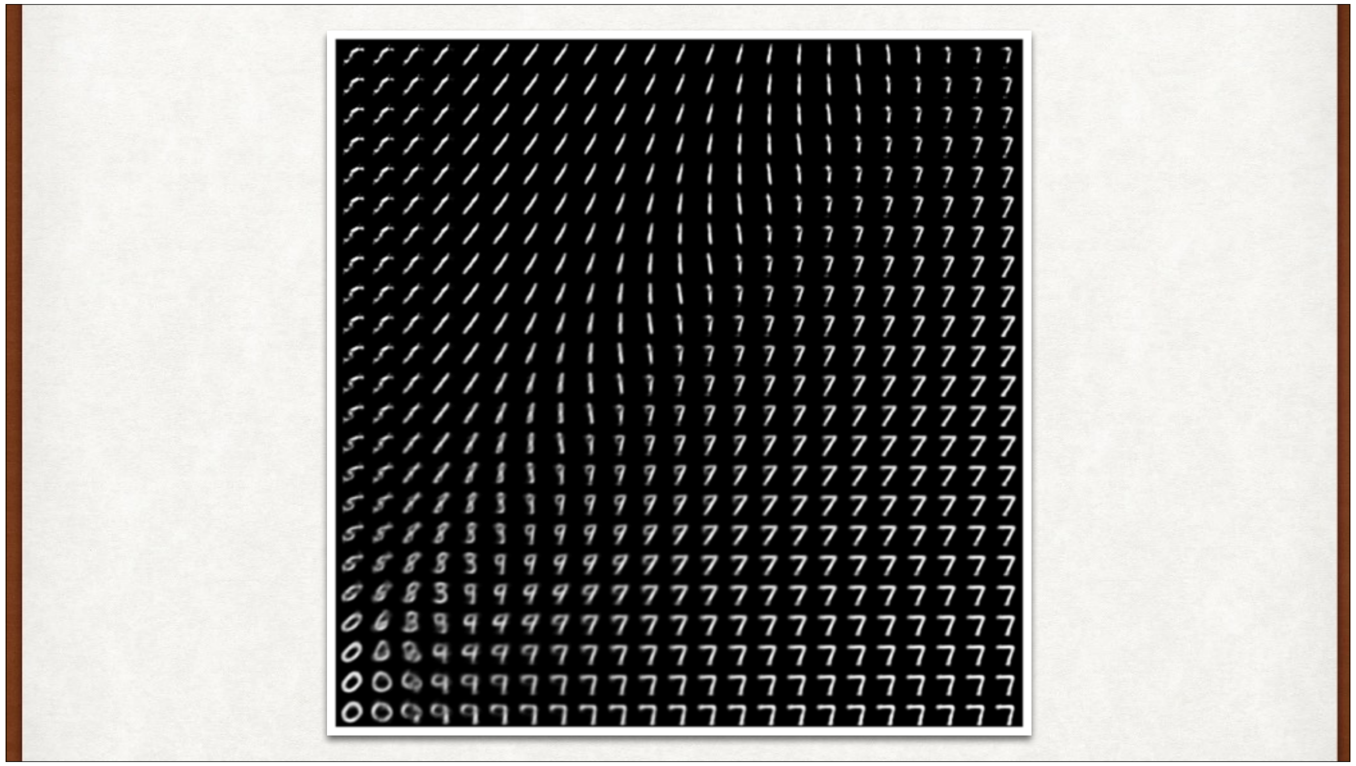
Blending in latent space. Each row shows equally-spaced steps along the corresponding arrow. Interpolated “digits” aren’t great, but they’re not random nonsense.



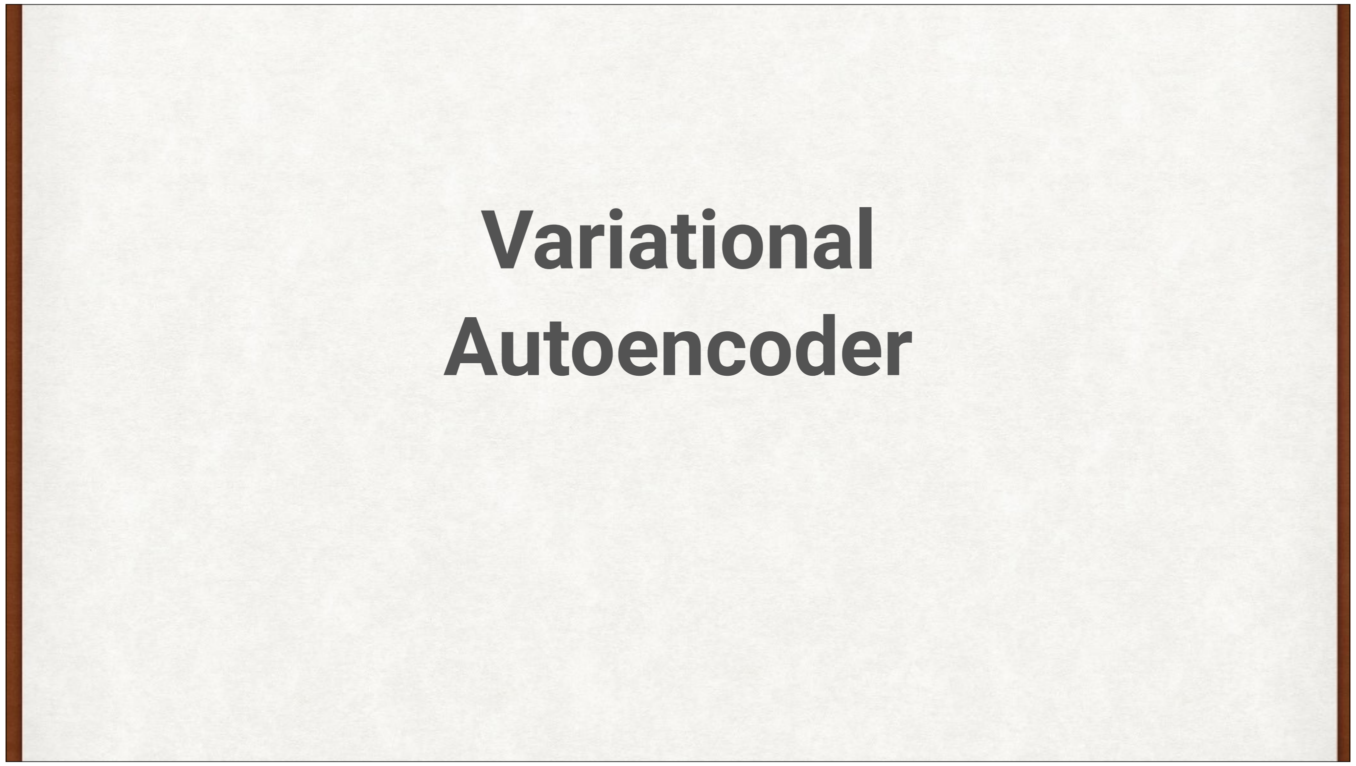
Blending in latent space. Each row shows equally-spaced steps along the corresponding arrow. Interpolated “digits” aren’t great, but they’re not random nonsense.



Blending in latent space. Each row shows equally-spaced steps along the corresponding arrow. Interpolated “digits” aren’t great, but they’re not random nonsense.

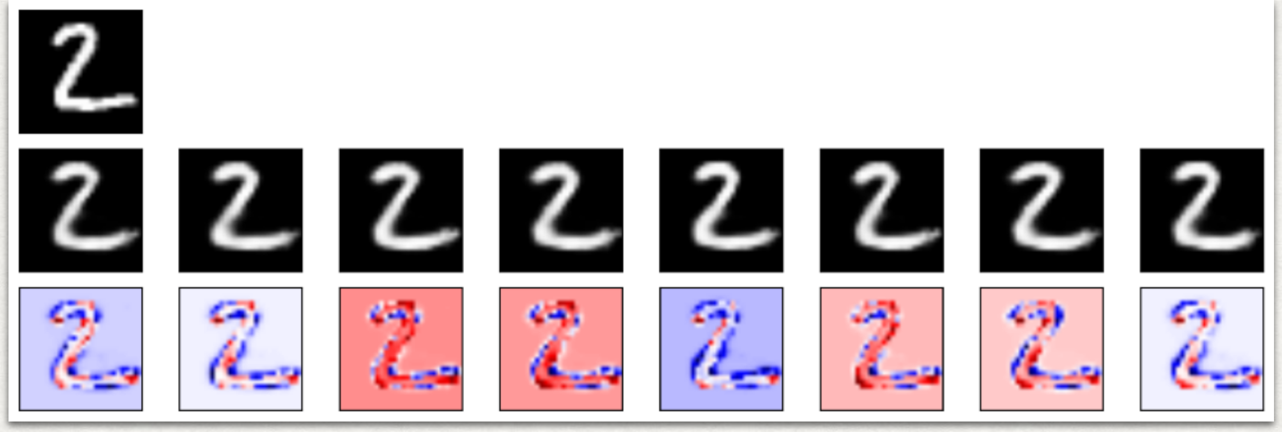


Sampling a grid over the 2D latent variables, and drawing the result for each pair of values. The 1s and 7s take up most of the space, but we can spot the other digits in the bottom-left corner.

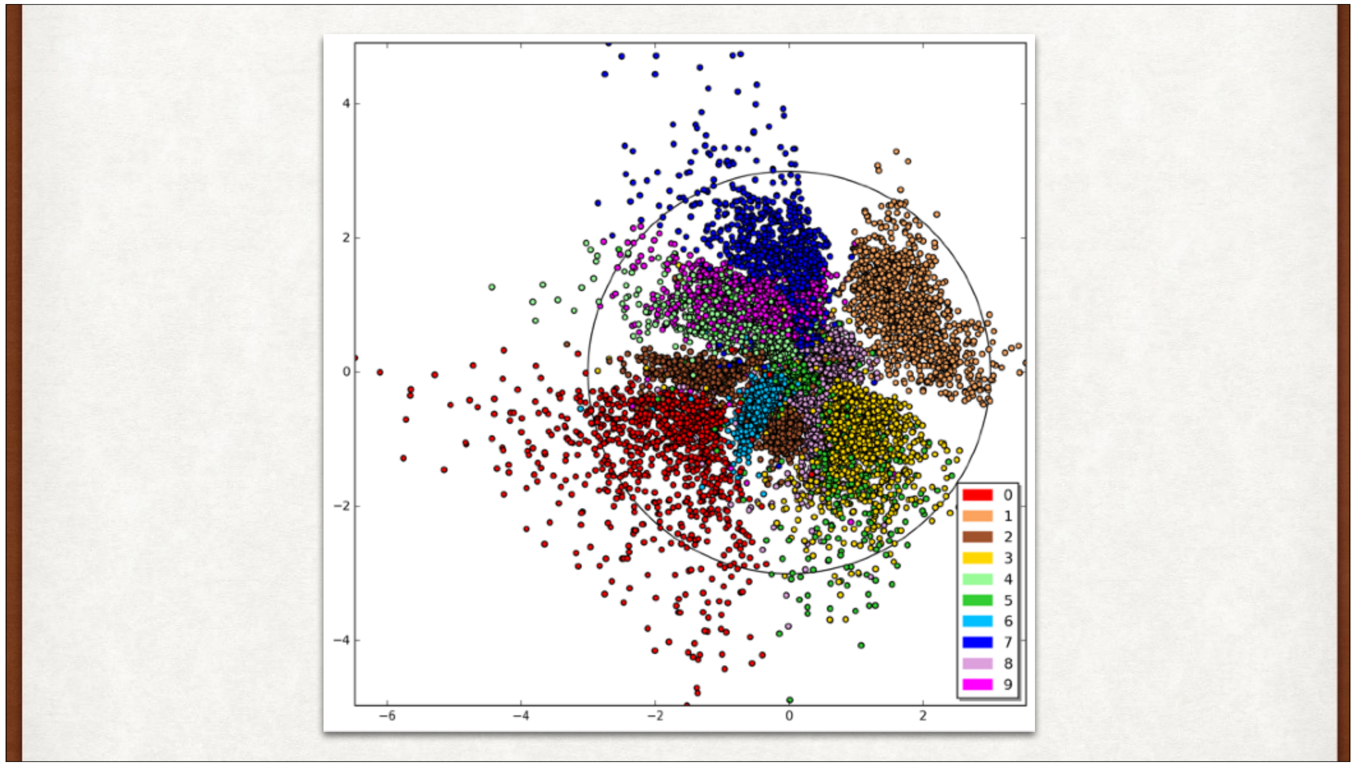


Variational Autoencoder

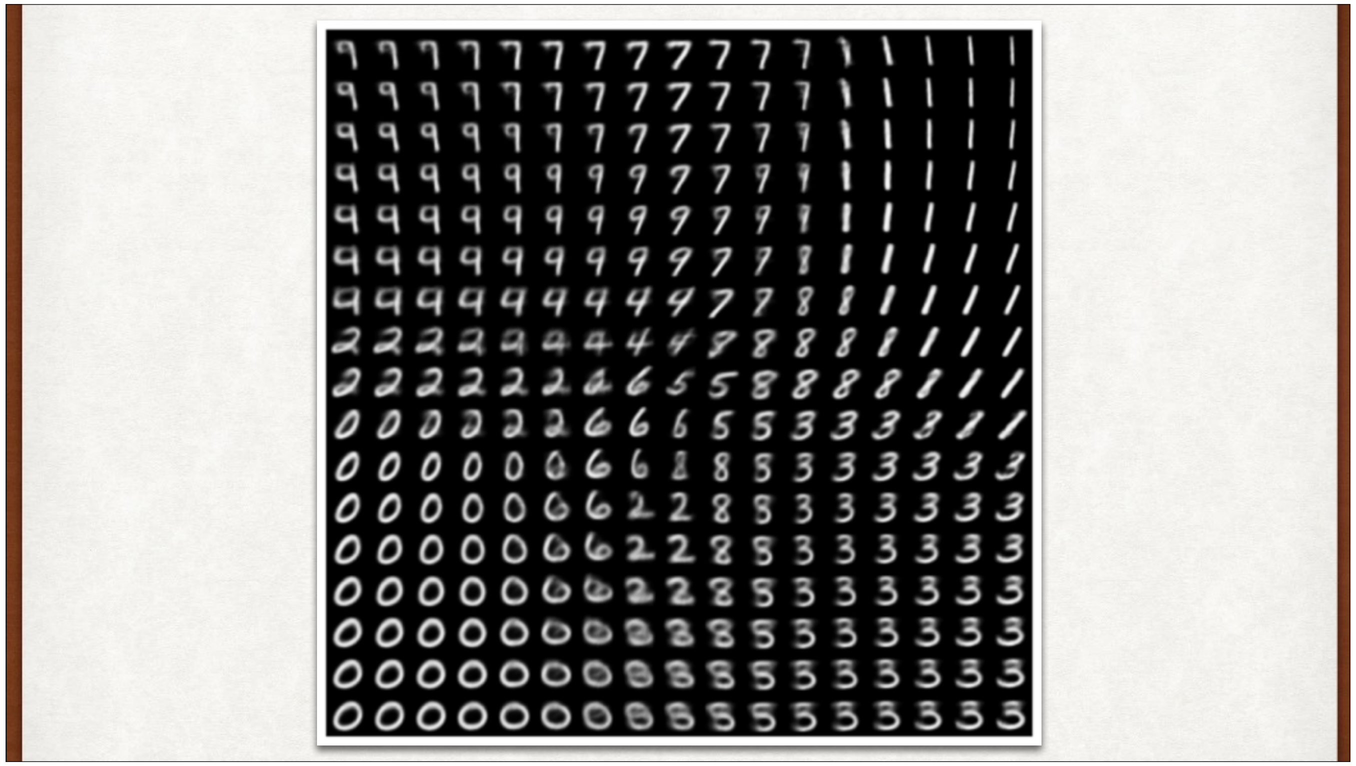
Another type of autoencoder. It’s more complicated, but produces nice results.



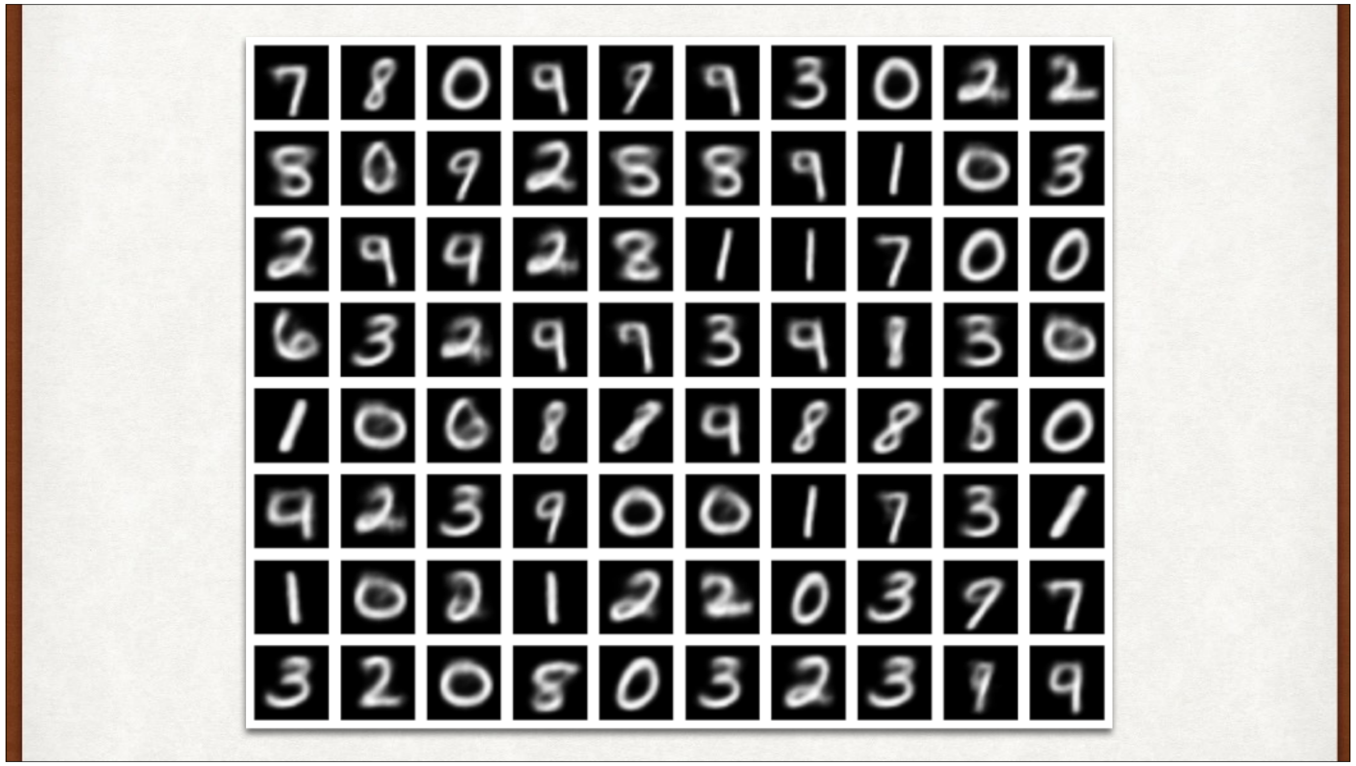
Thanks to its random nature, each output of a VAE is different, even from the same input. The top 2 was fed to a trained VAE repeatedly. The images below are the outputs. The images in the bottom row are the difference between the input and output.



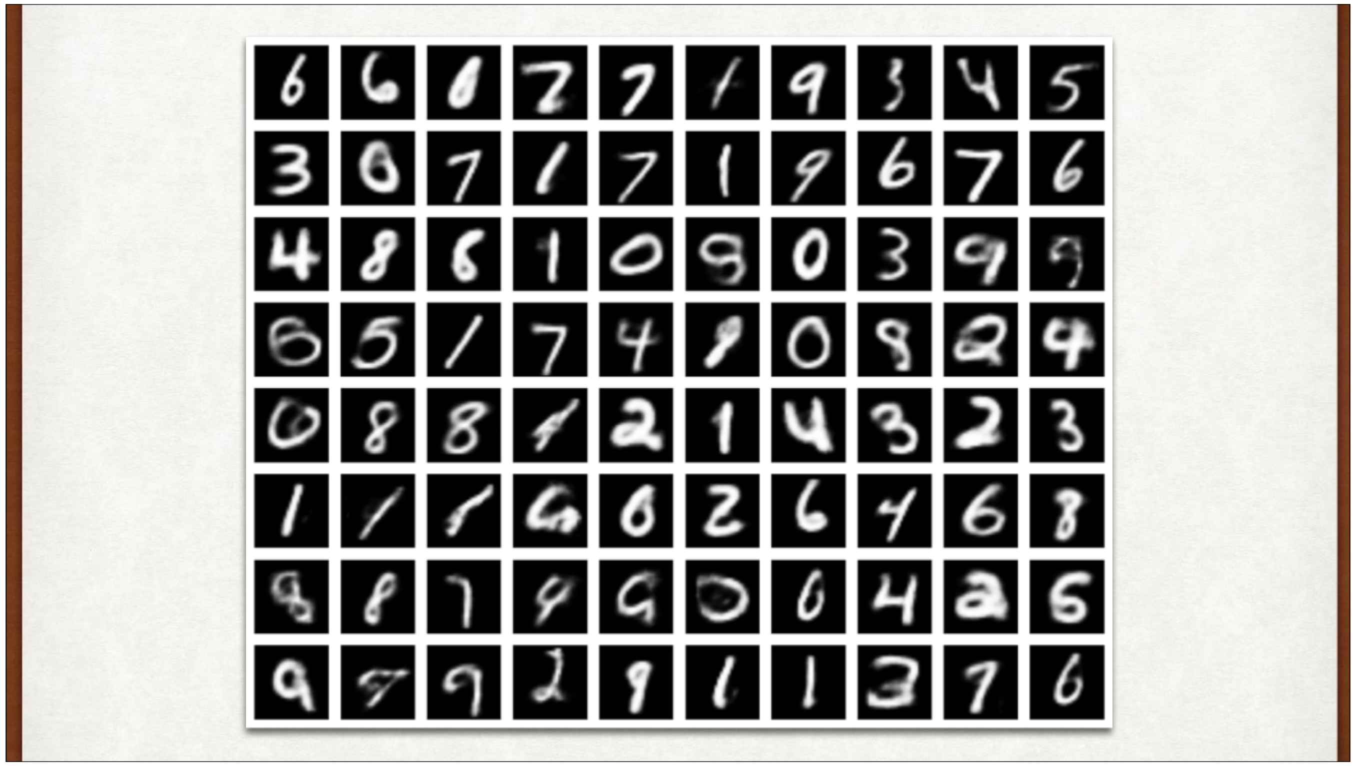
Visualizing the 2D latent variable space from a VAE. The digits are much better isolated and grouped than with our earlier autoencoder.



Sampling the VAE’s latent variables in 2D space. The images are fuzzy because we’re using only 2 latent variables. Notice how much more nicely they’re distributed than in the previous autoencoder.

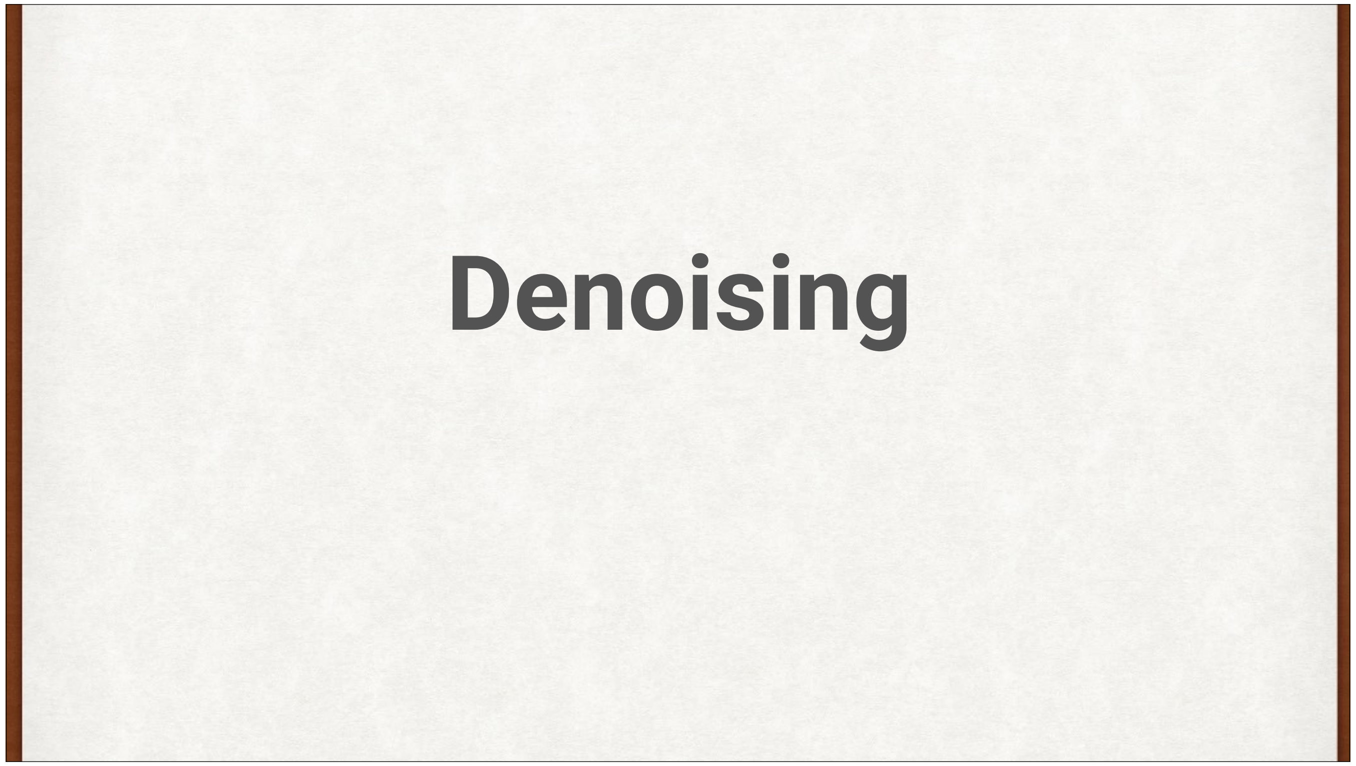


80 images generated by a VAE with 2 latent variables. Each image is the result of 2 random variables used as input.



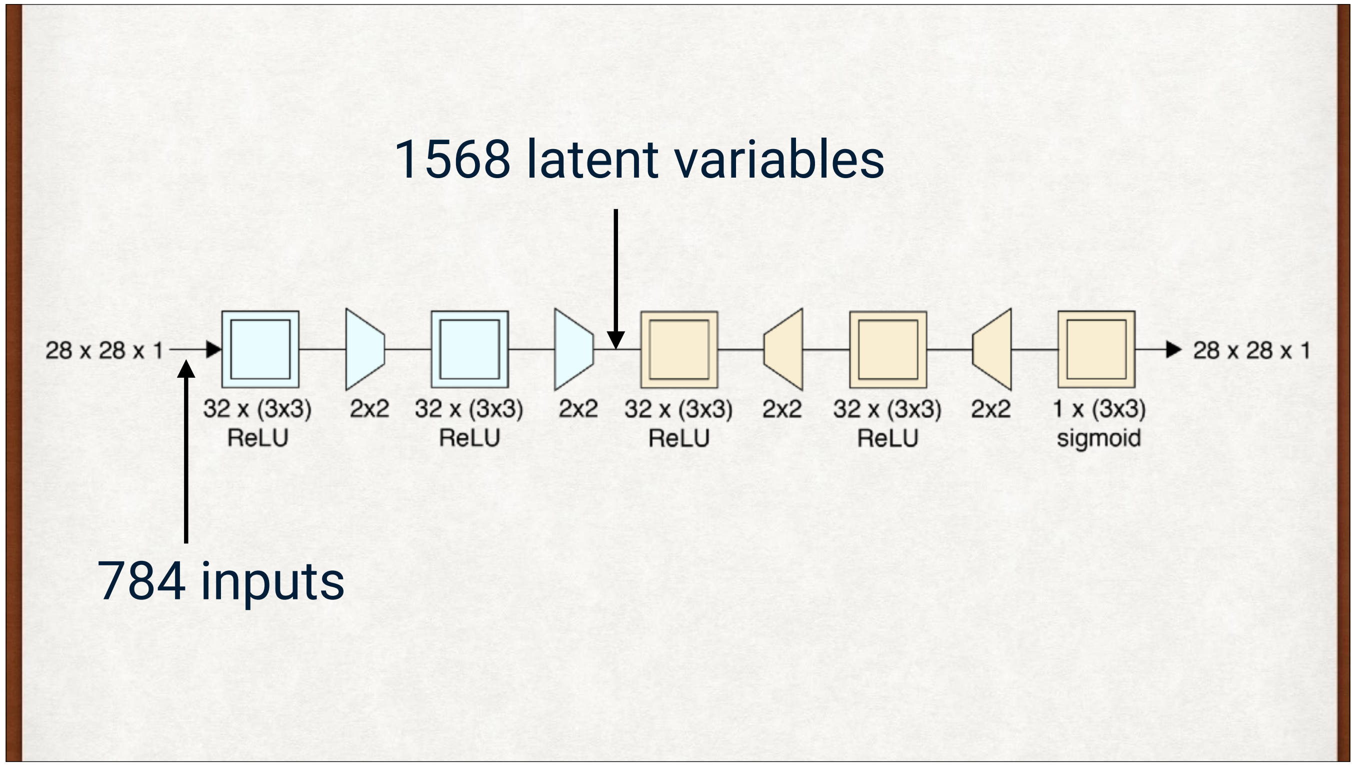
More random digits, only now we’re using a VAE with 50 latent variables. So the input to the VAE was a list of 50 random numbers, and these images popped out at the end.

Take a breath, think about nothing for a moment. Time for a new topic.



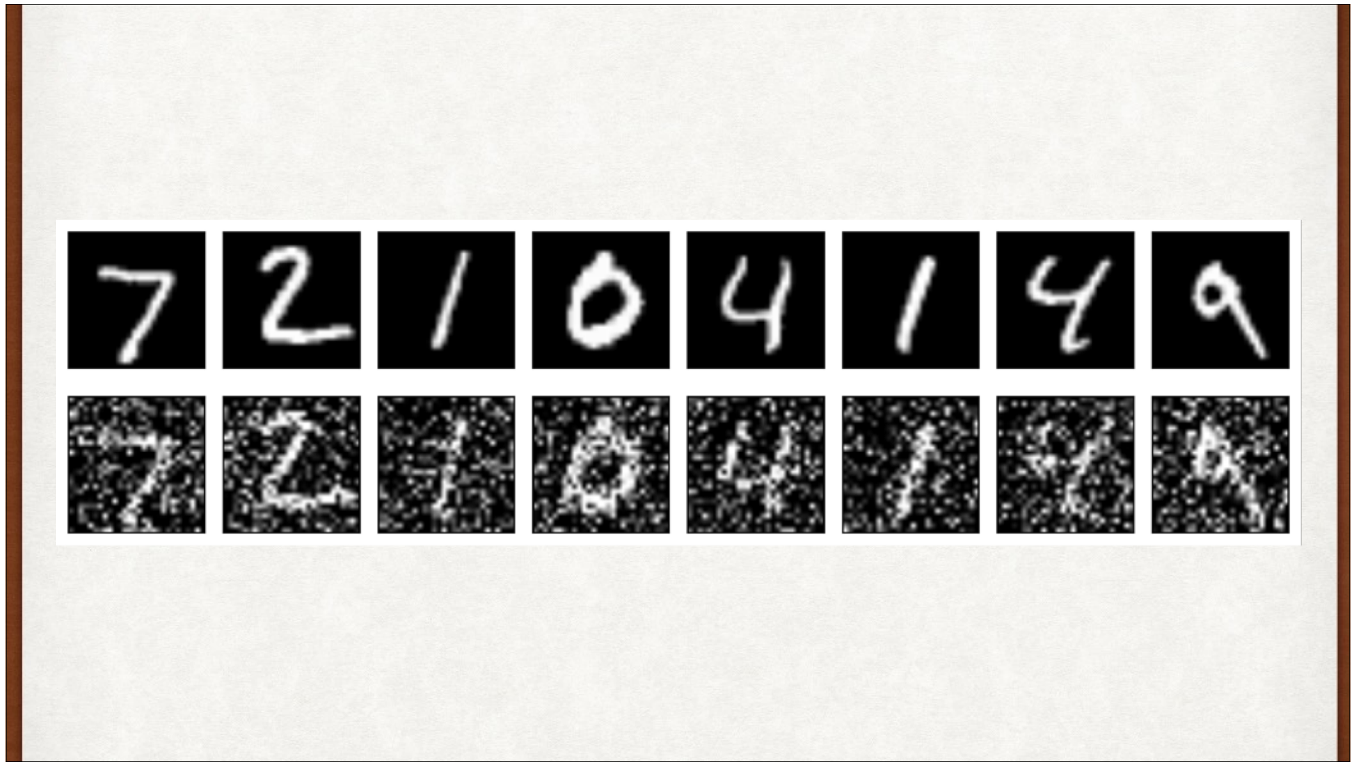
Denoising

Autoencoders are great for removing noise.

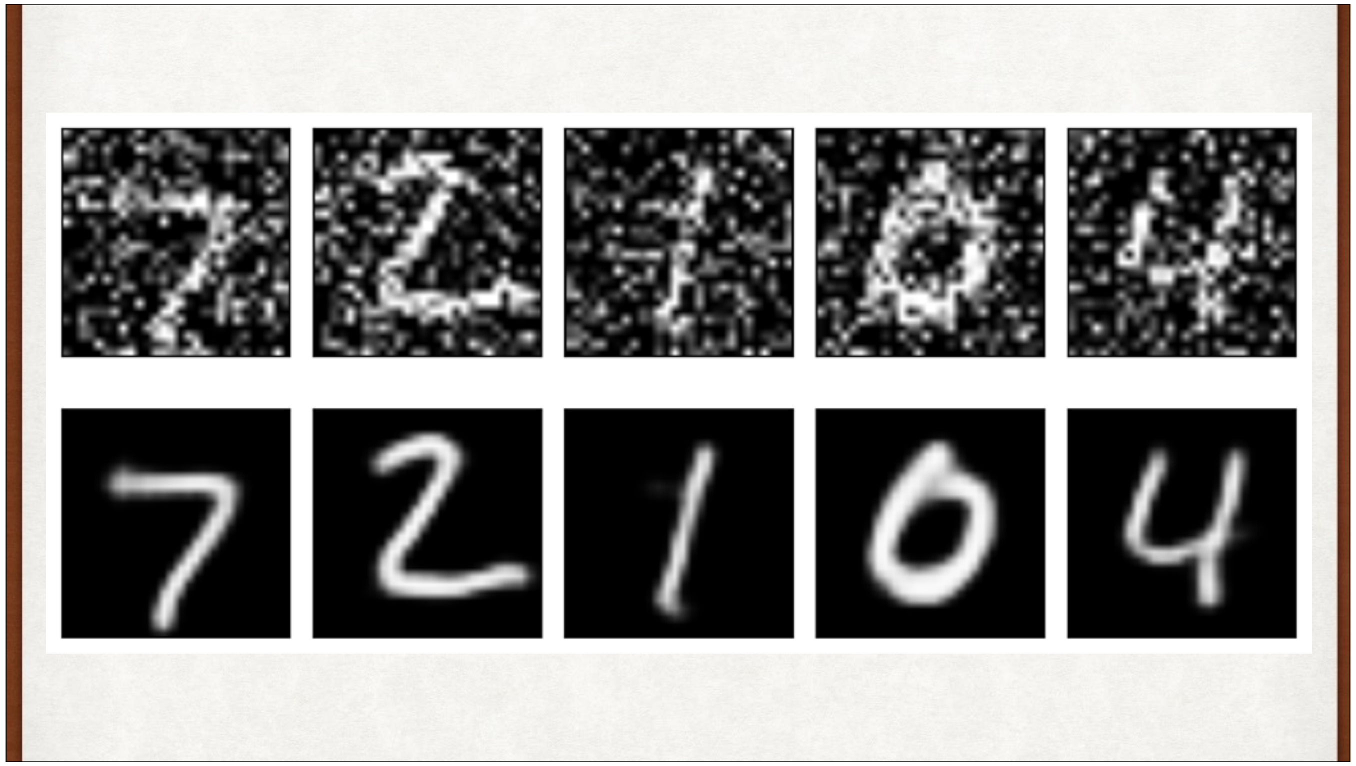

1568 latent variables

784 inputs

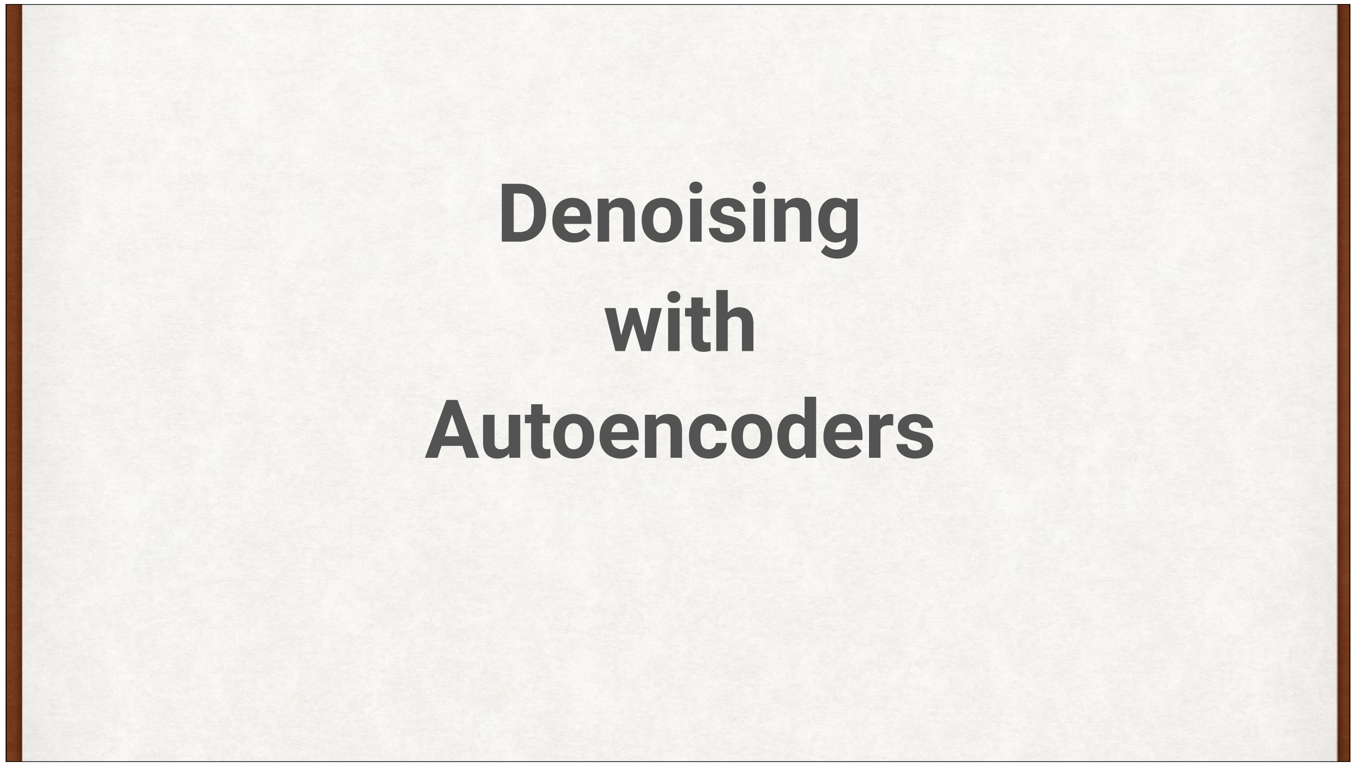
An autoencoder for denoising. There are 32*7*7 = 1568 latent variables. Note that this is more than the 784 inputs. But our goal now is to remove noise, not to compress the input.



