





Presented at SIGGRAPH 2018 in Vancouver, BC on August 12, 2018. The notes here are just breadcrumbs. You can see the talk itself on YouTube at <https://www.youtube.com/watch?v=r0Ogt-q956I> . These slides have been slightly updated since the course. Images have been saved at only “good” quality to keep the file size down. This file is part 2 of the course, picking up after the break.



Photography & Recording Permitted

© 2018 SIGGRAPH. All Rights Reserved

2

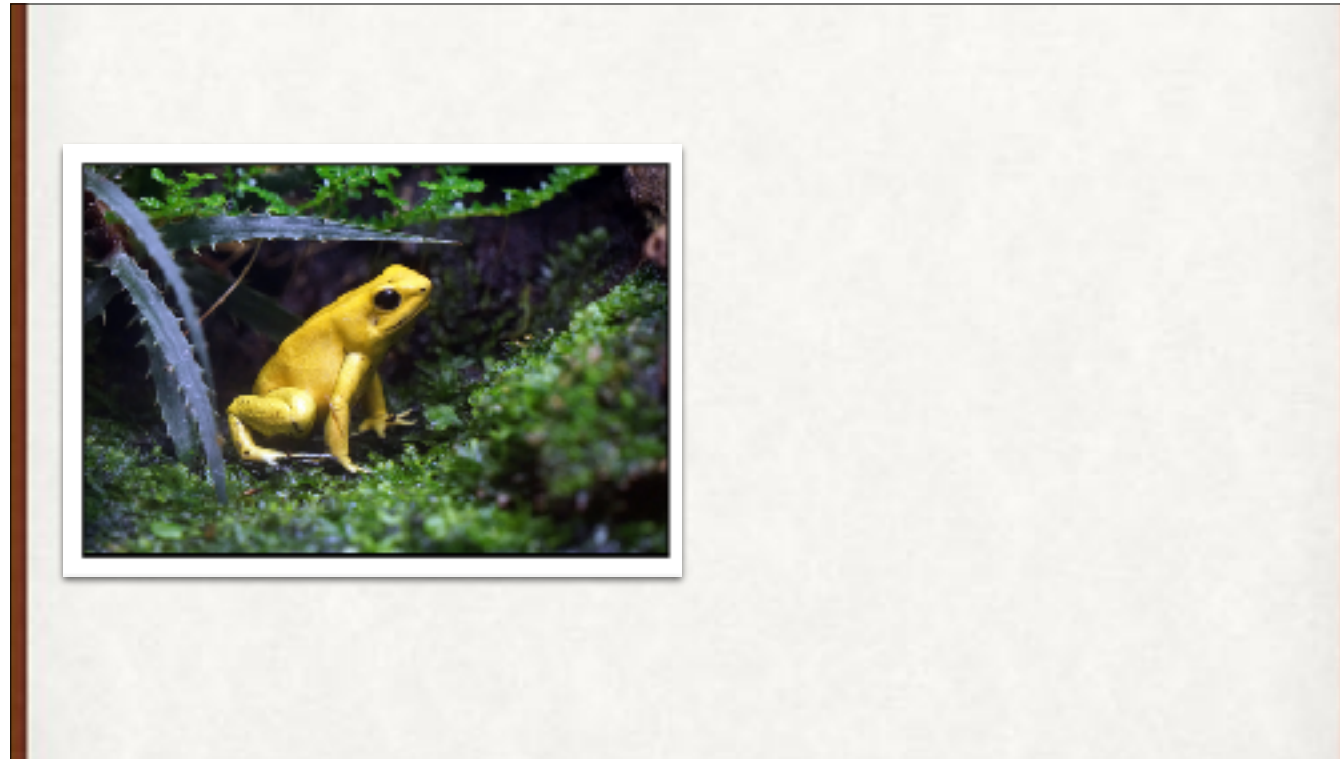
Almost all of these slides are available in high-res format, for free use in your own talks, classes, and other presentations. Go to GitHub under my userid, blueberrymusic. Look in the repos of the figures for my Deep Learning book. This Keynote file is also at GitHub.



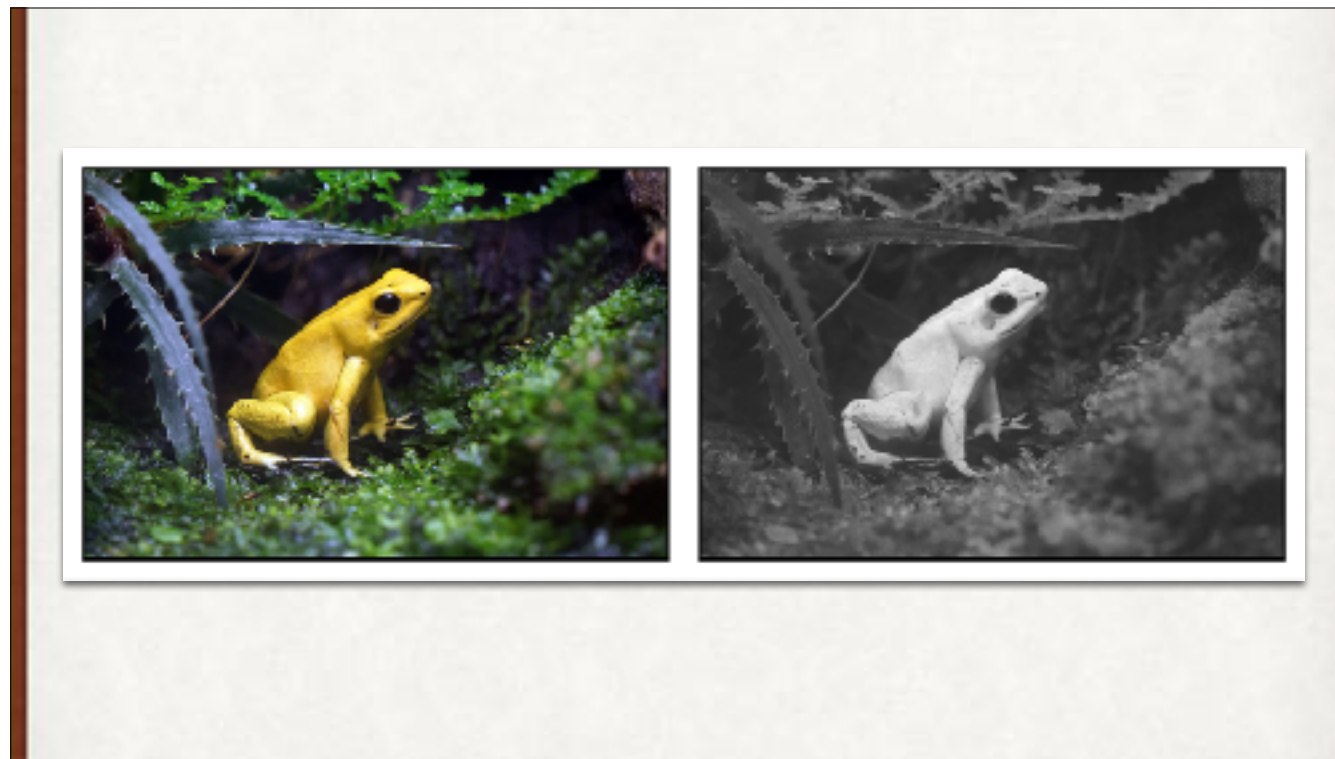
Concepts, terminology, structures, no math, no code. Free open-source libraries do the hard work
This course is based on my books, "Deep Learning from Basics to Practice," available in digital form at <http://amzn.to/2F4nz7k> and <http://amzn.to/2EQtPR2>

