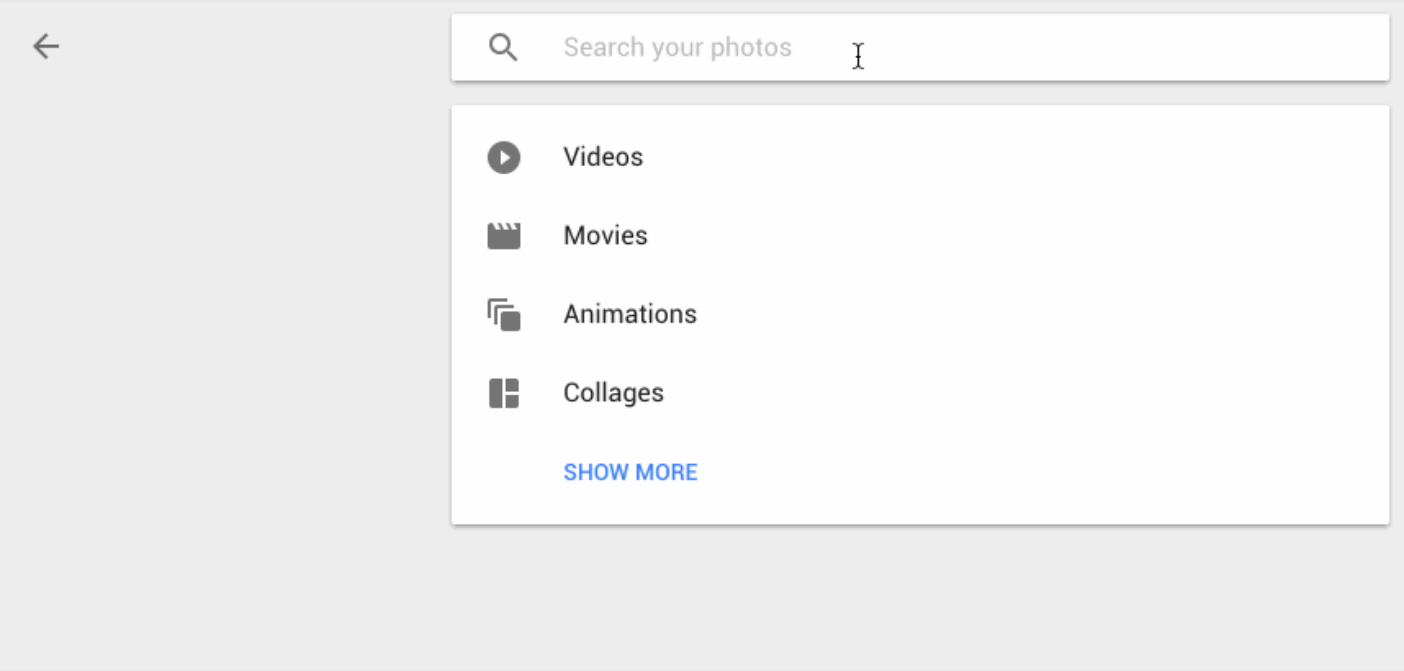
Deep Learning and Convolutional Neural Networks

Deep Learning

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

- In this session, we are going to learn how to write programs that recognize objects in images using deep learning.

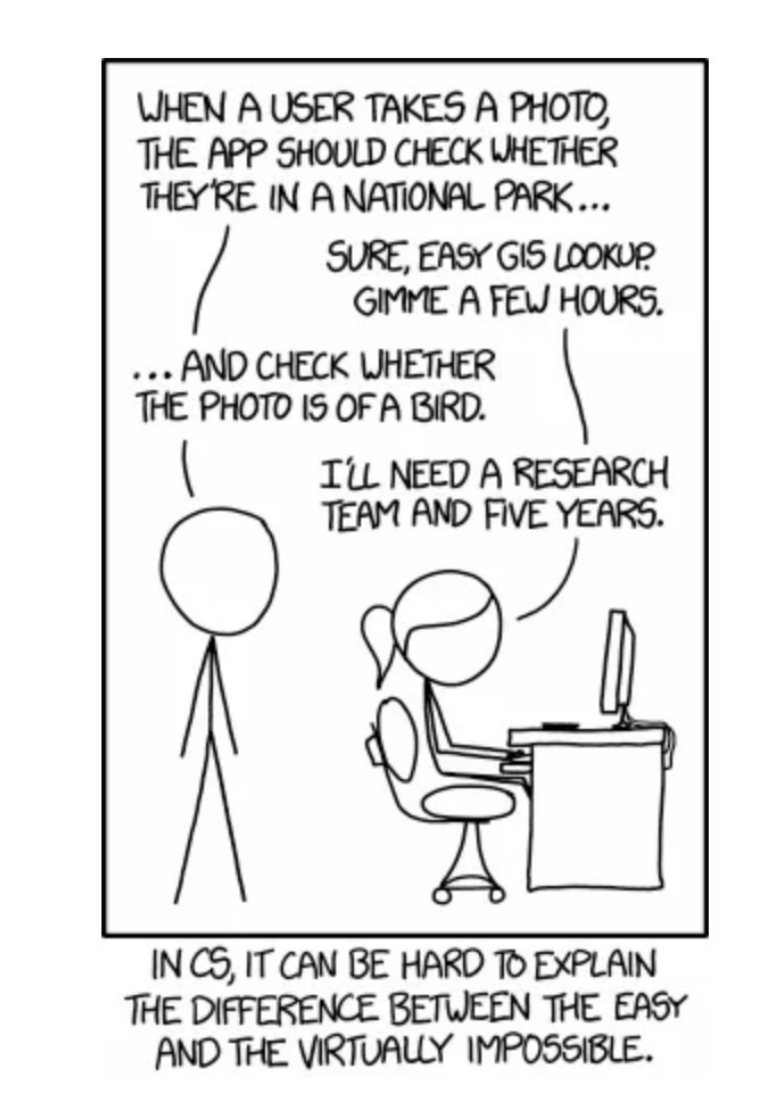
Google lets you search your own photos by description — even if they aren't tagged! How does this work?