Adding noise. Top: MNIST digits. Bottom: Digits plus noise. We train on the noisy digits, and ask the autoencoder to produce the clean versions.

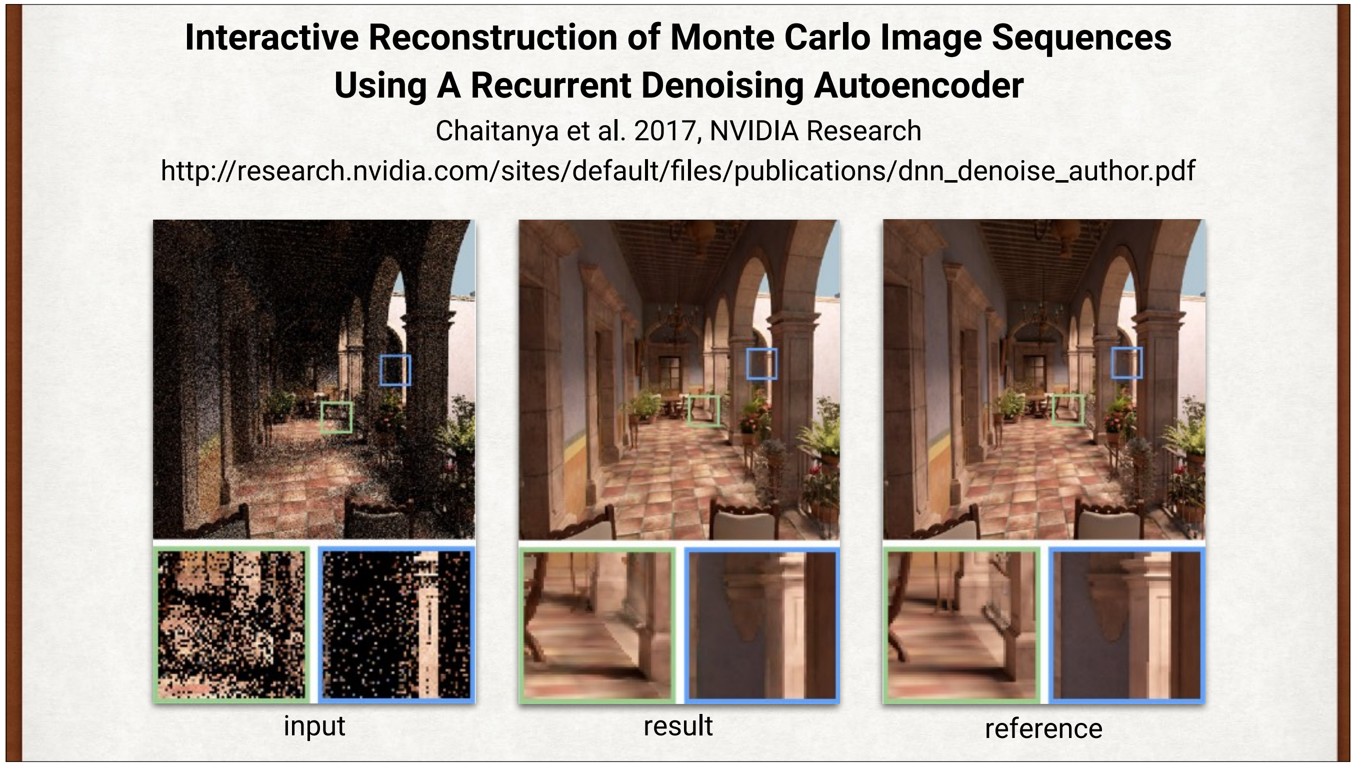


Desnoising success! Top: input images. Bottom: outputs.



Denoising with Autoencoders

Let’s see autoencoders in use to denoise images.



Interactive Reconstruction of Monte Carlo Image Sequences Using A Recurrent Denoising Autoencoder

Chaitanya et al. 2017, NVIDIA Research

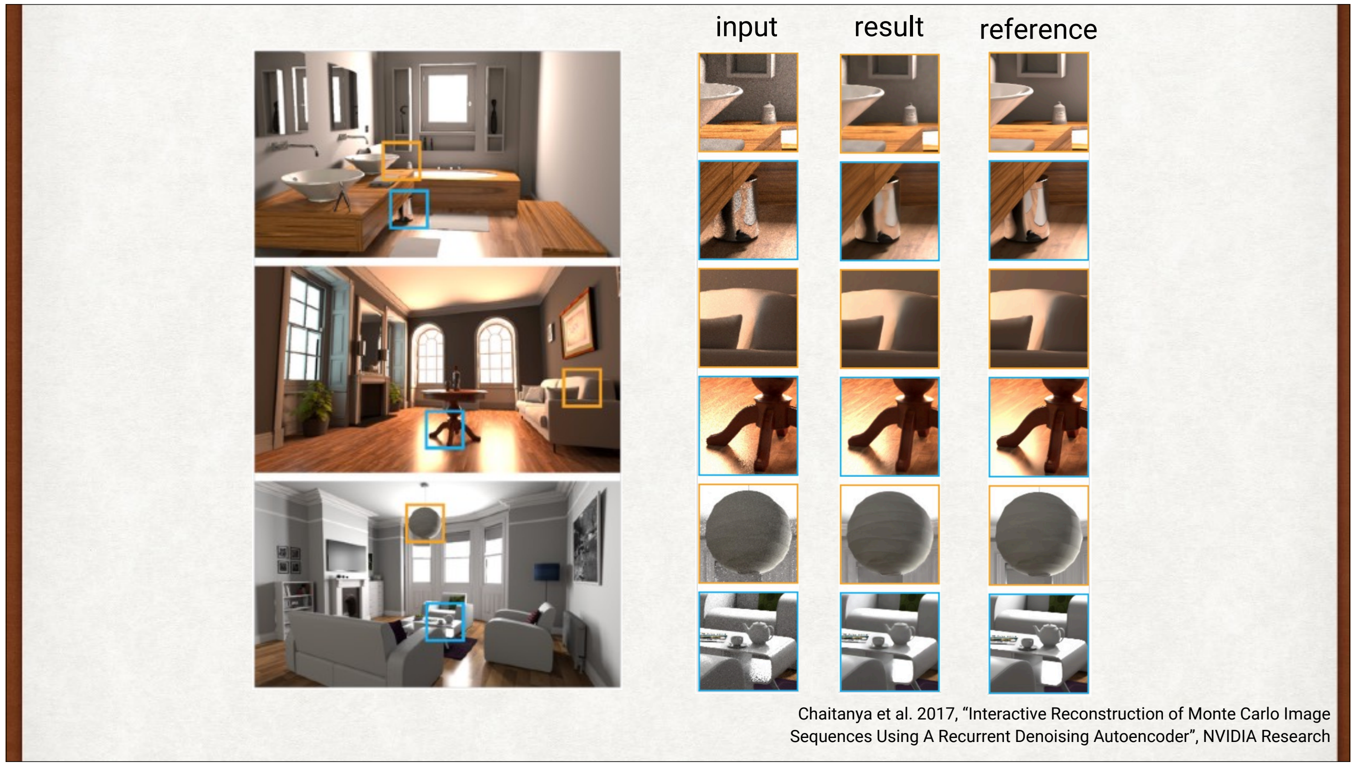
http://research.nvidia.com/sites/default/files/publications/dnn_denoise_author.pdf

input

result

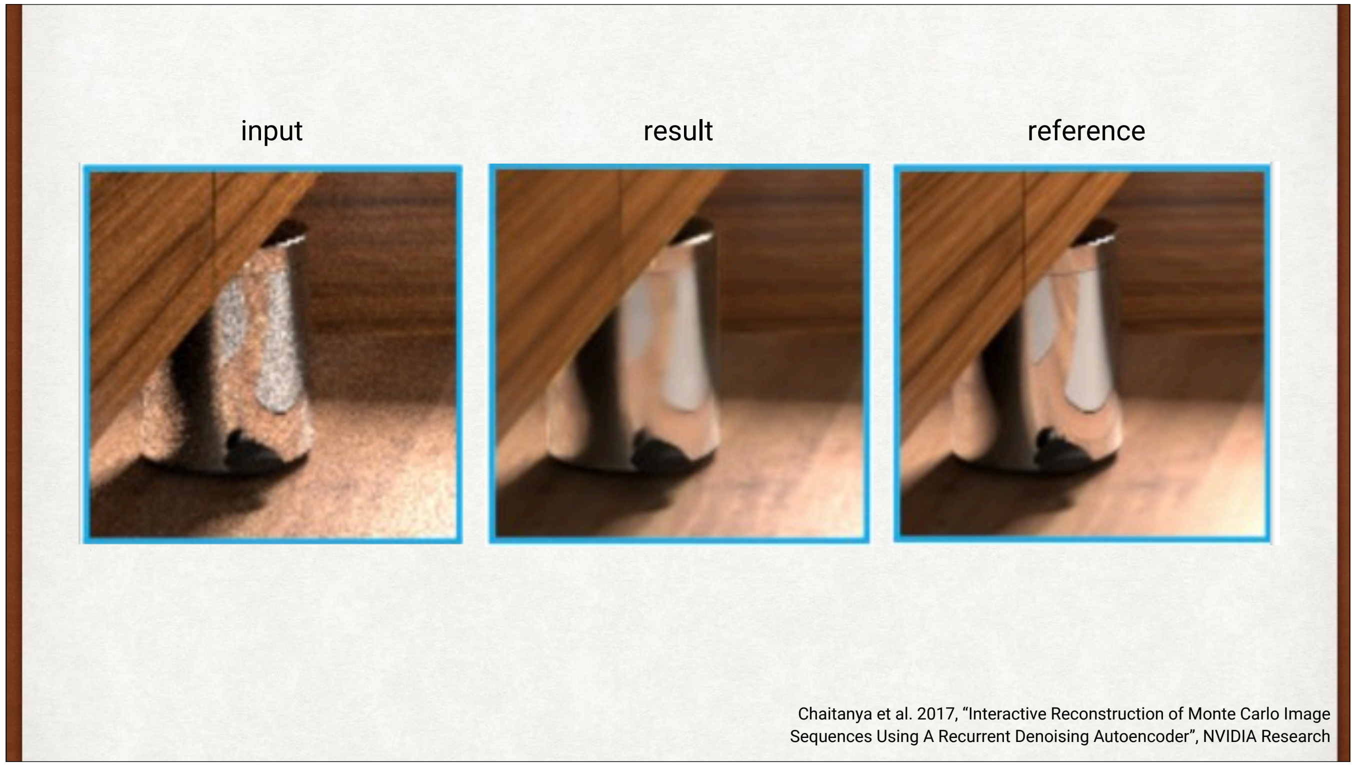
reference

Left: 1 sample per pixel. Middle: reconstruction of the leftmost image with autoencoder. Right: the reference (high-quality) image. Notice how much correct detail the autoencoder is able to extract, even in regions where the values are mostly missing.



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

More reconstructions from the autoencoder.



input

result

reference

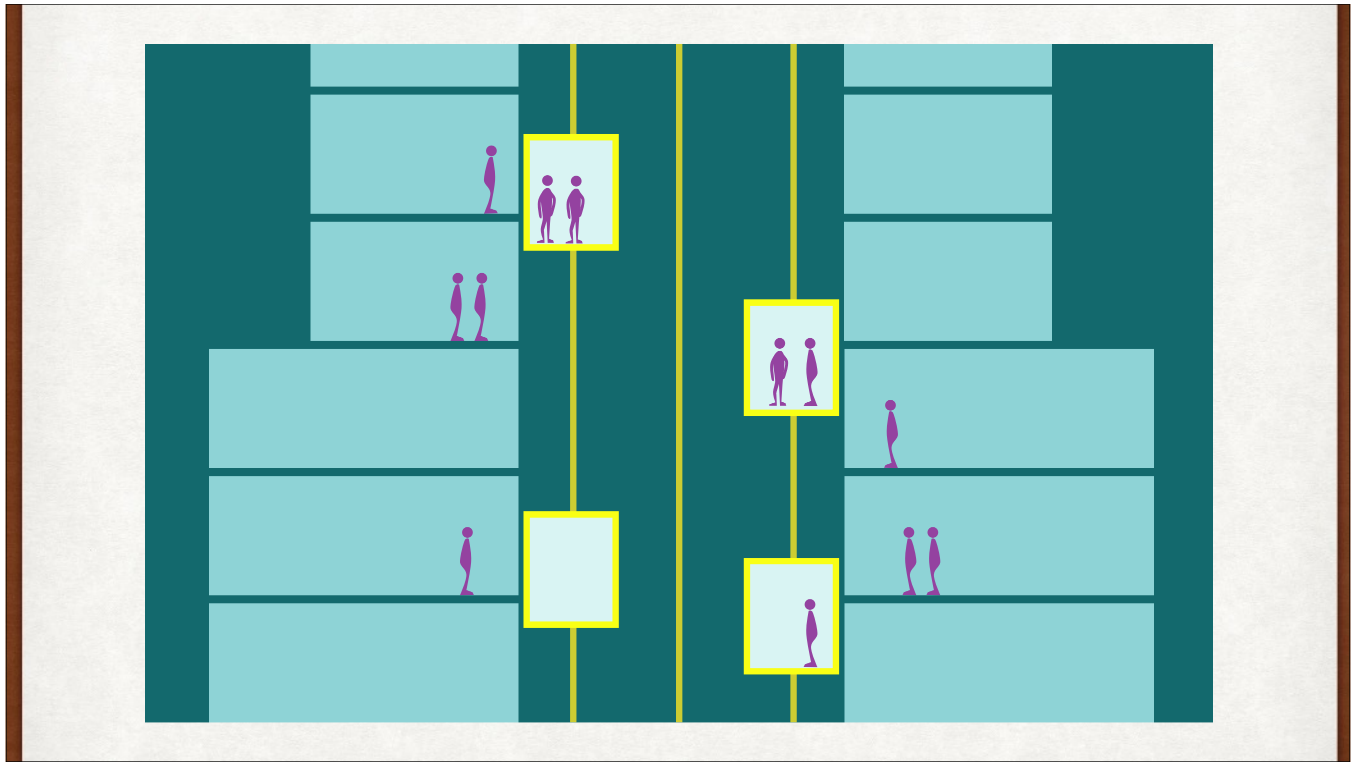
Chaitanya et al. 2017, “Interactive Reconstruction of Monte Carlo Image Sequences Using A Recurrent Denoising Autoencoder”, NVIDIA Research

Close-up reconstructions from the autoencoder.

Take a breath, enjoy some palette cleanser.

Reinforcement Learning

Another way to learn. An “agent” takes actions in an environment. The environment tells the agent how good each action is. The agent seeks to find actions that earn big rewards.



Another way to learn. An “agent” takes actions in an environment. The environment tells the agent how good each action is. The agent seeks to find actions that earn big rewards.

Taking turns. Who is the agent, what is the environment (everything else!).

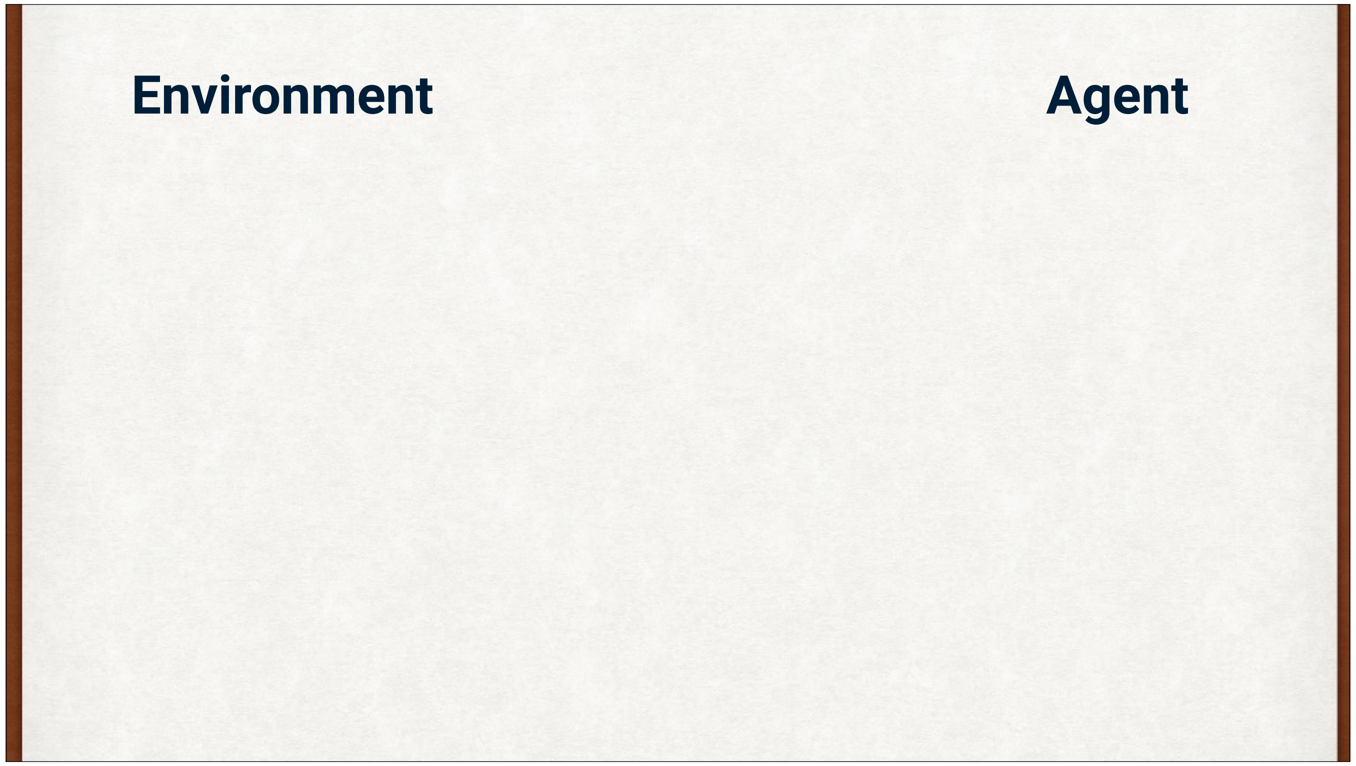
https://pixabay.com/en/play-chess-elephant-mouse-snail-2647368/

https://pixabay.com/en/play-chess-elephant-mouse-snail-2647368/

https://pixabay.com/en/play-chess-elephant-mouse-snail-2647368/

https://pixabay.com/en/play-chess-elephant-mouse-snail-2647368/

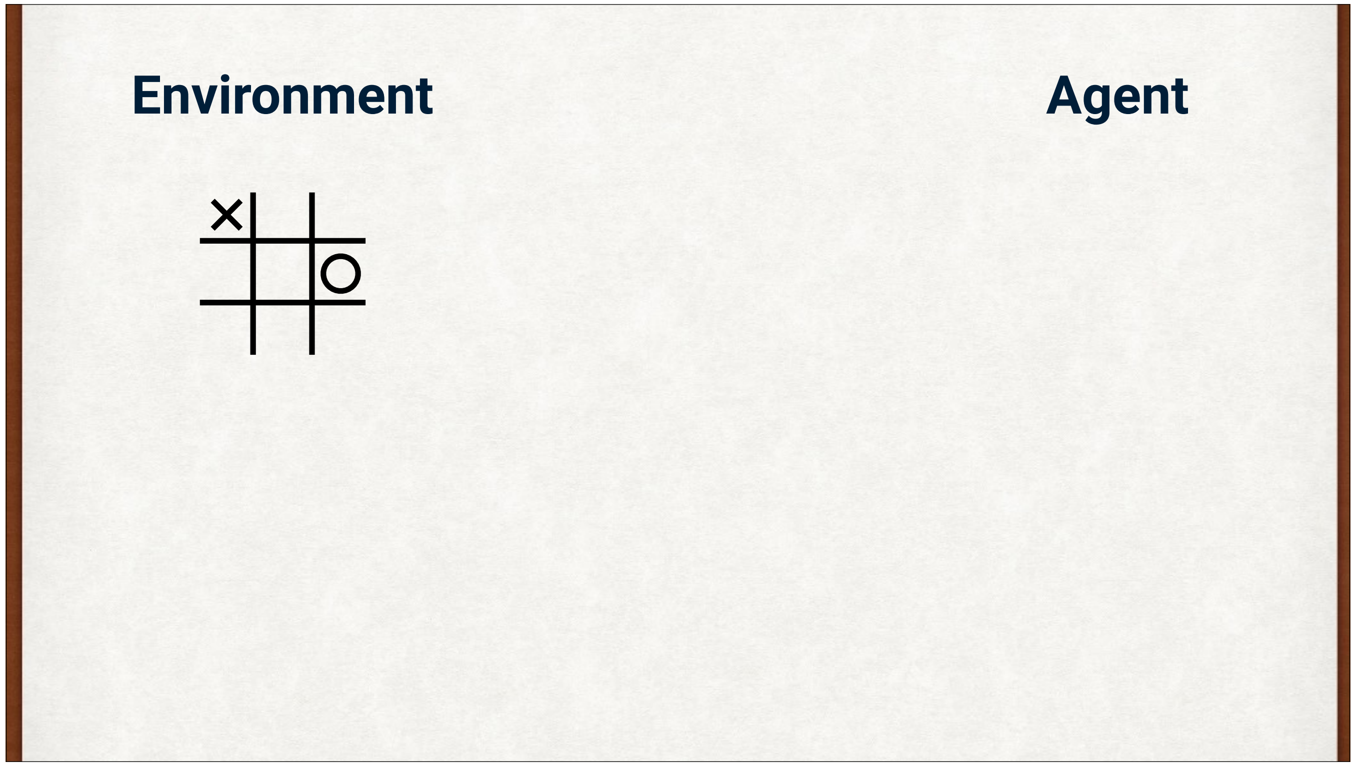
https://pixabay.com/en/play-chess-elephant-mouse-snail-2647368/



Environment

Agent

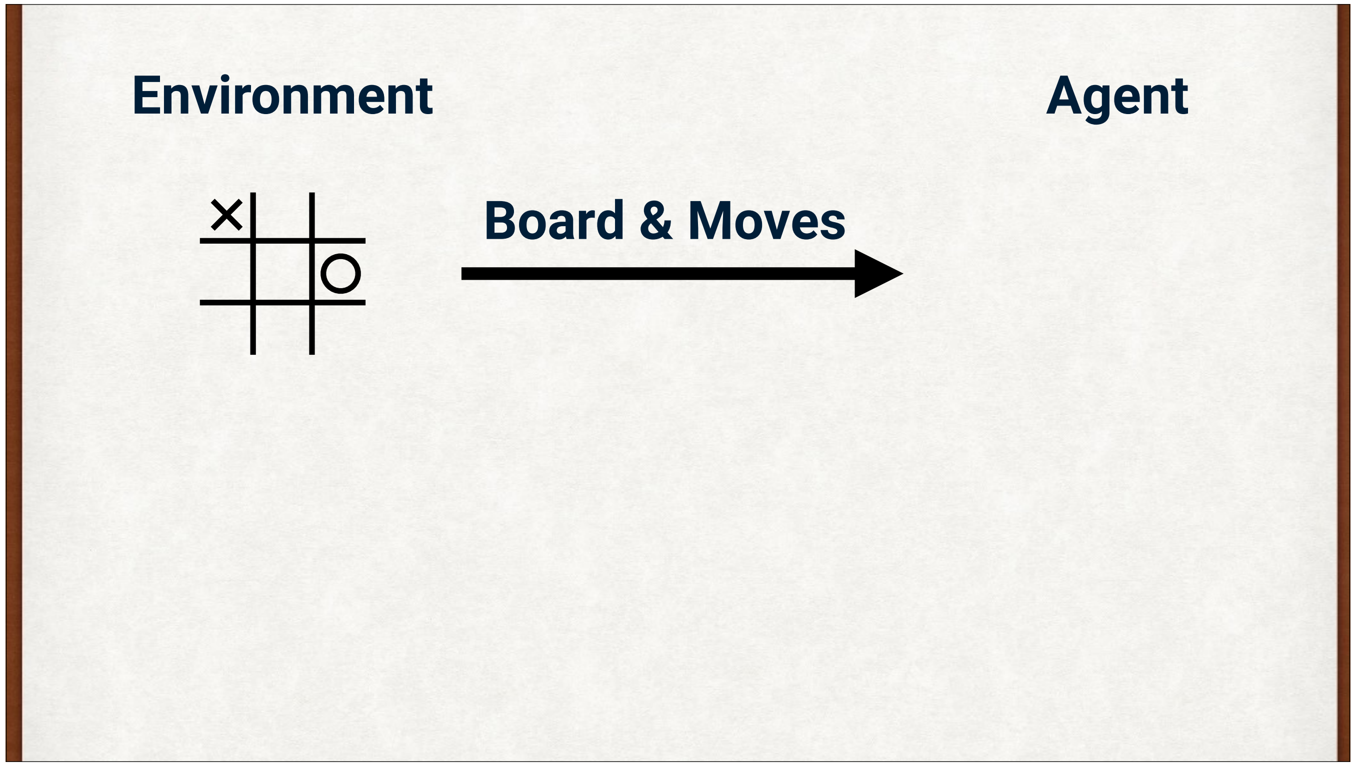
Learning to play tic-tac-toe. The agent (blue) chooses moves. The environment (yellow) does everything else, and tells the agent how good (or not) each move is. Good moves get the agent closer to winning. Great moves are those that win the game.



Environment

Agent

Learning to play tic-tac-toe. The agent chooses moves. The environment does everything else, and tells the agent how good (or not) each move is. Good moves get the agent closer to winning. Great moves are those that win the game.

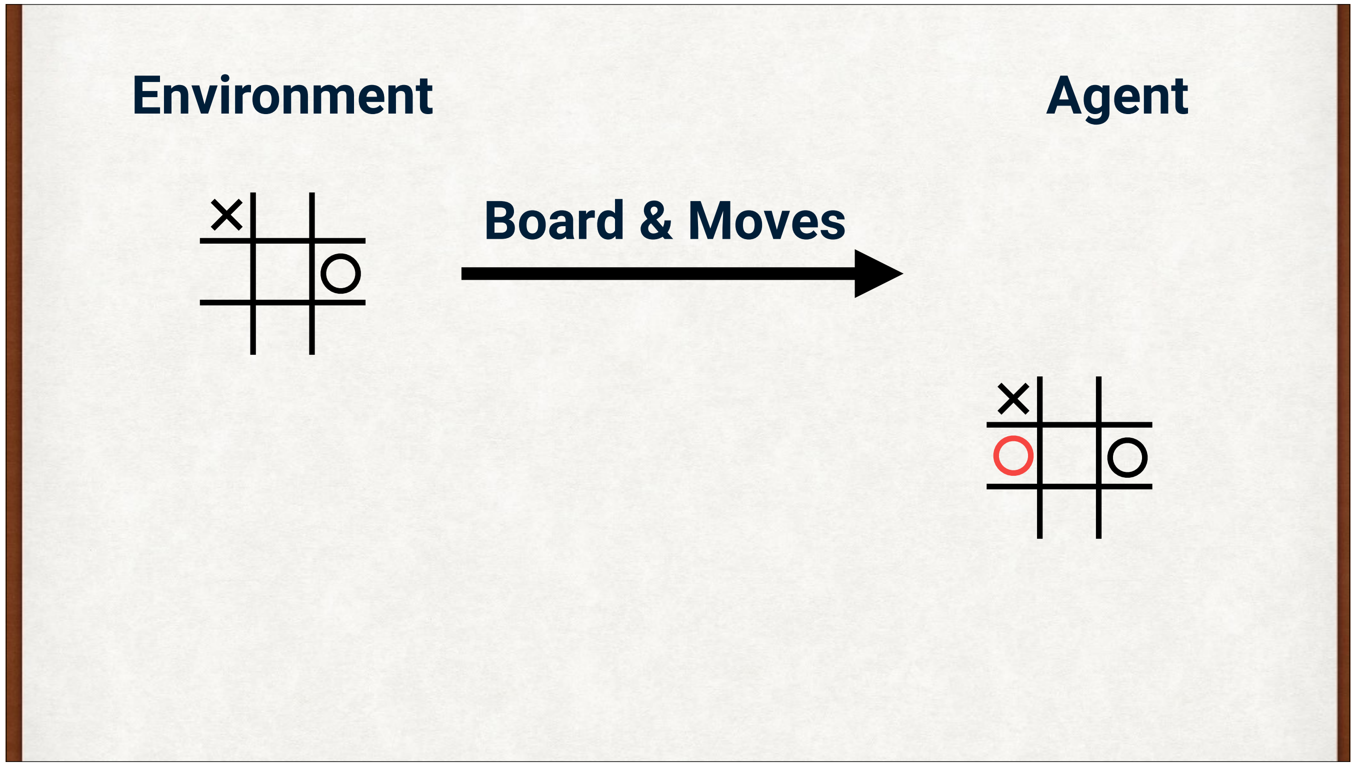


Environment

Agent

Board & Moves

Another way to learn. An “agent” takes actions in an environment. The environment tells the agent how good each action is. The agent seeks to find actions that earn big rewards.

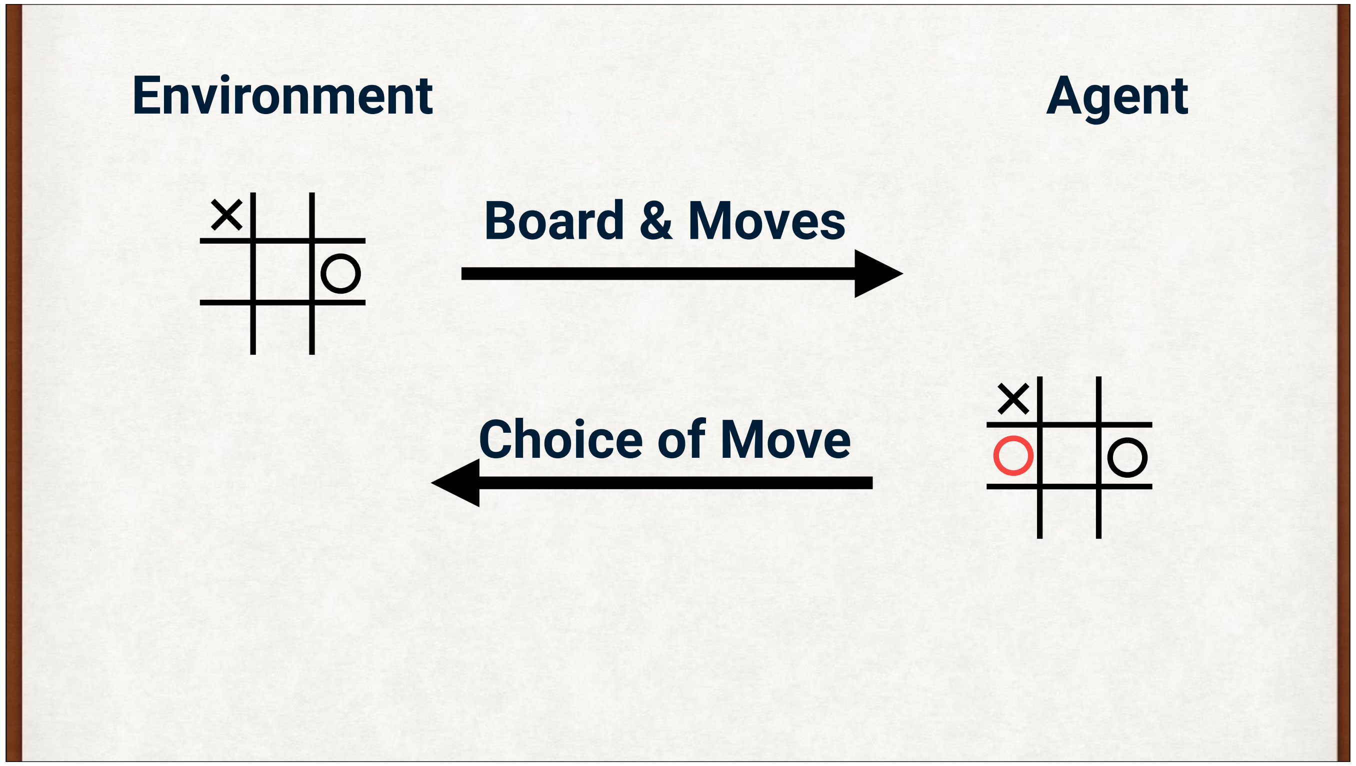


Environment

Board & Moves

Agent

Another way to learn. An “agent” takes actions in an environment. The environment tells the agent how good each action is. The agent seeks to find actions that earn big rewards.



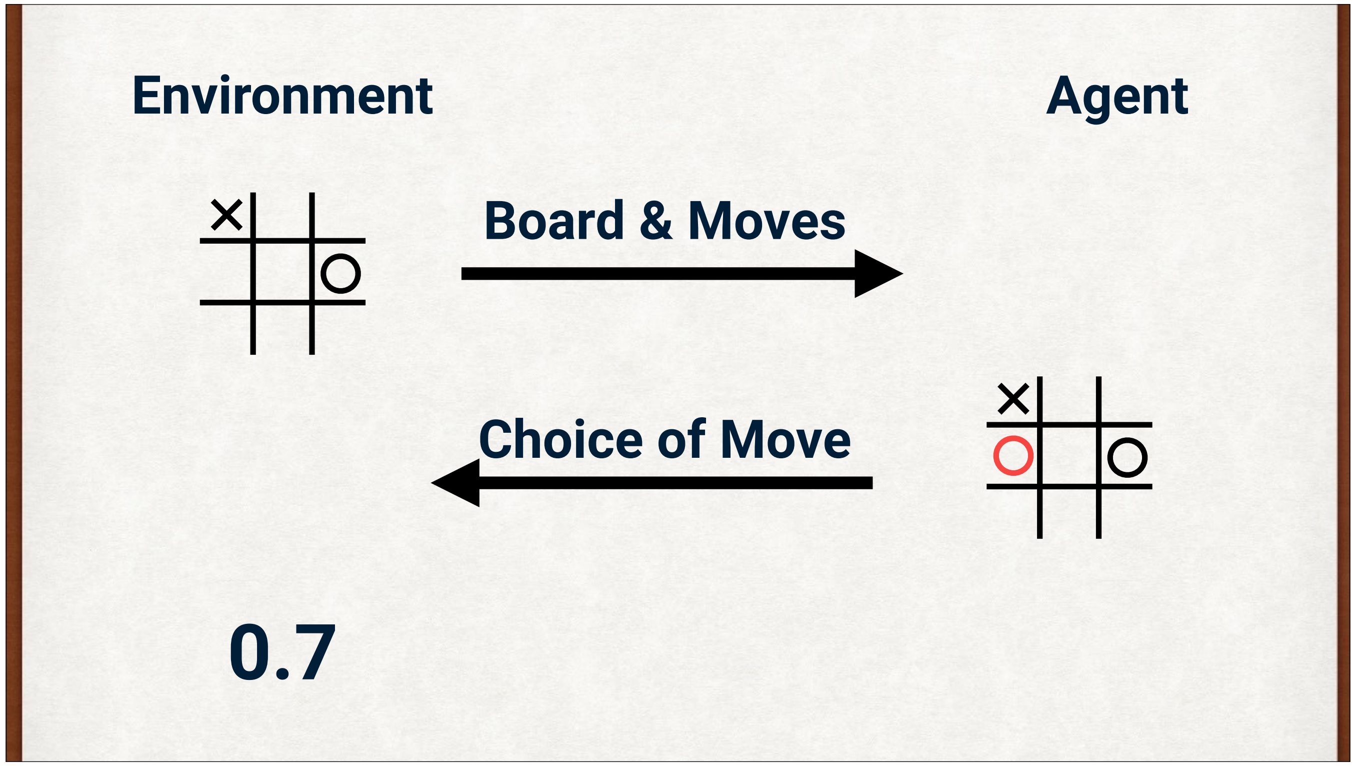
Environment

Agent

Board & Moves

Choice of Move

Another way to learn. An “agent” takes actions in an environment. The environment tells the agent how good each action is. The agent seeks to find actions that earn big rewards.