Convolution

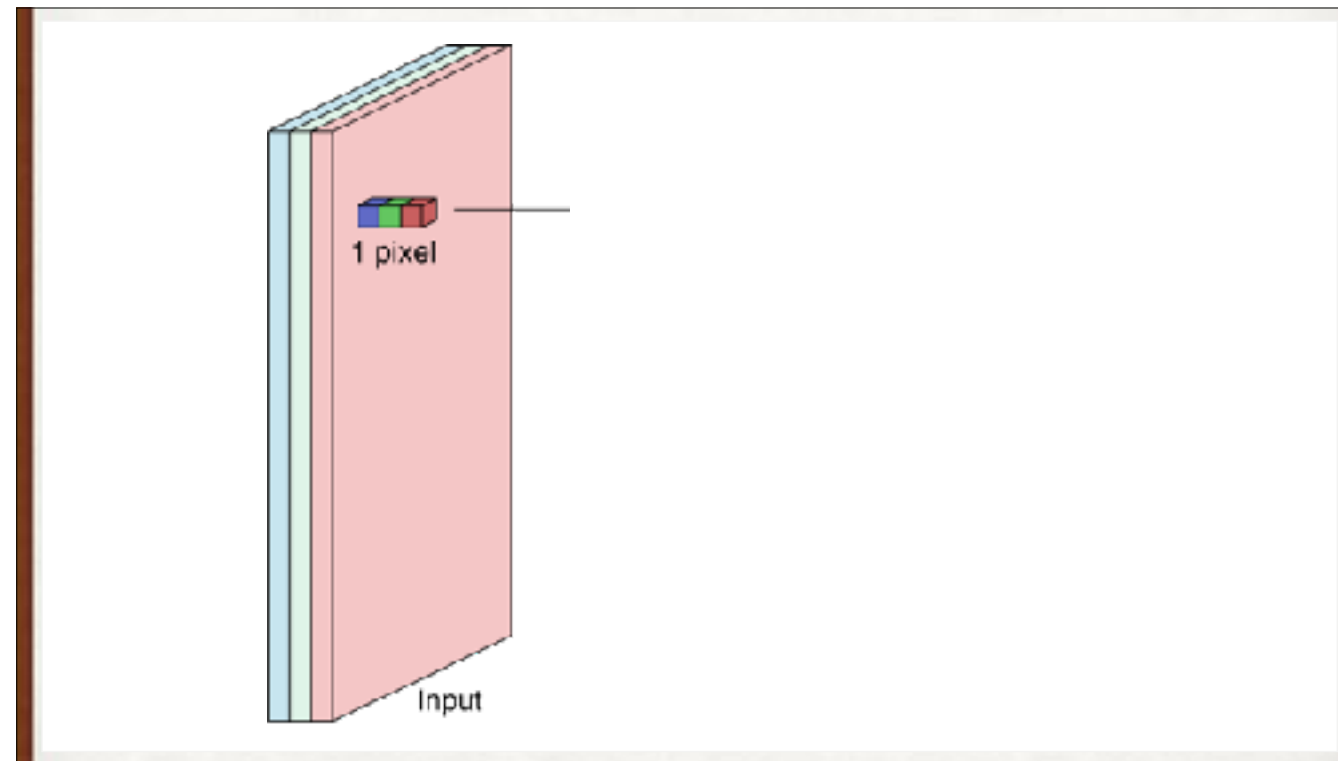
Convolution is hugely popular for image-based systems. Let's see what it's all about, without math!



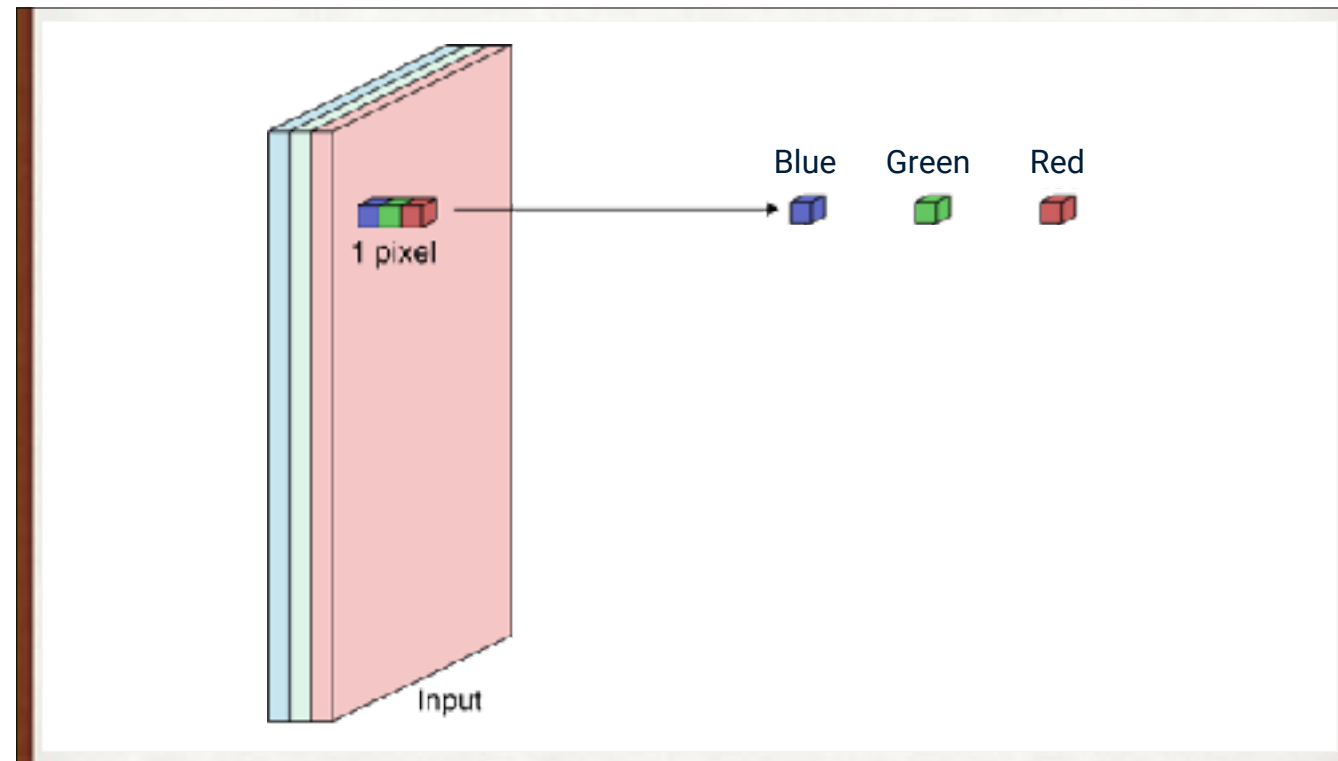
Our yellow filter applied to a frog, with the results scaled to 0=black, 2=white.