A note

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

Recognizing Objects with Deep Learning

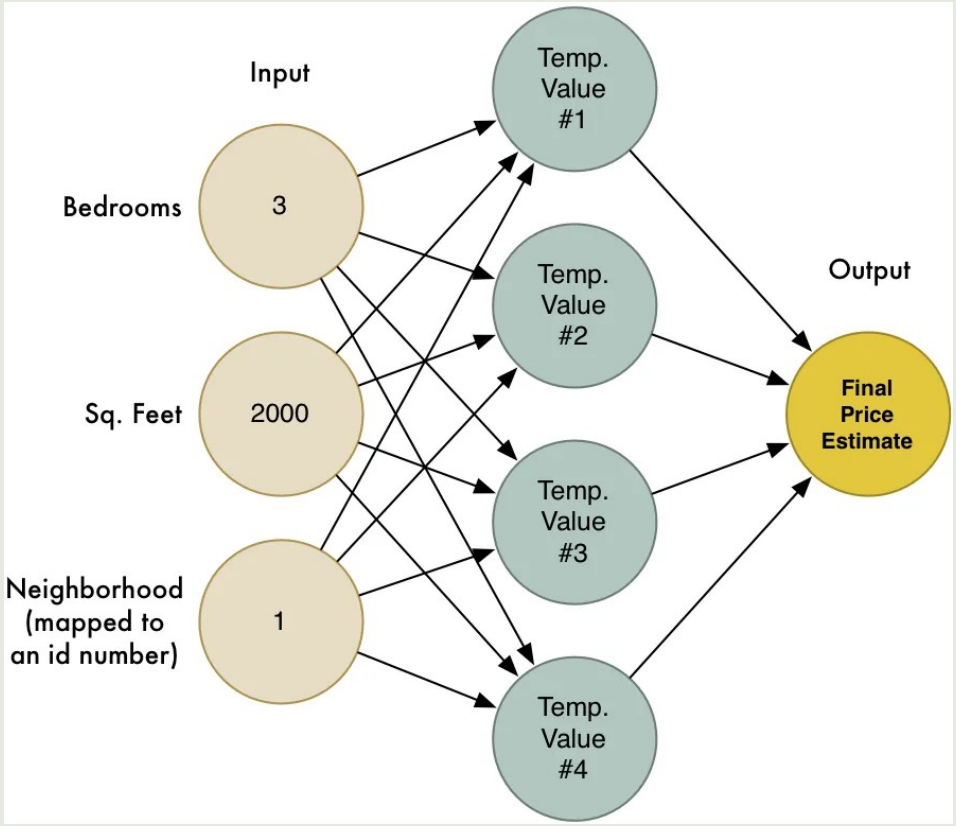


Starting Simple

- Before we learn how to recognize pictures of birds, let’s learn how to recognize something much simpler — the handwritten number “8”.

- In the previous sessions, we created a small neural network to estimate the price of a house based on how many bedrooms it had, the square footage of the house, and which neighborhood it was in

Neural network to estimate house price

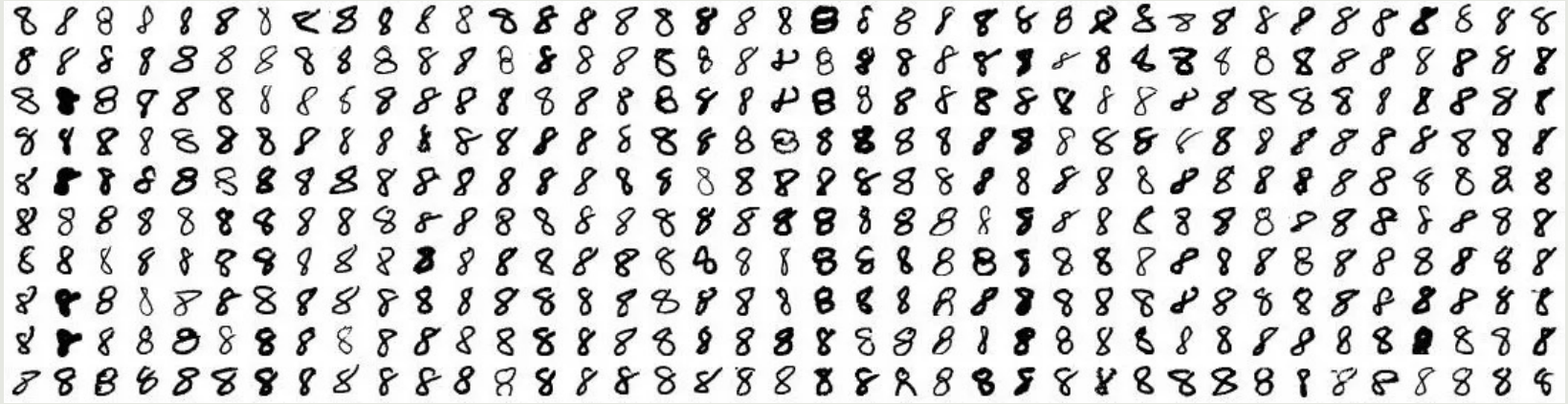


Making changes

- We know that the idea of machine learning is that the same generic algorithms can be reused with different data to solve different problems.
- Let’s modify this same neural network to recognize handwritten text.
- To make the job really simple, we’ll only try to recognize one letter — the numeral “8”.

Data

- We need lots and lots of handwritten “8”s to get started. Luckily, researchers created the MNIST data set of handwritten numbers for this very purpose.
- MNIST provides 60,000 images of handwritten digits, each as an 18x18 pixels image.