Environment

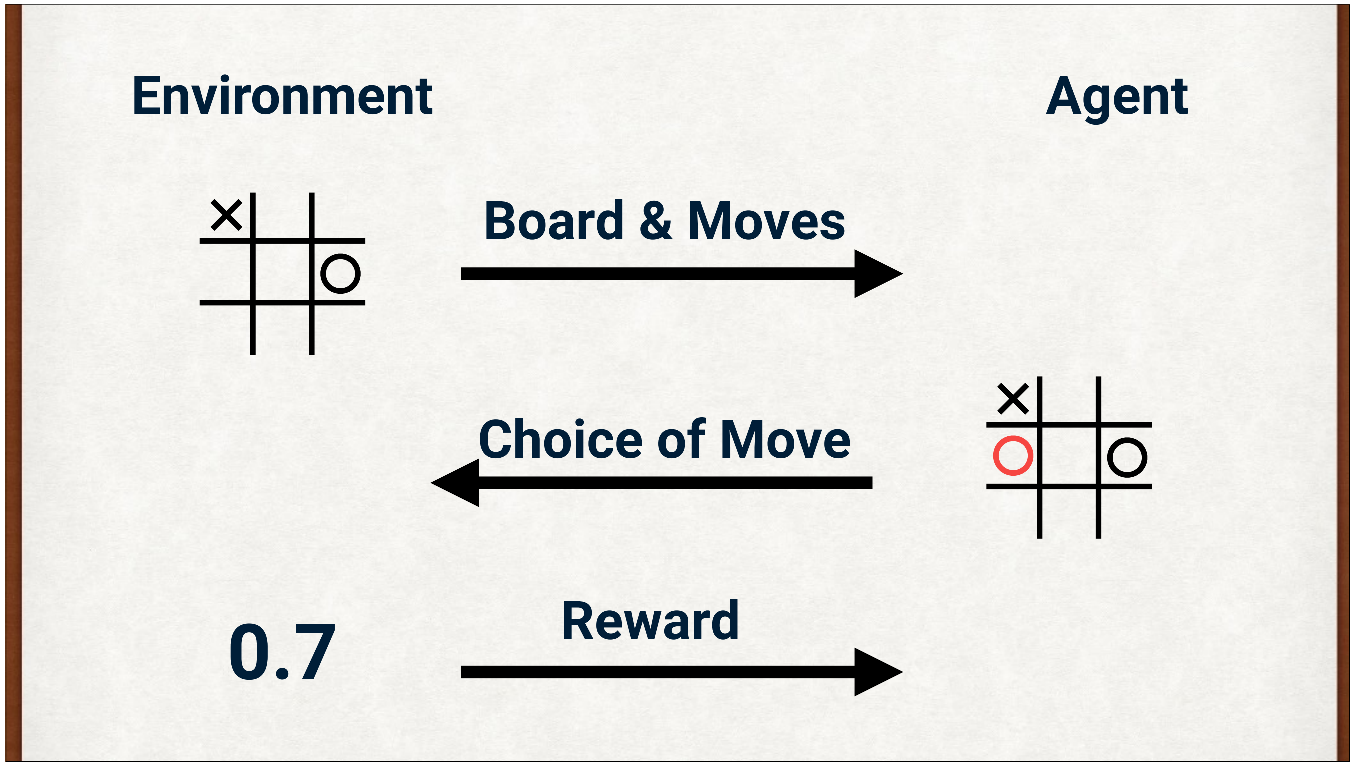
Agent

Board & Moves

Choice of Move

0.7

Another way to learn. An “agent” takes actions in an environment. The environment tells the agent how good each action is. The agent seeks to find actions that earn big rewards.



Environment

Agent

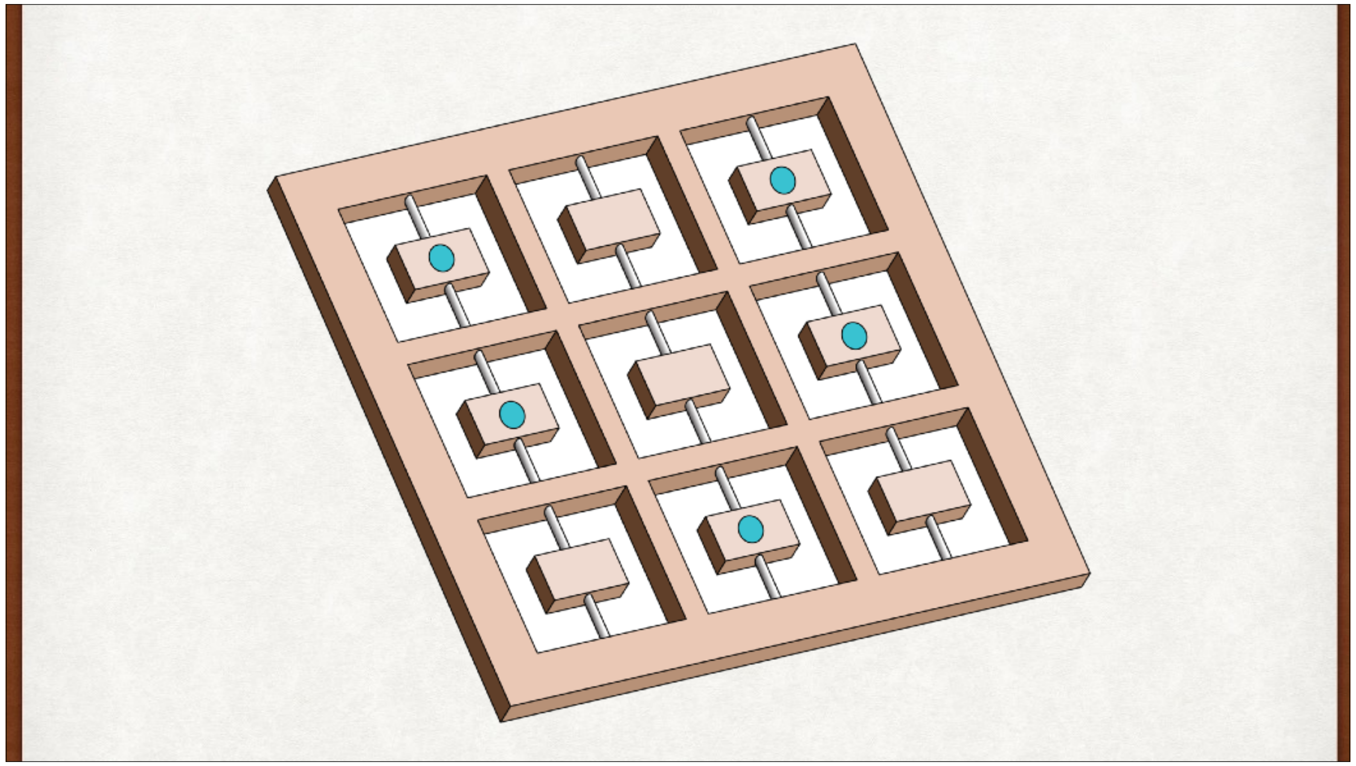
Board & Moves

Choice of Move

0.7

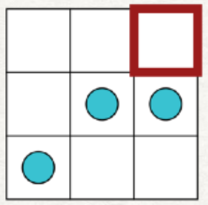
Reward

Another way to learn. An “agent” takes actions in an environment. The environment tells the agent how good each action is. The agent seeks to find actions that earn big rewards.



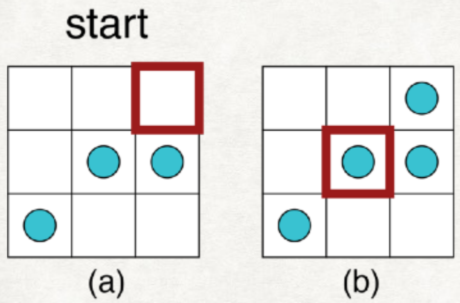
The absurd game of solitaire tic-tac-toe, called Flippers. Each tile has a blue dot on only one side. We want to get three dots horizontal or vertical, and blank tiles everywhere else. A move consists of flipping one tile.

start

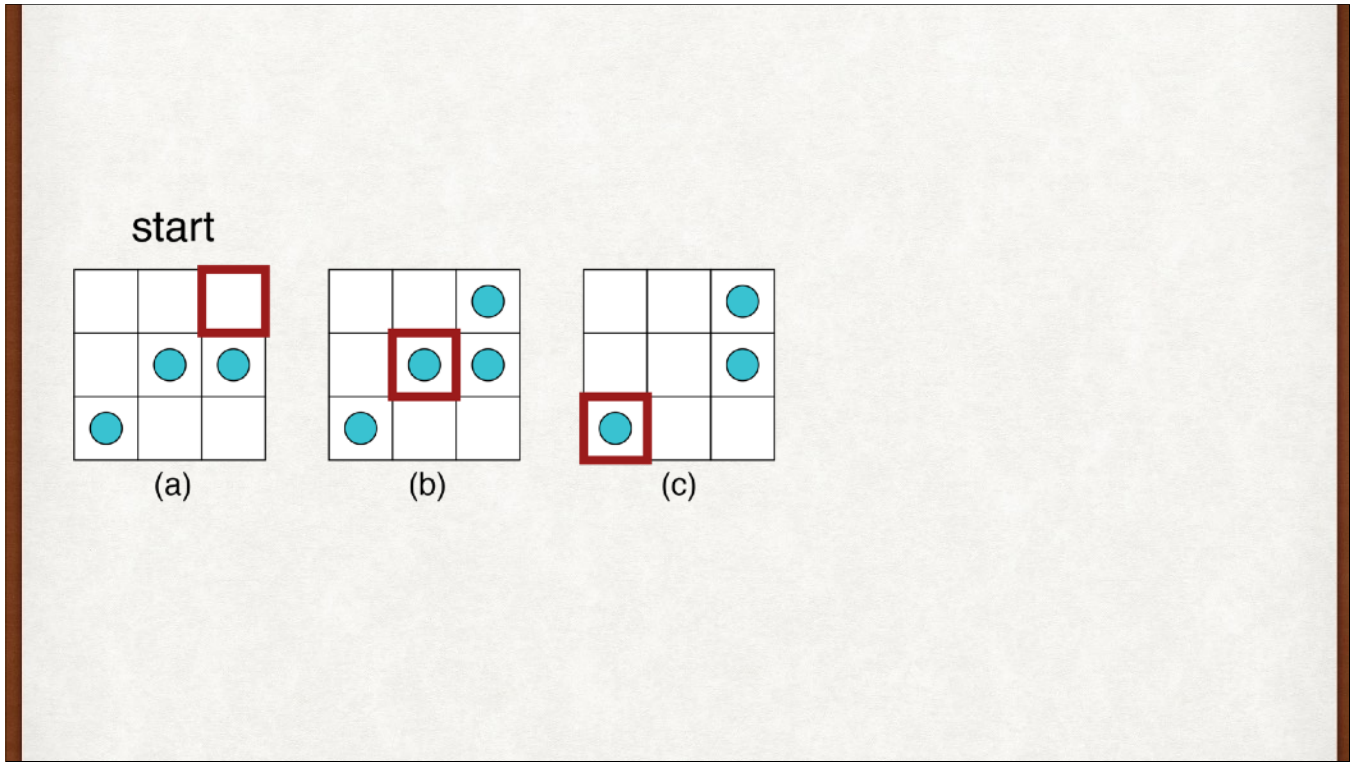


(a)

Playing Flippers. Red cells are where we’re going to flip next.

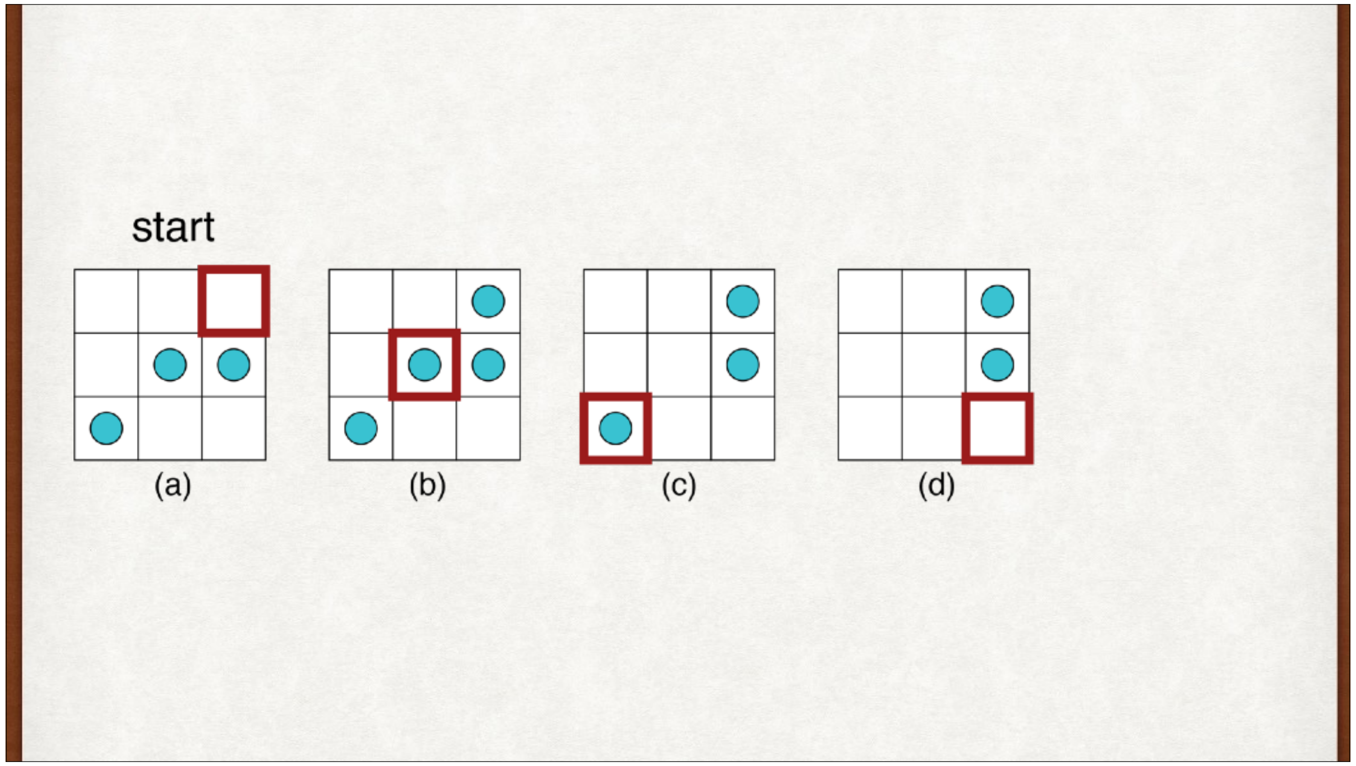


Playing Flippers. Red cells are where we’re going to flip next.

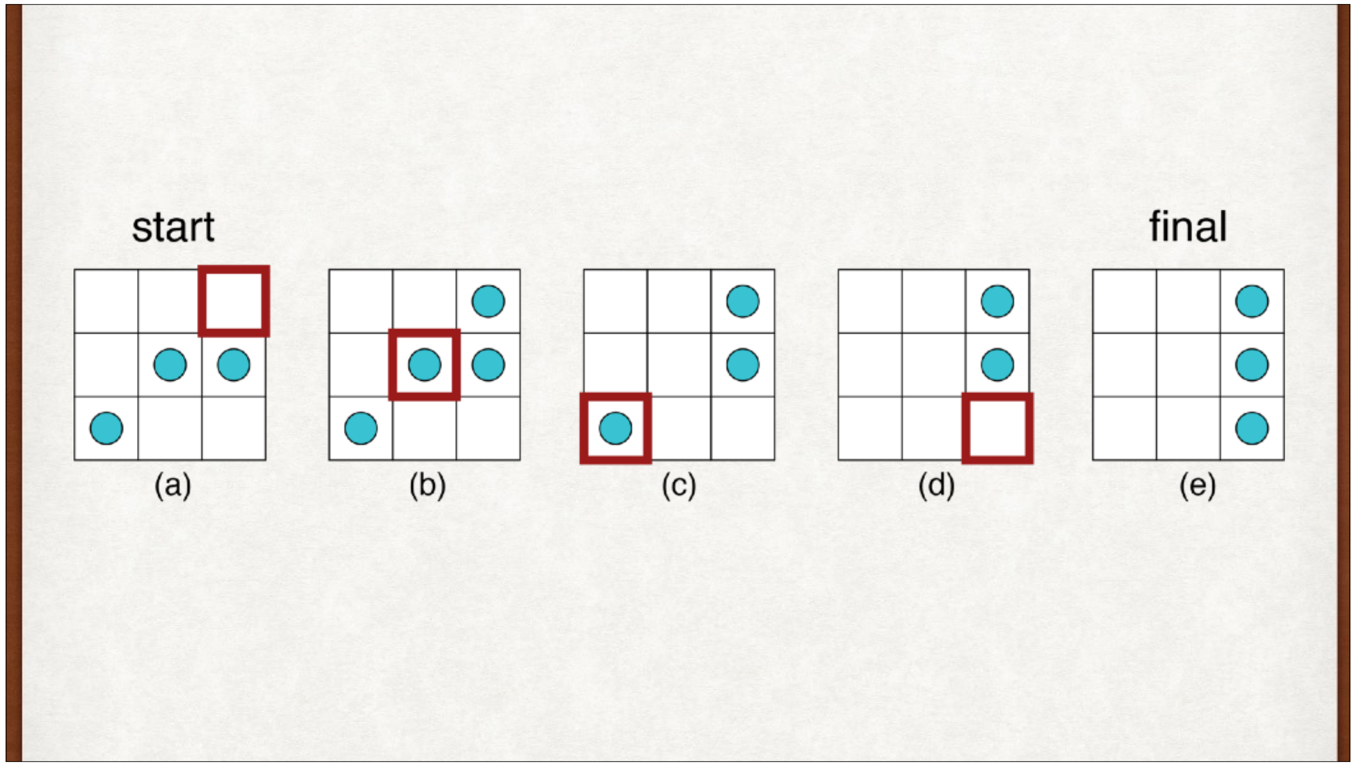


(a) (b) (c)

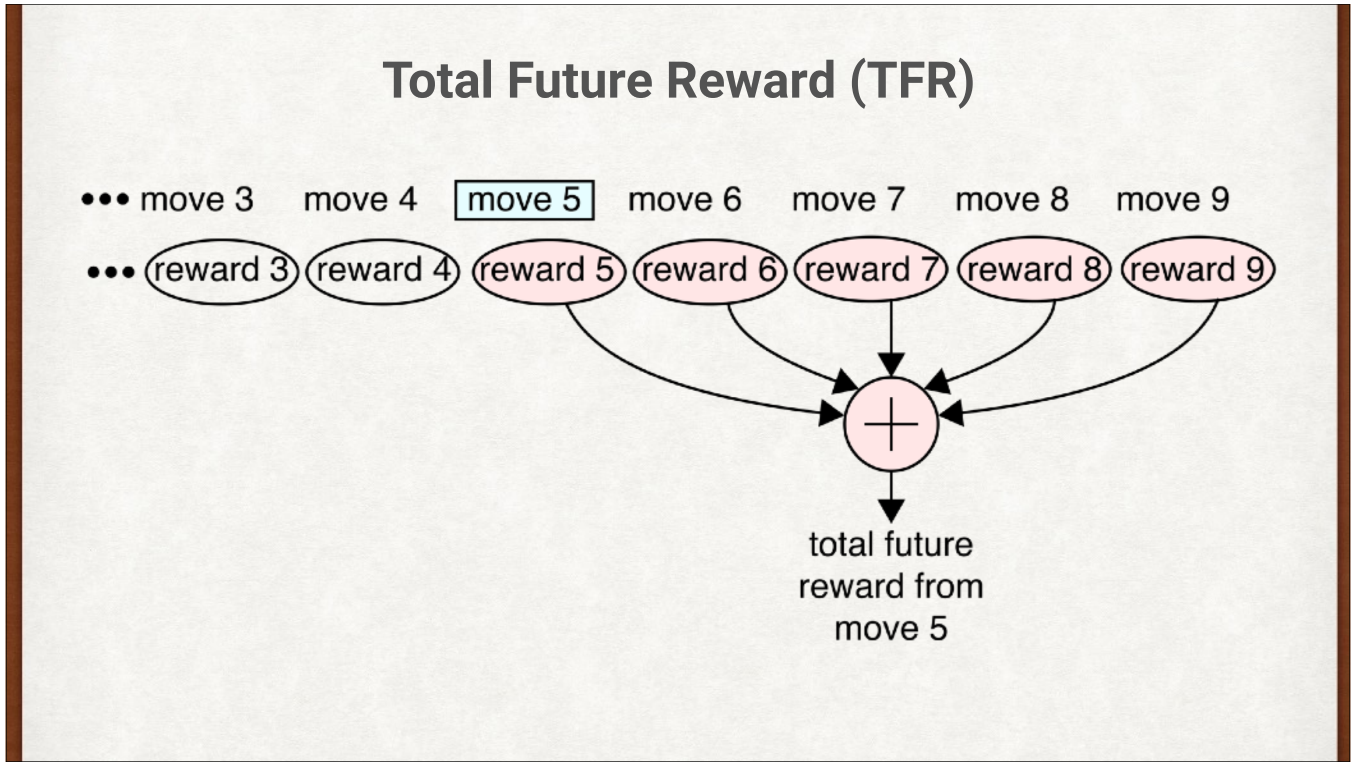
Playing Flippers. Red cells are where we’re going to flip next.



Playing Flippers. Red cells are where we’re going to flip next.

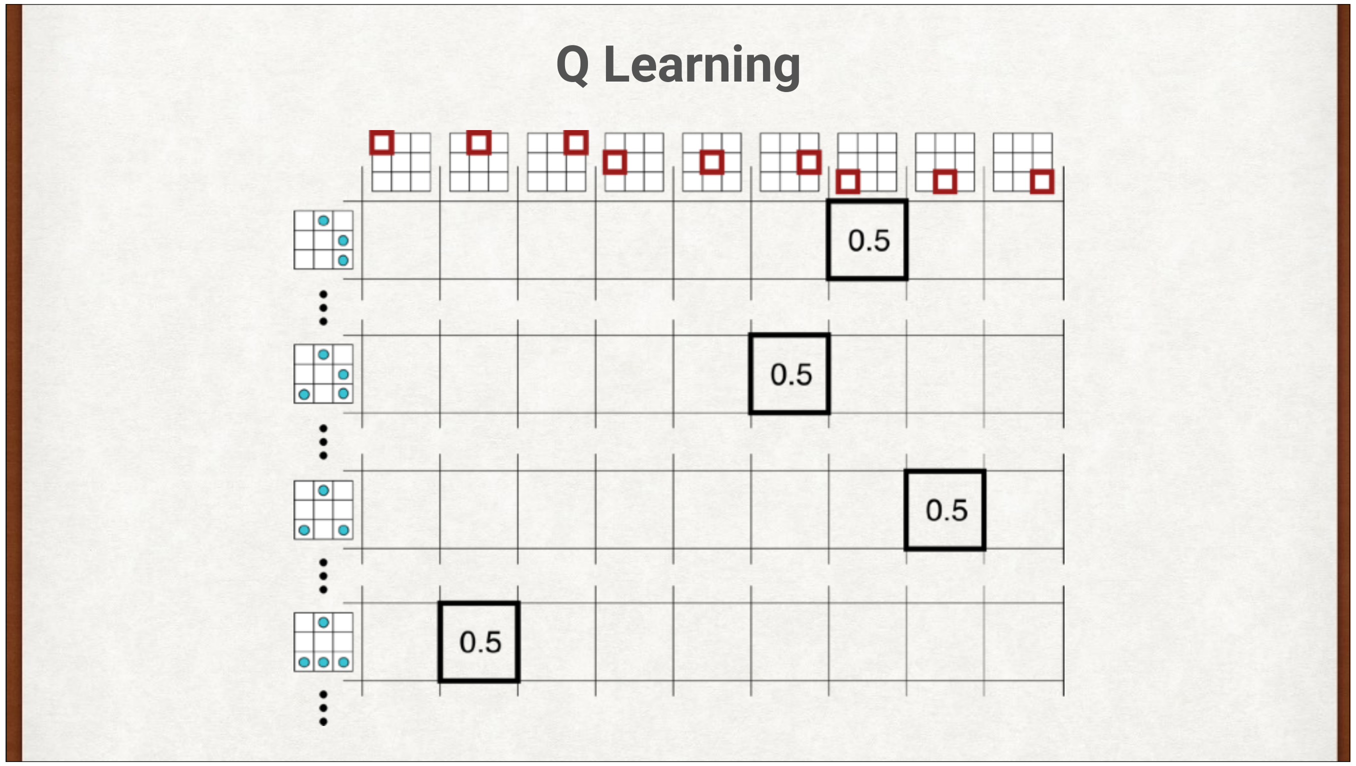


Playing Flippers. Red cells are where we’re going to flip next.

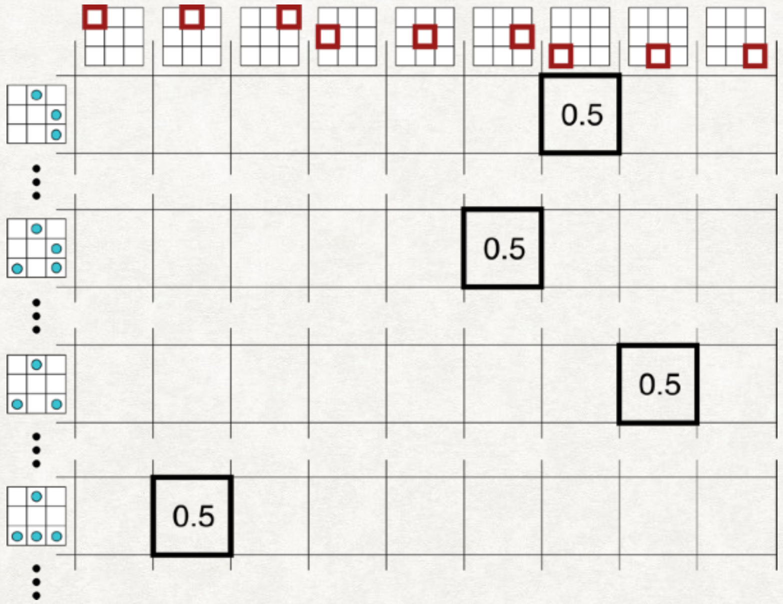


Total Future Reward (TFR)

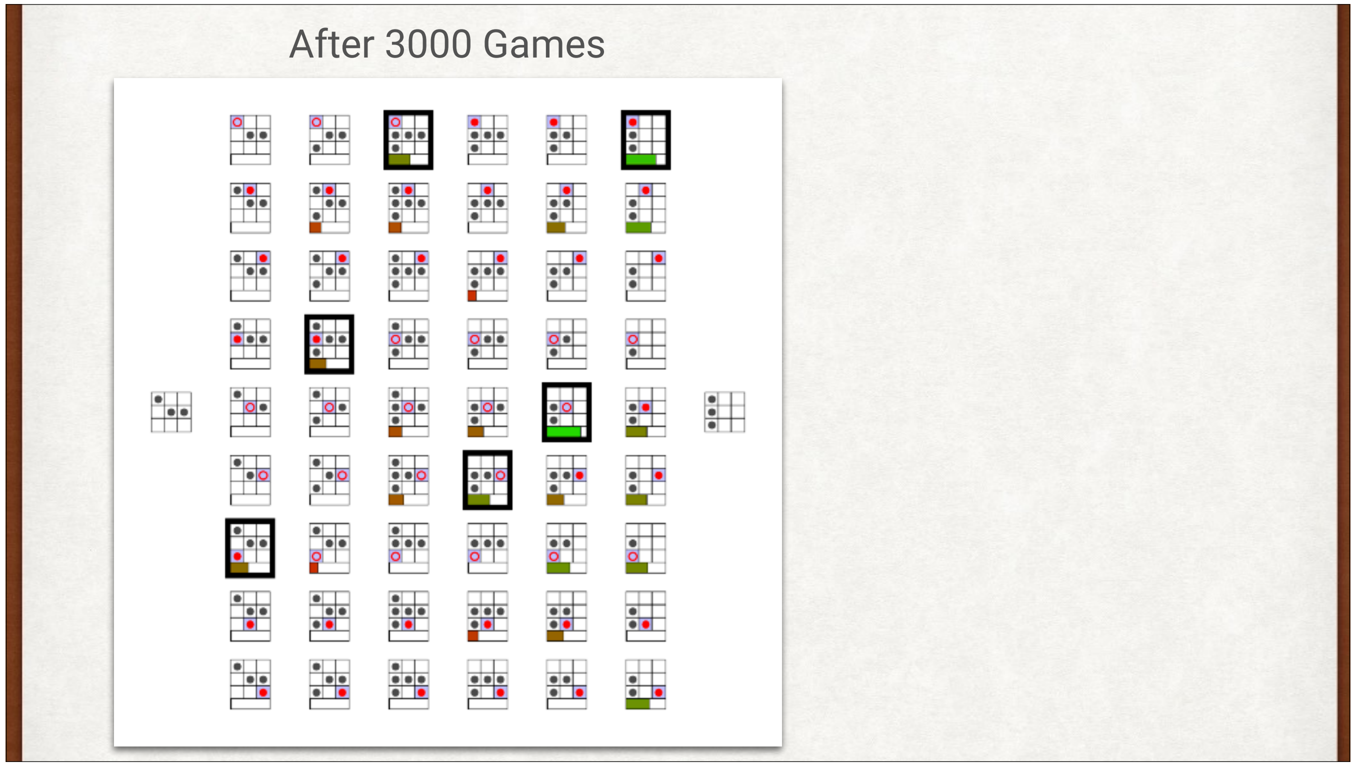
The Total Future Reward tells us how much reward we’ll get from all moves starting with this one.



Q Learning

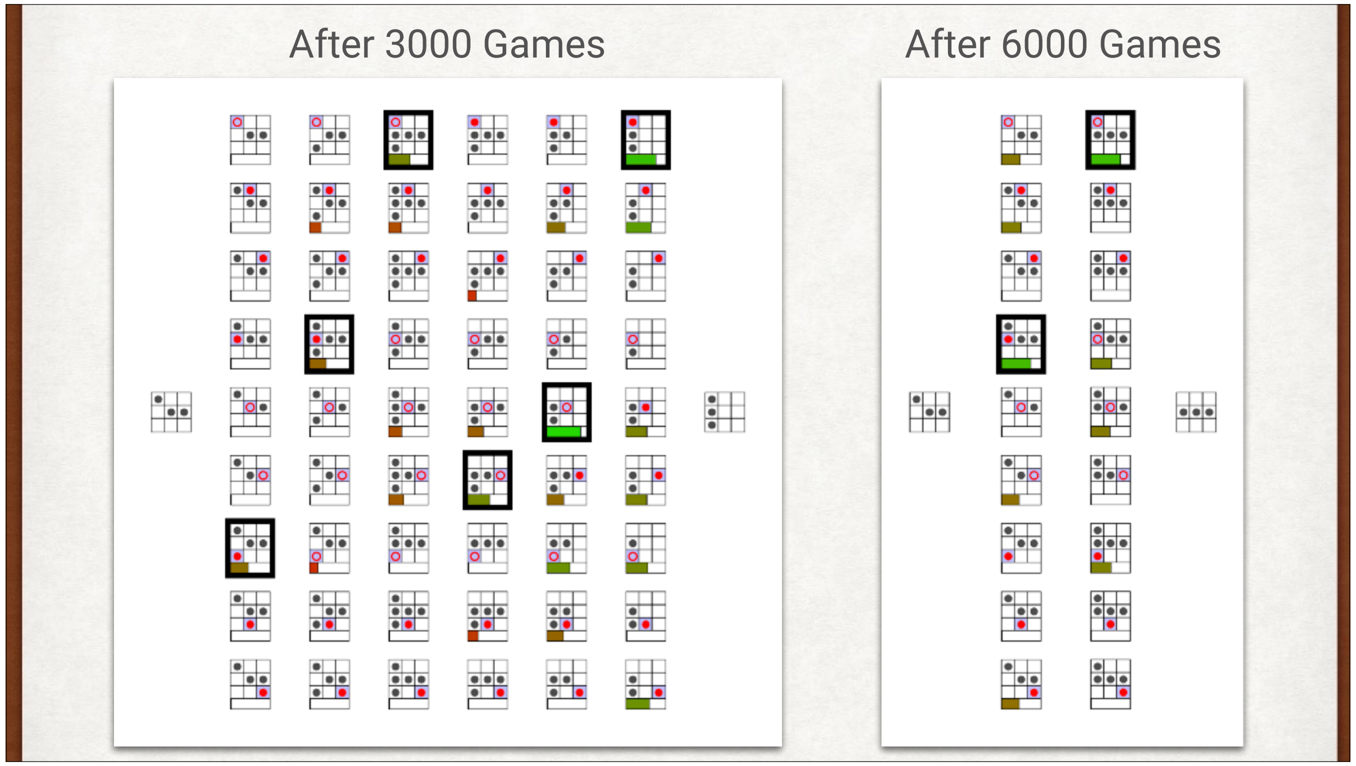


We can store the TFRs in a table indexed by board configuration and choice of move. This is part of Q Learning. **Exploration vs. exploitation**.

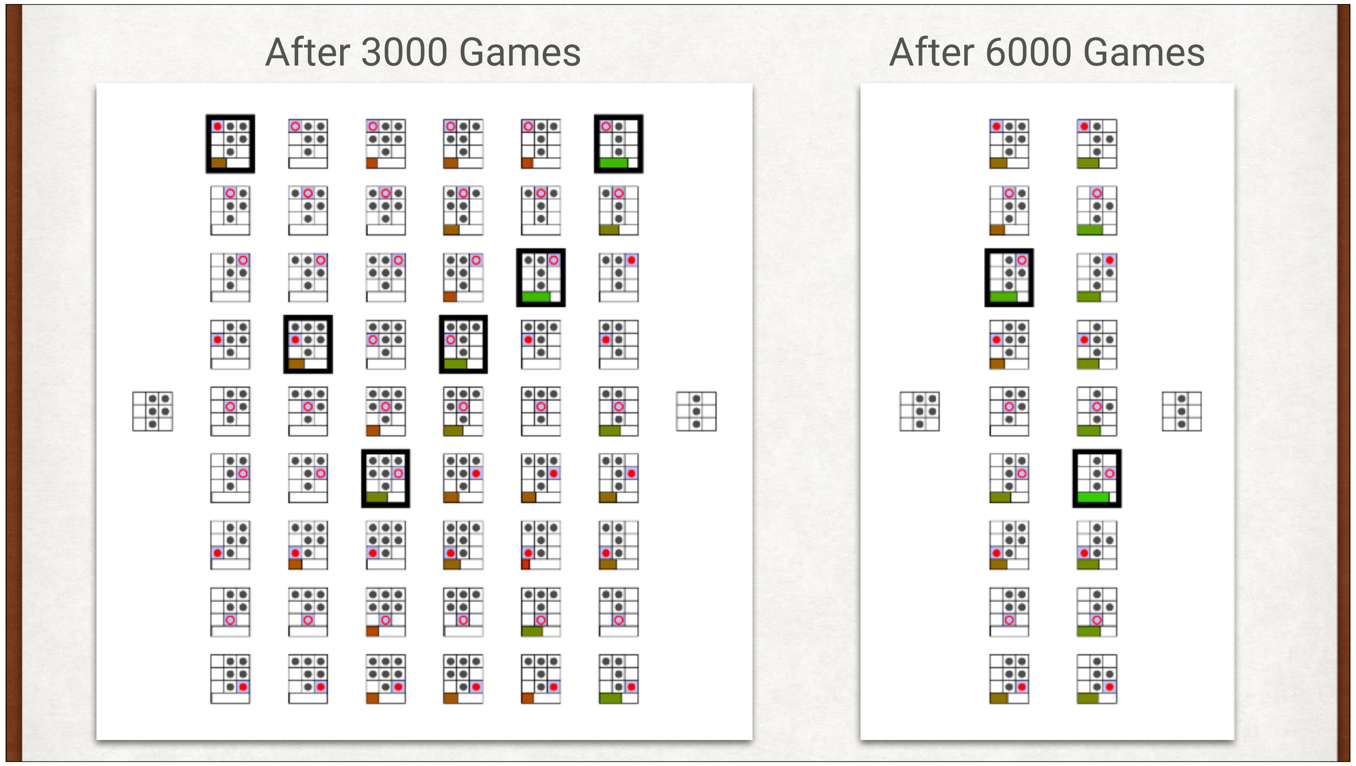


After 3000 Games

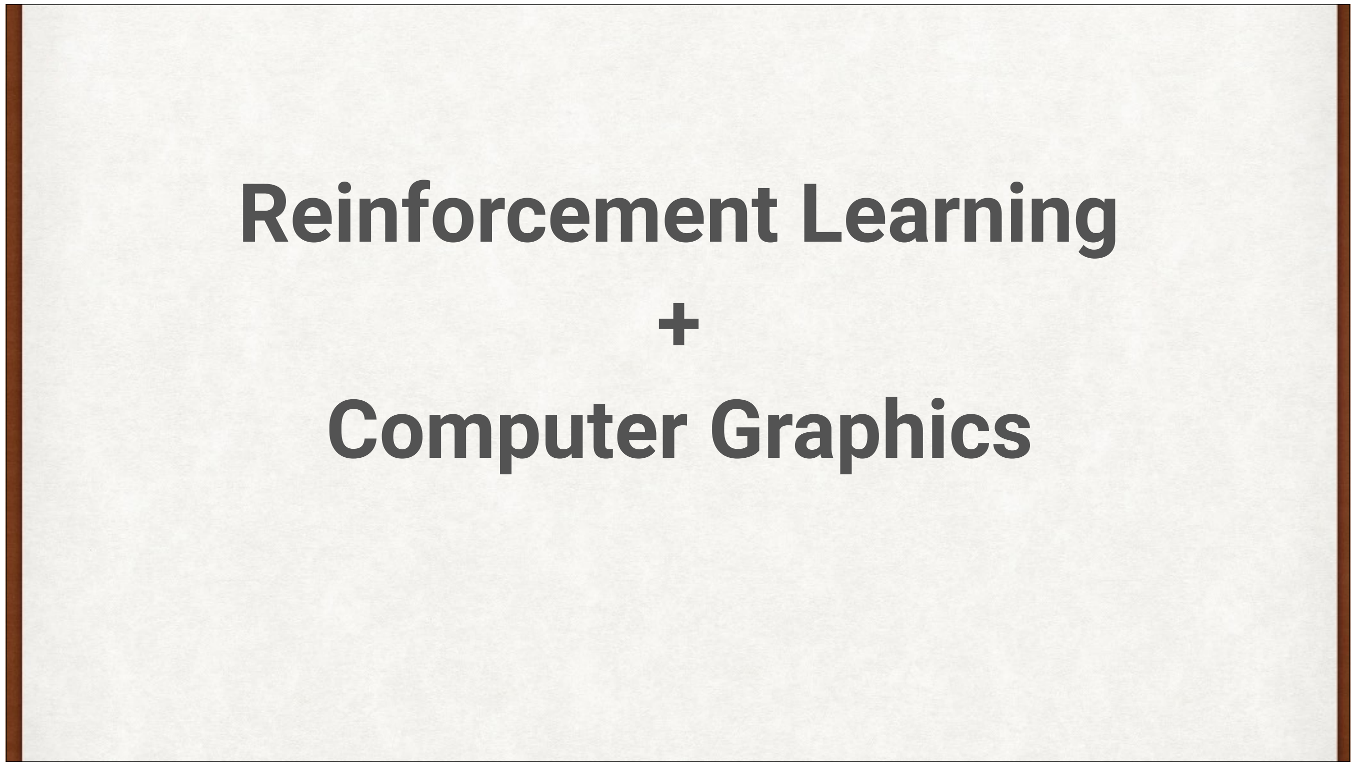
Learning and experience helps you get better! Here the bar at the bottom (coded by color and length) shows the stored TFR for each move. After 3000 games of training, it took a meandering 6 moves to win the game. After 6000 games, the learned TFRs guide us to a victory in just 2 moves, the minimum needed to win this board.