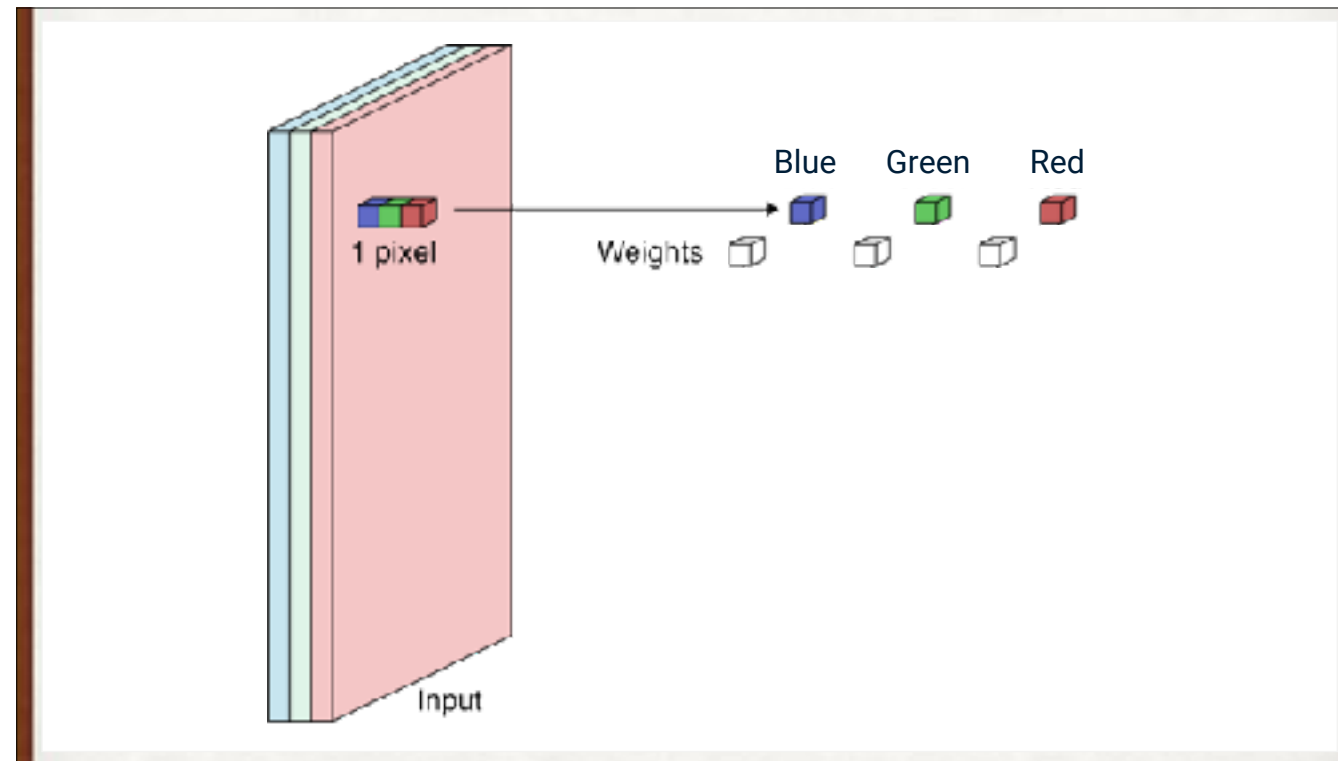
Our yellow filter applied to a frog, with the results scaled to 0=black, 2=white.



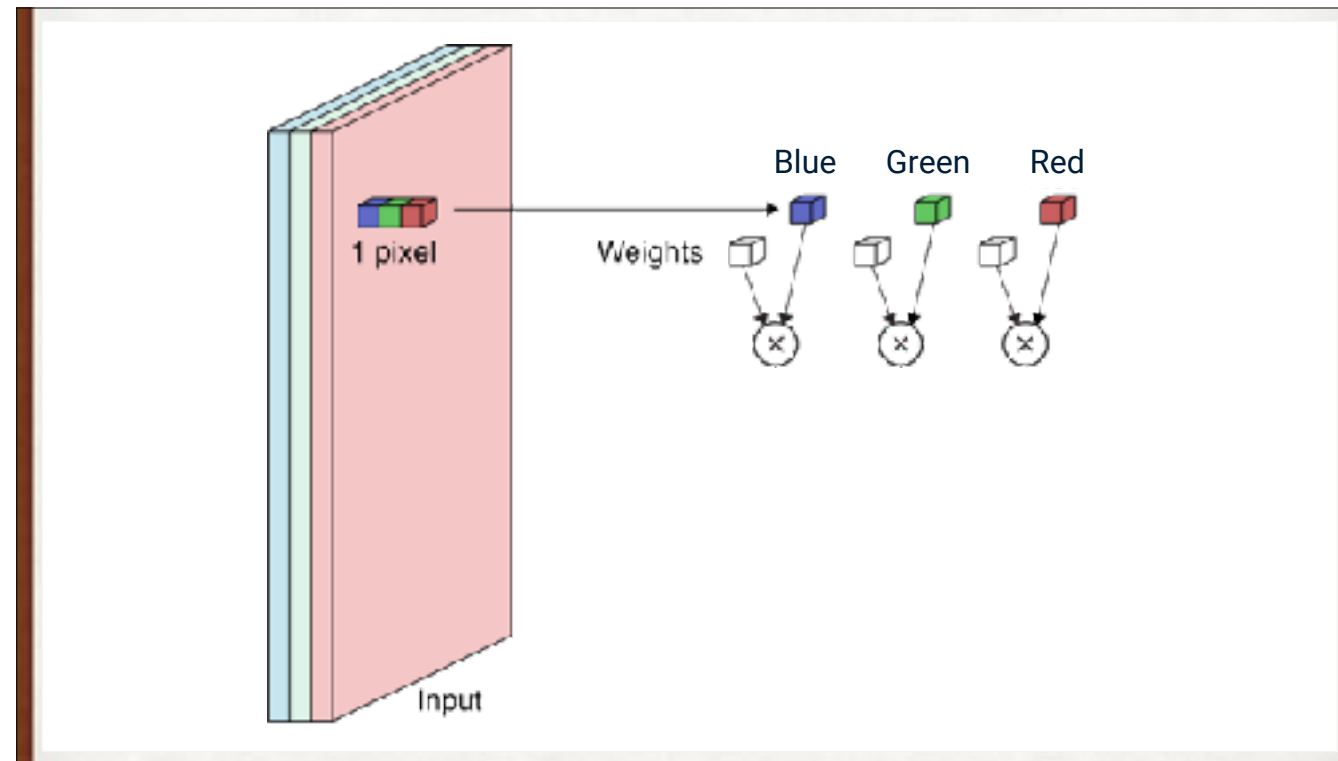
An RGB picture at left. We take a “core sample” of 1 pixel. We multiply the red, green, and blue values by associated weights. Then we add up the results. For our purposes, this is “convolution”. We say that we have “convolved” the input (the core sample) with a filter (the three weights).