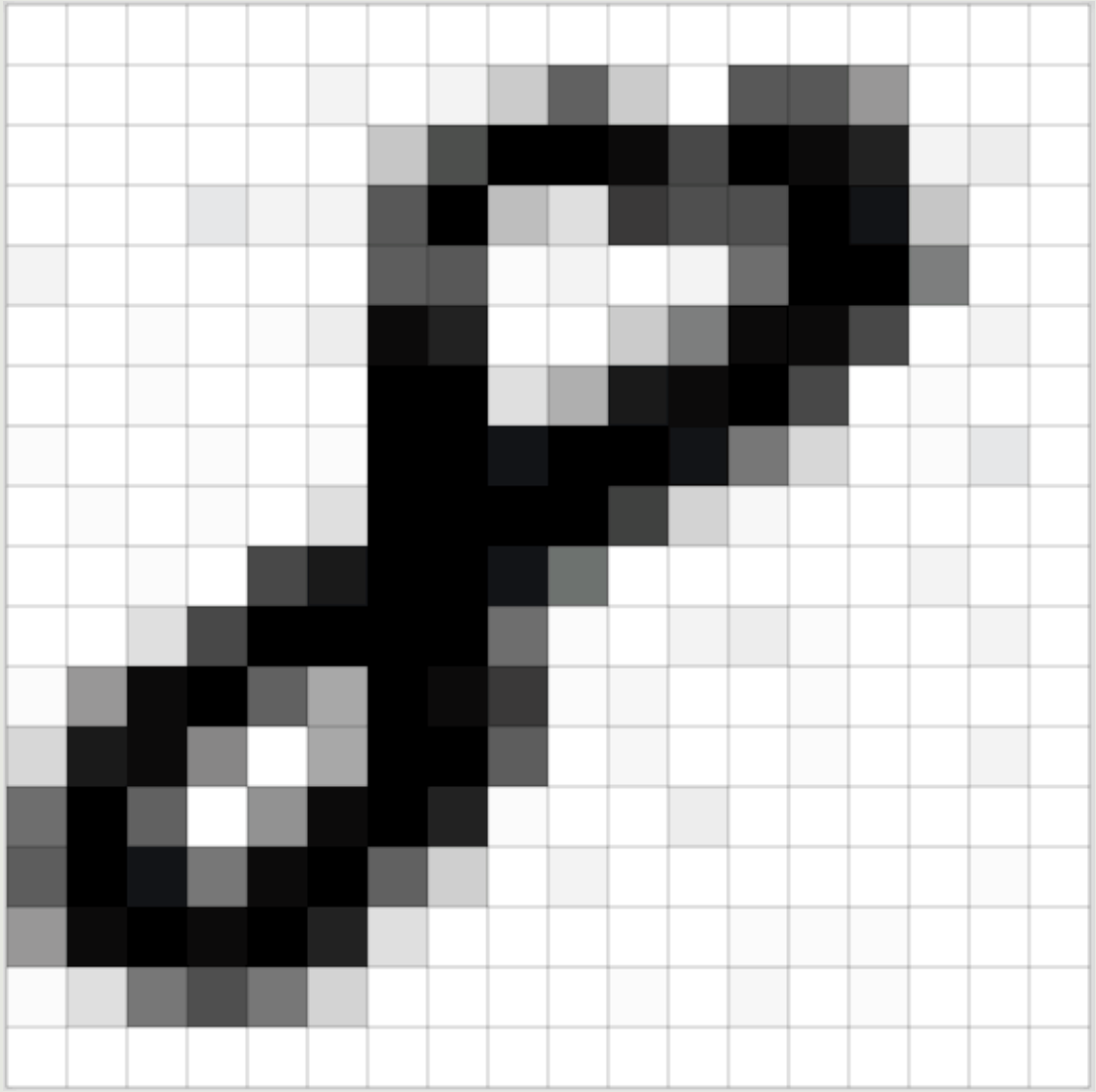


Neural Network Input

- The neural network we made for the house price estimation data only took in a three numbers as the input (“3” bedrooms, “2000” sq. feet , etc.).
- But now we want to process images with our neural network.
- How can we feed images into a neural network instead of just numbers?

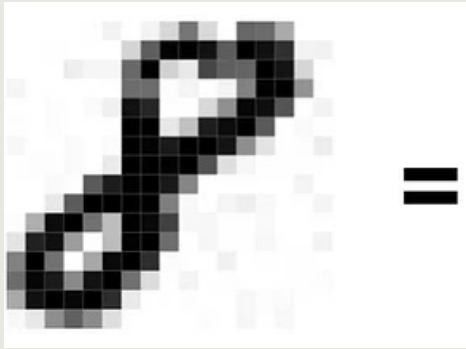
Images as numbers

- To a computer, an image is just a grid of numbers that represent how dark each pixel is.