Learning and experience helps you get better! Here the bar at the bottom (coded by color and length) shows the stored TFR for each move. After 3000 games of training, it took a meandering 6 moves to win the game. After 6000 games, the learned TFRs guide us to a victory in just 2 moves, the minimum needed to win this board.

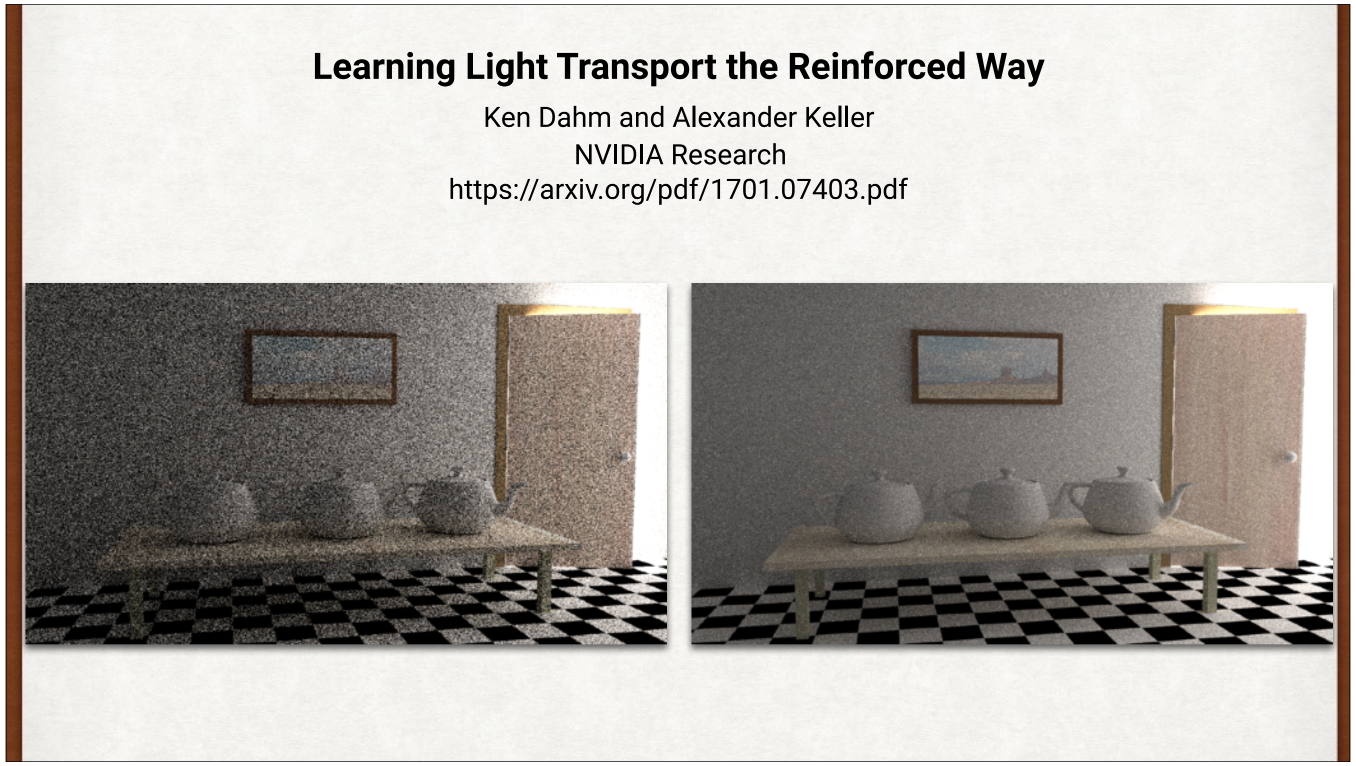


Another game after 3000 games of training, and 6000. **Exploitation vs. exploration**. **SARSA. Determinism!**



Reinforcement Learning
+
Computer Graphics

Using RL to create images.



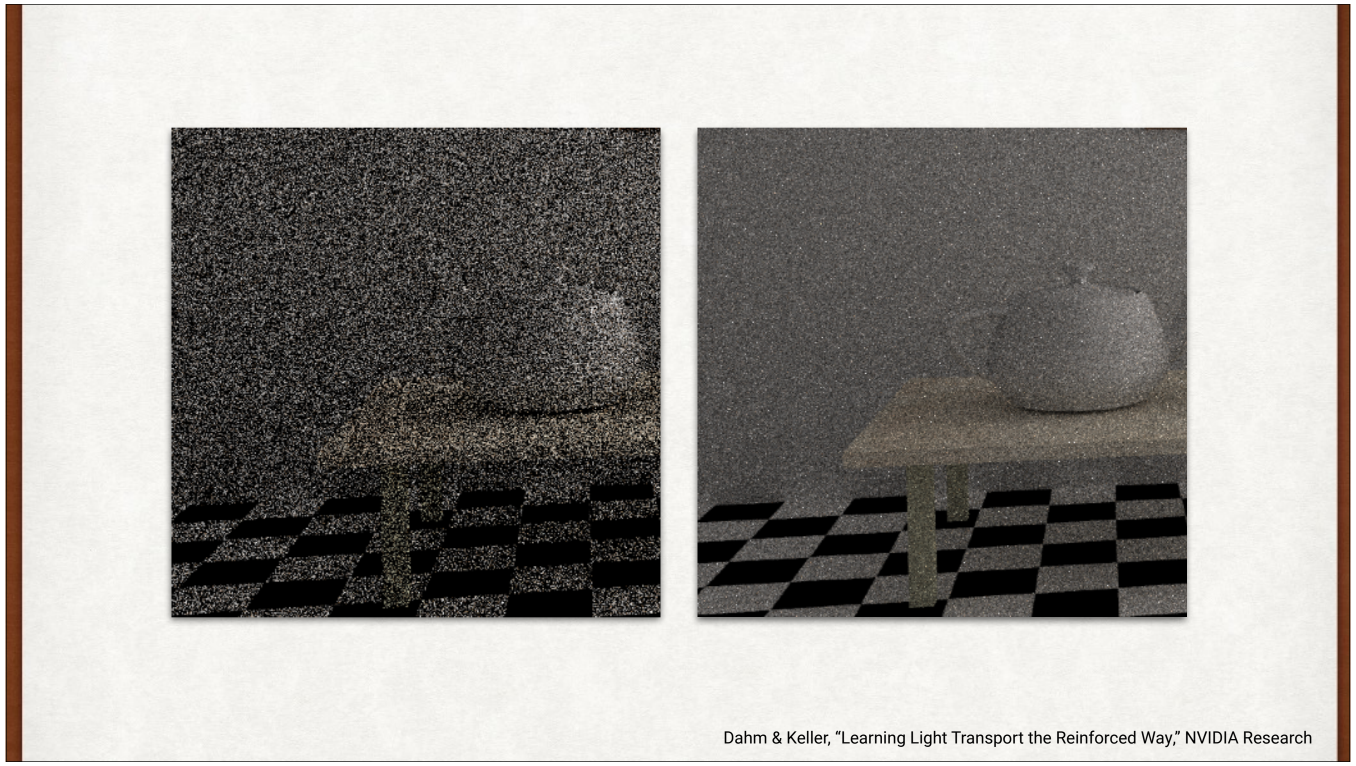
Learning Light Transport the Reinforced Way

Ken Dahm and Alexander Keller

NVIDIA Research

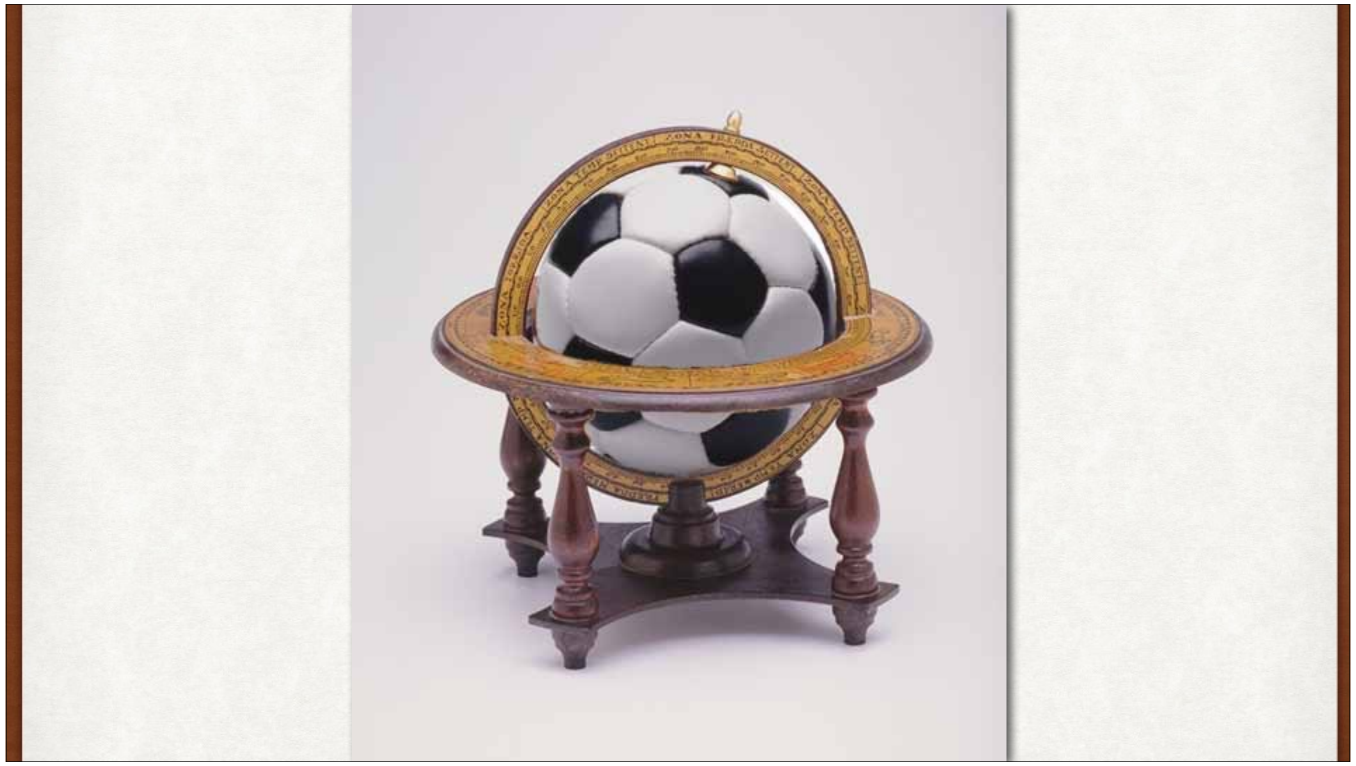
https://arxiv.org/pdf/1701.07403.pdf

Left: Results from a simple path tracer, 1024 samples per pixel. It’s still very noisy. Right: the same number of samples, only their colors are computed using Q learning and BSDF weighting.

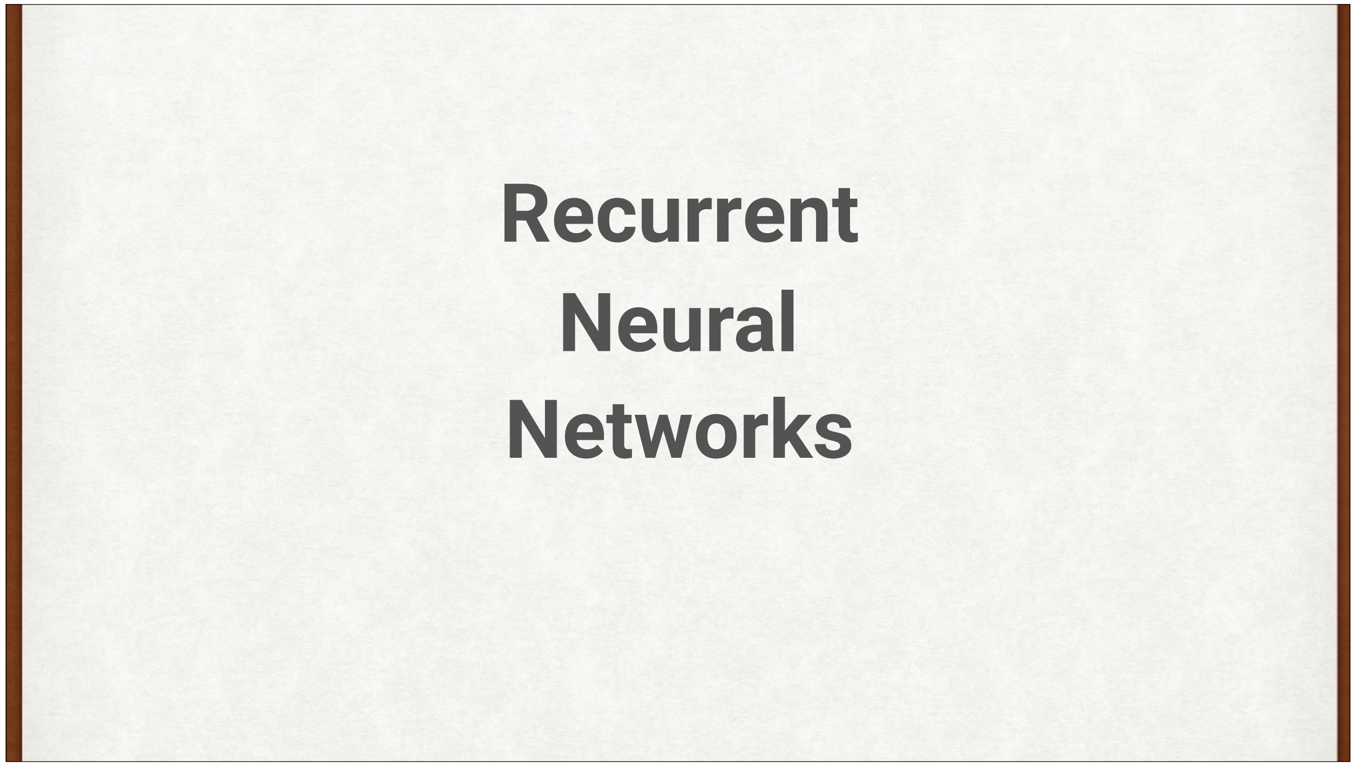


Dahm & Keller, “Learning Light Transport the Reinforced Way,” NVIDIA Research

Closeups: Left: Simple path tracer, 1024 samples per pixel. Right: The same number of samples, but with illumination computed with Q learning and BSDF weighting.

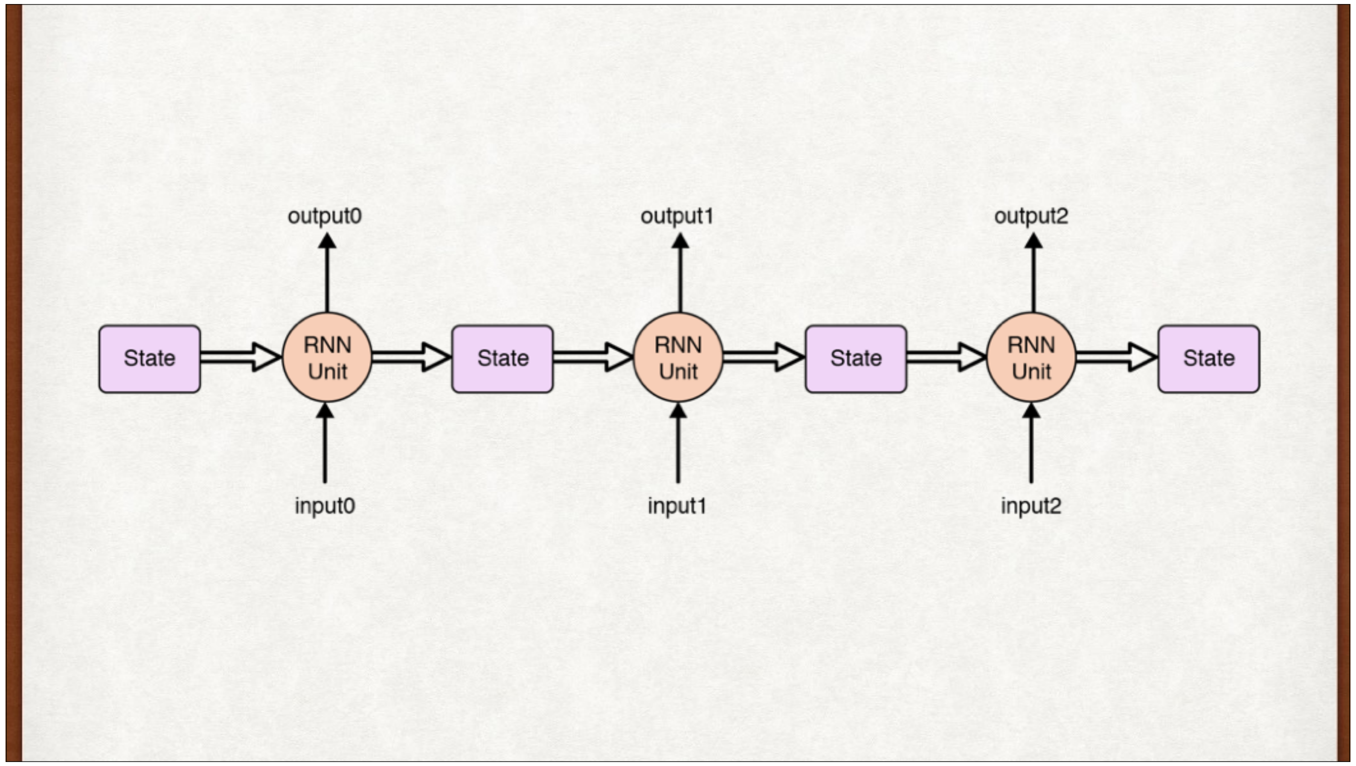


Another moment of rest for another change of topic.

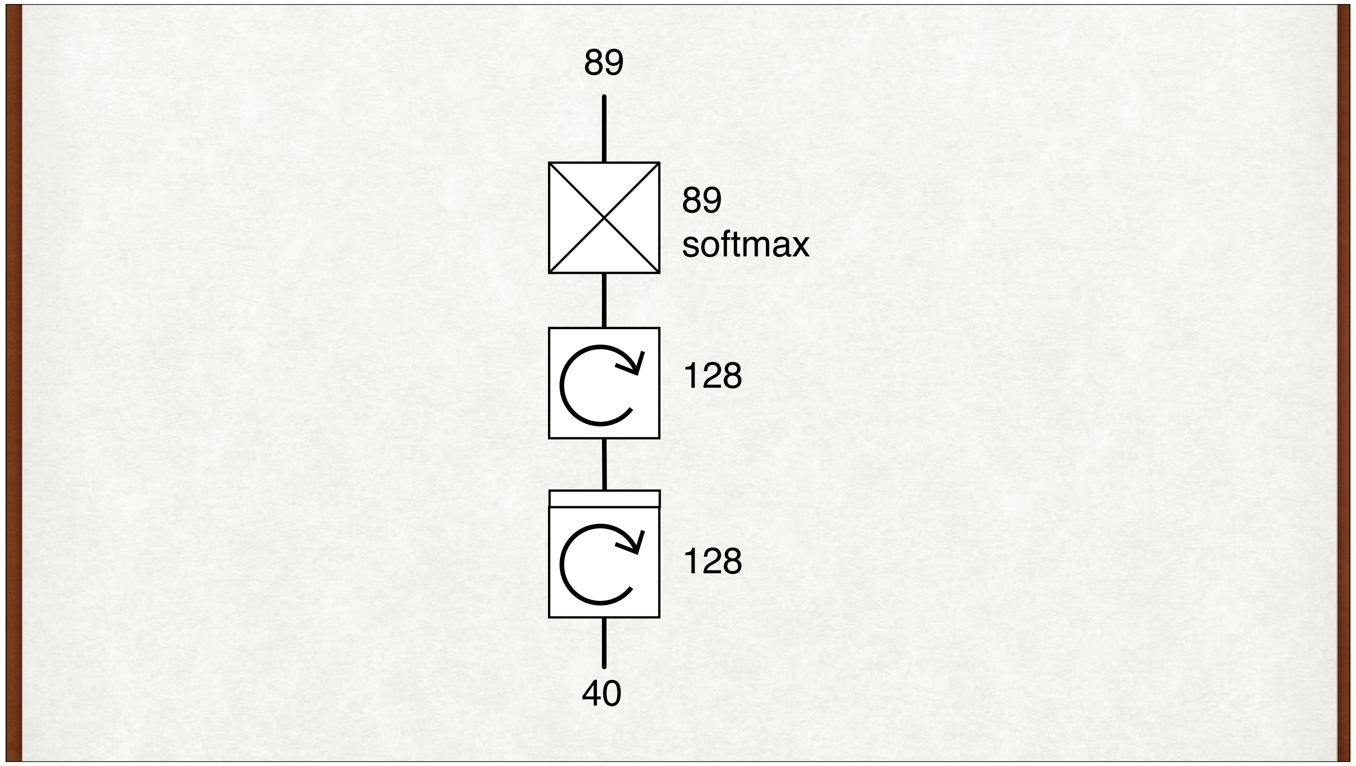


Recurrent Neural Networks

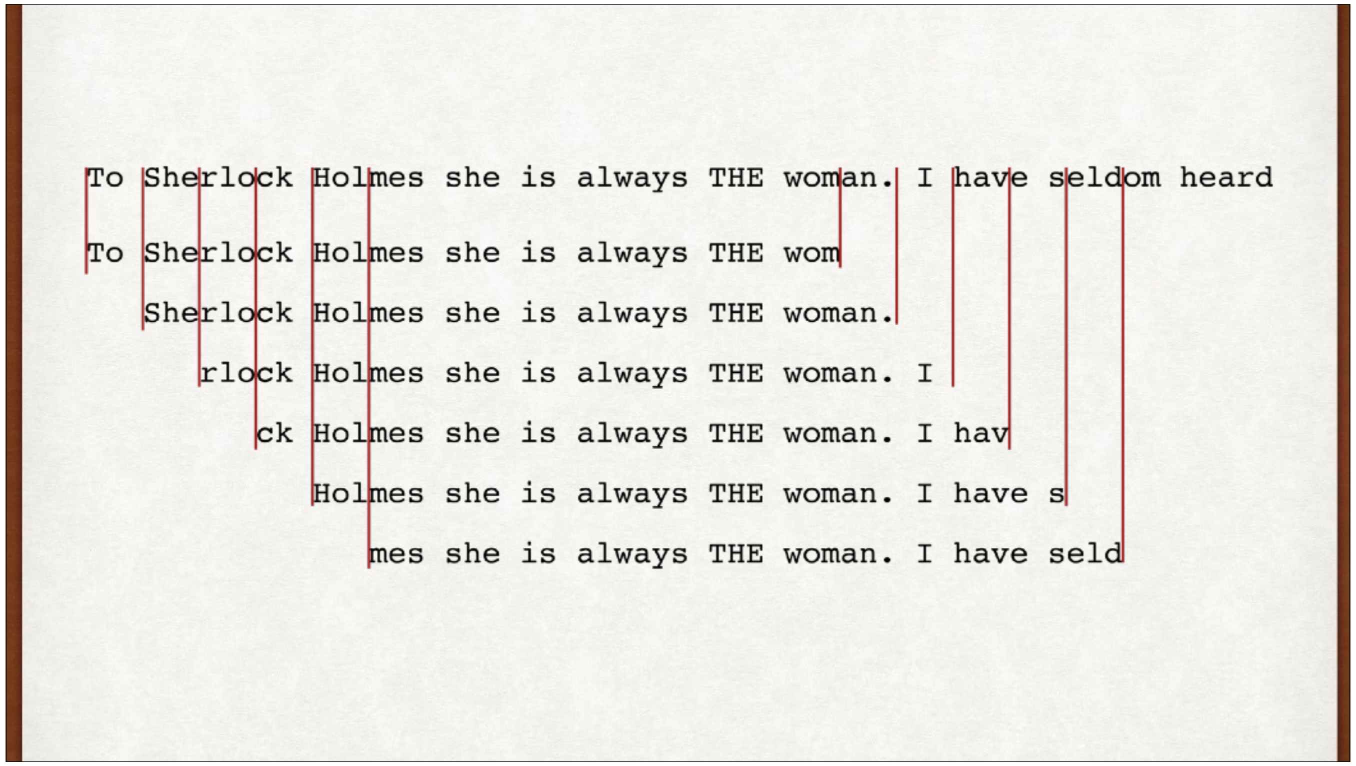
RNNs are a way to remember stuff from the past, which is useful when working with **sequences** of data, like audio or video. There is a growing body of evidence that lots of sequence-related operations can be handled by a CNN as well as, or better than, an RNN. For example, “An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling”. At arxiv.org/abs/1803.01271



Changing the labels, an RNN reads and writes a single collection of values called the “state”. So an RNN operates on its input, and updates and saves its state for the next input.



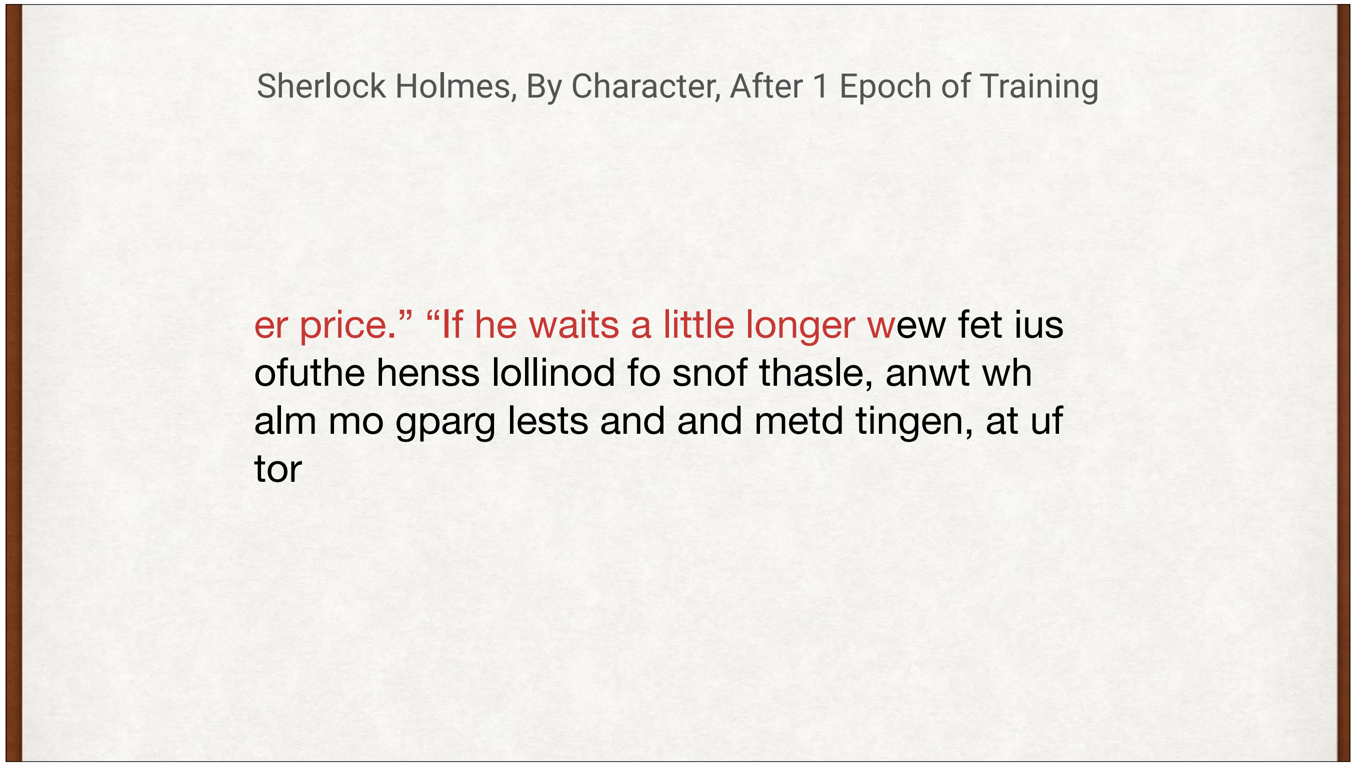
Just one tiny example of RNNs in action. Here is a recurrent network for generating new text from Sherlock Holmes stories. The two RNNs each have 128 elements of state. The box on the lower RNN means it returns an output for every input, rather than just the last one in a sequence. 40 characters of text go in, and 89 probabilities (one for each letter and punctuation mark) come out. The most probable character is chosen and added to the growing output.



```
To Sherlock Holmes she is always THE woman. I have seldom heard

To Sherlock Holmes she is always THE wom
   Sherlock Holmes she is always THE woman.
      rlock Holmes she is always THE woman. I
         ck Holmes she is always THE woman. I hav
            Holmes she is always THE woman. I have s
               mes she is always THE woman. I have seld
```

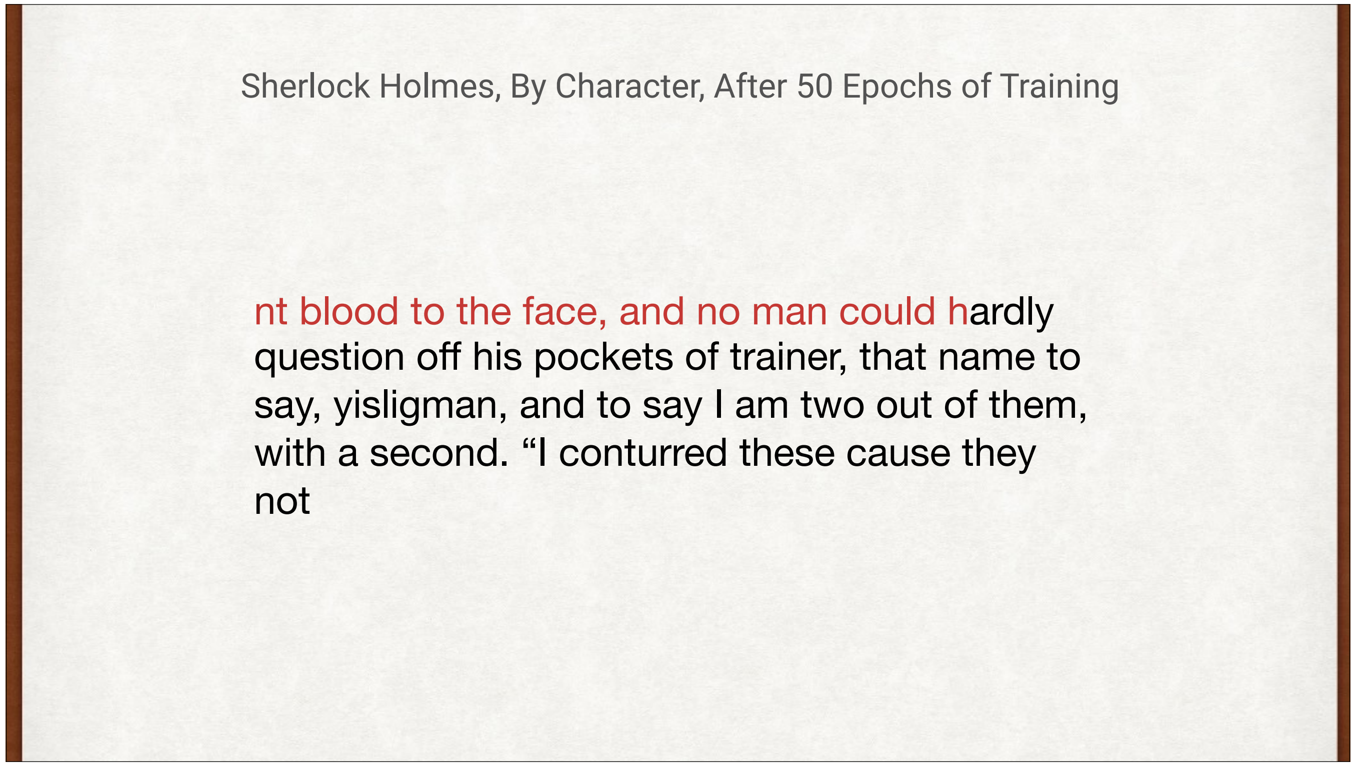
Chopping up text into overlapping, 40-character chunks.



Sherlock Holmes, By Character, After 1 Epoch of Training

er price.” “If he waits a little longer wew fet ius ofuthe henss lollinod fo snof thasle, anwt wh alm mo gparg lests and and metd tingen, at uf tor

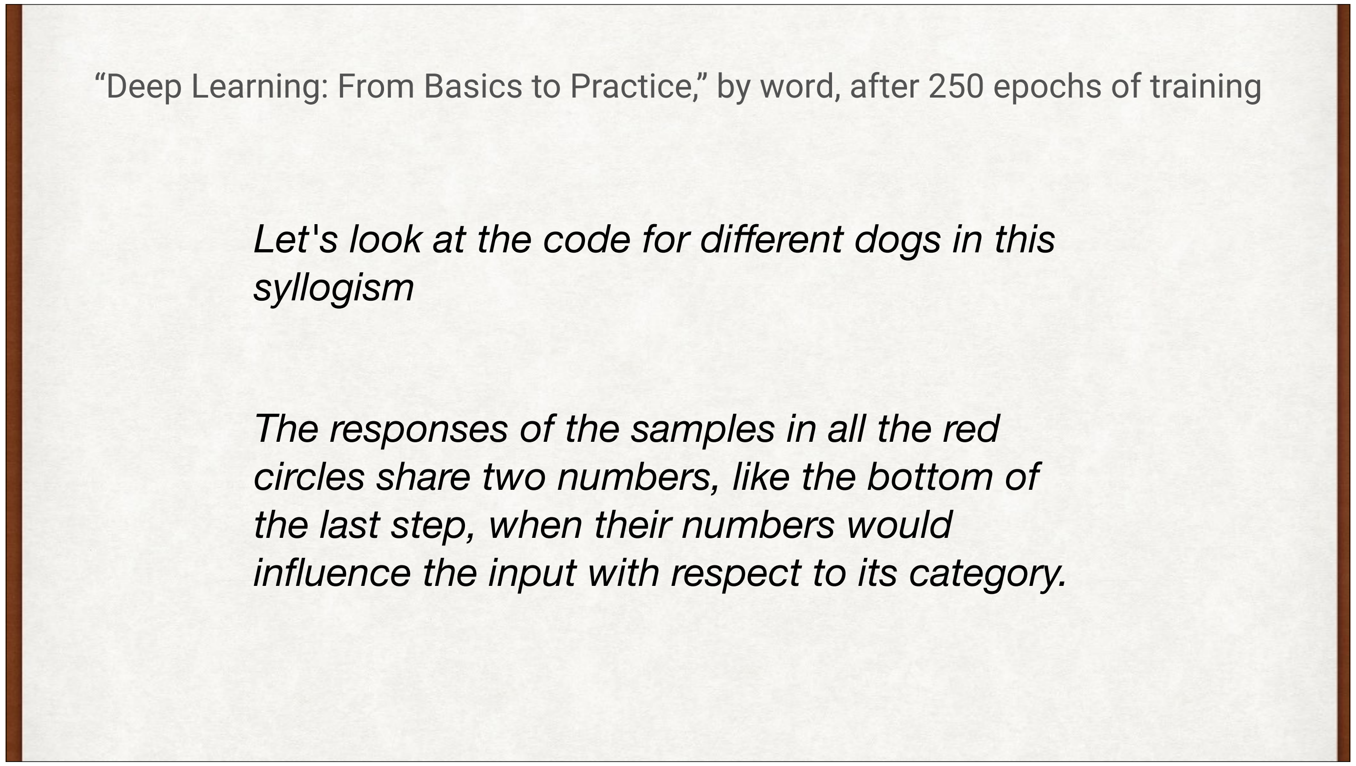
The results after 1 epoch are not so great. The first 40 characters, not in italics, are the randomly-chosen starting text.