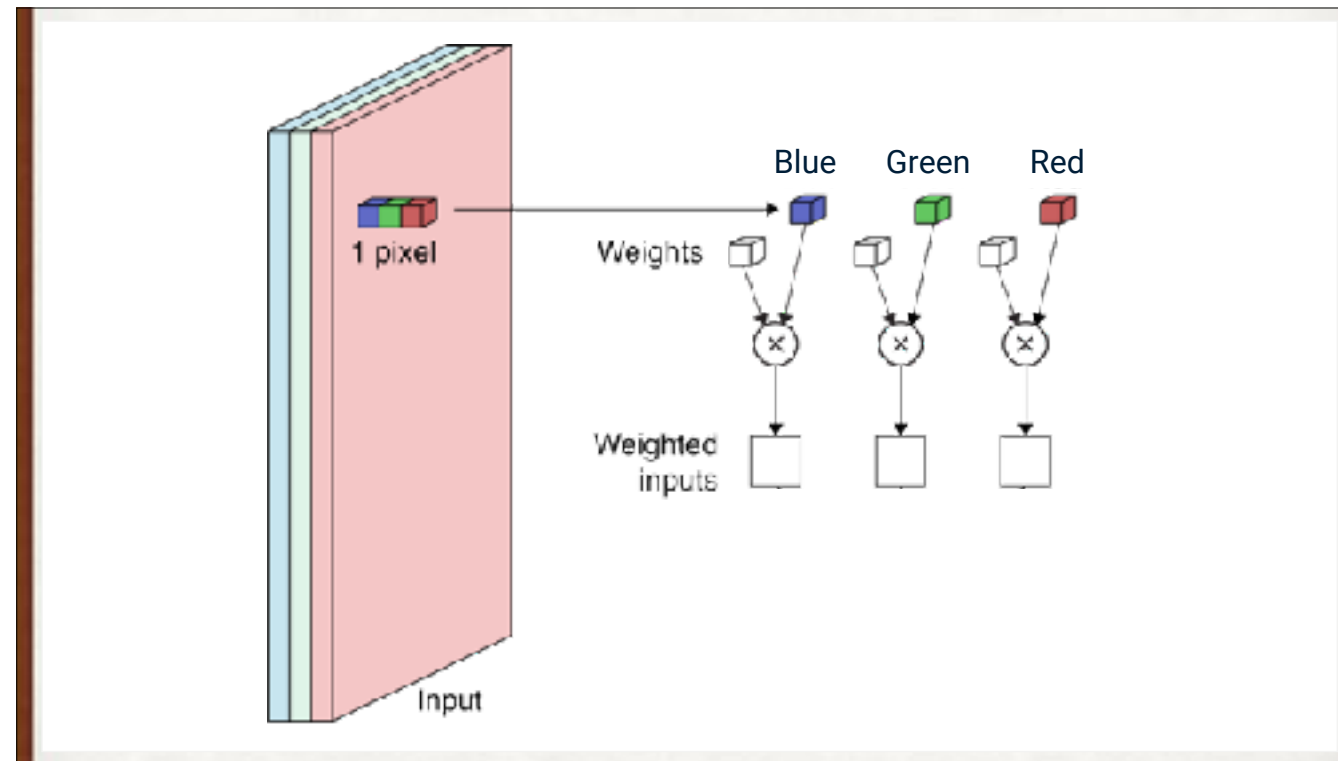
An RGB picture at left. We take a “core sample” of 1 pixel. We multiply the red, green, and blue values by associated weights. Then we add up the results. For our purposes, this is “convolution”. We say that we have “convolved” the input (the core sample) with a filter (the three weights).



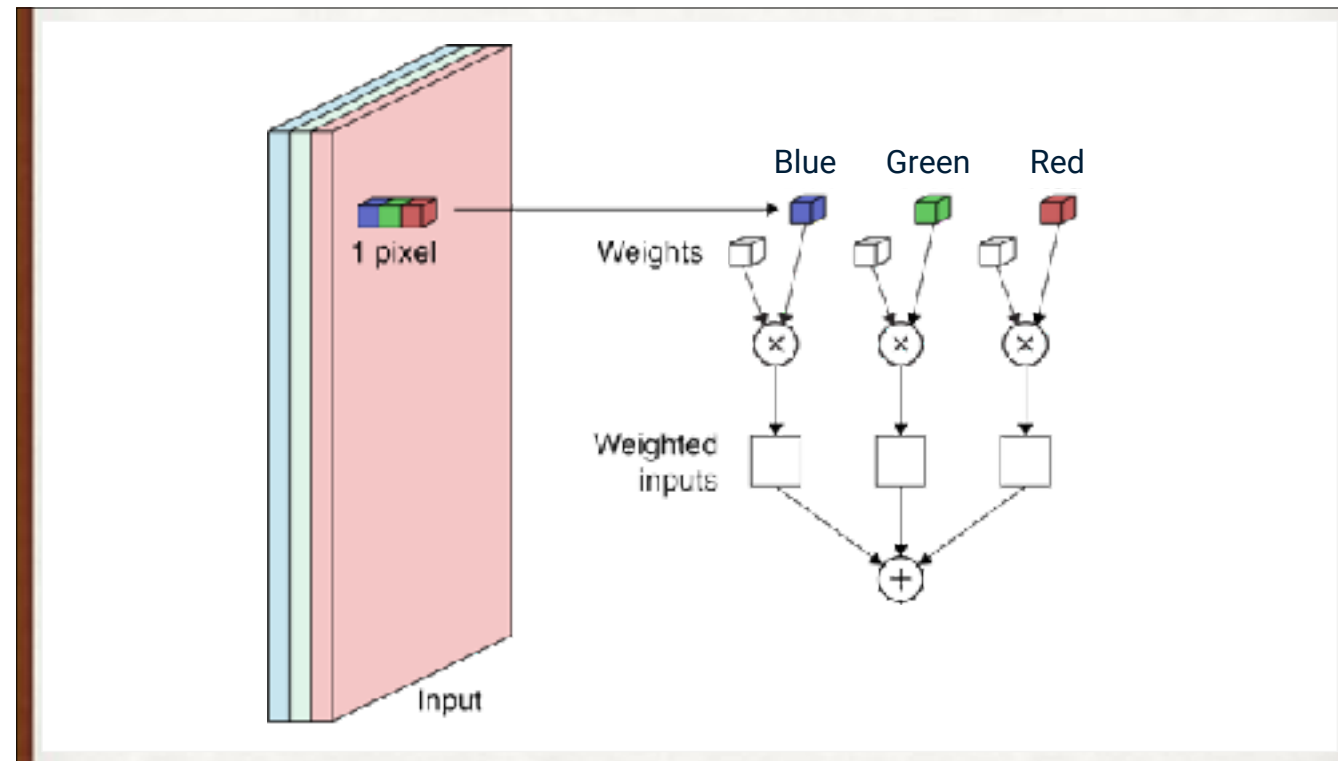
An RGB picture at left. We take a “core sample” of 1 pixel. We multiply the red, green, and blue values by associated weights. Then we add up the results. For our purposes, this is “convolution”. We say that we have “convolved” the input (the core sample) with a filter (the three weights).