Images as numbers

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

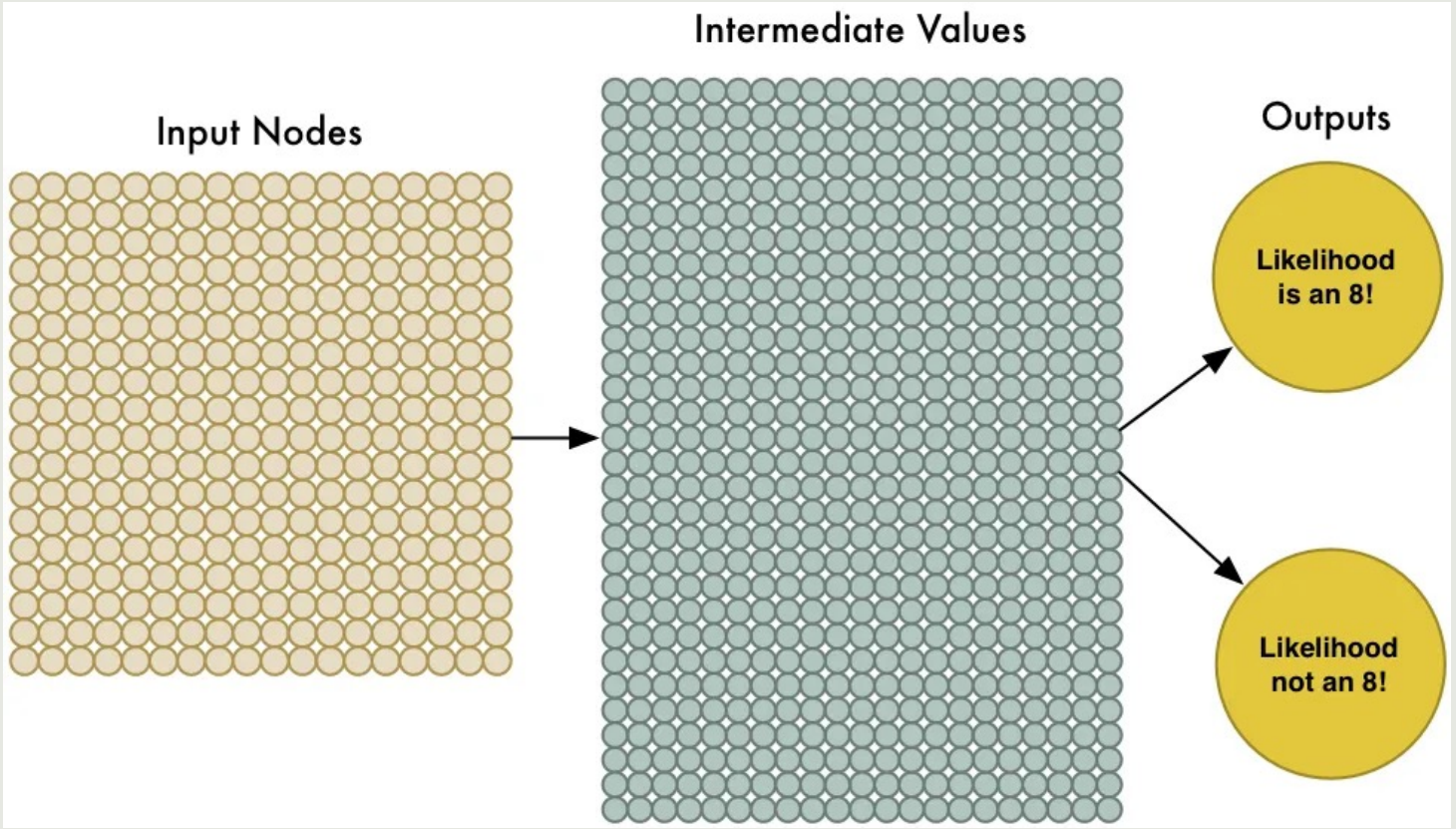


=

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

Neural Network

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



Some observations

- Our neural network also has two outputs now (instead of just one)
- The first output will predict the likelihood that the image is an “8” and thee second output will predict the likelihood it isn’t an “8”.
- By having a separate output for each type of object we want to recognize, we can use a neural network to classify objects into groups.
- Our neural network is a lot bigger than last time (324 inputs instead of 3!). But any modern computer can handle a neural network with a few hundred nodes without blinking. This would even work fine on your cell phone.

Training the network

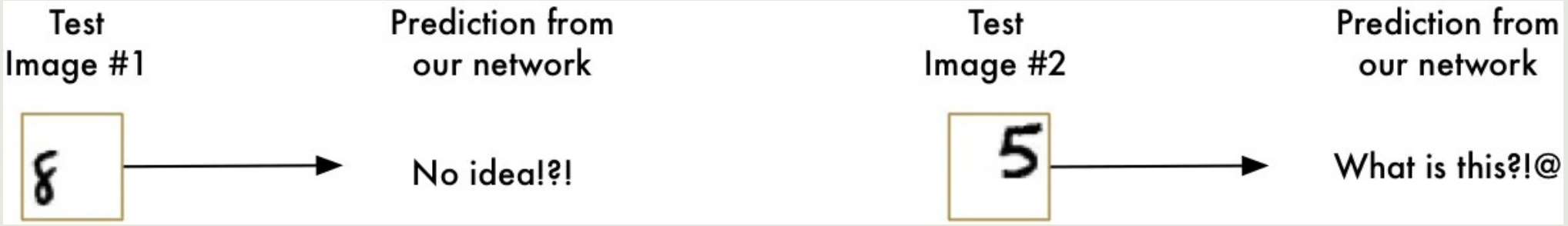
- We need to train the neural network with images of “8”s and not-“8"s so it learns to tell them apart. When we feed in an “8”, we’ll tell it the probability the image is an “8” is 100% and the probability it’s not an “8” is 0%.
- We can train this kind of neural network in a few minutes on a modern laptop with a pretty high accuracy.
- Welcome to the world of (late 1980’s-era) image recognition!

Tunnel Vision

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

Tunnel Vision

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

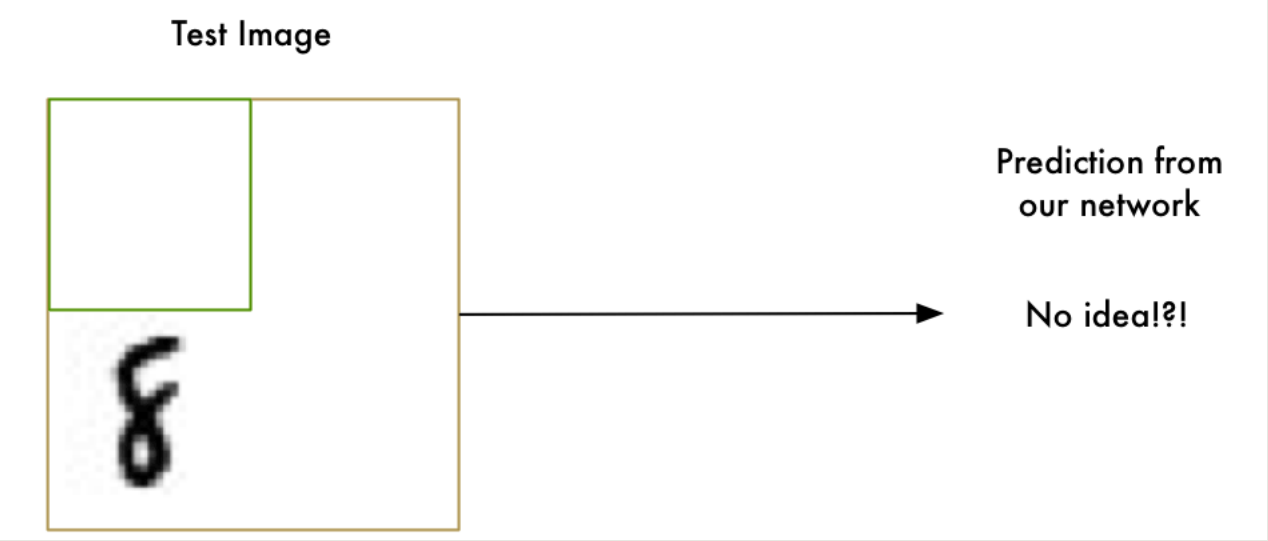


Tunnel Vision

- This is because our network only learned the pattern of a perfectly-centered “8”.
- It has absolutely no idea what an off-center “8” is. It knows exactly one pattern and one pattern only.
- Real world problems are never that clean and simple. So we need to figure out how to make our neural network work in cases where the “8” isn’t perfectly centered.