Sherlock Holmes, By Character, After 50 Epochs of Training

nt blood to the face, and no man could hardly question off his pockets of trainer, that name to say, yisligman, and to say I am two out of them, with a second. “I conturred these cause they not

After 50 epochs, it’s a lot better.

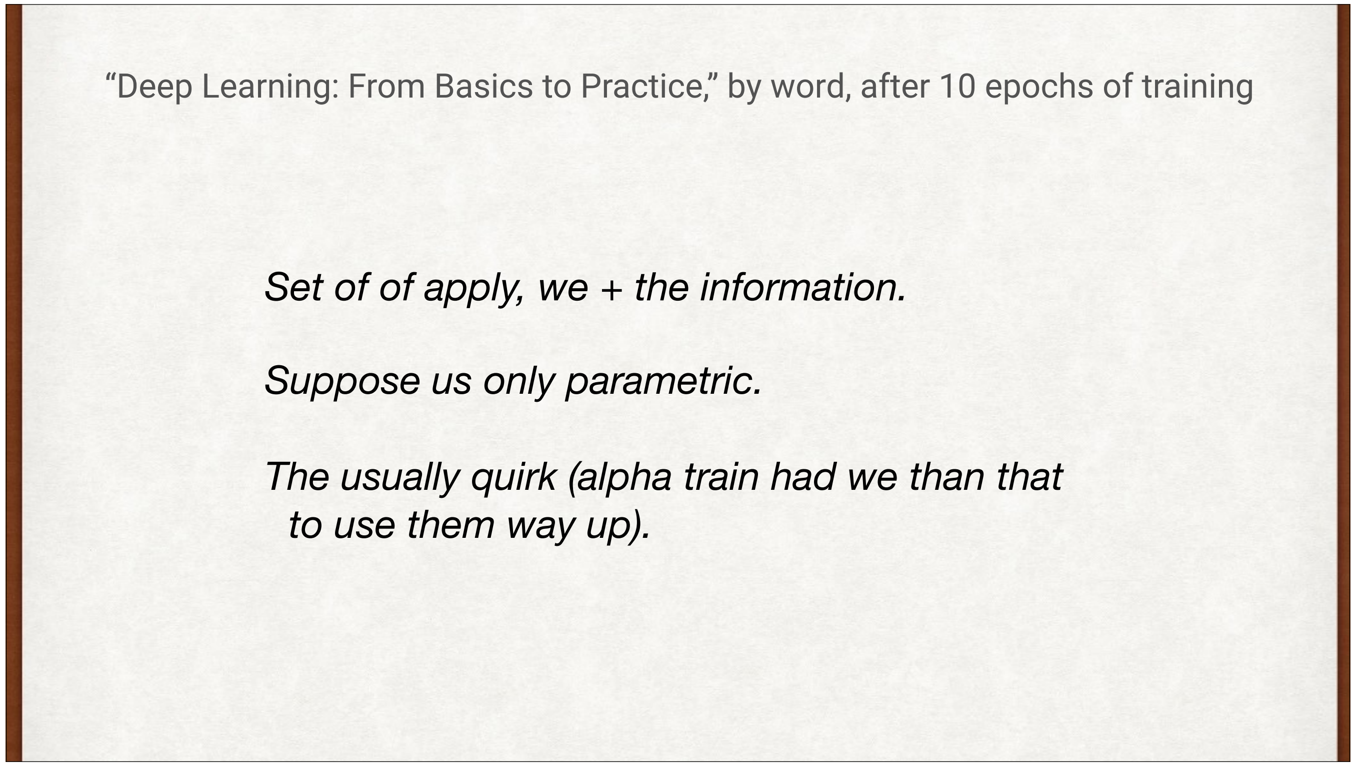


“Deep Learning: From Basics to Practice,” by word, after 250 epochs of training

Let's look at the code for different dogs in this syllogism

The responses of the samples in all the red circles share two numbers, like the bottom of the last step, when their numbers would influence the input with respect to its category.

Text generated using my book as the source, but using words rather than characters.



“Deep Learning: From Basics to Practice,” by word, after 10 epochs of training

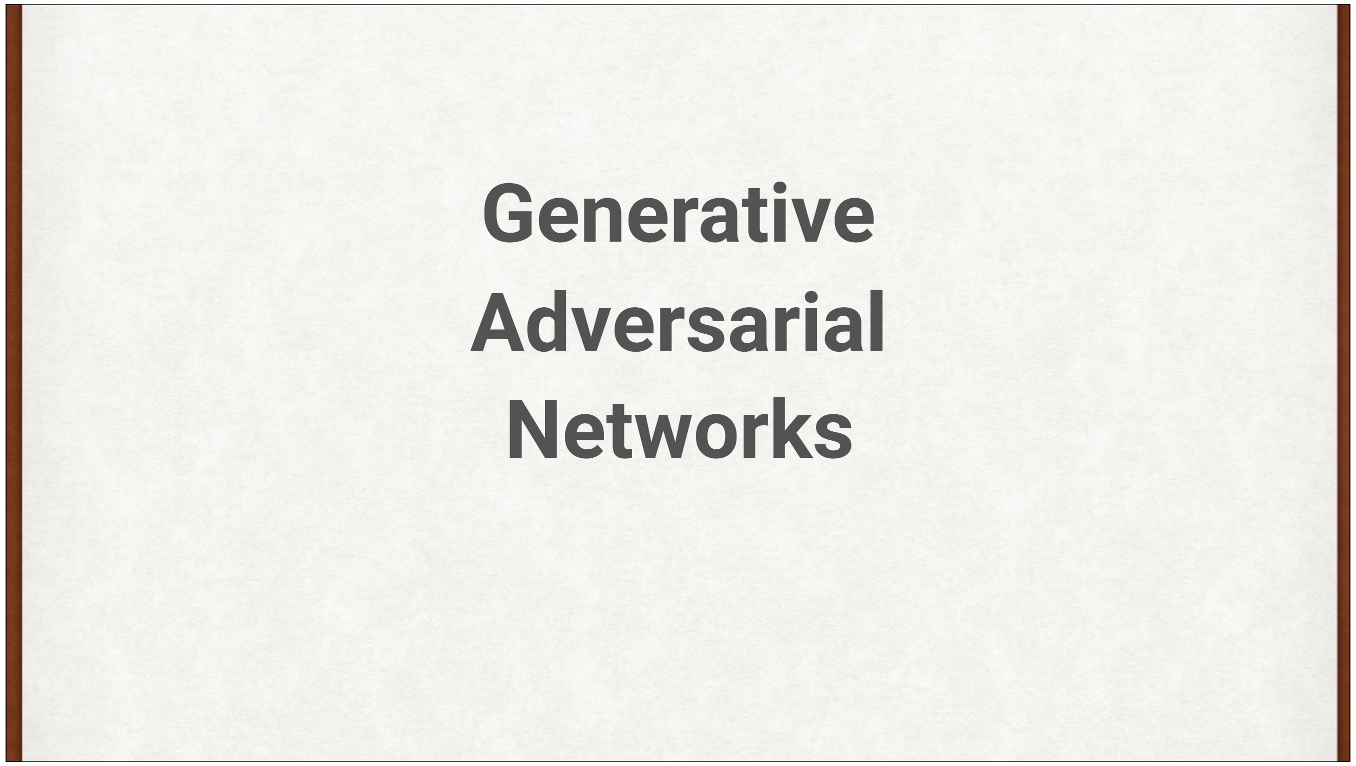
Set of of apply, we + the information.

Suppose us only parametric.

The usually quirk (alpha train had we than that to use them way up).

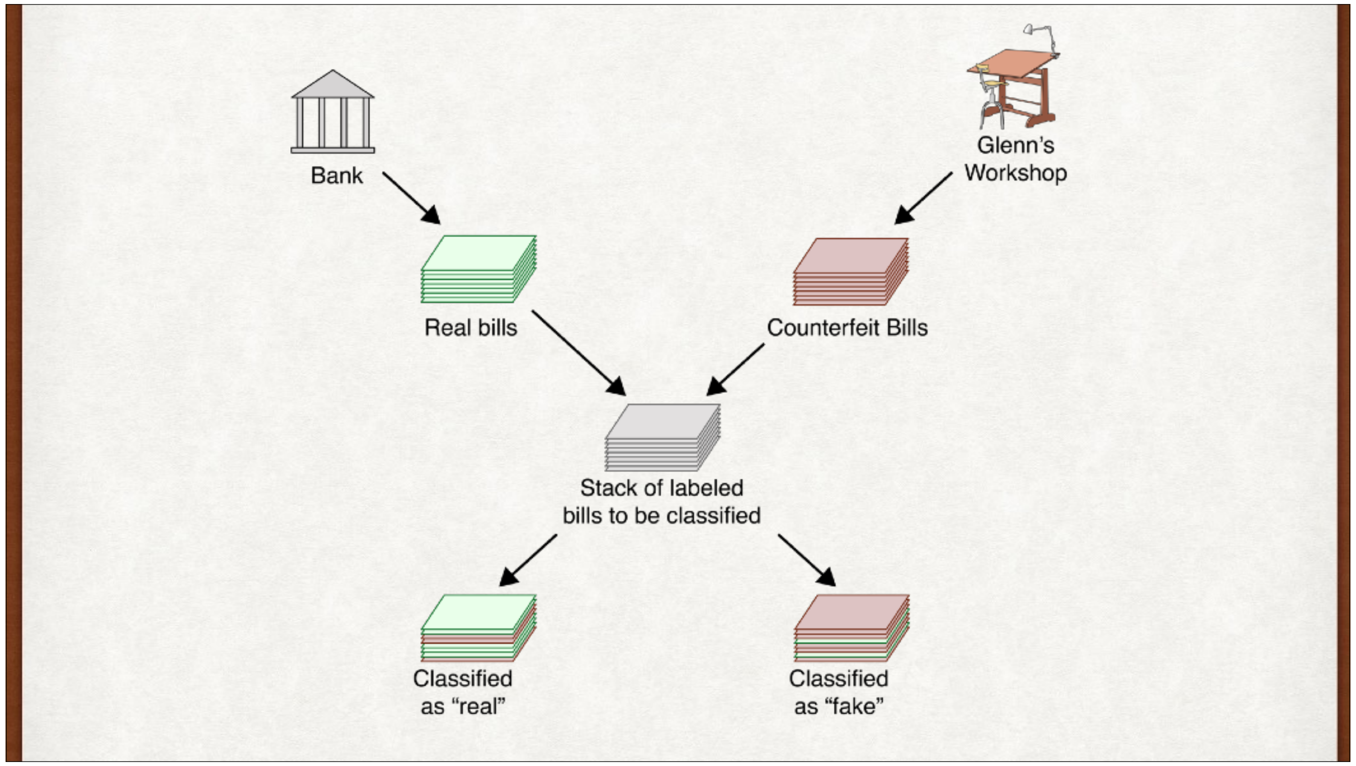
Short phrases are more fun!

A moment of relaxation before our next topic.

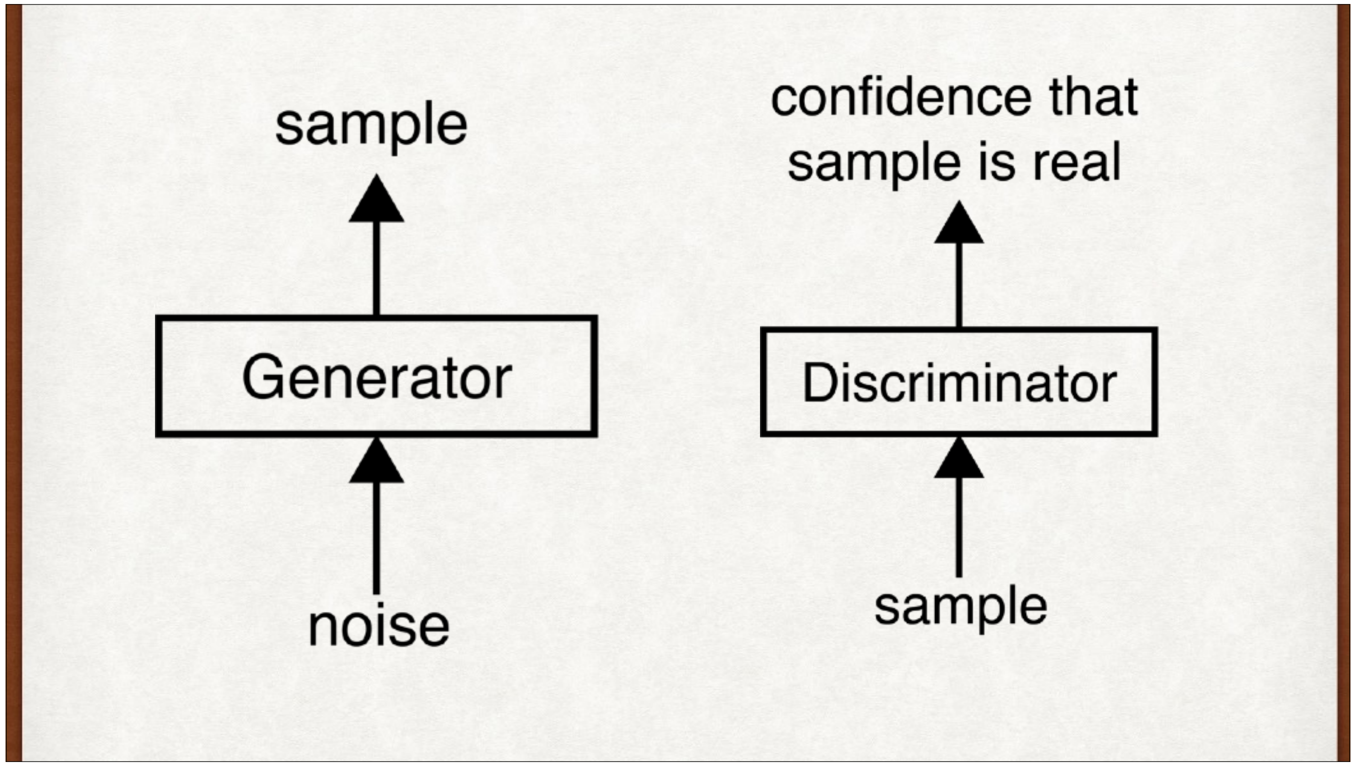


Generative Adversarial Networks

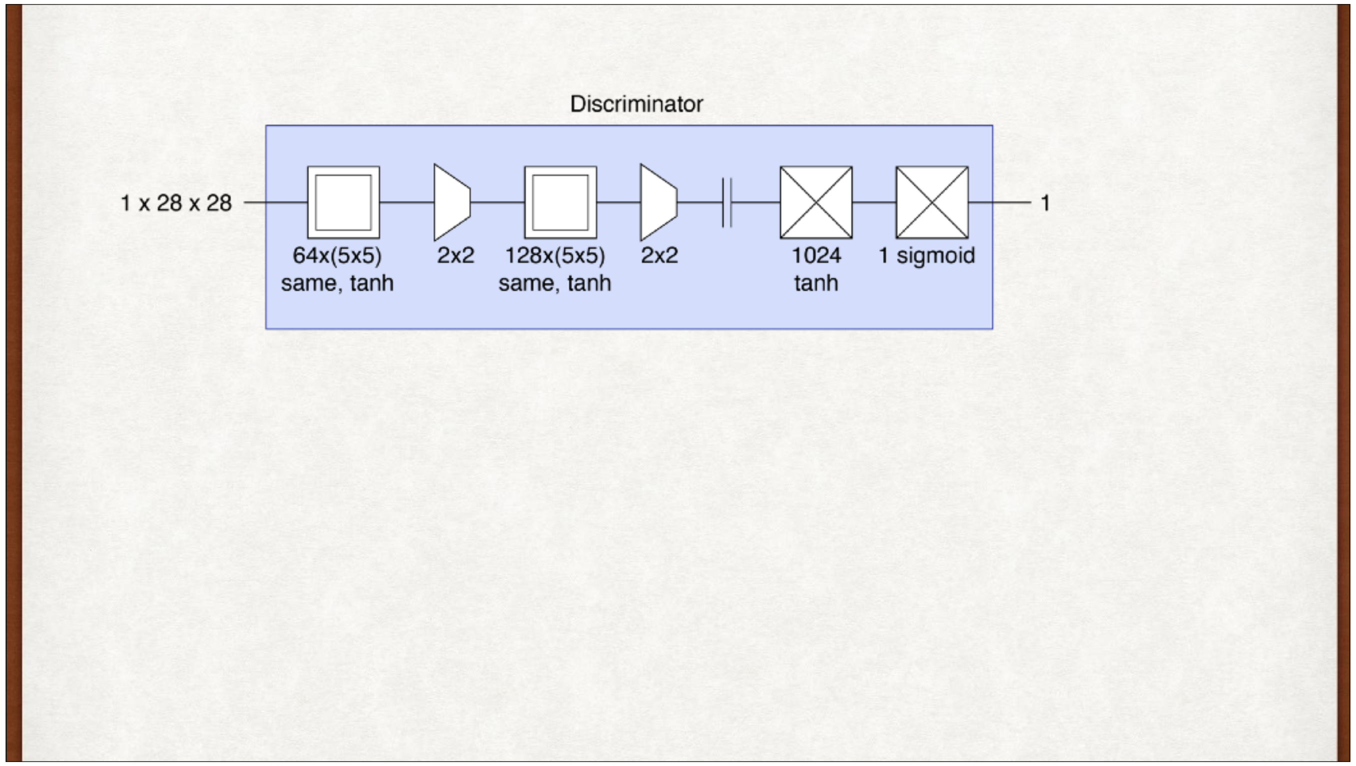
GANs learn how to create new data by running a competition between a data generator and a detector that distinguishes between original input data and generated data.



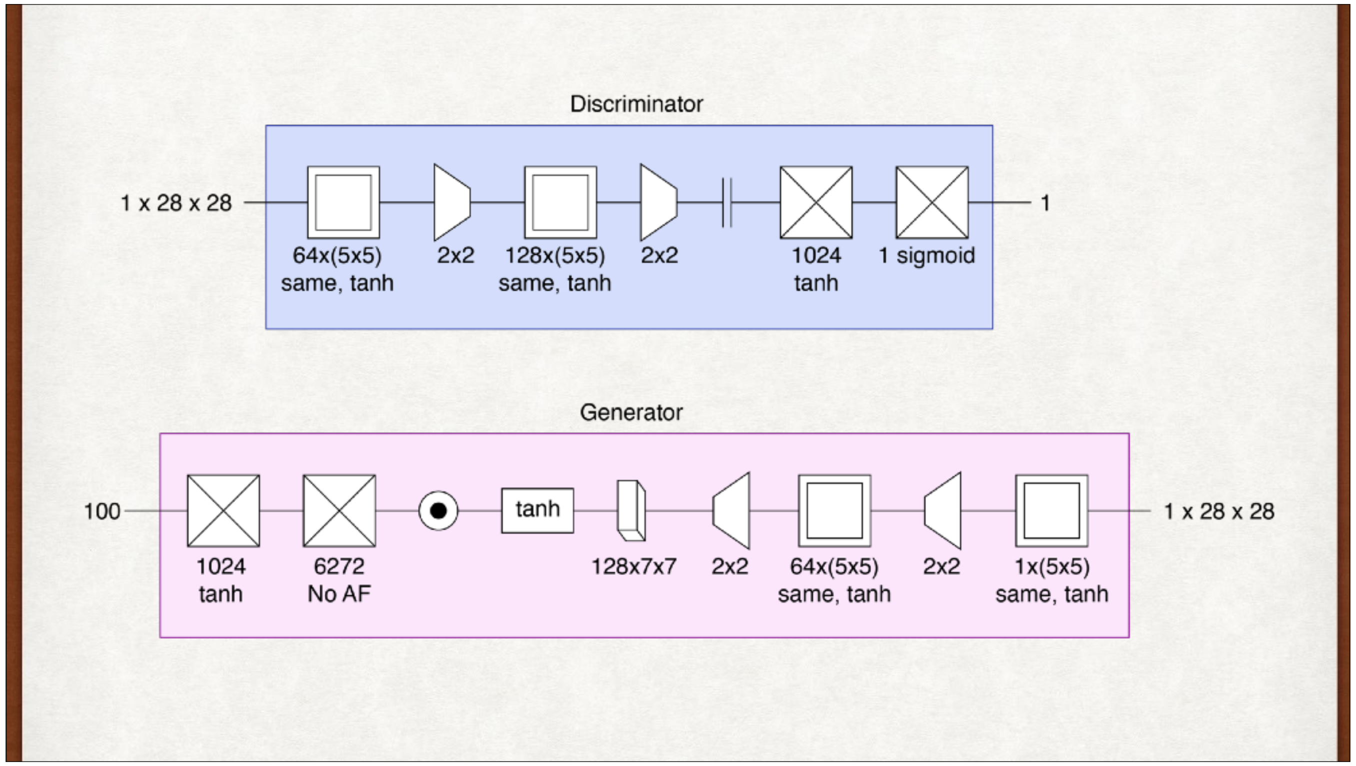
Glenn wants to make counterfeit money. Dawn takes each day's output, plus some real money from the bank, and tries to tell which bills are real and which are counterfeits. Glenn is the "generator," and Dawn is the "detector." Each tries to become as skilled as possible at their task, and that simultaneous challenge drives both of them to improve.



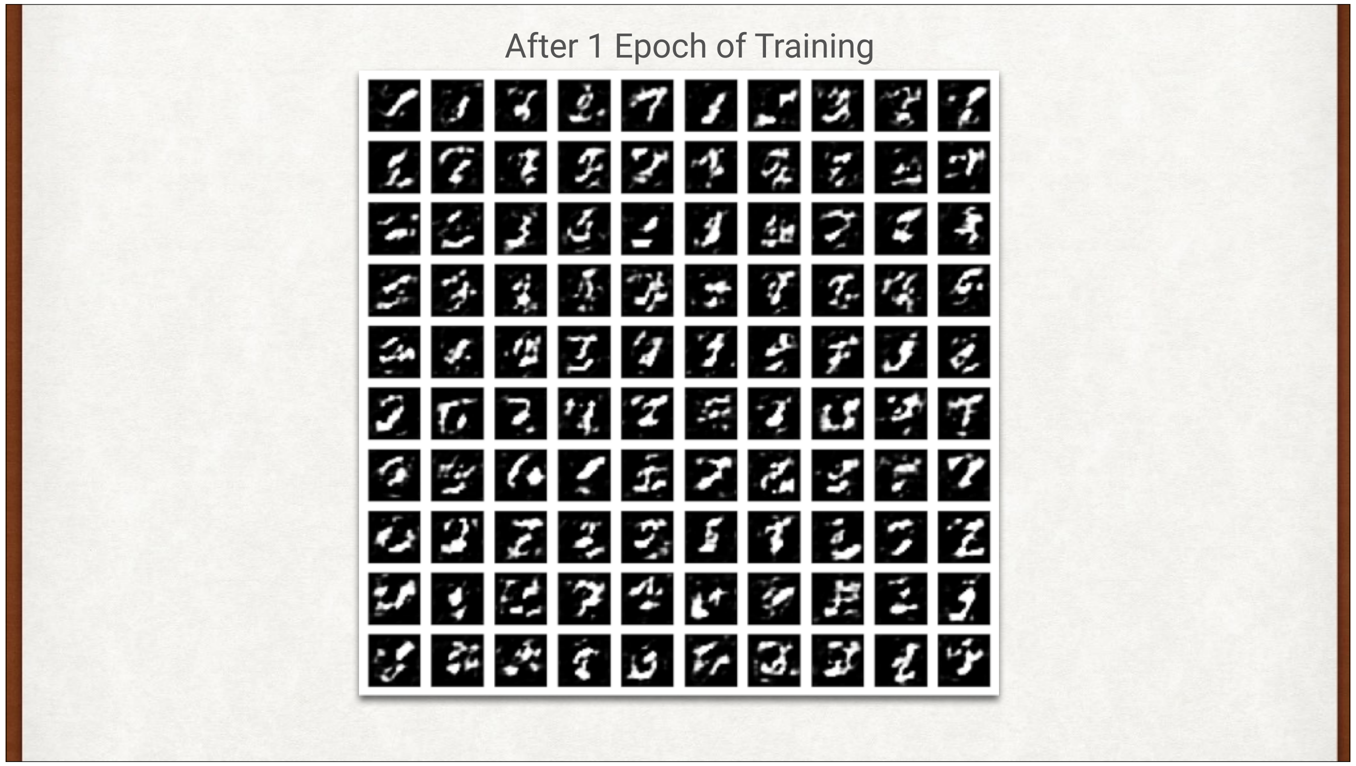
The two pieces of a GAN. The generator turns noise into fake bills. The discriminator tries to spot fake bills.



A discriminator and generator for a GAN using MNIST data.

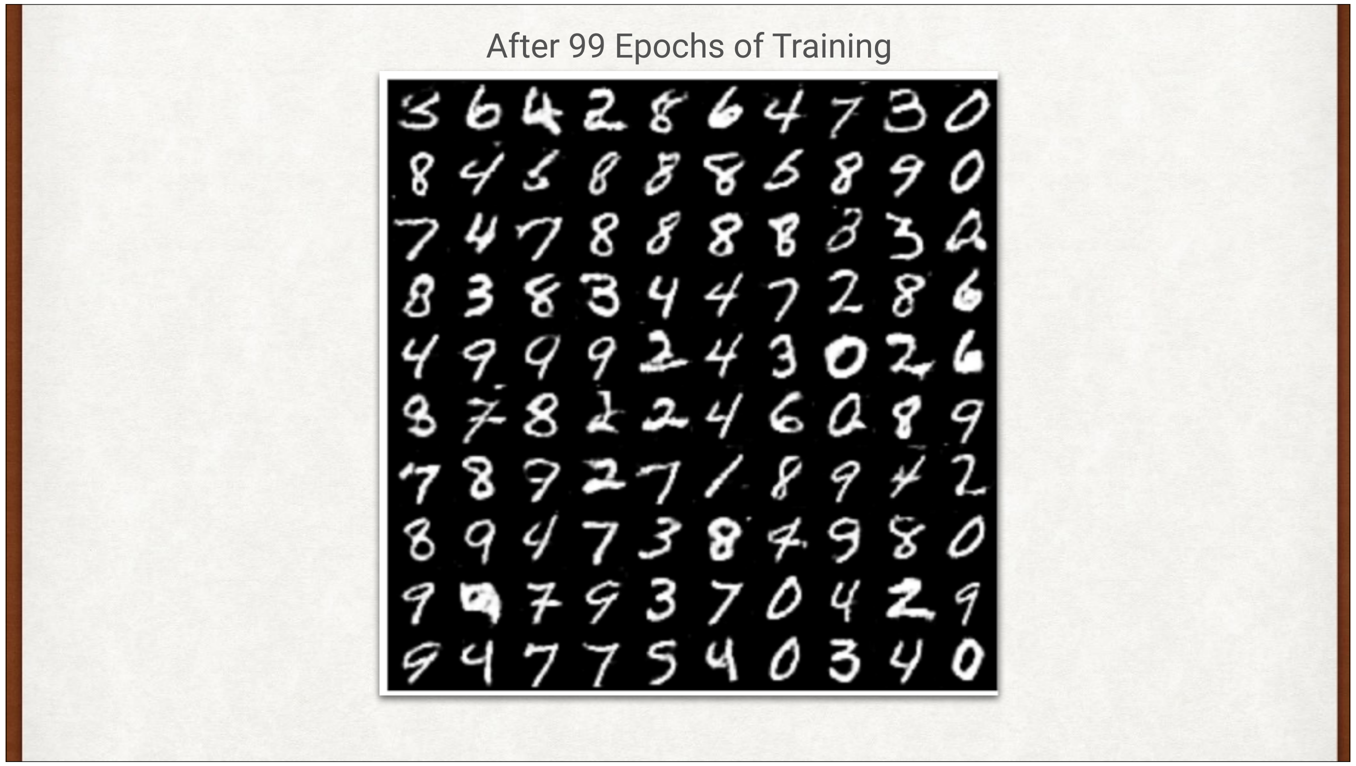


A discriminator and generator for a GAN using MNIST data. The circle with a dot is a “batchnorm” operation, which is a form of regularization. We place this between each neuron’s summation and activation function, so the second fully-connected layer has no AF, then batchnorm, and then a tanh AF.



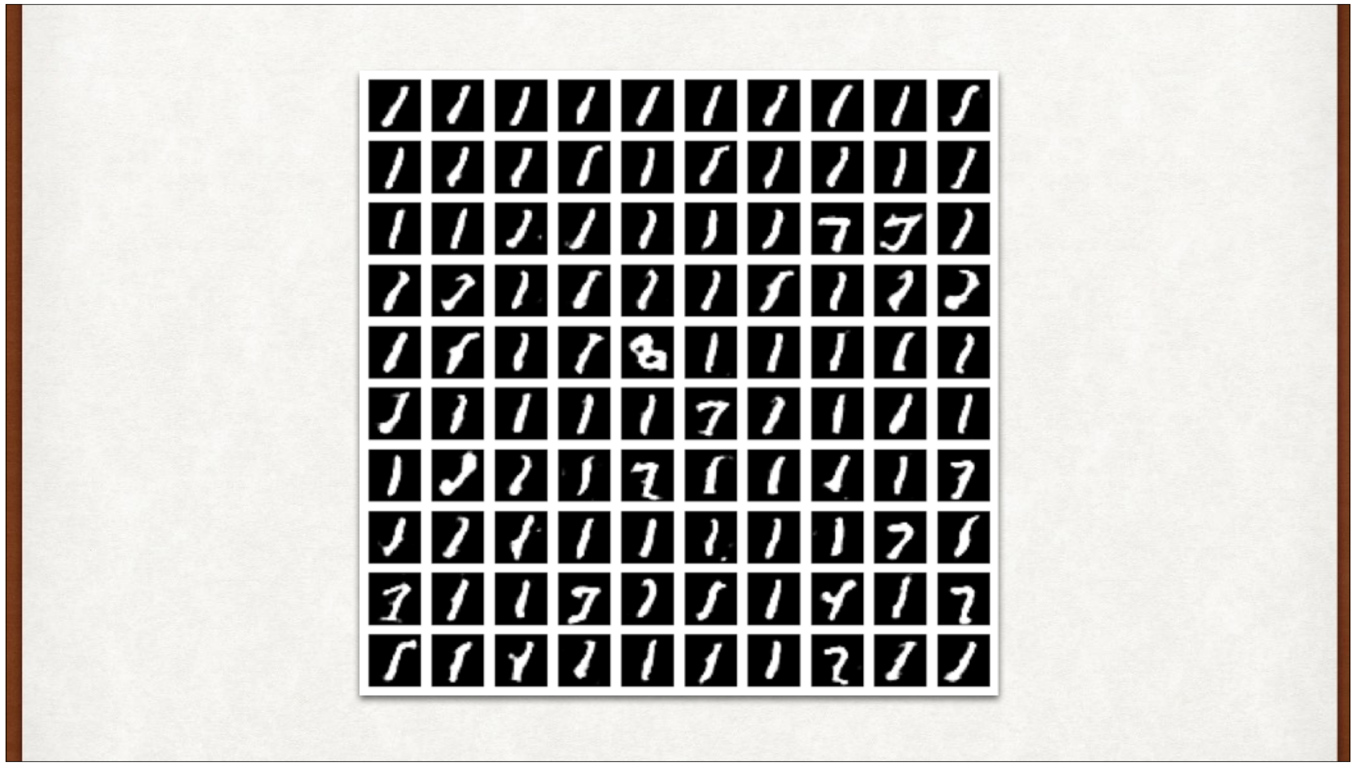
After 1 Epoch of Training

After 1 epoch, we get MNIST-y splotches. This is encouraging!

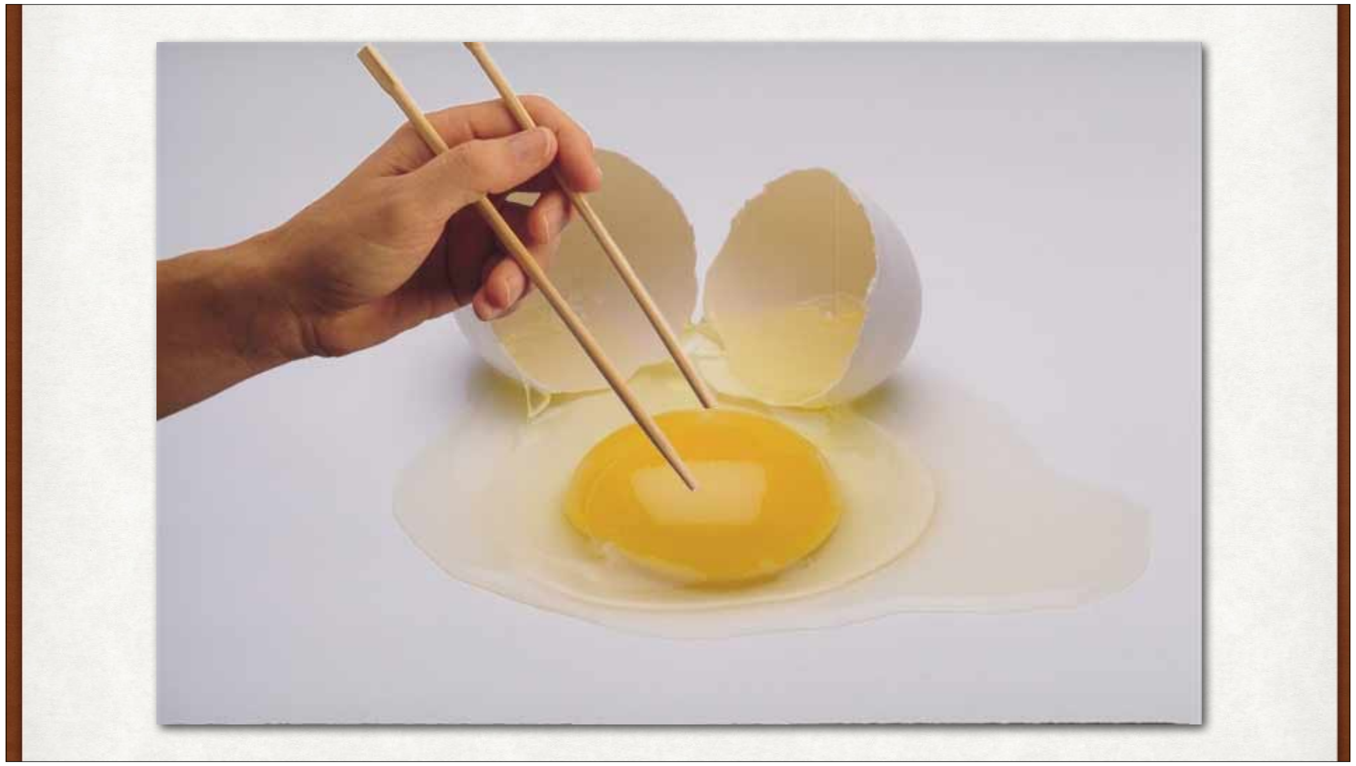


After 99 Epochs of Training

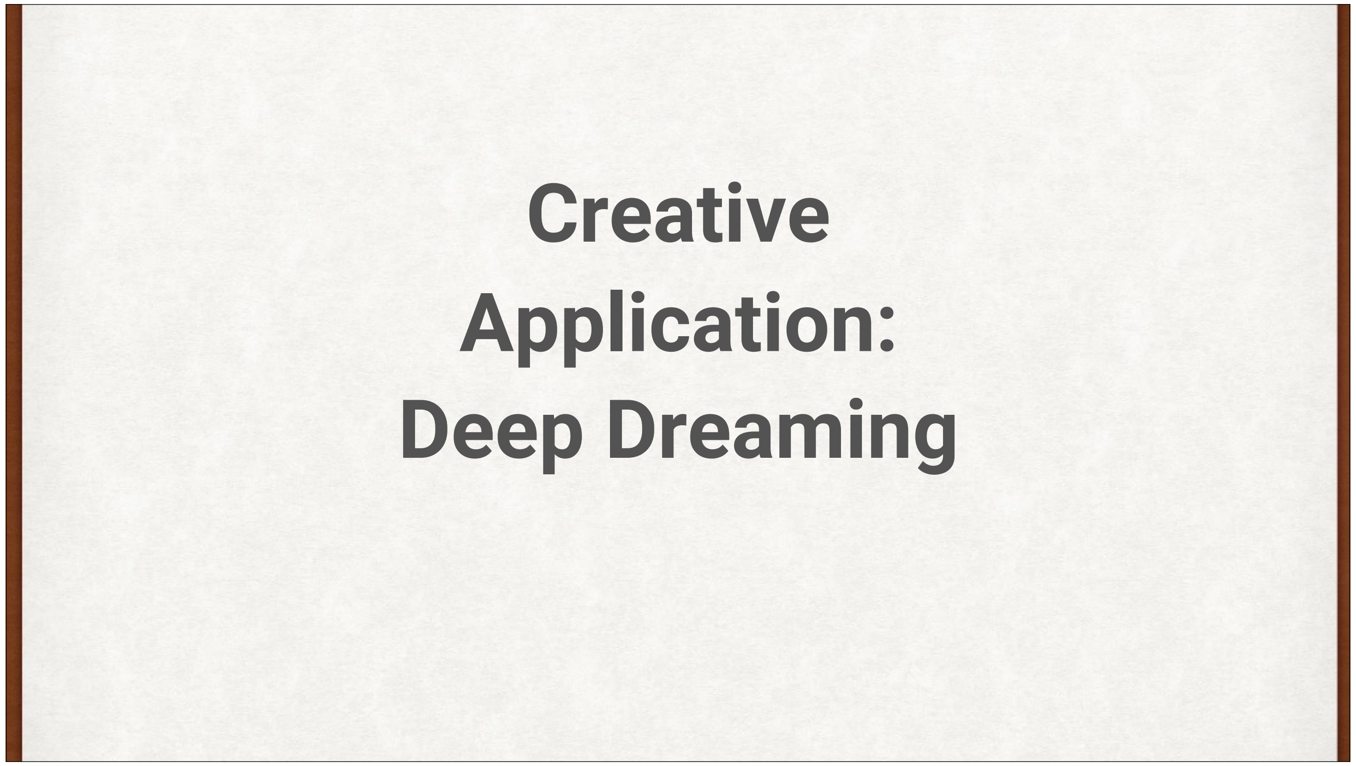
Whoa. Really? Totally synthetic images that started with noise. These were not hand-picked.