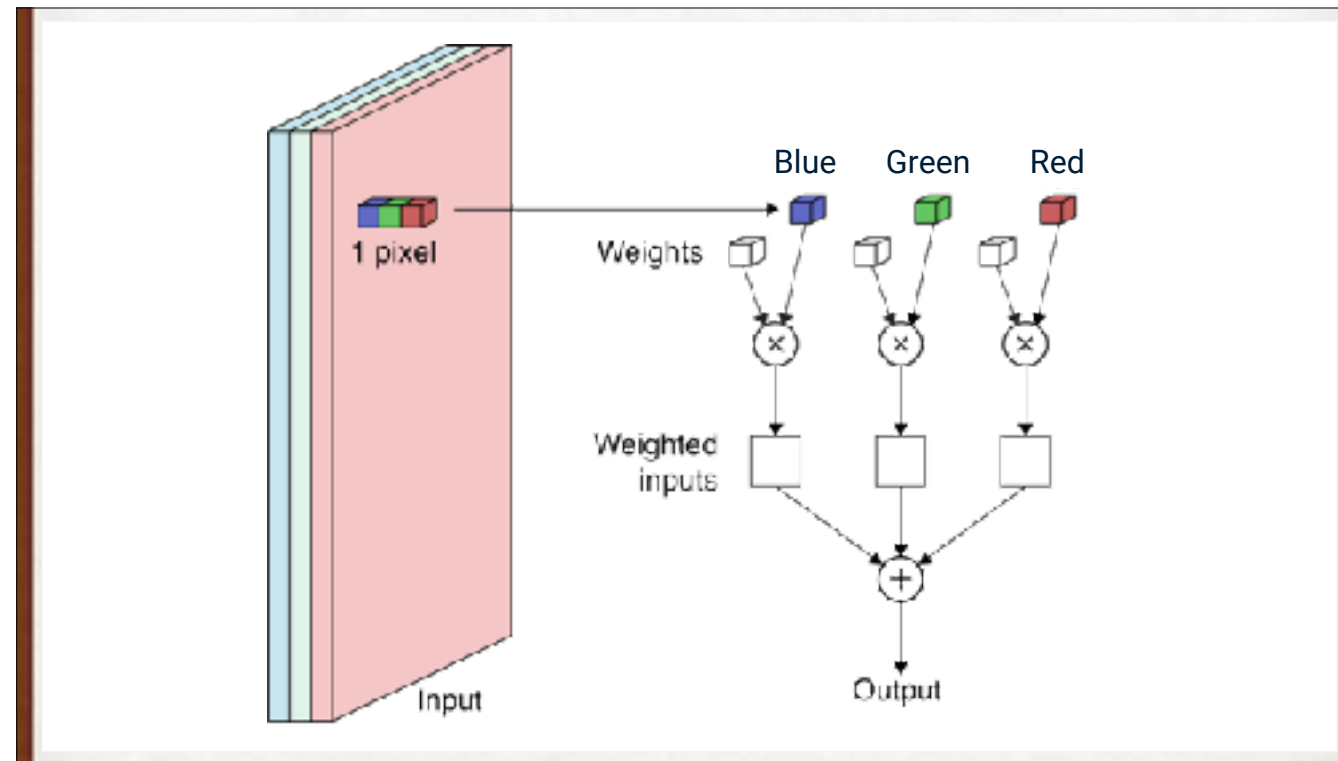
An RGB picture at left. We take a “core sample” of 1 pixel. We multiply the red, green, and blue values by associated weights. Then we add up the results. For our purposes, this is “convolution”. We say that we have “convolved” the input (the core sample) with a filter (the three weights).



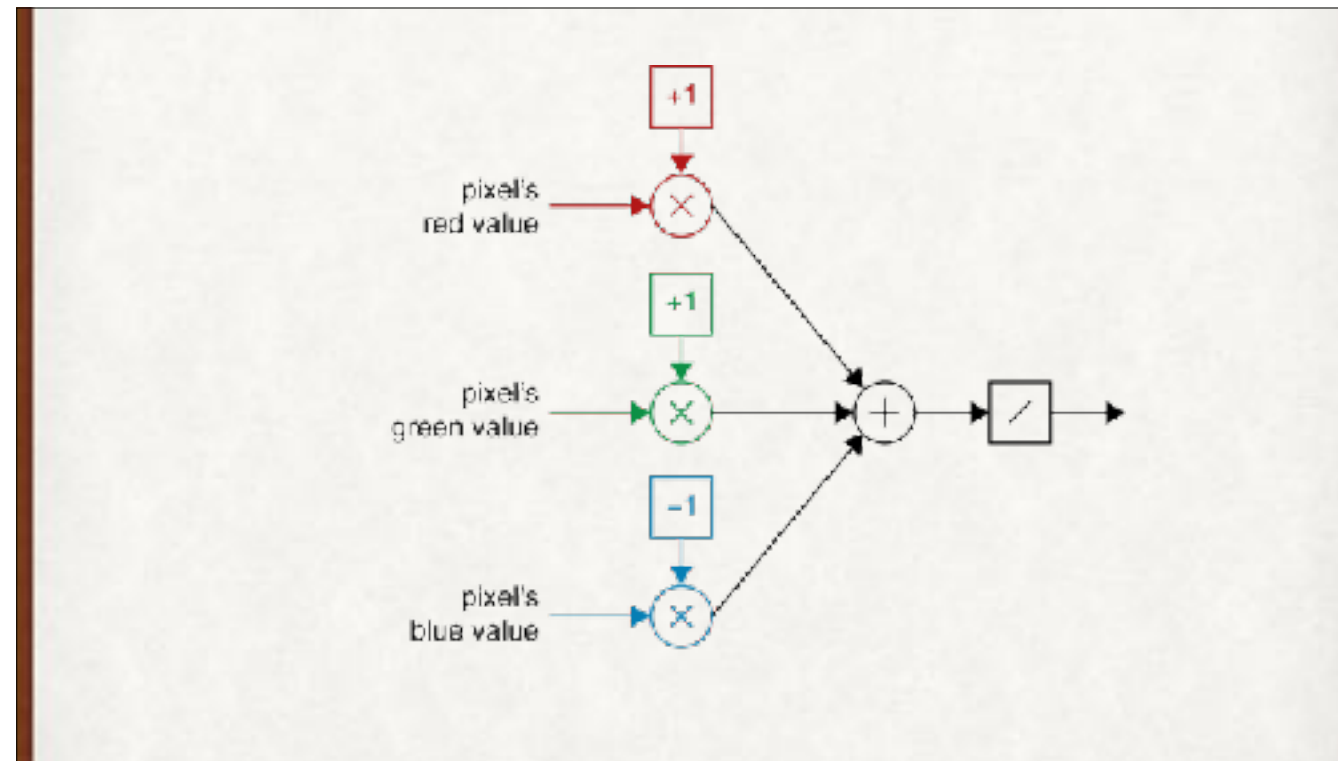
An RGB picture at left. We take a “core sample” of 1 pixel. We multiply the red, green, and blue values by associated weights. Then we add up the results. For our purposes, this is “convolution”. We say that we have “convolved” the input (the core sample) with a filter (the three weights).