Brute Force Idea #1: Searching with a Sliding Window

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



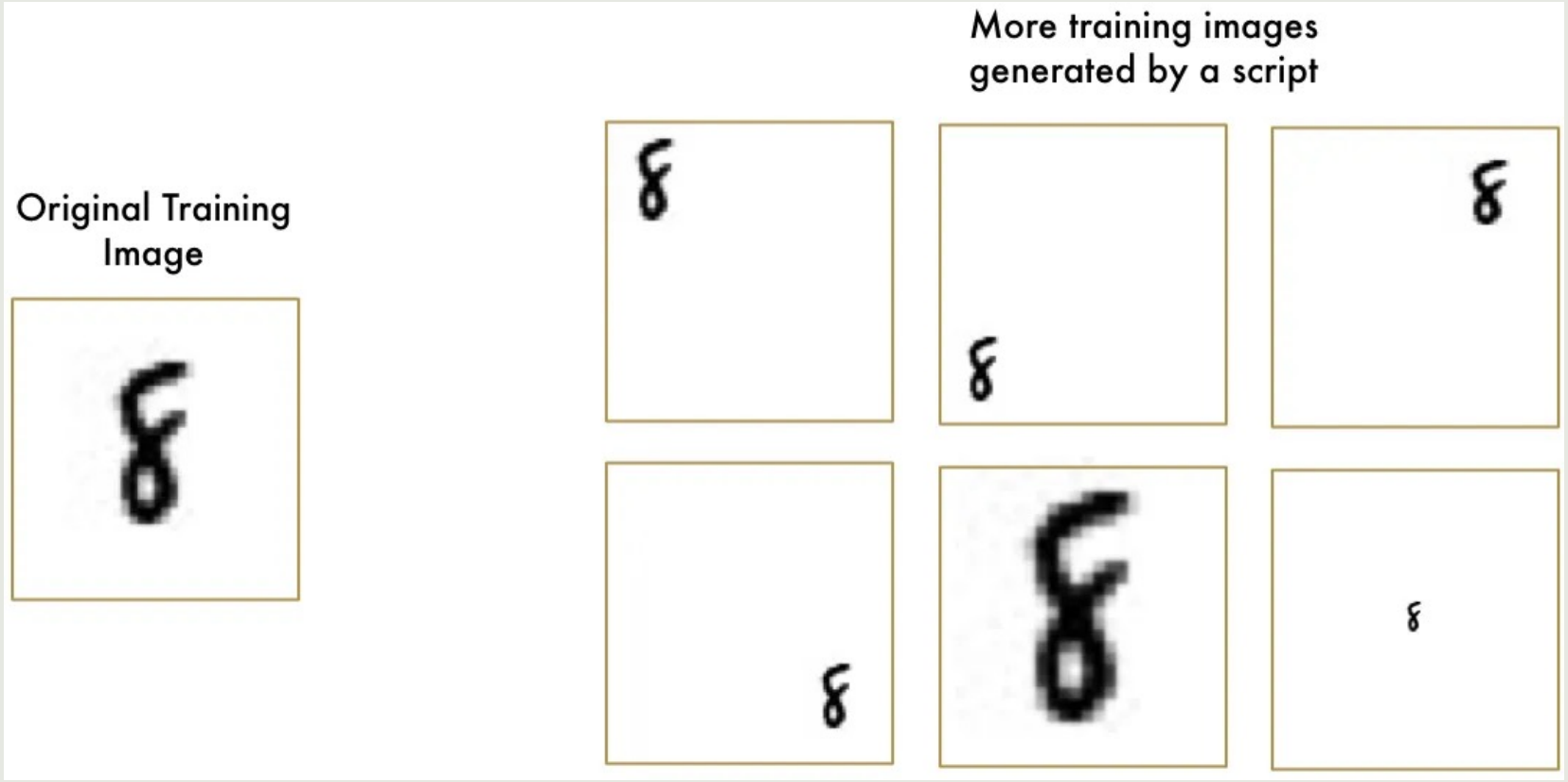
Brute Force Idea #1: Searching with a Sliding Window

- This approach called a sliding window.
- It works well in some limited cases, but it’s very inefficient.
- We need to check the same image over and over, looking for objects of different sizes.

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

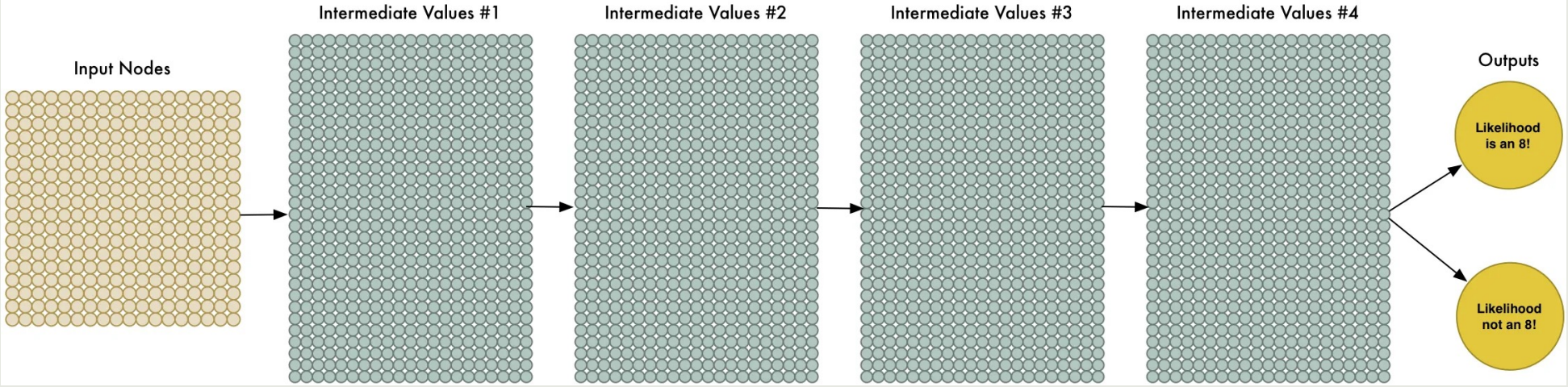
- When we trained our network, we only showed it “8”s that were perfectly centered. What if we train it with more data, including “8”s in all different positions and sizes all around the image?
- We don’t even need to collect new training data. We can just write a script to generate new images with the “8”s in all kinds of different positions in the image.
- This is a very useful creation technique called Synthetic Training Data.