Beware modal collapse! Another sneaky way for AI to do what we asked for, not what we want. If this image of a 1 passes the detector, the GAN could just output that every time (or almost every time) and pass the test. This GAN is on its way to doing just that. We need to take extra steps to prevent this shortcut by the generator.

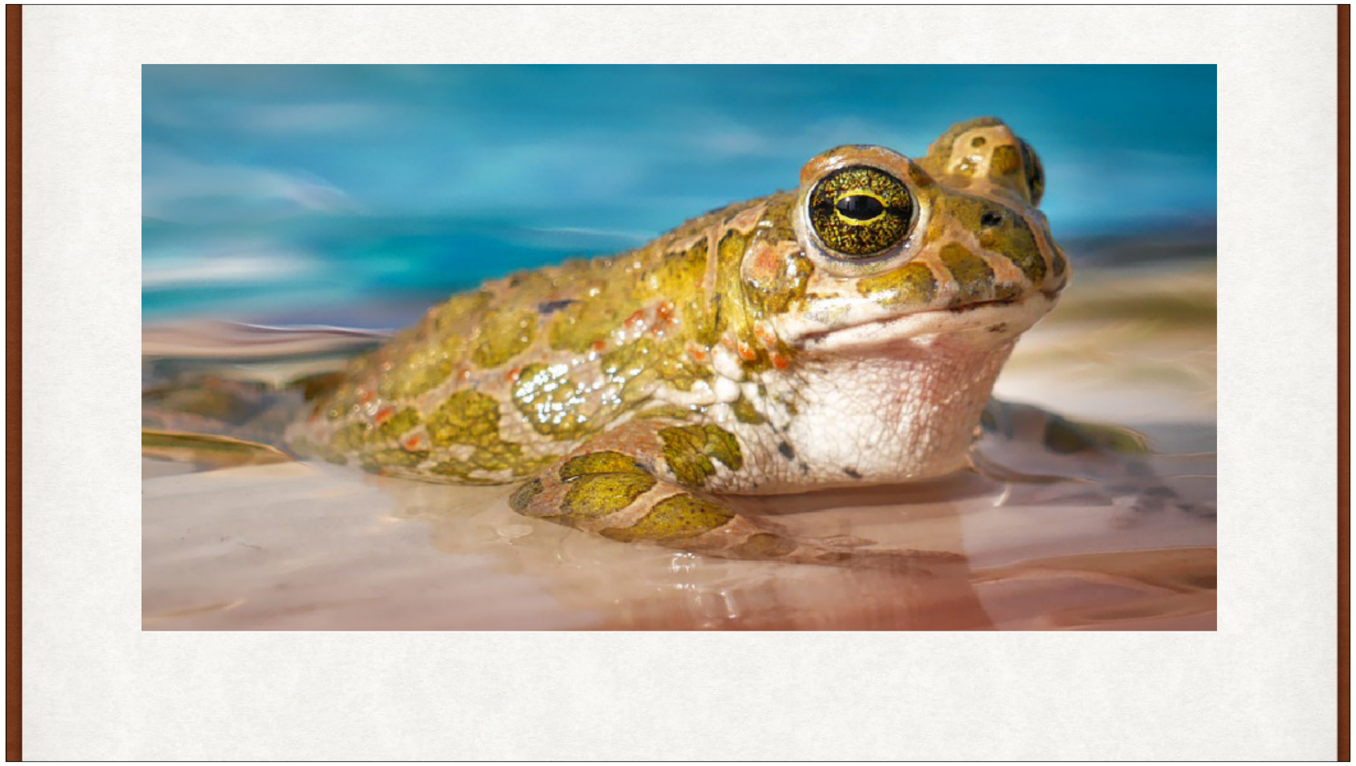


That’s it for the main technical stuff. Let’s see some fun applications.

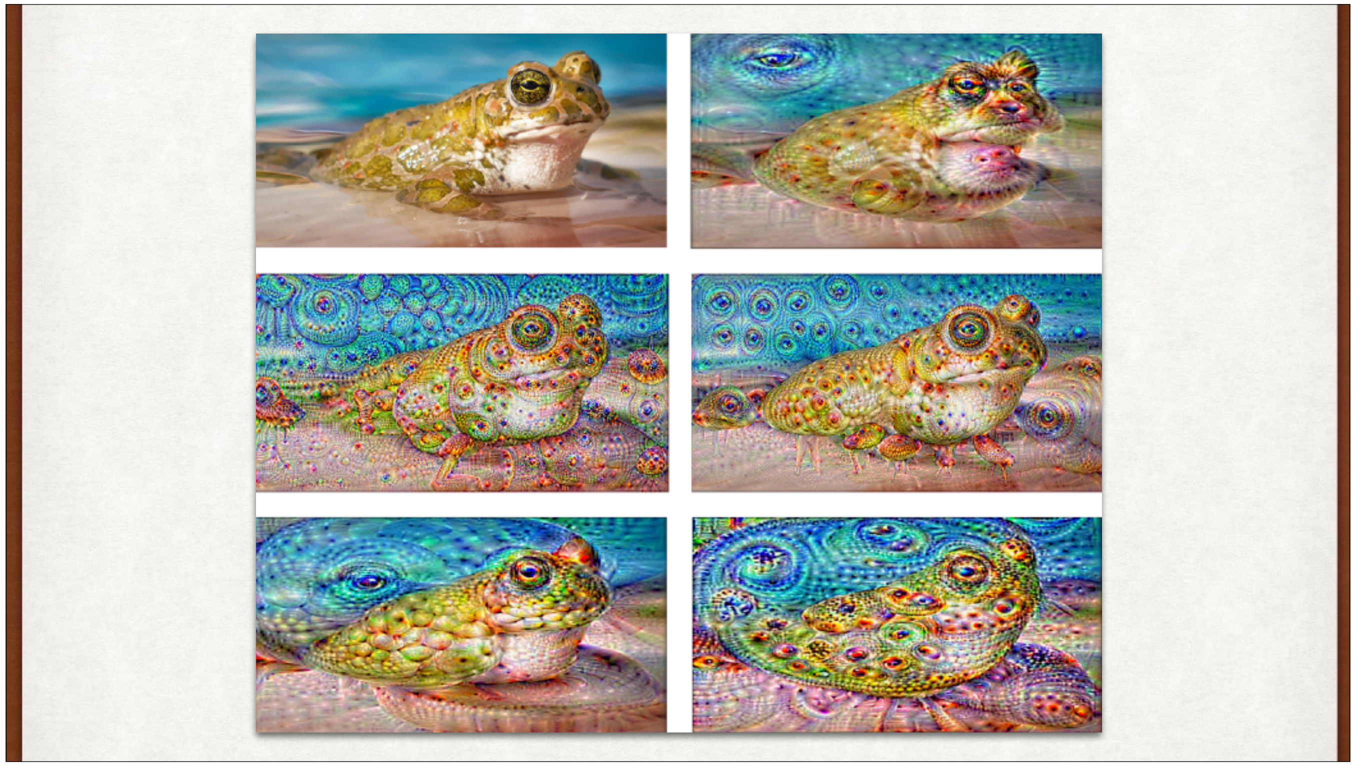


Creative Application: Deep Dreaming

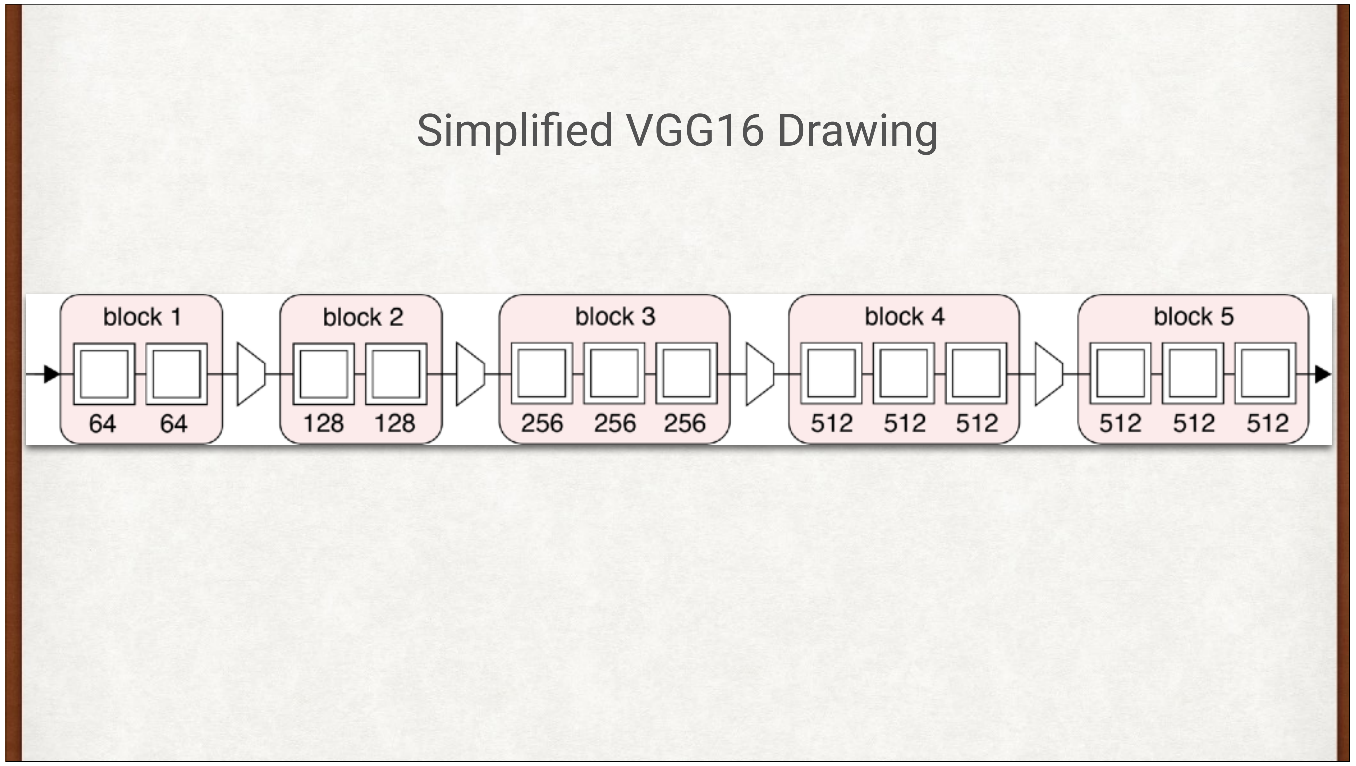
Let’s have fun with a creative application of DL.



A spectacular frog. This frog is 100% frog. This frog is all frogs. Frogs are so cool.

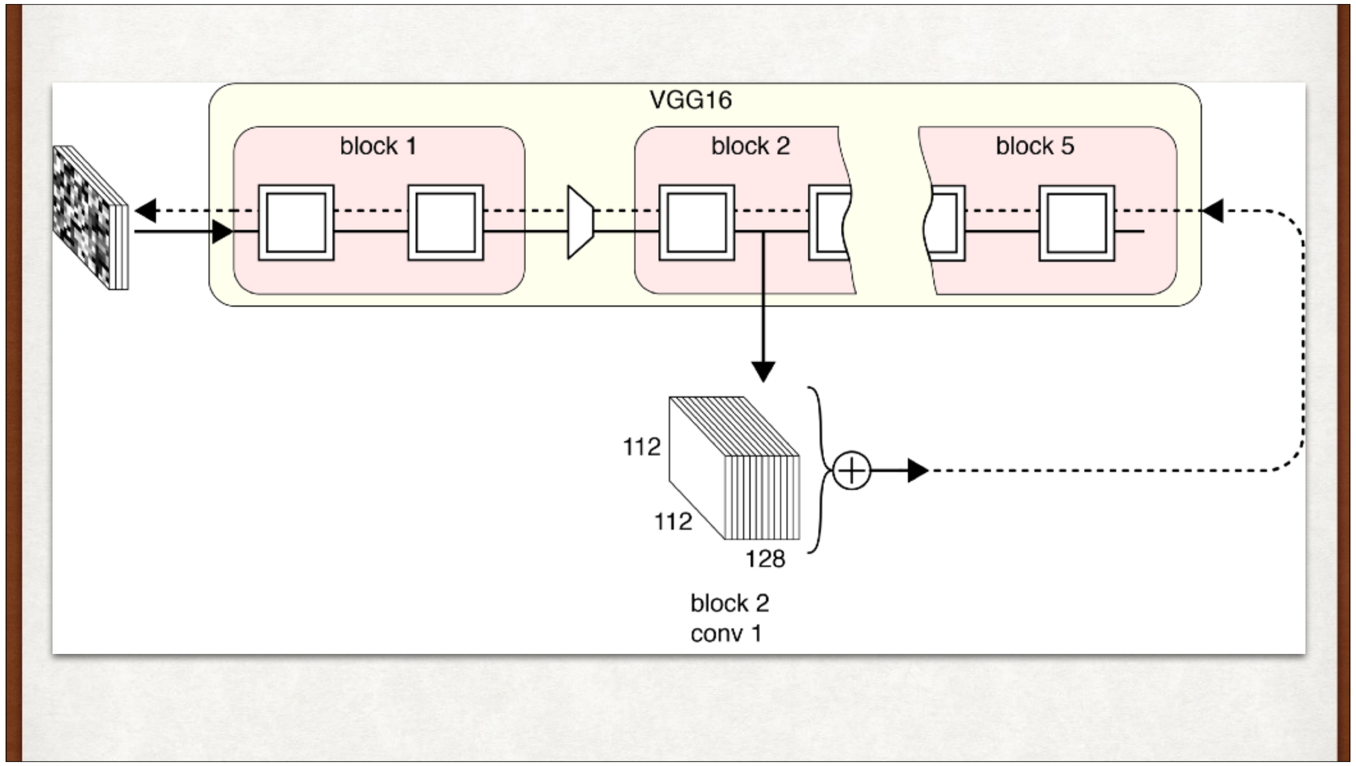


This is where we’re going: deep dreaming from the frog.

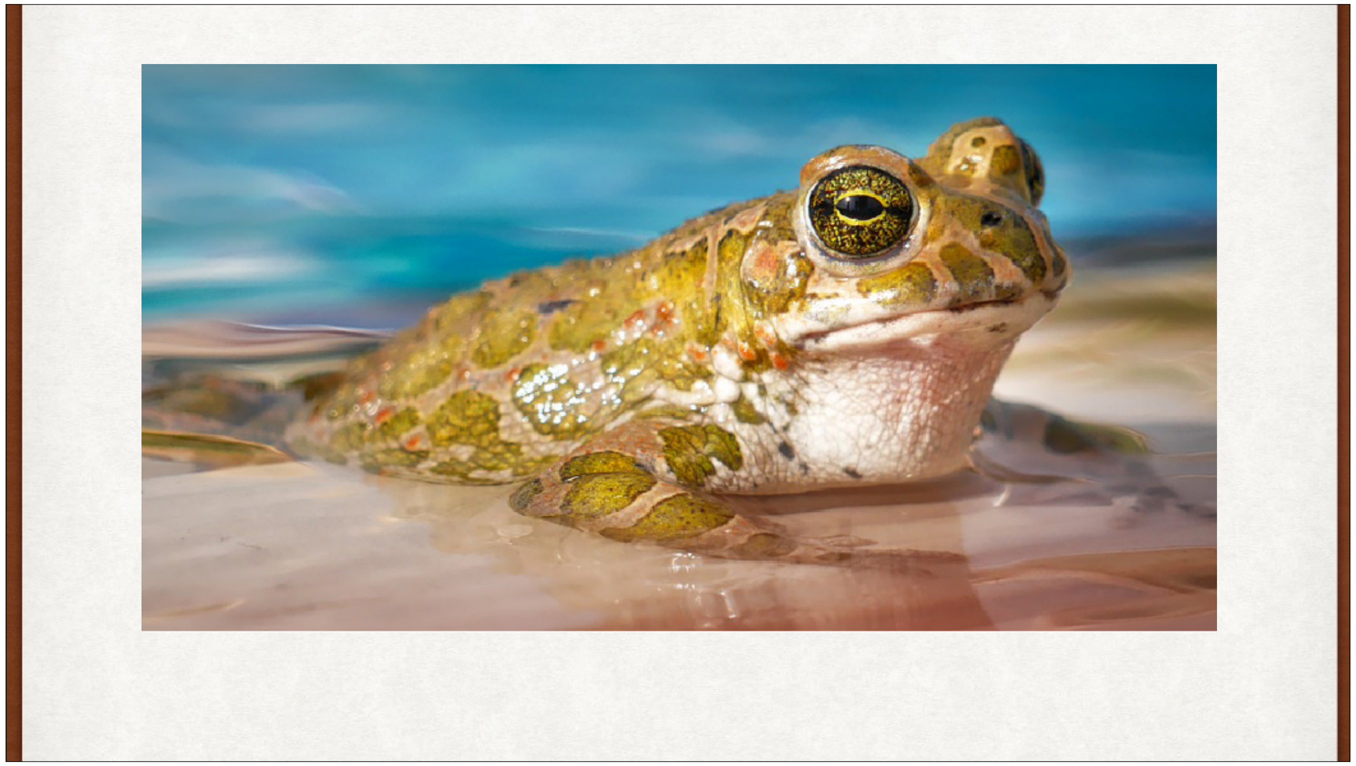


Simplified VGG16 Drawing

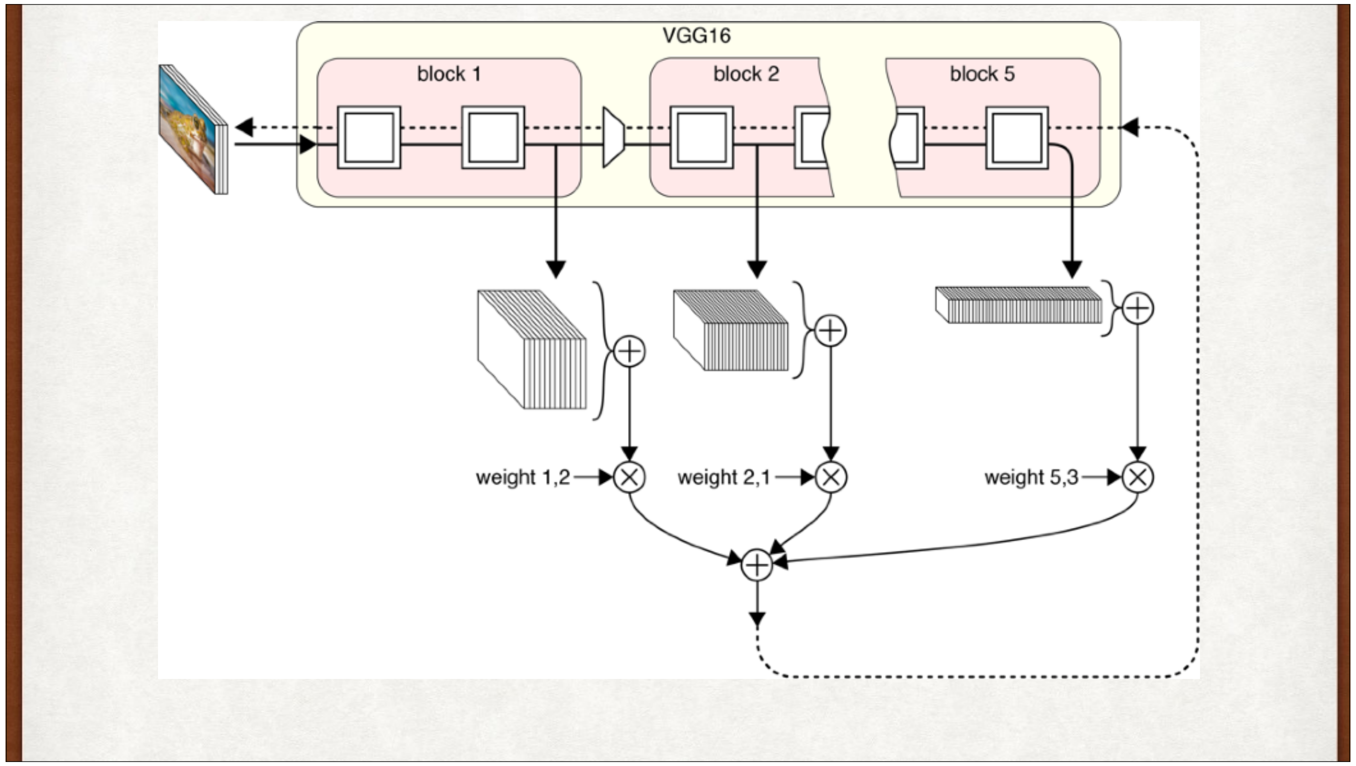
We’ll simplify the drawing of VGG16 (but not the network!) to show just the convolution and downsampling layers.



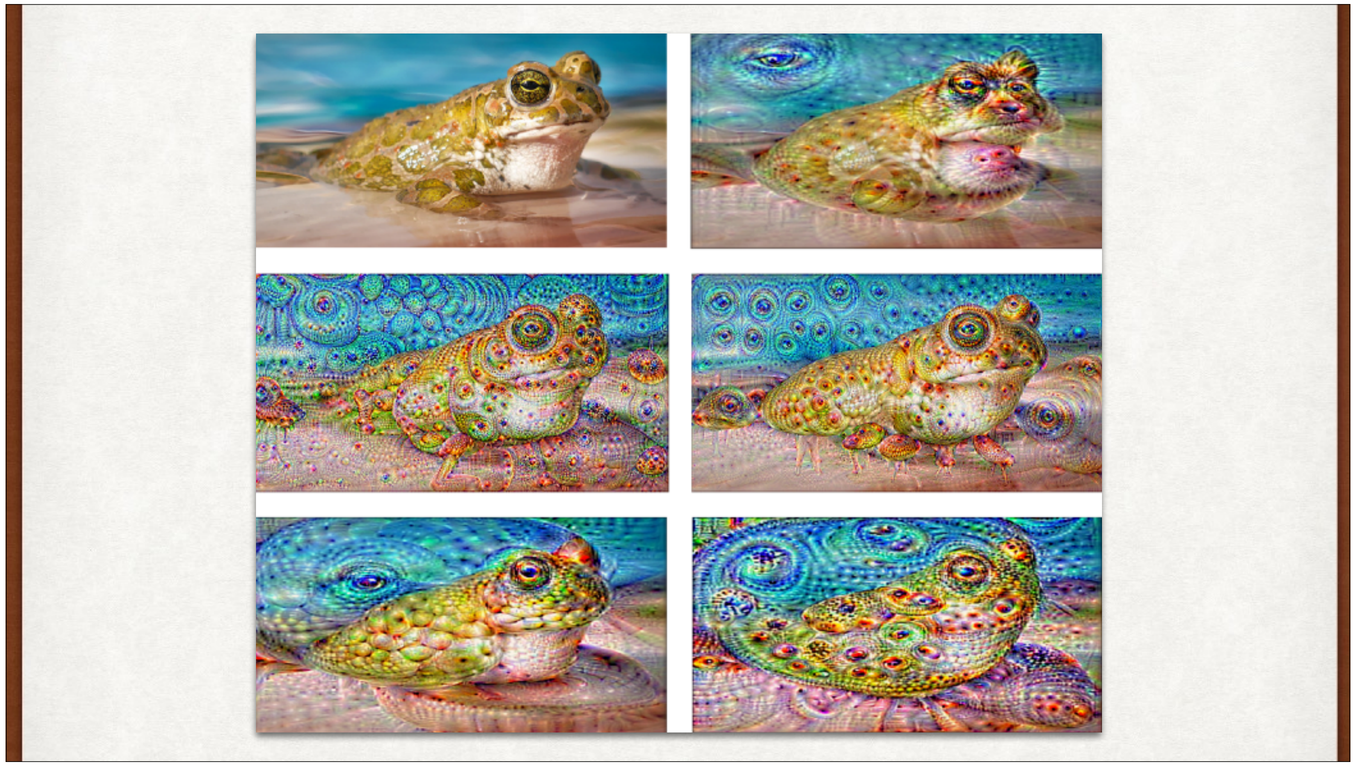
Let’s sum up the responses of all the filters after a layer, and use that as our error. Again, the network doesn’t change in any way. But we do modify the pixel values in the noise image so that they will cause a bigger stimulation to the outputs of this layer.



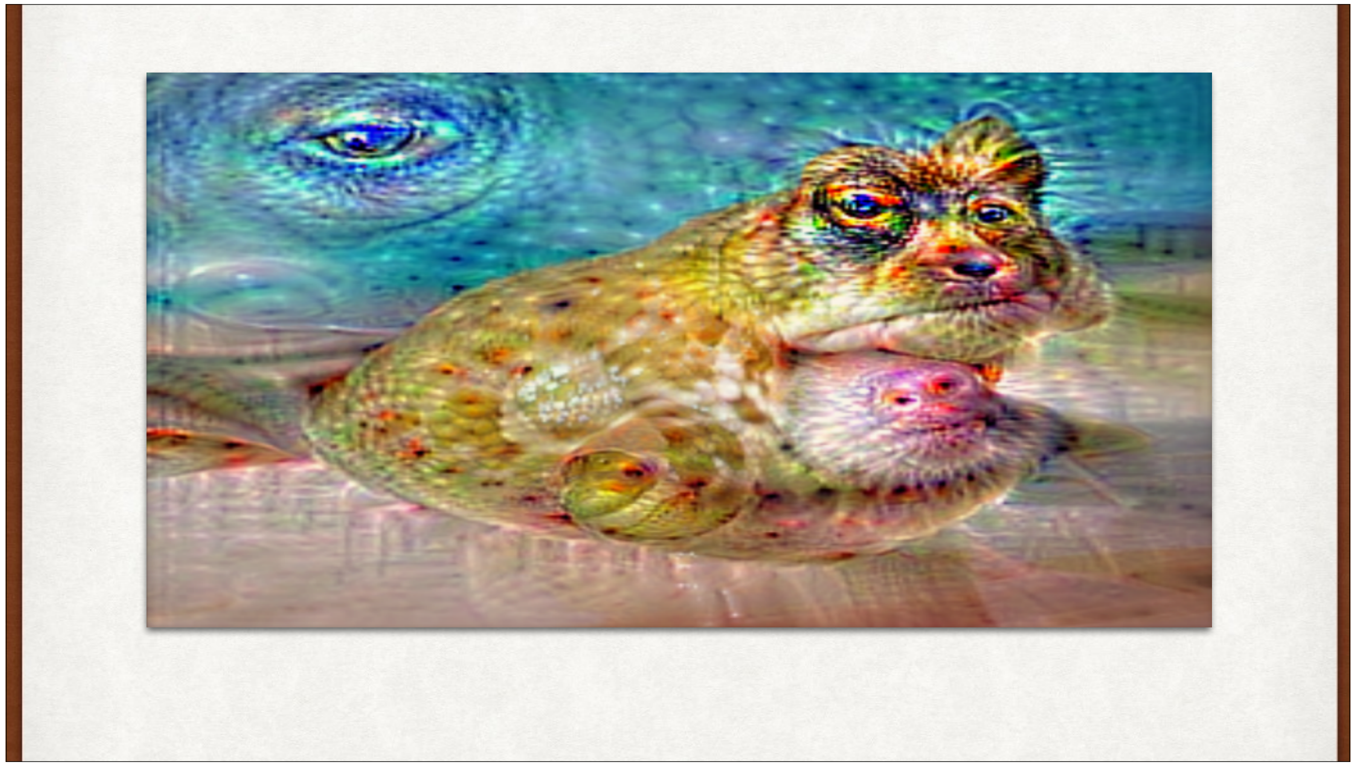
A spectacular frog. This frog is 100% frog. This frog is all frogs. Frogs are so cool.
https://pixabay.com/photos/waters-nature-frog-animal-swim-3038803/



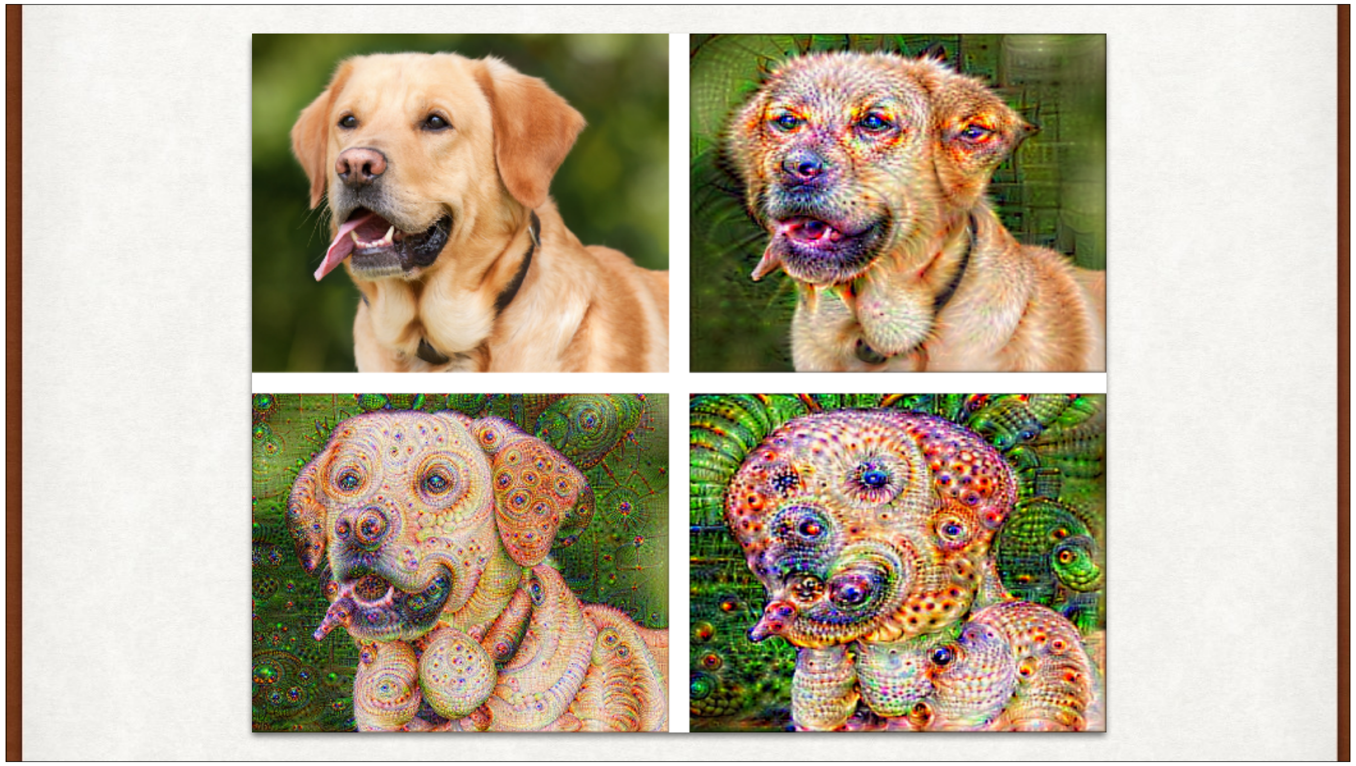
We’ll feed the frog to VGG16, instead of noise, and get the summed responses of different layers. We’ll scale those sums, add them together, that’s our new “error”. We’ll then modify the pixel values in the frog image to drive up this “error”. We’ll run the modified frog through the network again, compute this “error”, use that to modify the pixels, and so on. Every time around the loop, the input image changes a little bit to better stimulate the filters.



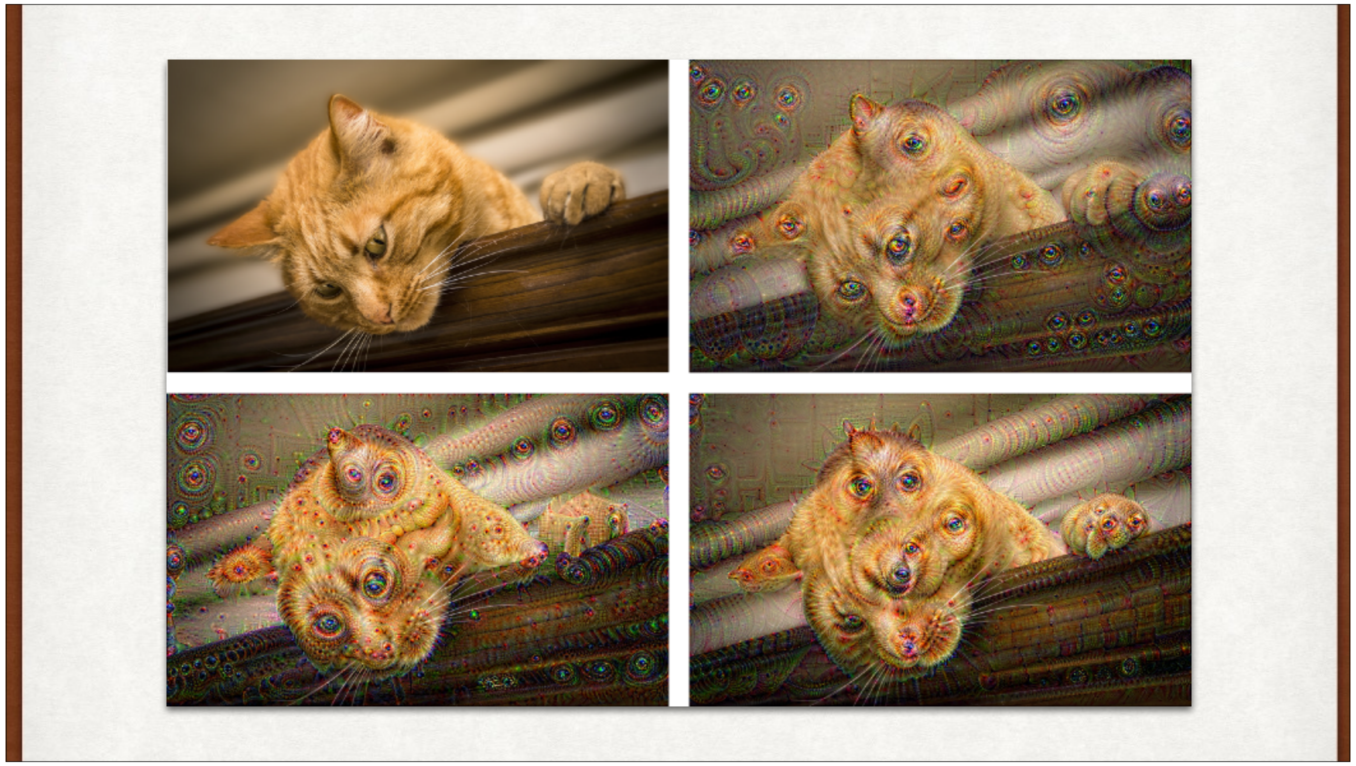
Obviously (hah!) this is what we get back. Originally called inceptionism, it’s usually called deep dreaming. The filters are responding to little variations in the input, and the loop we’ve built causes those variations to get amplified and enhanced, causing the filters to respond more strongly, causing those changes to get further amplified. Different images come from using different choices of filters and layers, weights, and times through the loop.