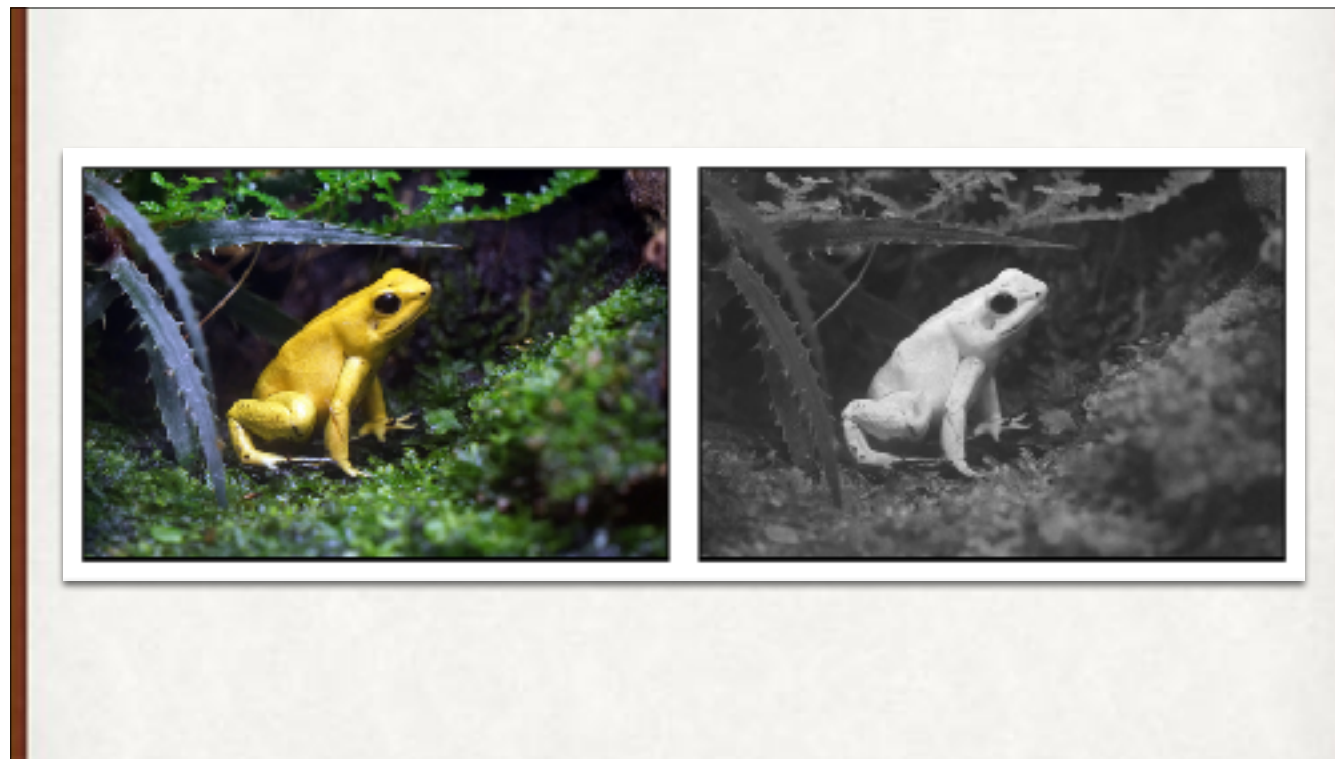
An RGB picture at left. We take a “core sample” of 1 pixel. We multiply the red, green, and blue values by associated weights. Then we add up the results. For our purposes, this is “convolution”. We say that we have “convolved” the input (the core sample) with a filter (the three weights).



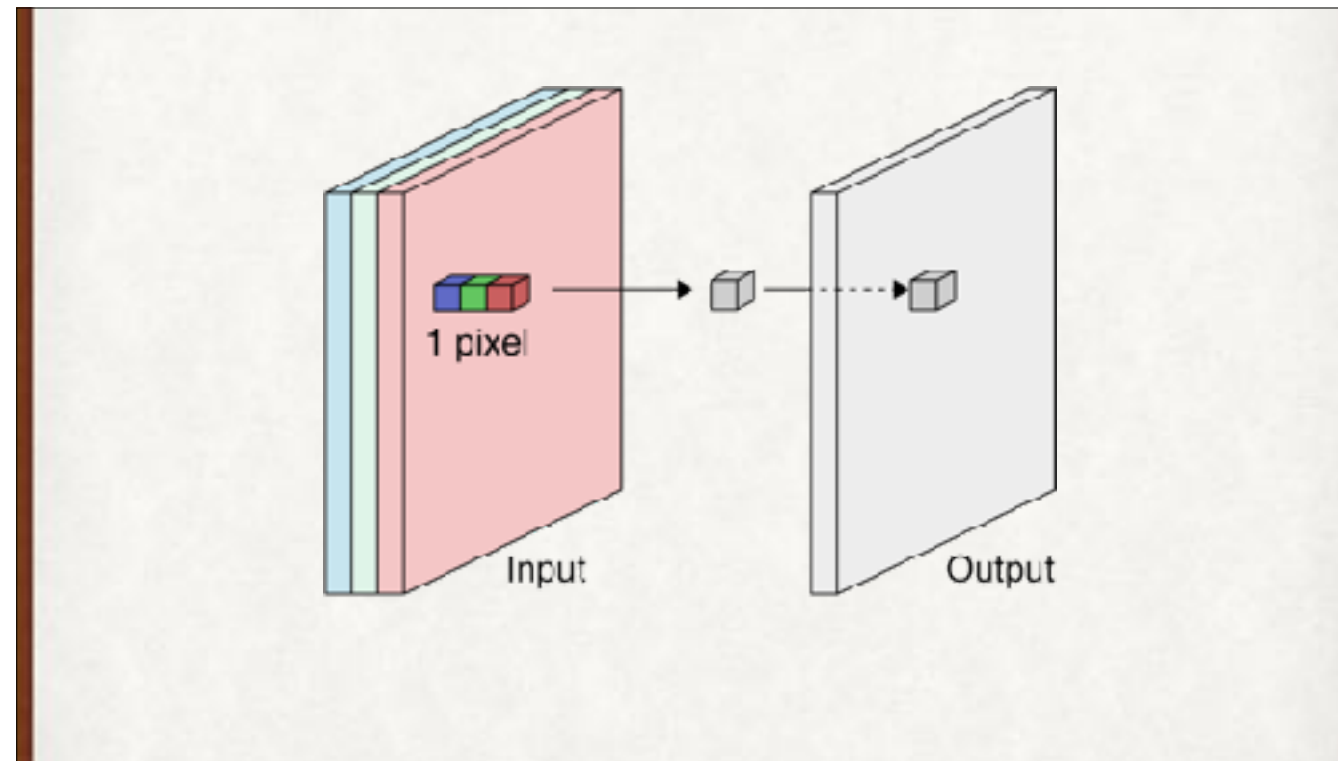
An RGB picture at left. We take a “core sample” of 1 pixel. We multiply the red, green, and blue values by associated weights. Then we add up the results. For our purposes, this is “convolution”. We say that we have “convolved” the input (the core sample) with a filter (the three weights).