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



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

- More data makes the problem harder for our neural network to solve, but we can compensate for that by making our network bigger and thus able to learn more complicated patterns.
- To make the network bigger, we just stack up layer upon layer of nodes



Deep Learning

- We call this a “deep neural network” because it has more layers than a traditional neural network.
- This idea has been around since the late 1960s. But until recently, training this large of a neural network was just too slow to be useful.
- The advent of new and fast pieces of hardware called Graphical Processing Unit (GPU) has made it feasible for us to train large neural networks quicker.
- But even though we can make our neural network really big and train it quickly with a GPU, that still isn’t going to get us all the way to a solution.

A Better Solution

- It doesn’t make sense to train a network to recognize an “8” at the top of a picture separately from training it to recognize an “8” at the bottom of a picture as if those were two totally different objects.
- There should be some way to make the neural network smart enough to know that an “8” anywhere in the picture is the same thing without all that extra training.
- This technique is called as Convolution.

The Solution is Convolution

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

The Solution is Convolution

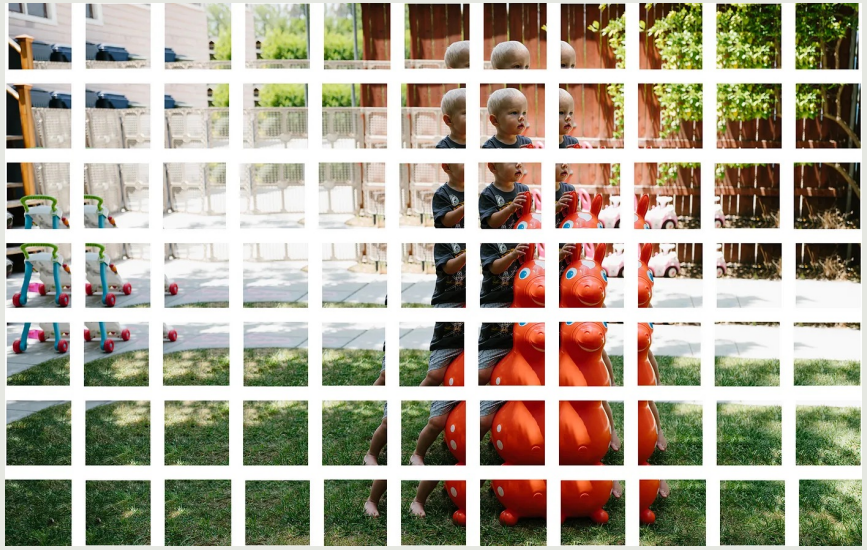
- As humans, we instantly recognize the hierarchy in this picture:
  - The ground is covered in grass and concrete
  - There is a child
  - The child is sitting on a bouncy horse
  - The bouncy horse is on top of the grass
- Most importantly, we recognize the idea of a *child* no matter what surface the child is on.
- We don’t have to re-learn the idea of *child* for every possible surface it could appear on.

Translation Invariance

- But right now, our neural network can't do this. It thinks that an "8" in a different part of the image is an entirely different thing.
- It doesn't understand that moving an object around in the picture doesn't make it something different. This means it has to re-learn the identify of each object in every possible position.
- We need to give our neural network understanding of *translation invariance* — an "8" is an "8" no matter where in the picture it shows up.
- We'll do this using a process called Convolution. The idea of convolution is inspired partly by computer science and partly by biology (i.e. mad scientists literally poking cat brains with weird probes to figure out how cats process images).

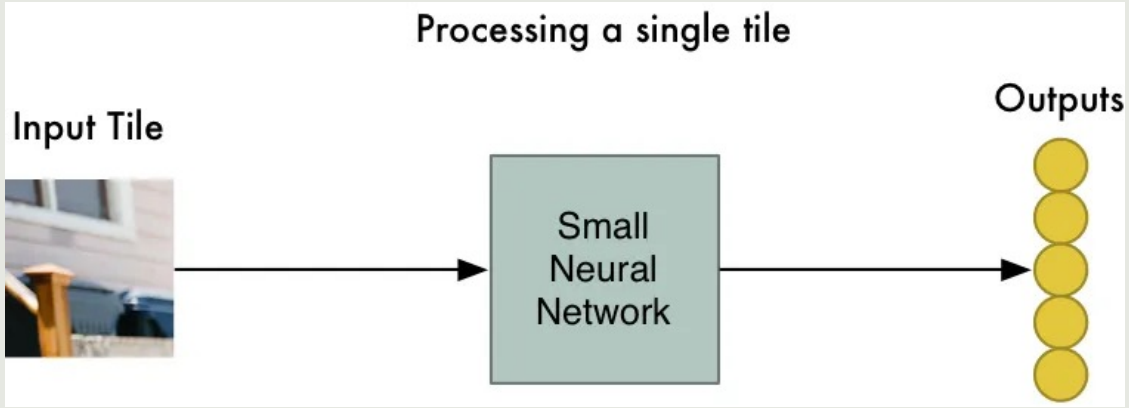
How Convolution Works – Step 1: Break the image into overlapping image tiles

- Similar to the sliding window search seen before, let's pass a sliding window over the entire original image and save each result as a separate, tiny picture tile. By doing this, we turned our original image into 77 equally-sized tiny image tiles.