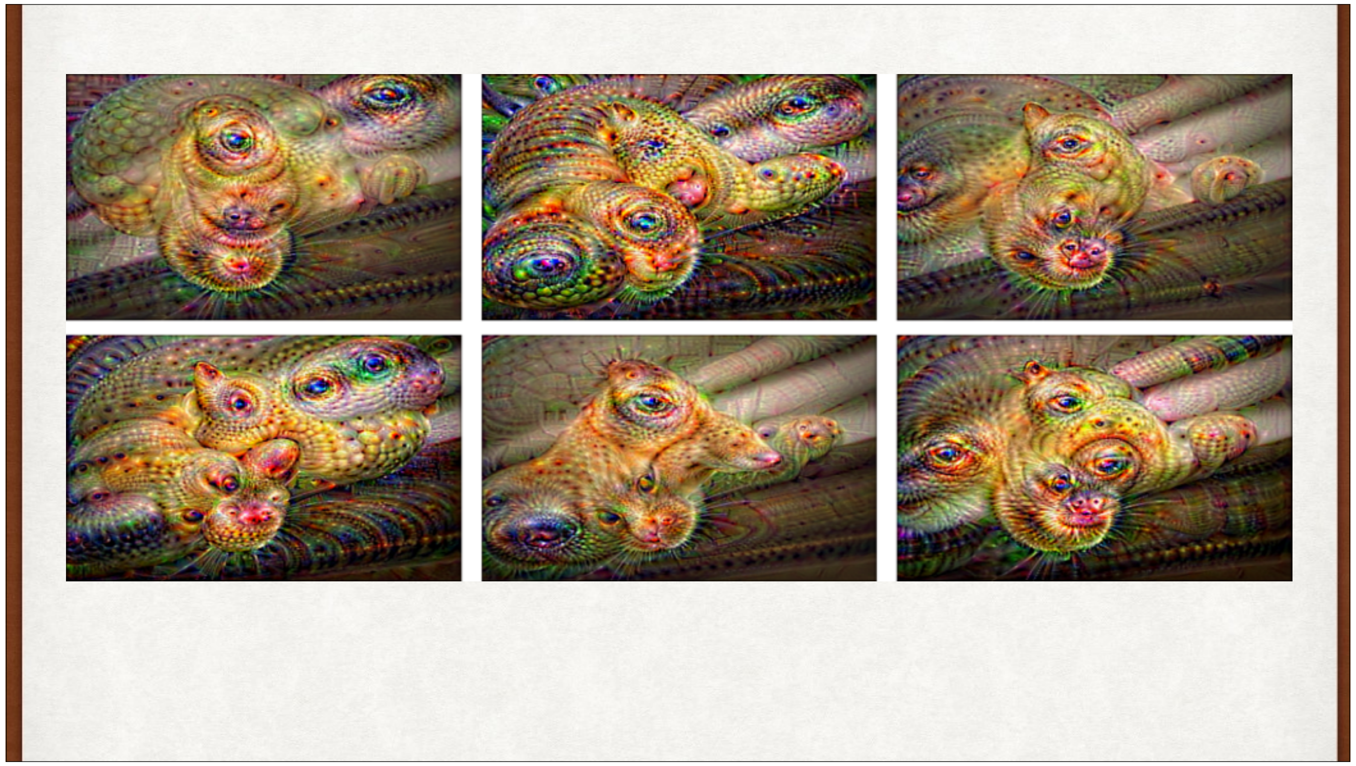
Obviously (hah!) this is what we get back. Originally called inceptionism, it’s usually called deep dreaming. The filters are responding to little variations in the input, and the loop we’ve built causes those variations to get amplified and enhanced, causing the filters to respond more strongly, causing those changes to get further amplified. Different images come from using different choices of filters and layers, weights, and times through the loop.



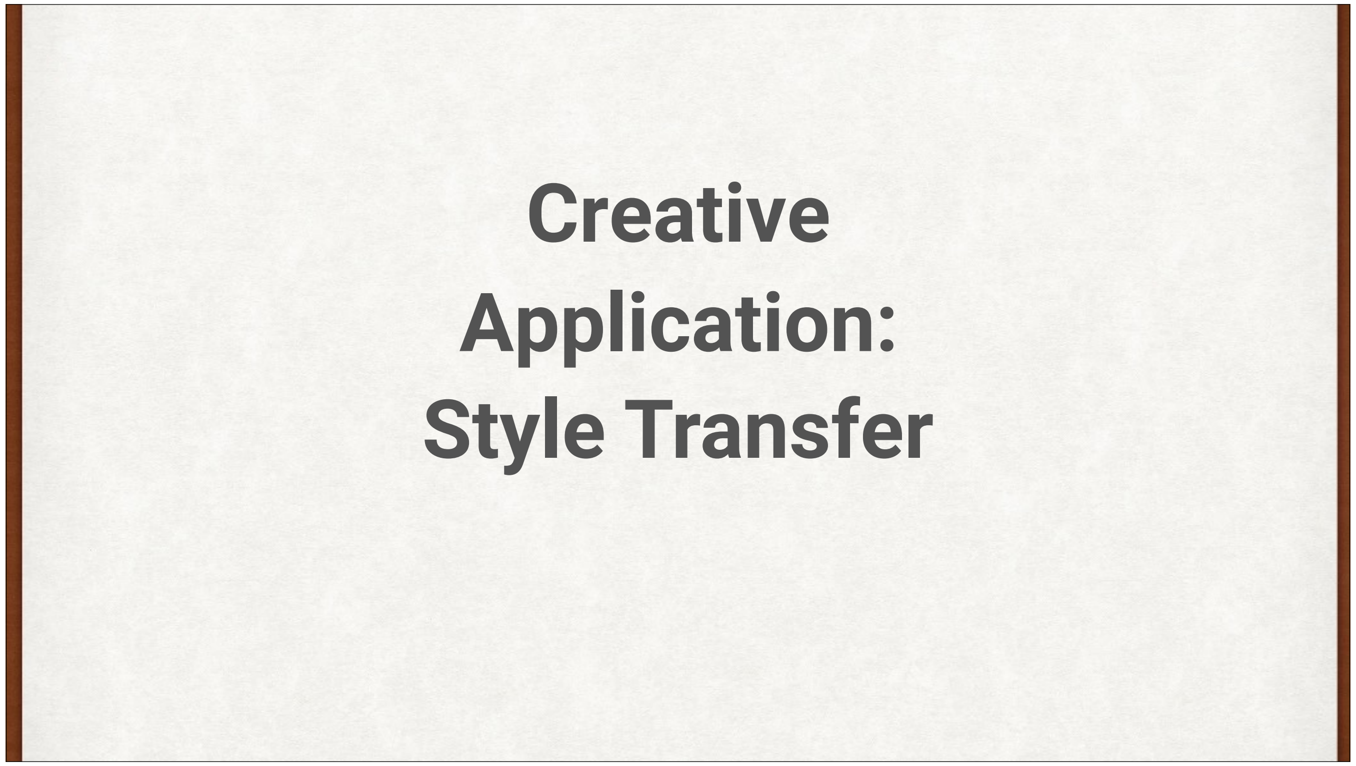
Deep dreaming on a friendly dog.
https://pixabay.com/photos/dog-labrador-light-brown-pet-1210559/



Deep dreaming on a picture of a cat.
https://pixabay.com/photos/cat-feline-animal-pet-puppy-hair-2184682/

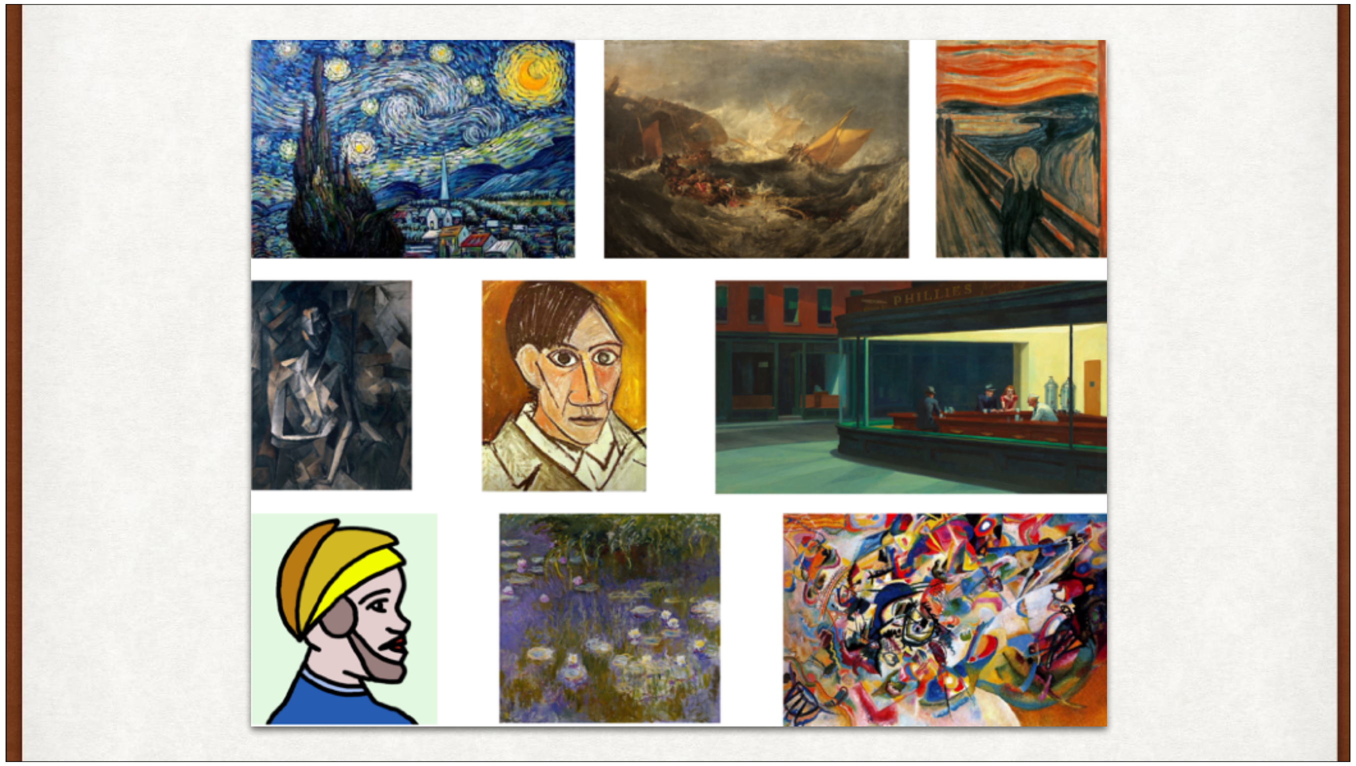


Using some other layers and weights to make other cat dreams. Different networks will give us different dreams, because their filters look for different kinds of image features. VGG16 is big on eyes, so we see lots of them.



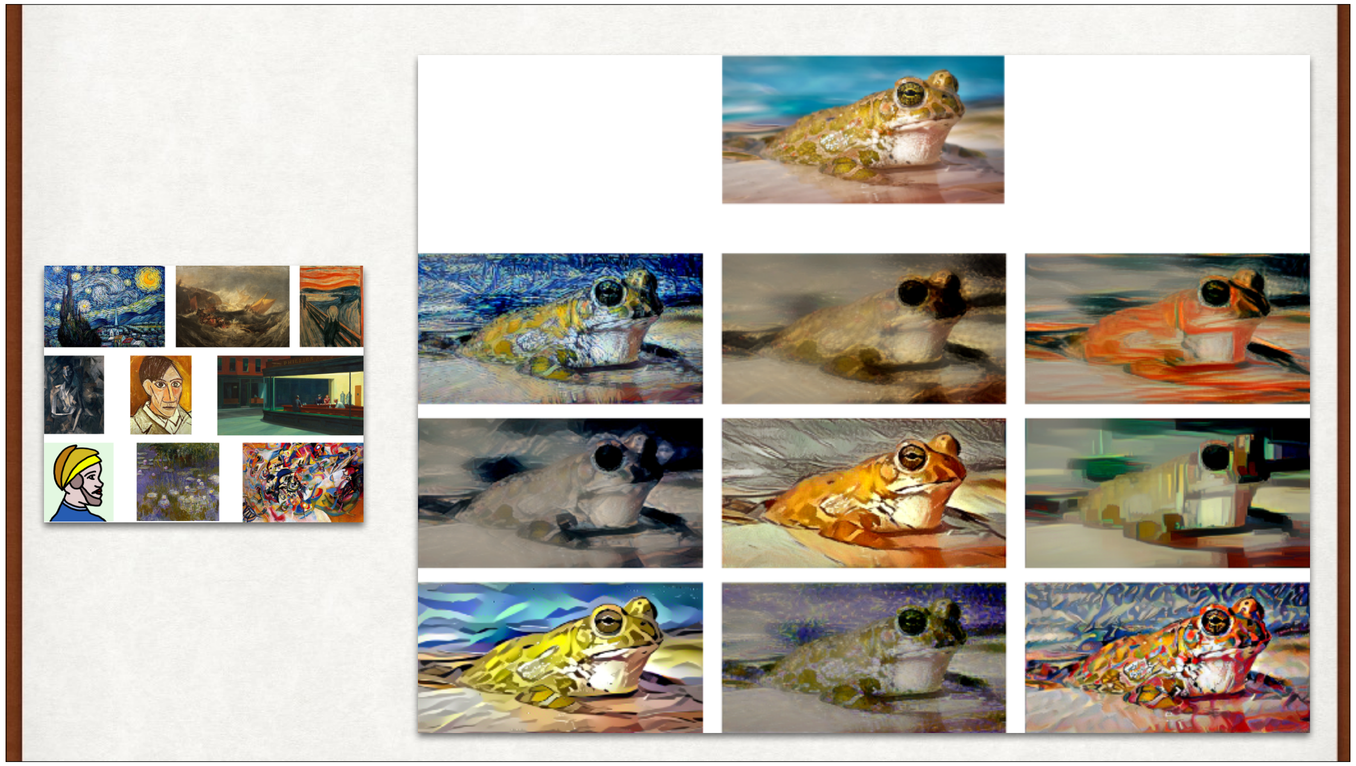
Creative Application: Style Transfer

Style transfer means changing an image to look like one created by someone with a recognizable style (we use the word “style” in its popular sense, and don’t try to nail it down more specifically).

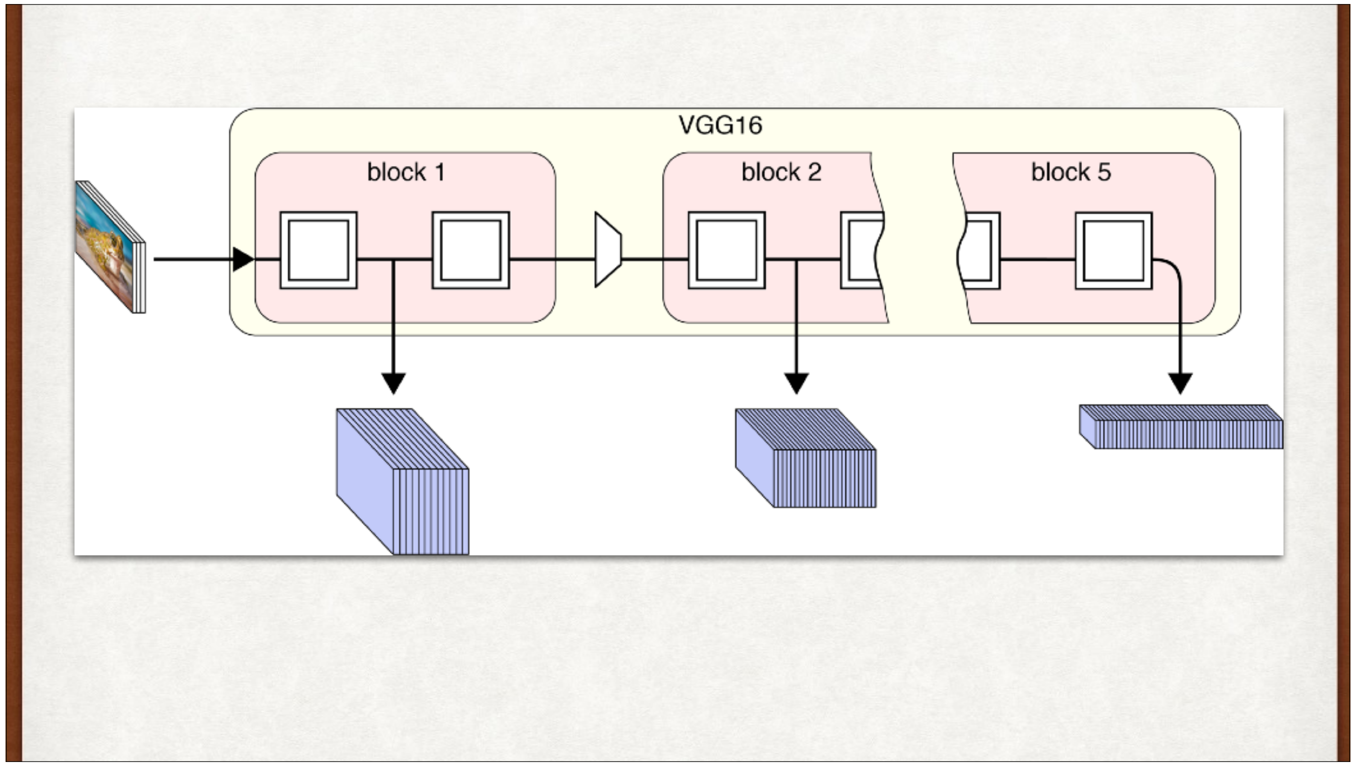


Nine images in different styles. We’ll apply these to the frog.

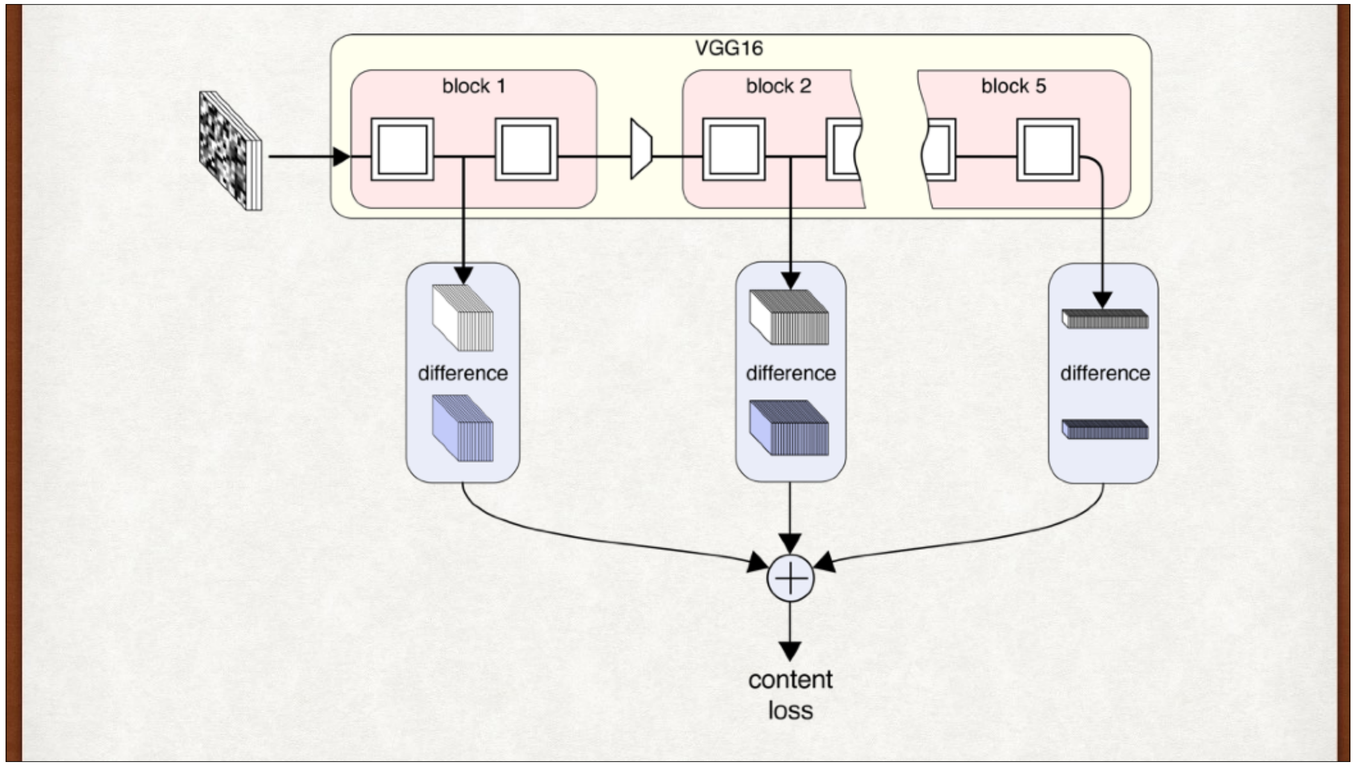
https://www.wikiart.org/en/wassily-kandinsky/composition-vii-1913
https://commons.wikimedia.org/wiki/File:Nighthawks_by_Edward_ Hopper_1942.jpg
https://www.wikiart.org/en/pablo-picasso/self-portrait-1907
https://www.wikiart.org/en/pablo-picasso/seated-female-nude-1910
https://www.wikiart.org/en/edvard-munch/the-scream-1893
https://commons.wikimedia.org/wiki/File:Shipwreck_of_the_ Minotaur_William_Turner.jpg
https://commons.wikimedia.org/wiki/File:VanGogh-starry_night_ ballance1.jpg
https://www.wikiart.org/en/claude-monet/ water-lilies-yellow-and-lilac-1917



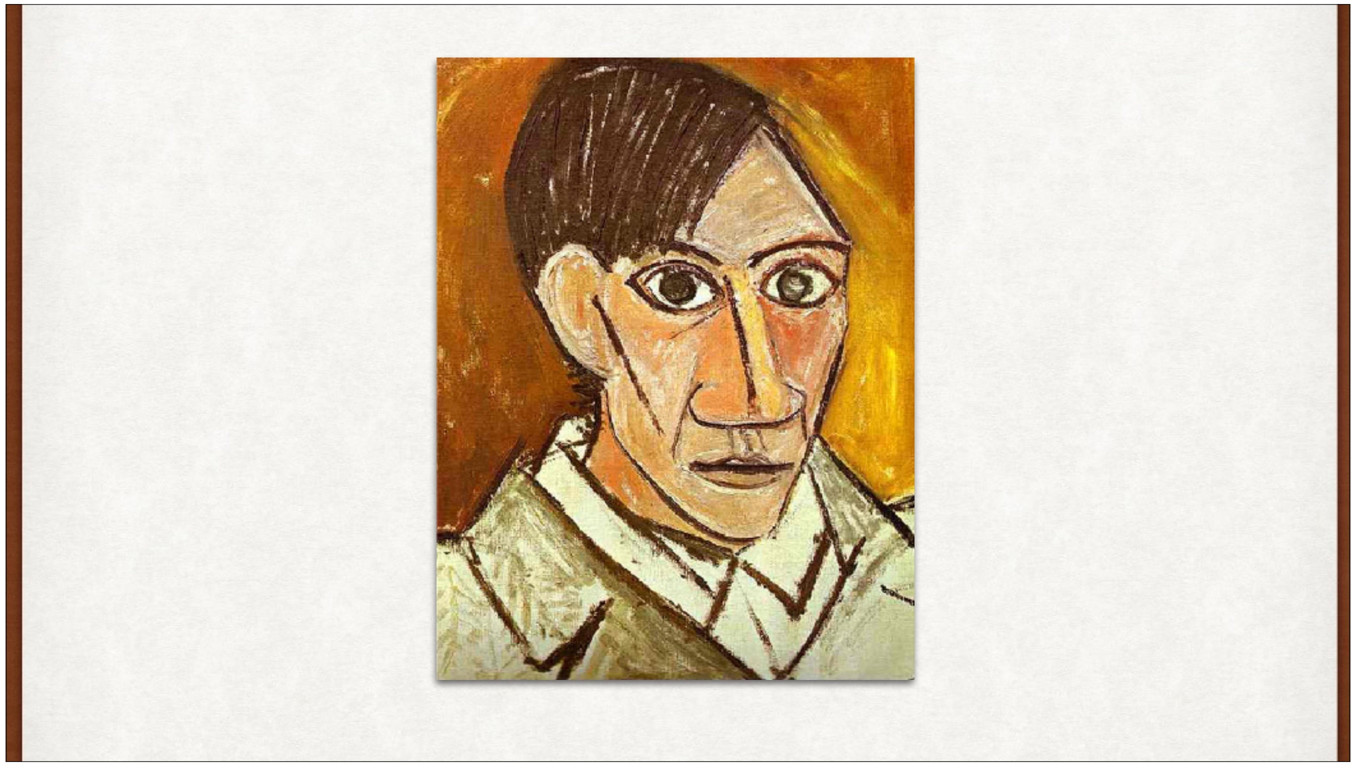
Wow. How do we do this?



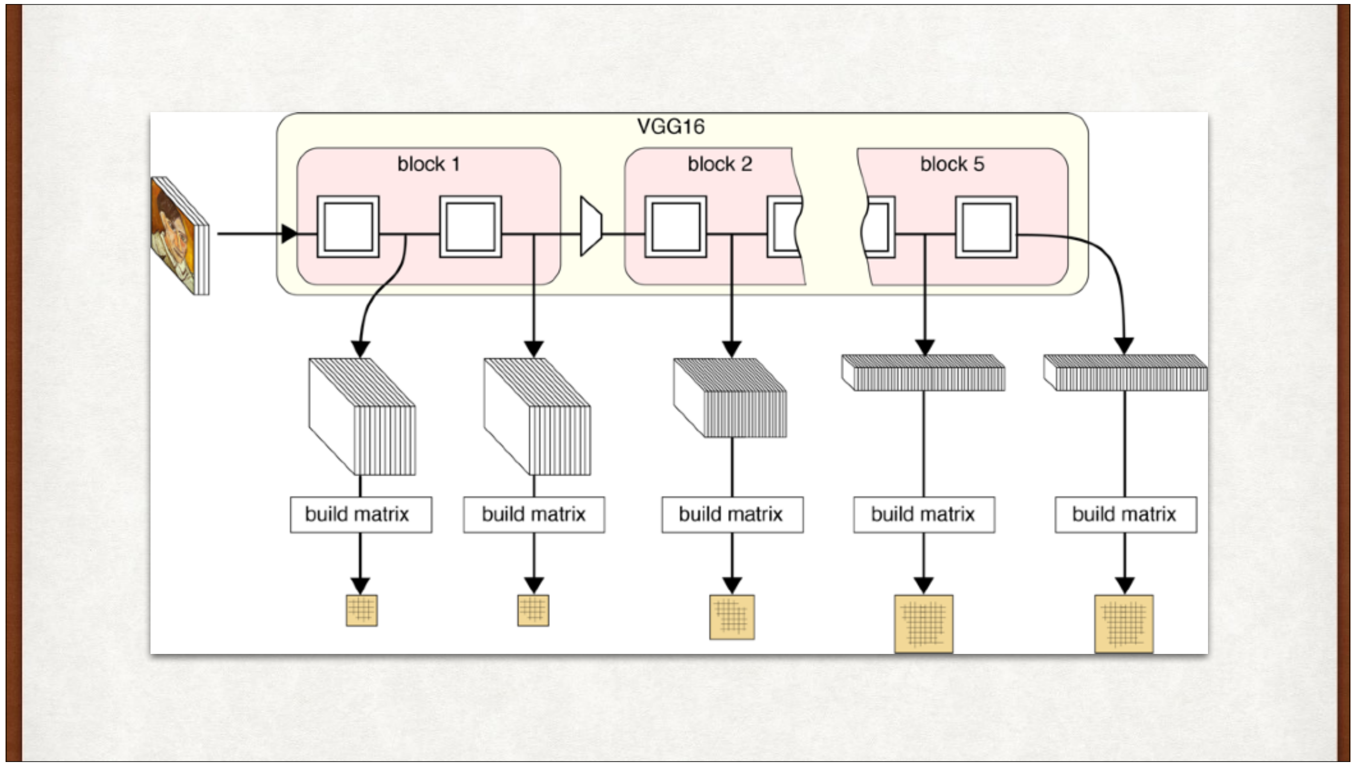
We’ll come back to the Gram matrices. For now, let’s feed the frog into VGG16, and save the outputs of each layer.



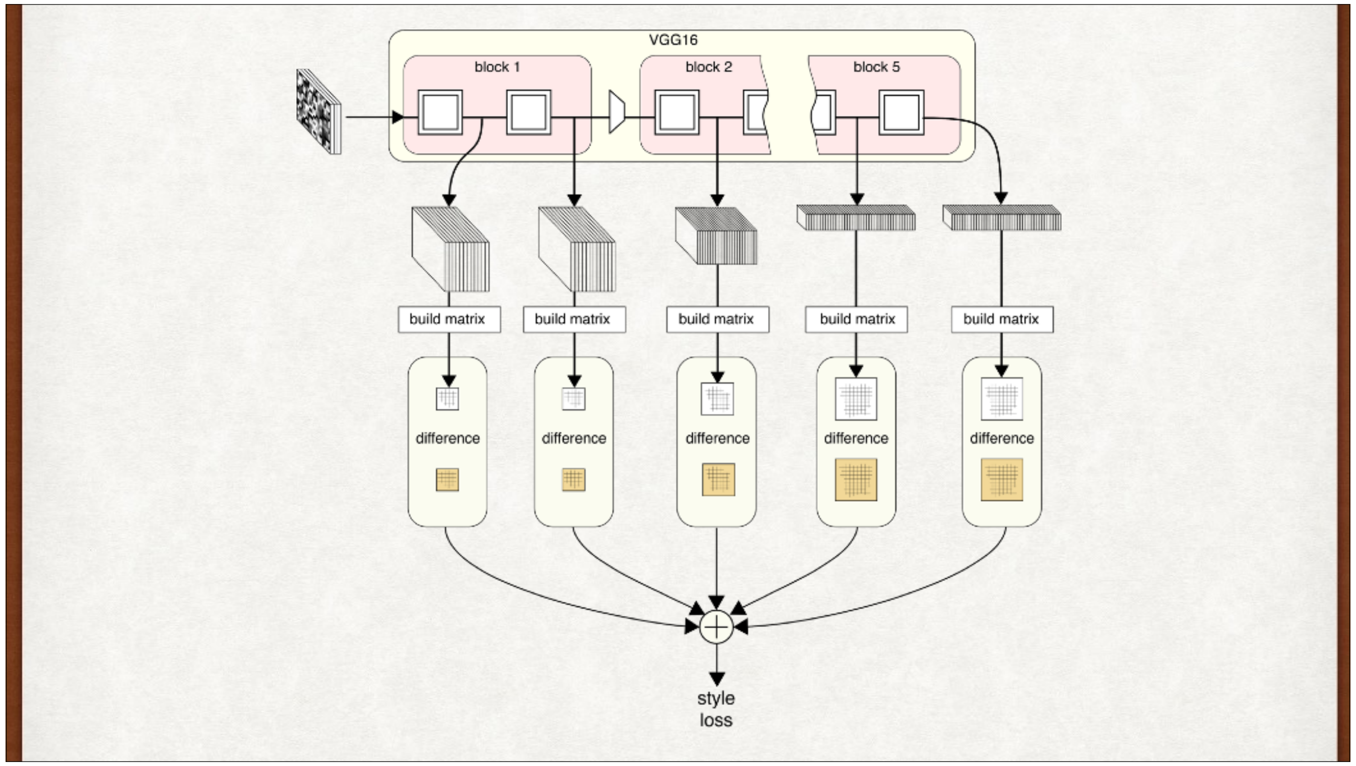
We’ll run noise into VGG16, and compare the total output of each layer with what we saved for the frog. Adding those all together gives us the “content loss,” or how different the noise is from the original picture of the frog.



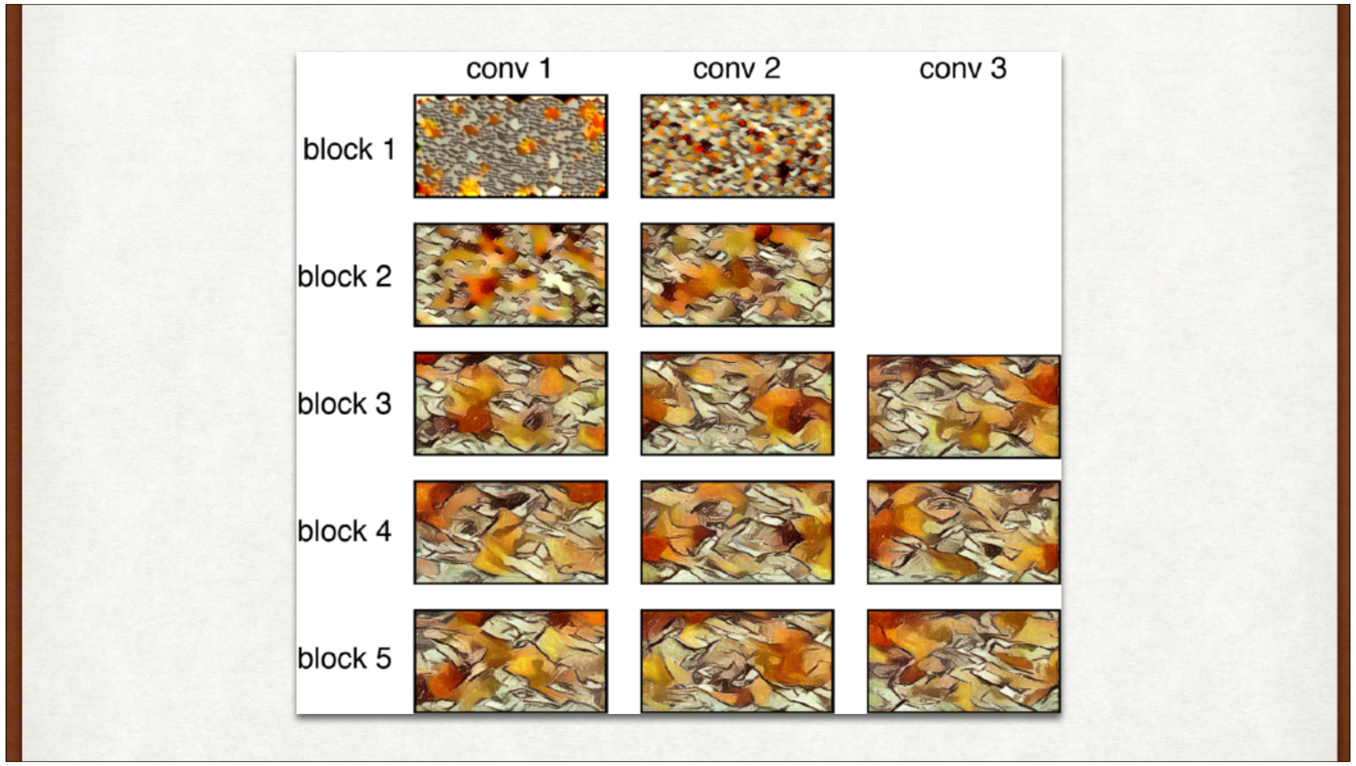
To do this, let’s pick this very stylized 1907 self-portrait by Pablo Picasso. People sure looked weird back then.



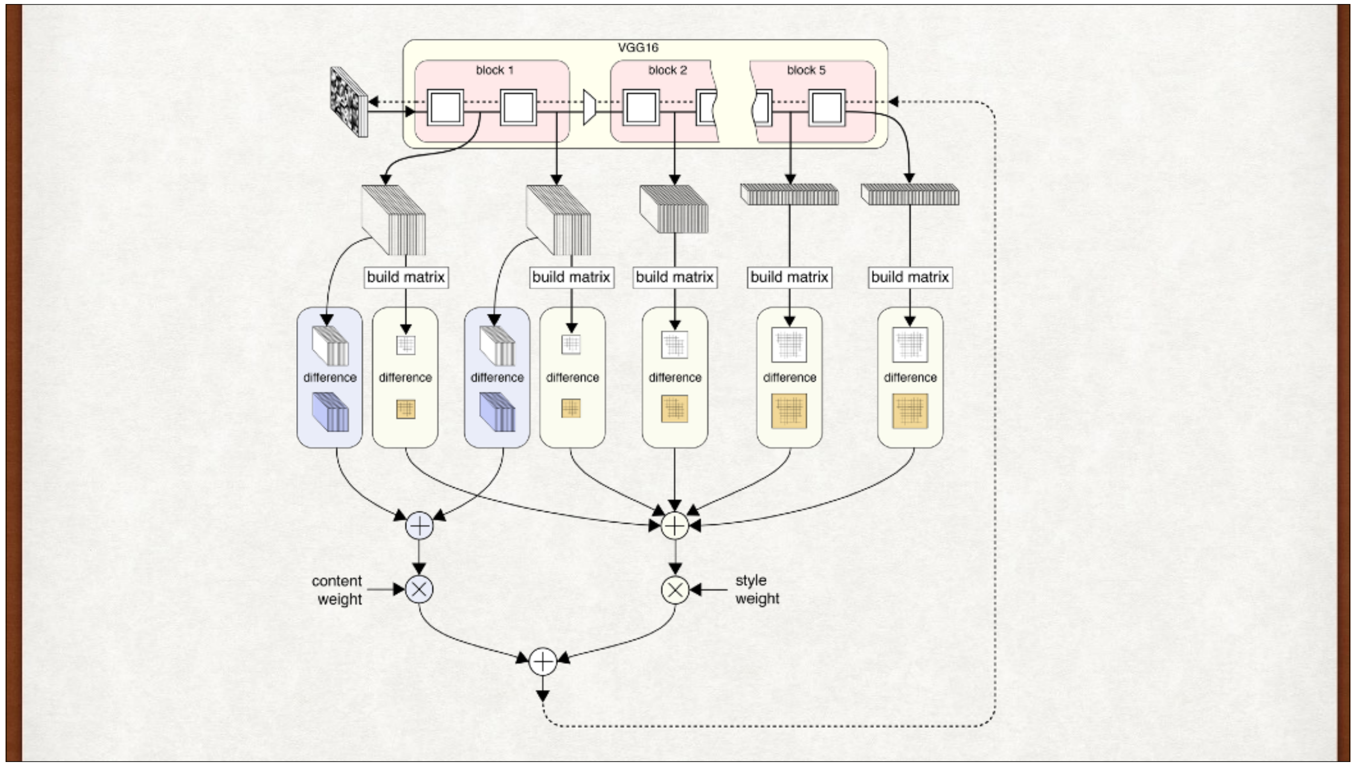
Run the painting through VGG16, get the output from each layer, and build the Gram matrix telling us where different filters activated in the same place. Save those matrices. Remember, these matrices tell us to what degree each pair of filters responded strongly in the same locations of their input.