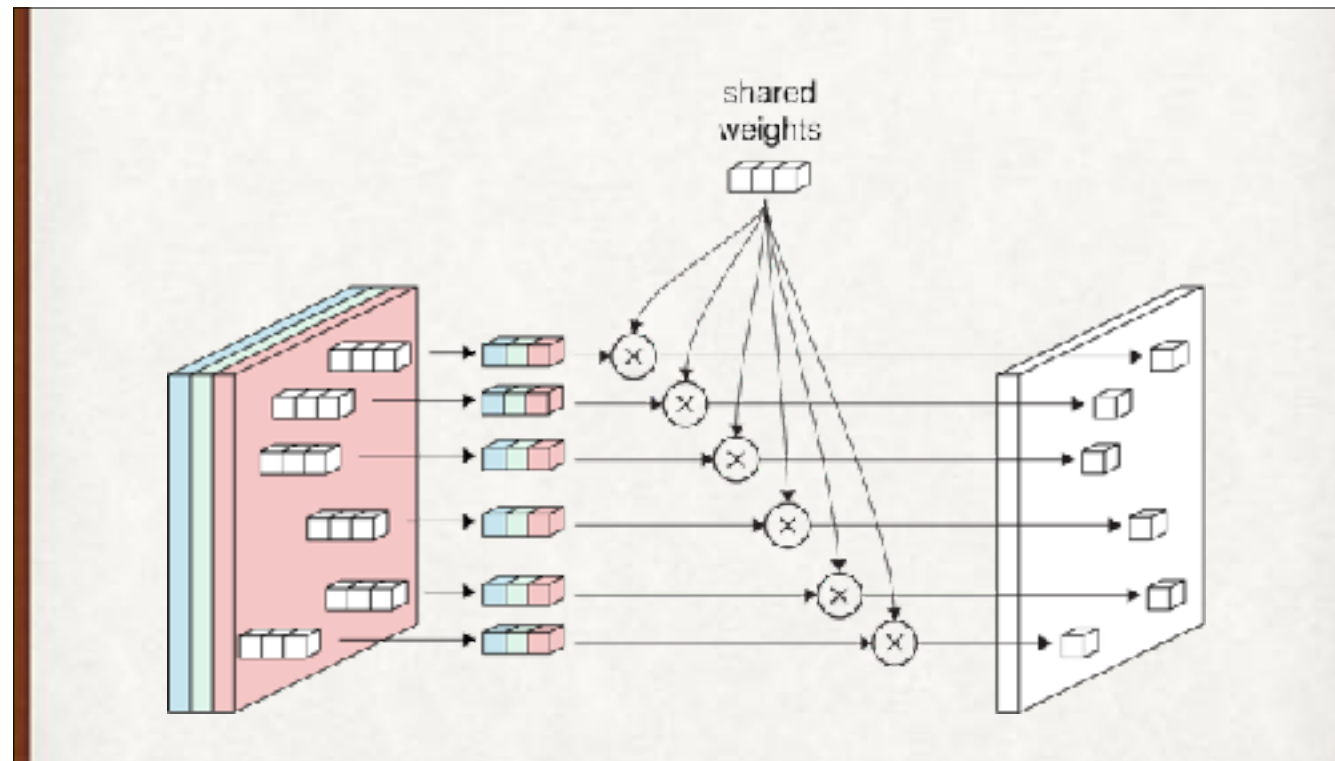
Using convolution to find yellow pixels (red=1, green=1, blue=0). The more yellow a pixel, the greater the output, to a maximum of +2. This looks a lot like an artificial neuron, or perceptron! The activation function in this case is just a line at 45 degrees (which is a bad idea in general, since it 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.



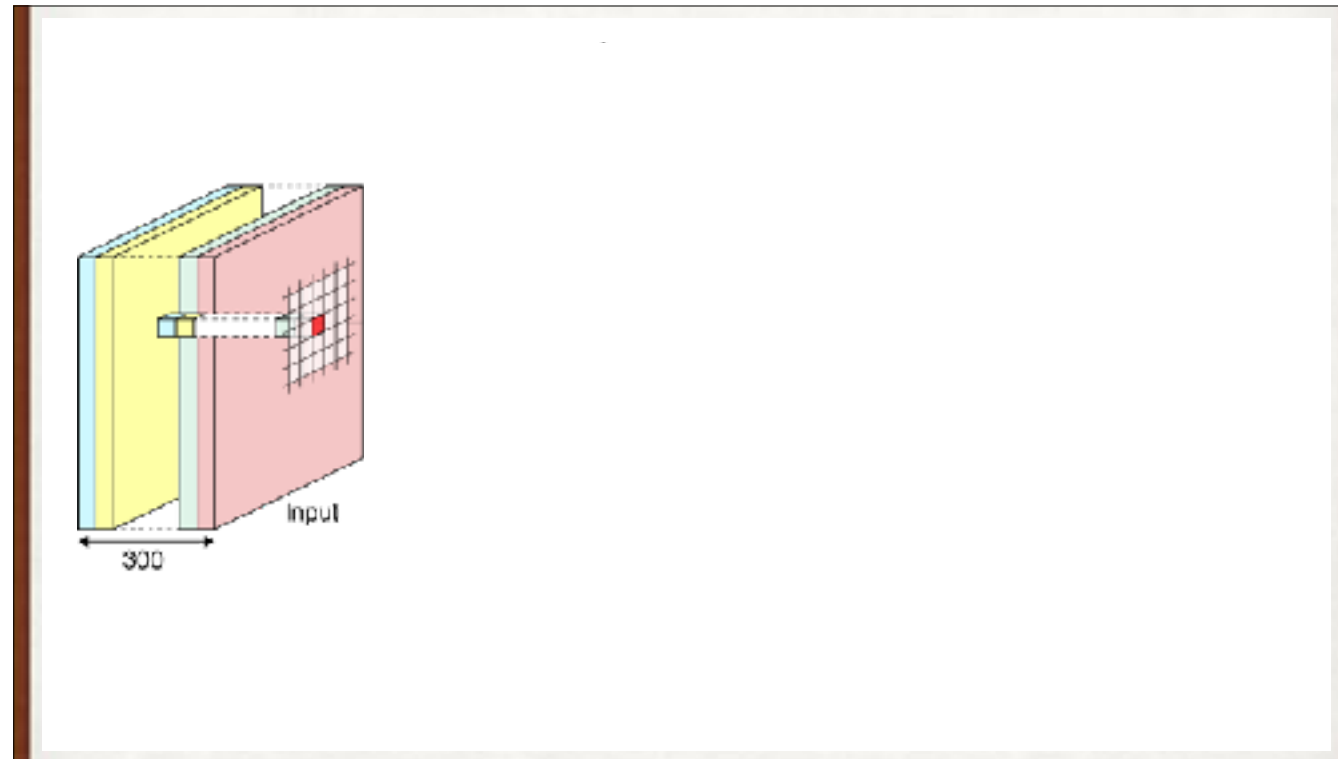
Our yellow filter applied to a frog, with the results scaled to 0=black, 2=white.