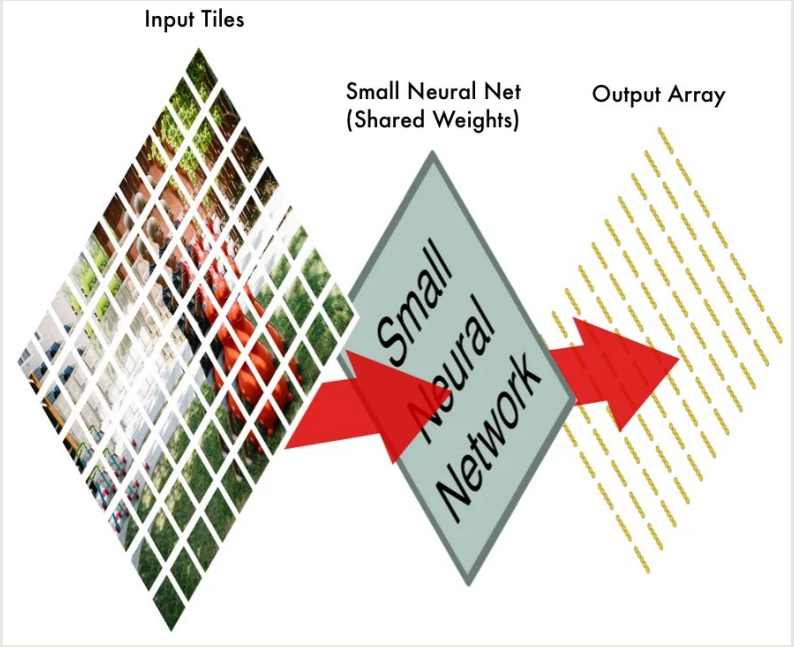
Step 2: Feed each image tile into a small neural network

- Earlier, we fed a single image into a neural network to see if it was an “8”. We’ll do the exact same thing here, but we’ll do it for each individual image tile.
- However, **there’s one big twist**: We’ll keep the **same neural network weights** for every single tile in the same original image. In other words, we are treating every image tile equally. If something interesting appears in any given tile, we’ll mark that tile as interesting.



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

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



Step 4: Downsampling

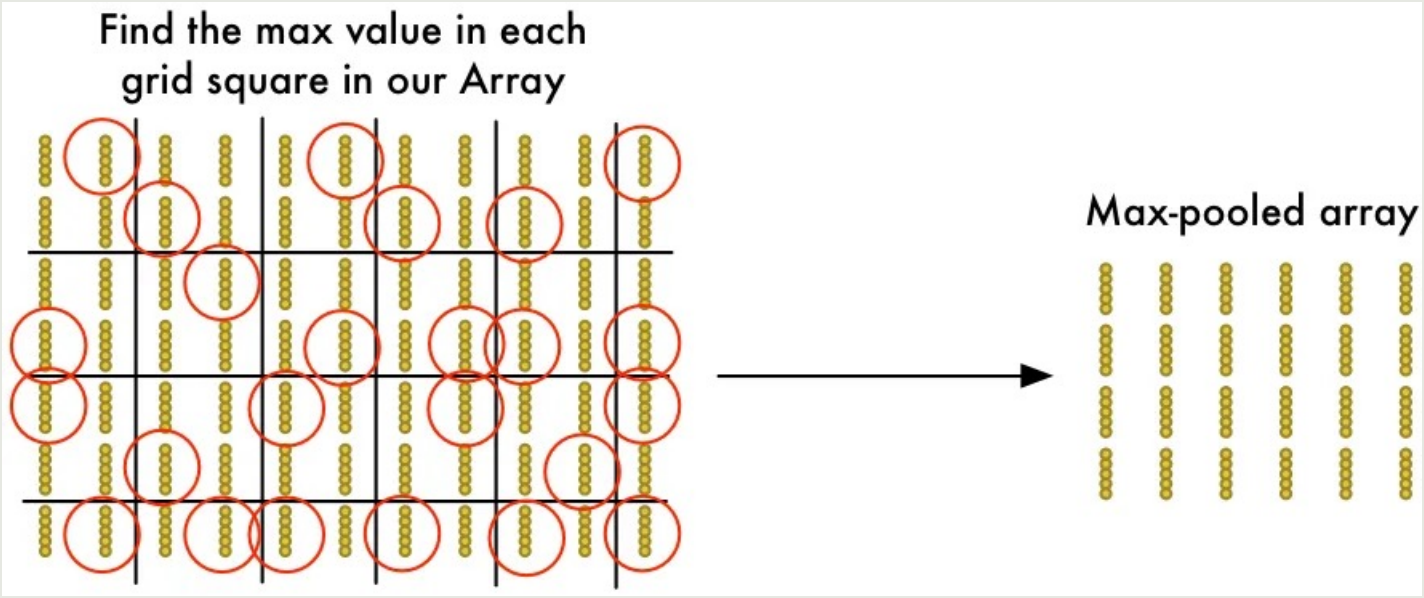
- The result of Step 3 was an array that maps out which parts of the original image are the most interesting. But that array is still pretty big

Original Input Image

Array resulting from convolution in Step 3

Step 4: Downsampling

- To reduce the size of the array, we *downsample* it using an algorithm called max pooling.
- We'll just look at each 2x2 square of the array and keep the biggest number



Step 4: Downsampling

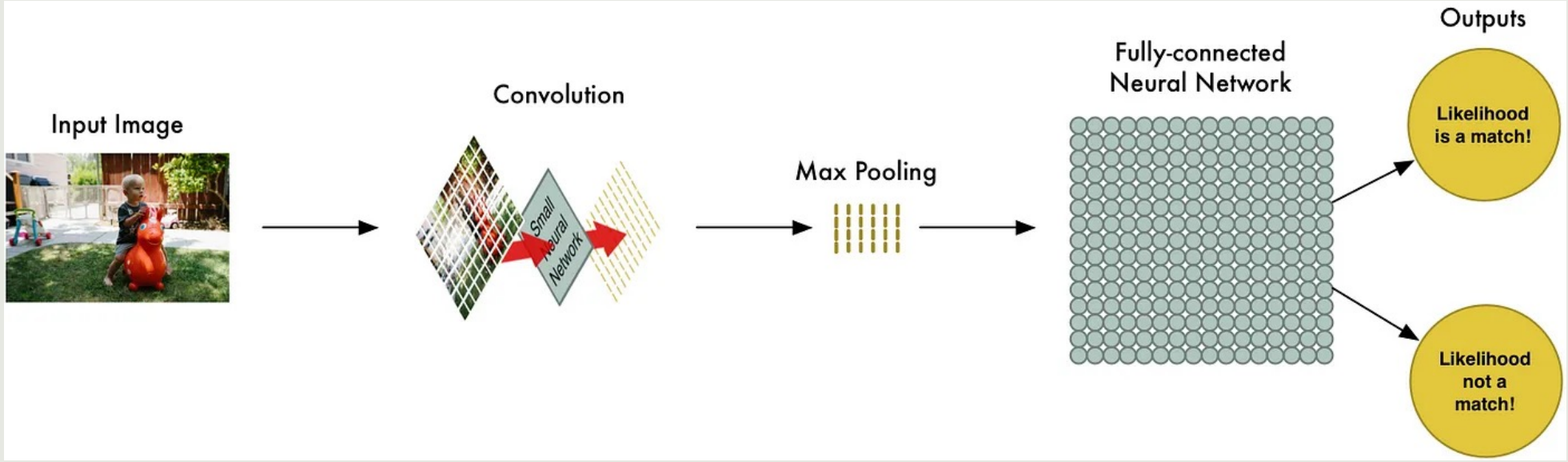
- The idea here is that if we found something interesting in any of the four input tiles that makes up each 2x2 grid square, we'll just keep the most interesting bit.
- This reduces the size of our array while keeping the most important bits.