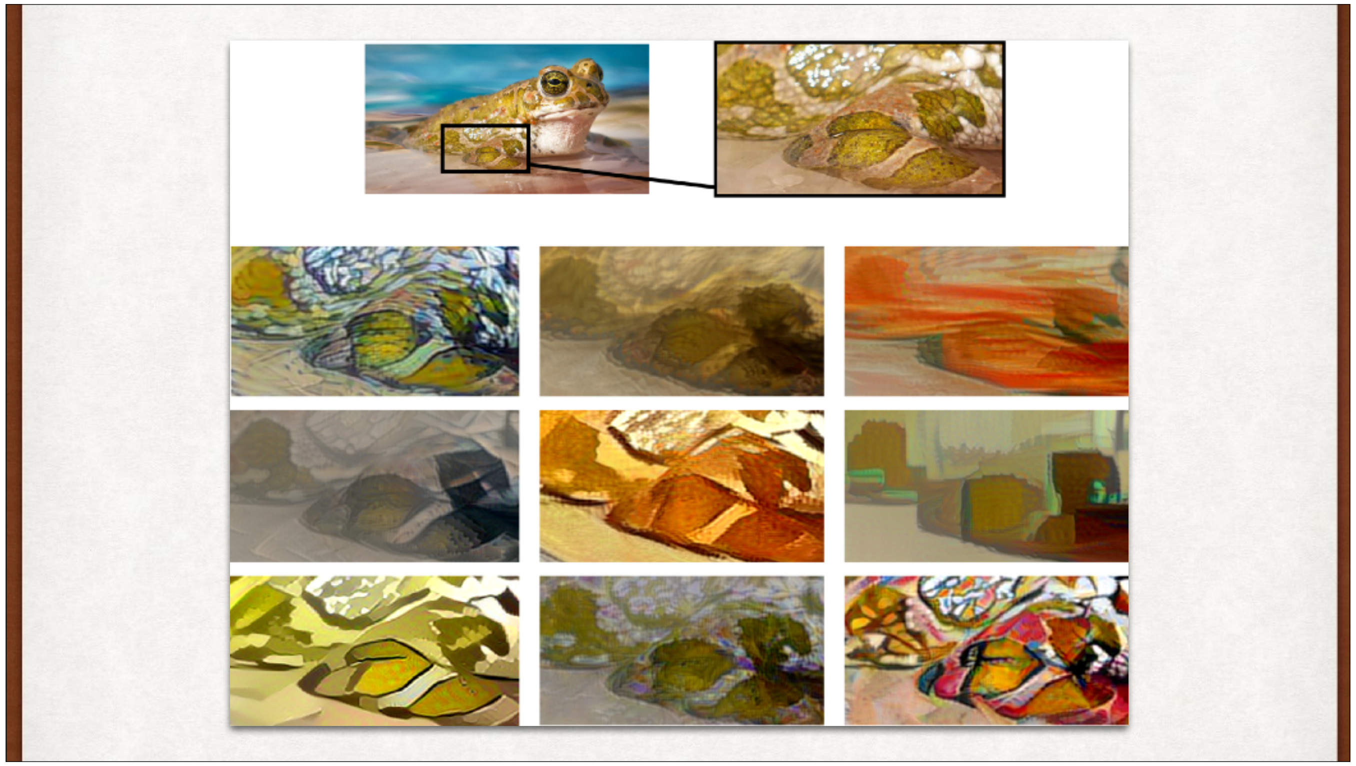
Now that we have the painting's Gram matrices saved, run noise through VGG16, build its Gram matrices, and find out how much they differ from the ones we saved. Add that all up to get the "Gram matrix loss." We'll see that this captures the "style" information we're looking for.



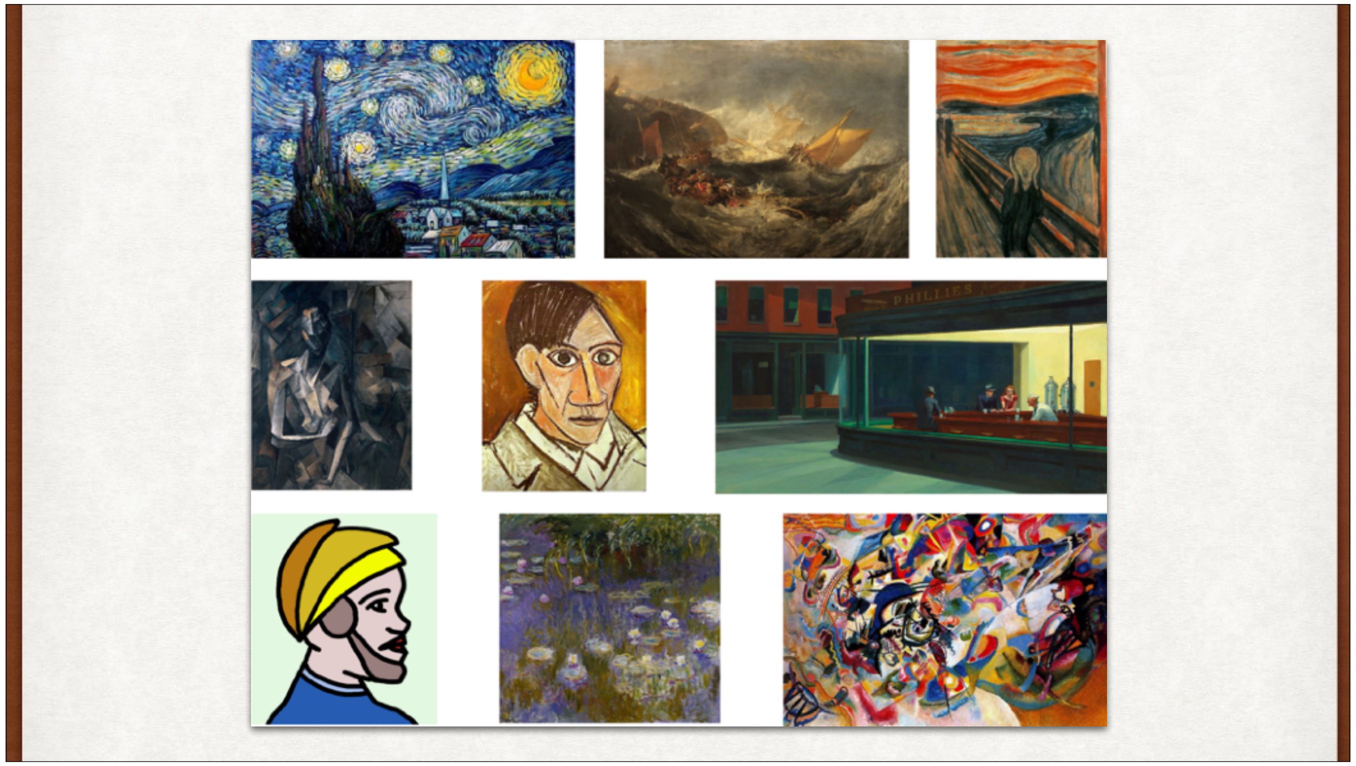
Let’s instead use the sum of all the outputs from the start up to and including a given layer. Starting with noise, these outputs are looking a lot like the style used in the self portrait, even though there’s no content information at all. They have the right colors, they’re grouped into coherent shapes with black outlines, and they definitely have the same “feeling” as the input. Just from modifying noise to have the same Gram matrix outputs as the painting! It feels like magic.



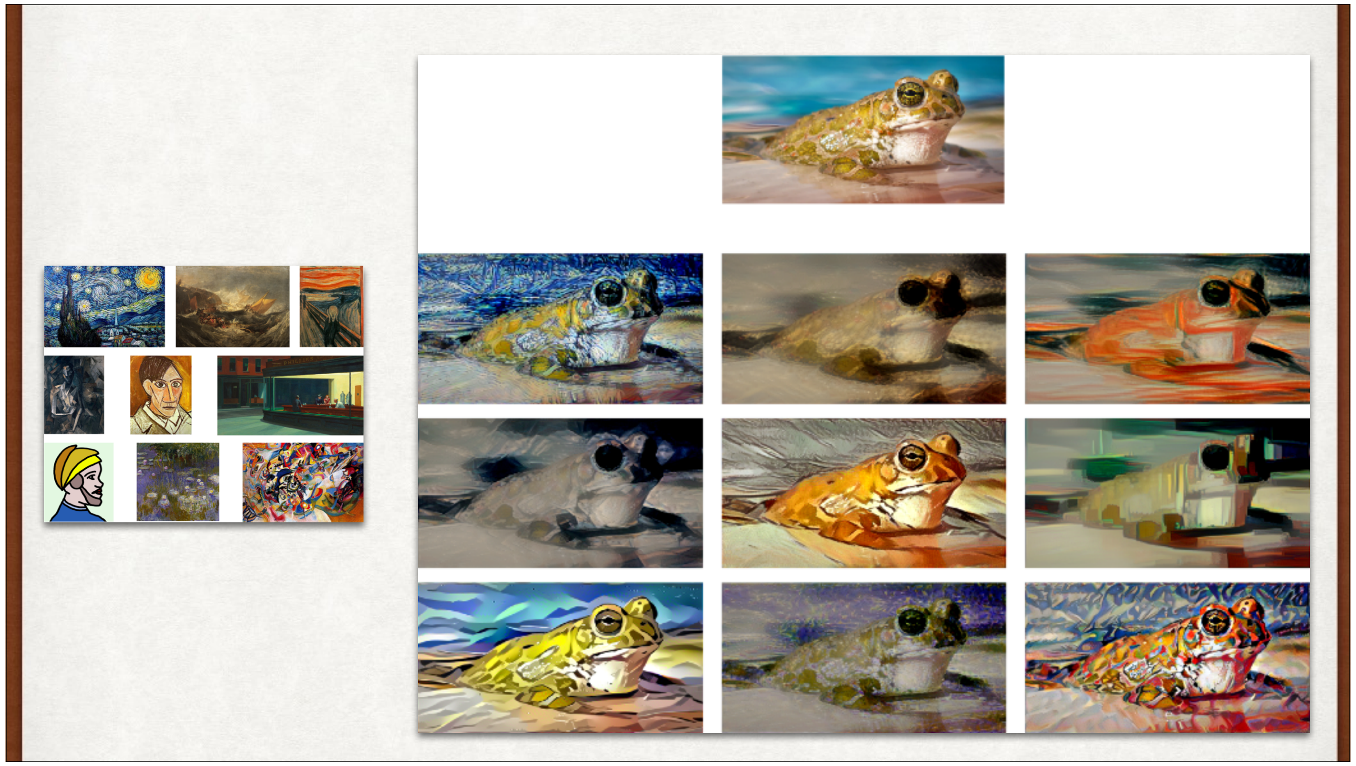
Here’s the overall technique for style transfer. We start with a noisy image. We’ll add the content and style losses together, after weighting them to give them the importances we desire. Here we’re only measuring content loss from the first two layers. The summed loss is our error signal, which we’ll try to minimize by changing the pixels in the input noise. Then we run it through again, and again. That’s the whole algorithm.



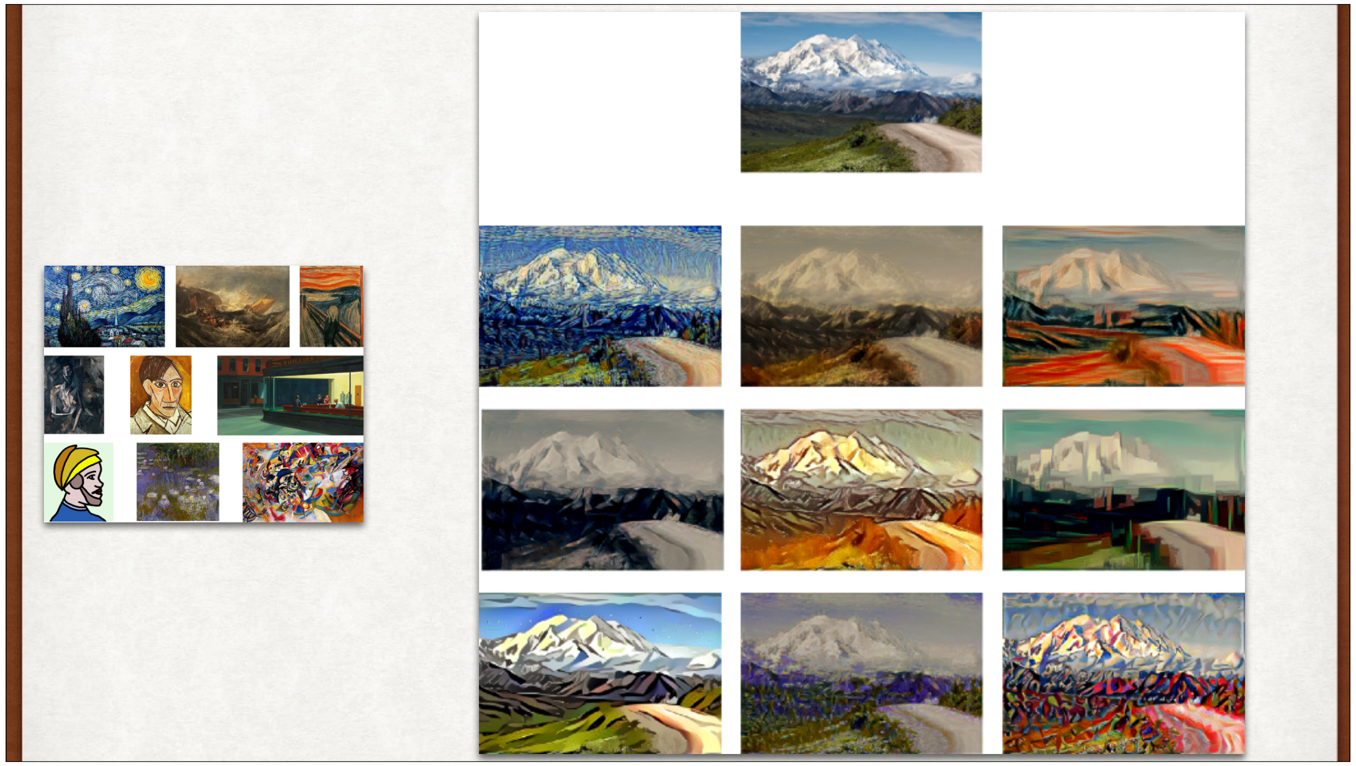
These are not just simple blends or color adjustments to the frog. They’re really unique images. Here’s a closeup of the near leg from each image.



Nine images in different styles. We’ll apply these to the frog.

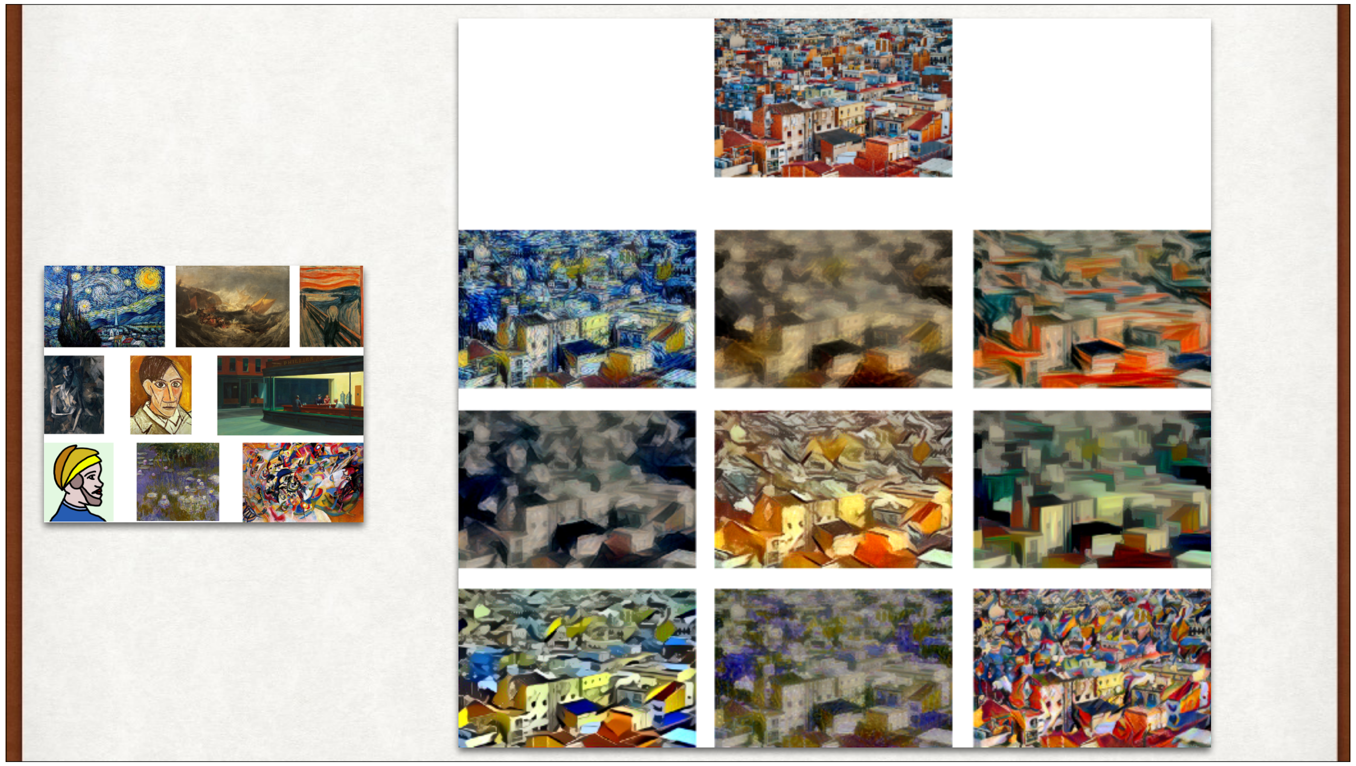


Wow. How do we do this?



The same style pictures, here applied to a photo of a mountain.

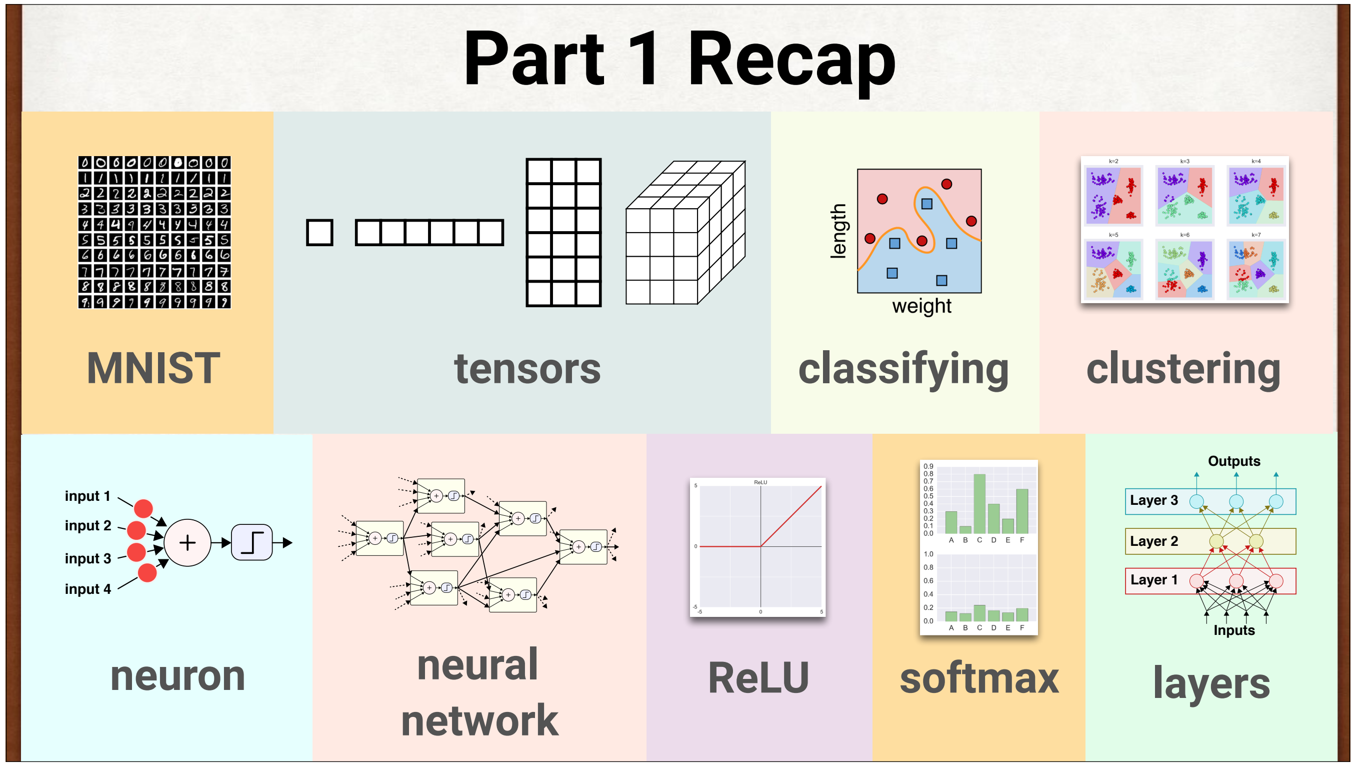
https://pixabay.com/en/mountains-landscape-snow-nature-1622731



And a photo of a town.
https://pixabay.com/en/town-building-urban-architecture-2430571

Big Recap!

GANs learn how to create new data by running a competition between a data generator and a detector that distinguishes between original input data and generated data.



Part 1 Recap

MNIST

tensors

classifying

clustering

neuron

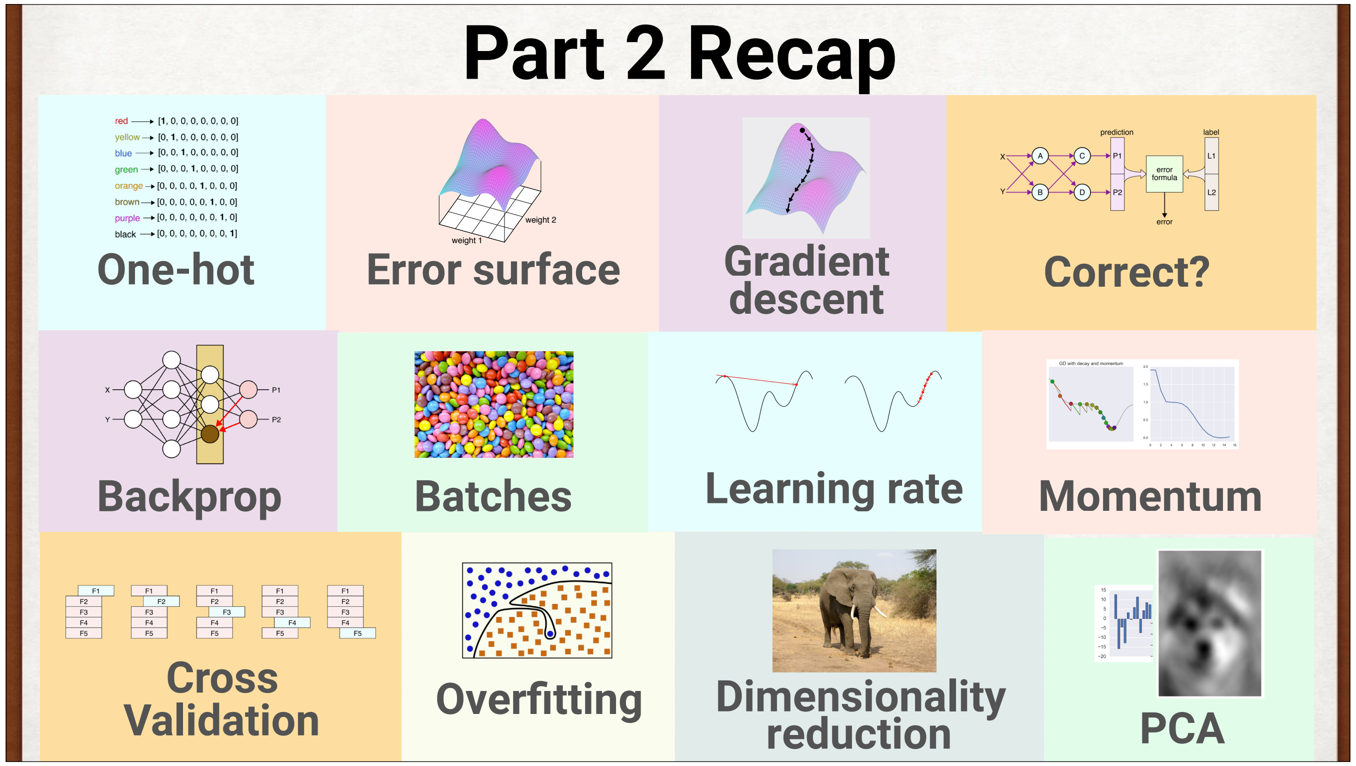
neural network

ReLU

softmax

layers

Part 1



Part 2 Recap

red → [1, 0, 0, 0, 0, 0, 0, 0]
yellow → [0, 1, 0, 0, 0, 0, 0, 0]
blue → [0, 0, 1, 0, 0, 0, 0, 0]
green → [0, 0, 0, 1, 0, 0, 0, 0]
orange → [0, 0, 0, 0, 1, 0, 0, 0]
brown → [0, 0, 0, 0, 0, 1, 0, 0]
purple → [0, 0, 0, 0, 0, 0, 1, 0]
black → [0, 0, 0, 0, 0, 0, 0, 1]

One-hot

Error surface

Gradient descent

Correct?

Backprop

Batches

Learning rate

Momentum

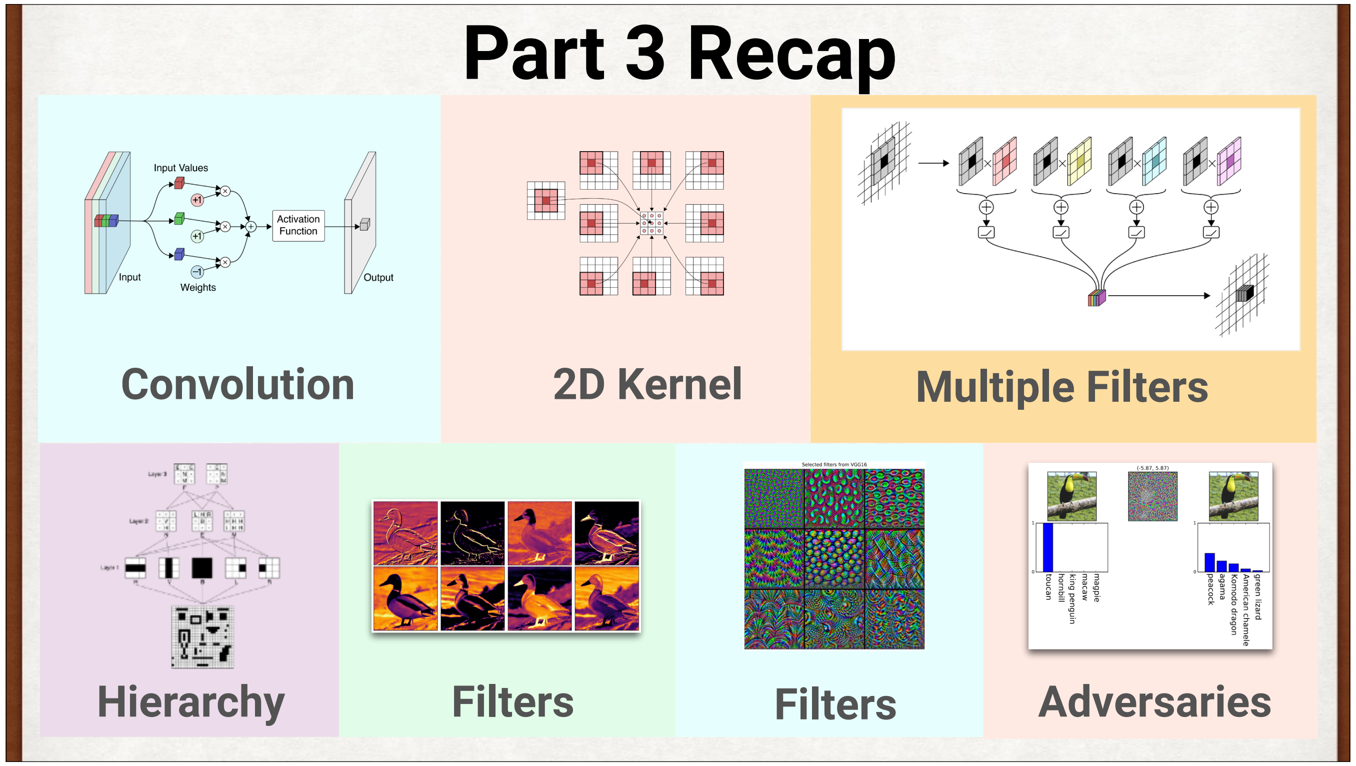
Cross Validation

Overfitting

Dimensionality reduction

PCA

Part 2



Part 3 Recap

Convolution

2D Kernel

Multiple Filters

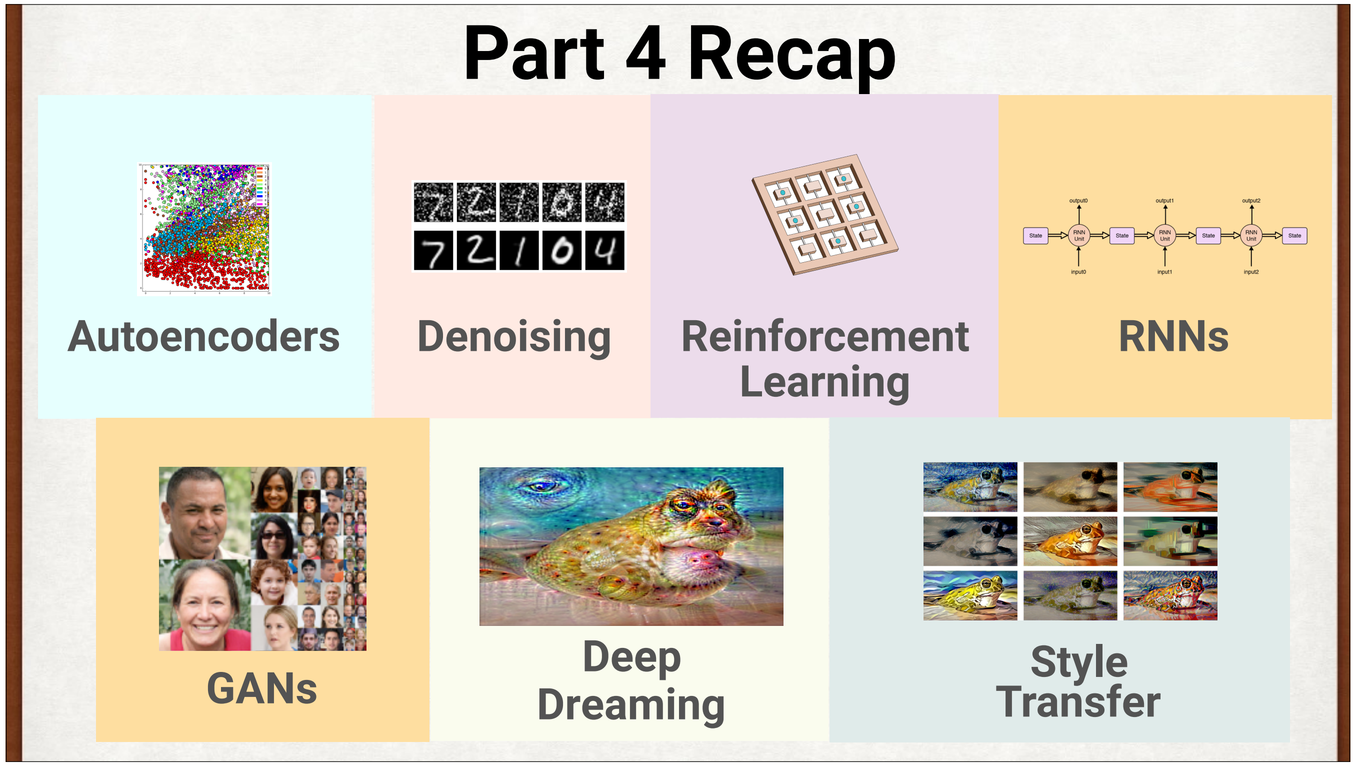
Hierarchy

Filters

Filters

Adversaries

Part 3



Part 4 Recap

Autoencoders

Denoising

Reinforcement Learning

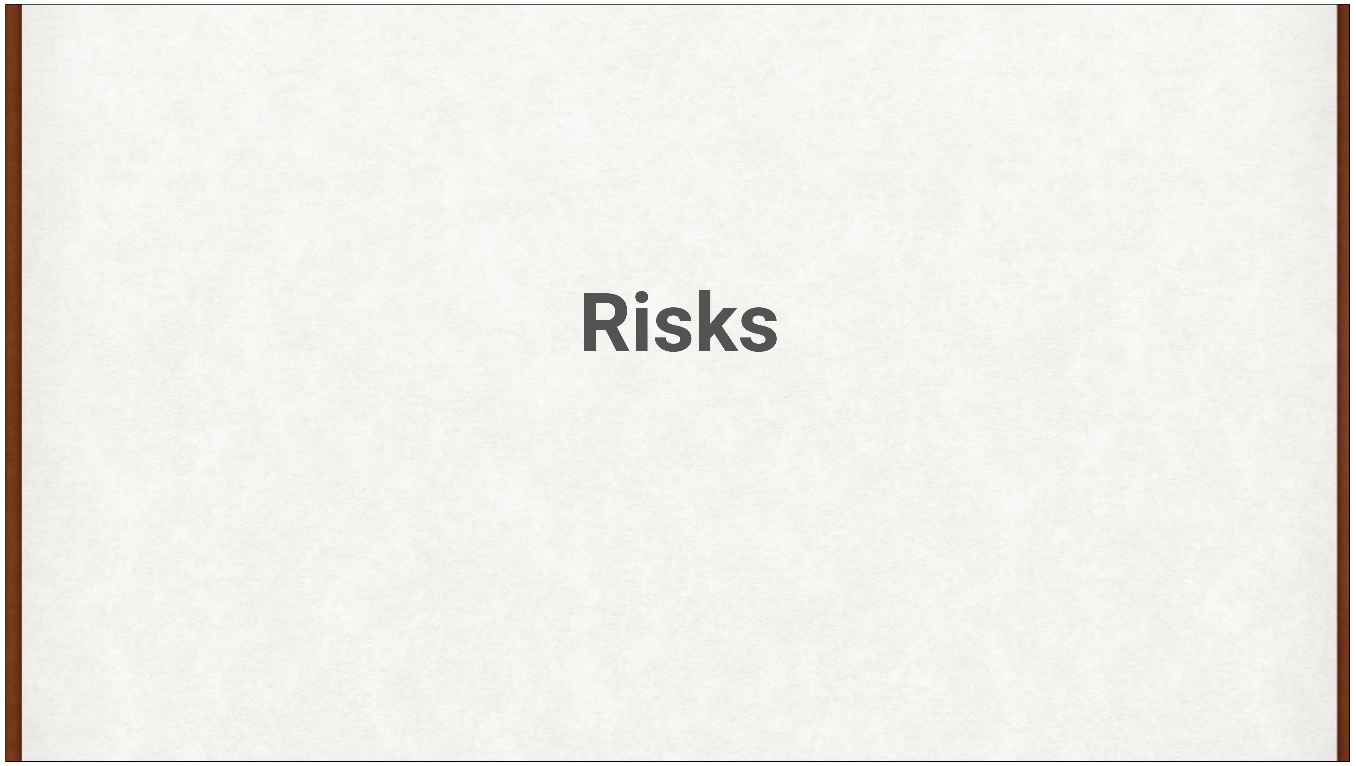
RNNs

GANs

Deep Dreaming

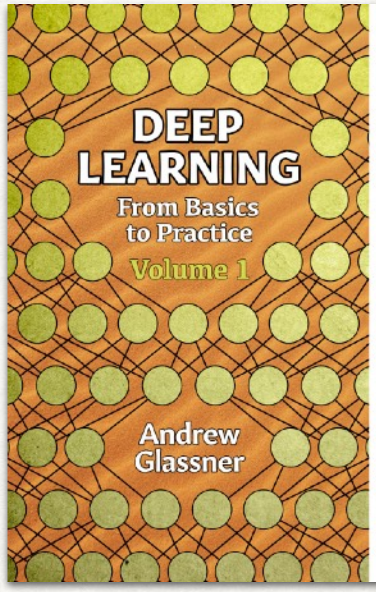
Style Transfer

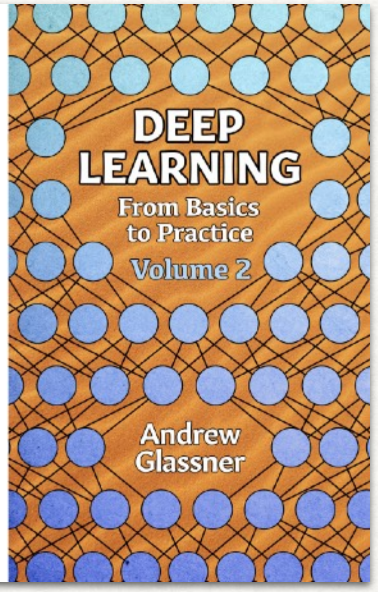
Part 4



Risks

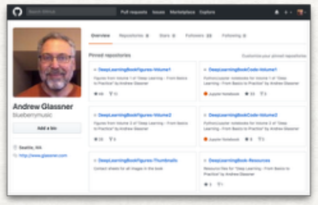
Jail story - DL for bank loans, school admissions, credit scores, promotions, raises, job recruitment/hiring - no recourse - inherent bias, self-reinforcing because bias causes results - no individuality, creative differences punished - permanent caste system - no anonymizing - data theft a sure thing -





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.

Andrew Glassner

glassner.com

@AndrewGlassner

in AndrewGlassner

✉ andrew.glassner@gmail.com

github.com/blueberrymusic

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.

Thank you!