Final step: Make a prediction

- So far, we've reduced a giant image down into a fairly small array.
- This array is just a bunch of numbers, so we can use that small array as input into *another neural network*.
- This final neural network will decide if the image is or isn't a match.
- To differentiate this neural network from the convolution step, we call it a "fully connected" network.

Overview of the entire pipeline

- From start to finish, our whole five-step pipeline looks like this

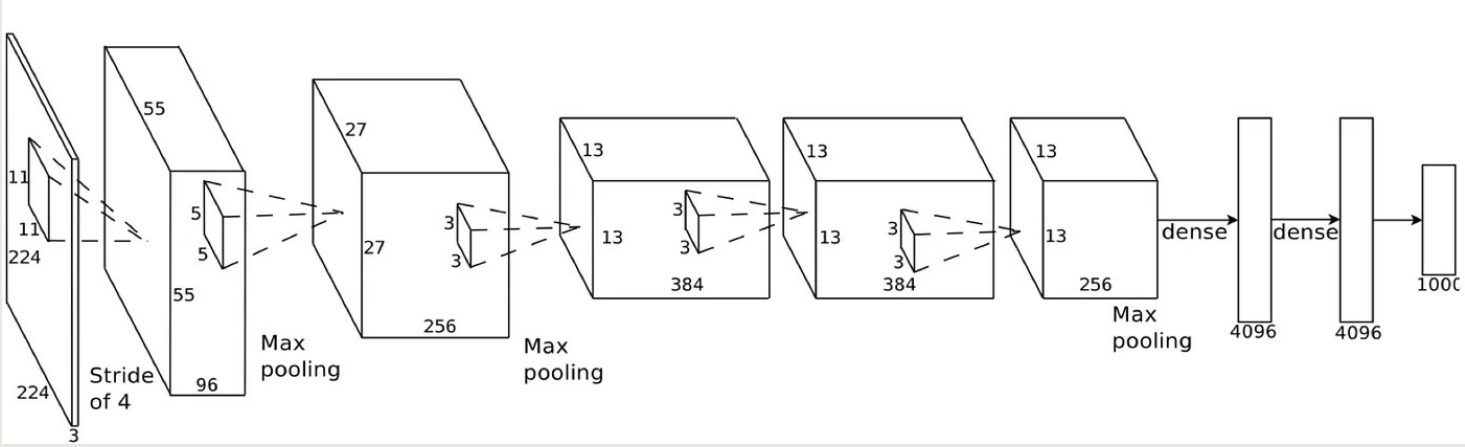


Adding Even More Steps

- Our image processing pipeline is a series of steps: convolution, max-pooling, and finally a fully-connected network.
- When solving problems in the real world, these steps can be combined and stacked as many times as you want! You can have two, three or even ten convolution layers. You can throw in max pooling wherever you want to reduce the size of your data.
- The basic idea is to start with a large image and continually boil it down, step-by-step, until you finally have a single result. The more convolution steps you have, the more complicated features your network will be able to learn to recognize.

Real world Deep CNN

- For example, the first convolution step might learn to recognize sharp edges, the second convolution step might recognize beaks using it's knowledge of sharp edges, the third step might recognize entire birds using it's knowledge of beaks, etc.
- Here's what a more realistic deep convolutional network (like you would find in a research paper) looks like



Real world Deep CNN

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

Constructing the Right Network

- So how do we know which steps you need to combine to make your image classifier work?
- There is no straightforward method. It involves a lot of a lot of experimentation and testing.
- We might have to train 100 networks before we find the optimal structure and parameters for the problem we are solving. Machine learning involves a lot of trial and error!

Building our Bird Classifier

- Now finally we know enough to write a program that can decide if a picture is a bird or not.
- As always, we need some data to get started. The free CIFAR10 data set contains 6,000 pictures of birds and 52,000 pictures of things that are not birds.
- But to get even more data we'll also add in the Caltech-UCSD Birds-2000–2011 data set that has another 12,000 bird pics.

Bird images

Non-Bird Images

About the data set

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

Model training

- We build a model which has a convolution layer followed by max pooling, then 2 more layers of convolution and one round of max-pooling. Finally, we send this array as input to a 512 node fully connected neural network.
- The output classifies an image as a bird or not a bird.
- You can view the code snippet here.

Testing our Network

- The data set I created held back 15,000 images for testing. When I ran those 15,000 images through the network, it predicted the correct answer 95% of the time.
- That seems pretty good, right? Well… it depends!
- Our network claims to be 95% accurate. That could mean all sorts of different things.

Accuracy: An inadequate metric

- For example, what if 5% of our training images were birds and the other 95% were not birds?
- A program that guessed “not a bird” every single time would be 95% accurate! But it would also be 100% useless.
- We need to look more closely at the numbers than just the overall accuracy.
- To judge how good a classification system really is, we need to look closely at *how* it failed, not just the percentage of the time that it failed.
- Instead of thinking about our predictions as “right” and “wrong”, let’s break them down into four separate categories

True Positives

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

True Negatives

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

False Positives

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

False Negatives

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

Results breakdown

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

Results for 15,000 Validation Images

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

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

Results breakdown

- Why do we break our results down like this? Because not all mistakes are created equal.
- Imagine if we were writing a program to detect cancer from an MRI image. If we were detecting cancer, we'd rather have false positives than false negatives.
- False negatives would be the worse possible case — that's when the program told someone they definitely didn't have cancer but they actually did.

Better metrics: Precision and Recall

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

Precision and Recall

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

Thank you!