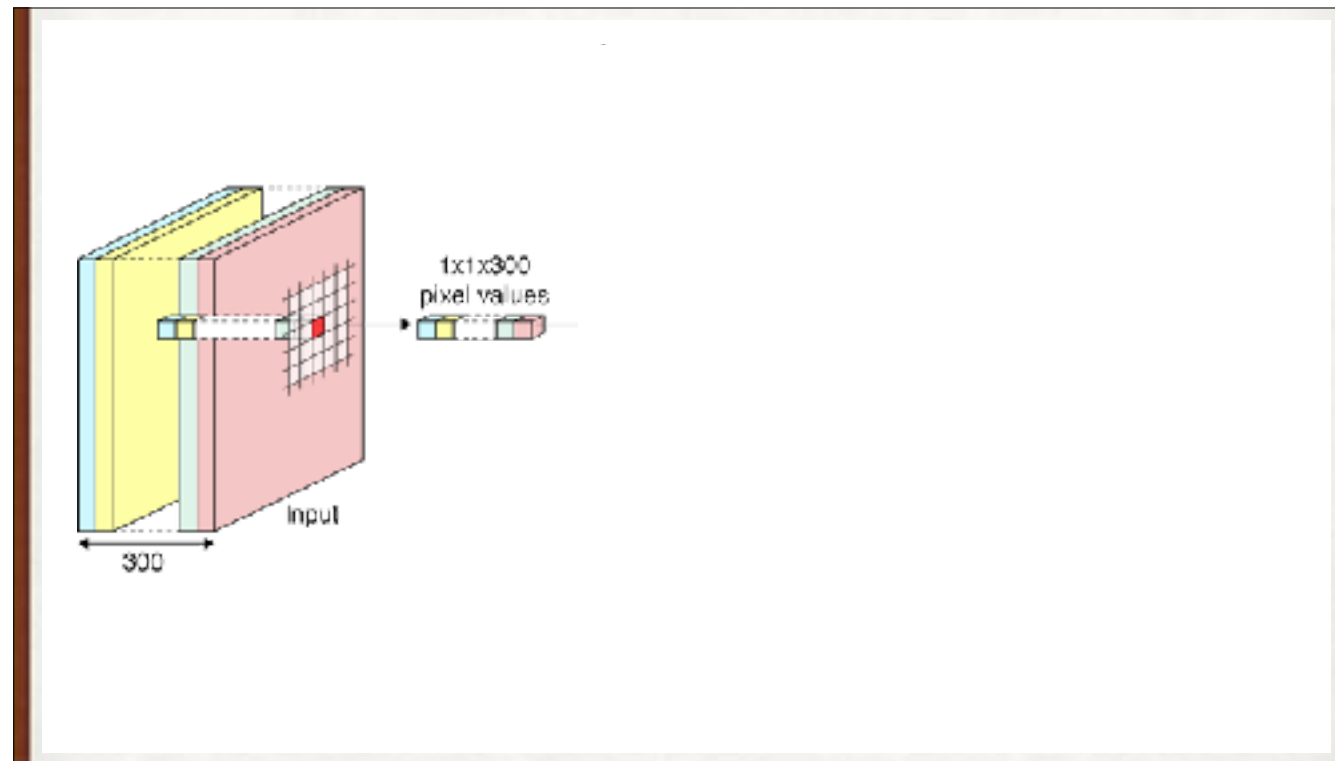
We'll save our weighted, summed core sample in an output image of just one channel (such as grayscale).



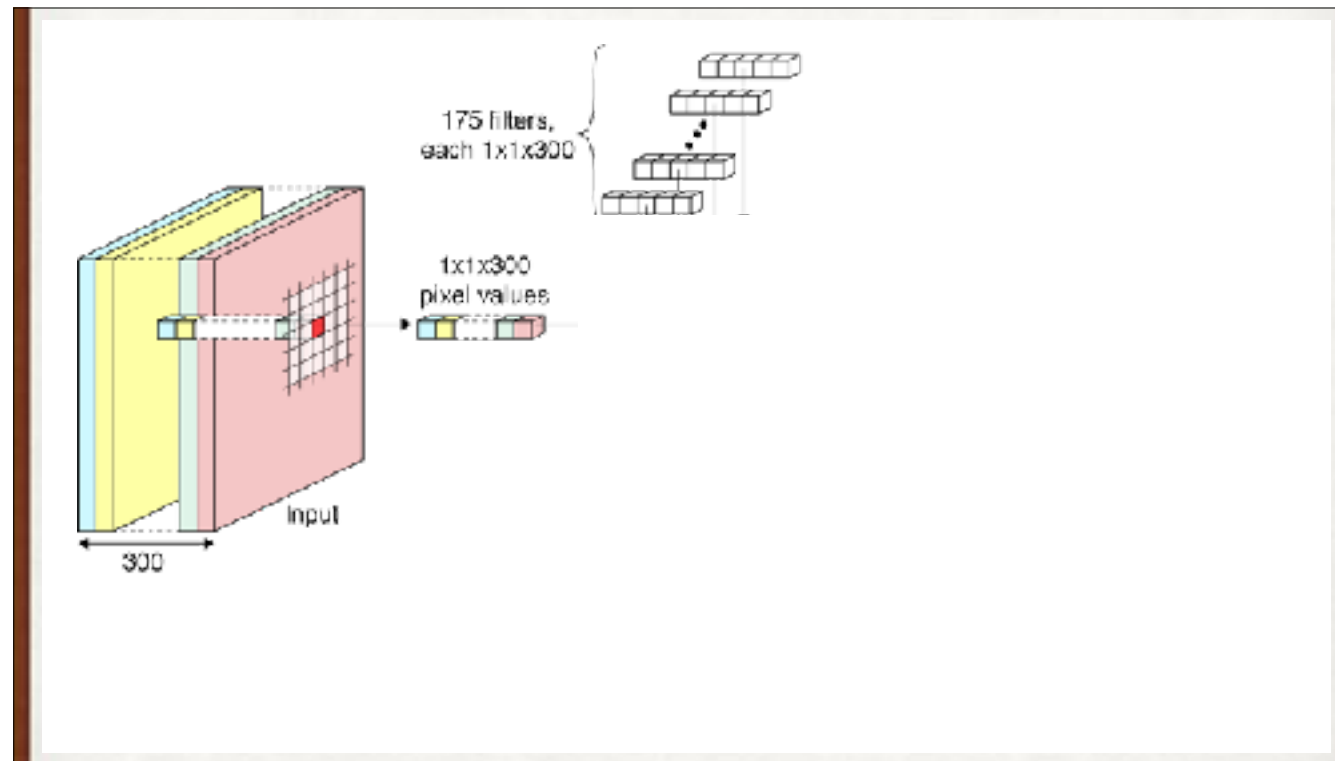
We apply the same weights to every pixel. Each of these “weight and add” convolution steps is using the same weights. We say they’re “sharing” those weights.



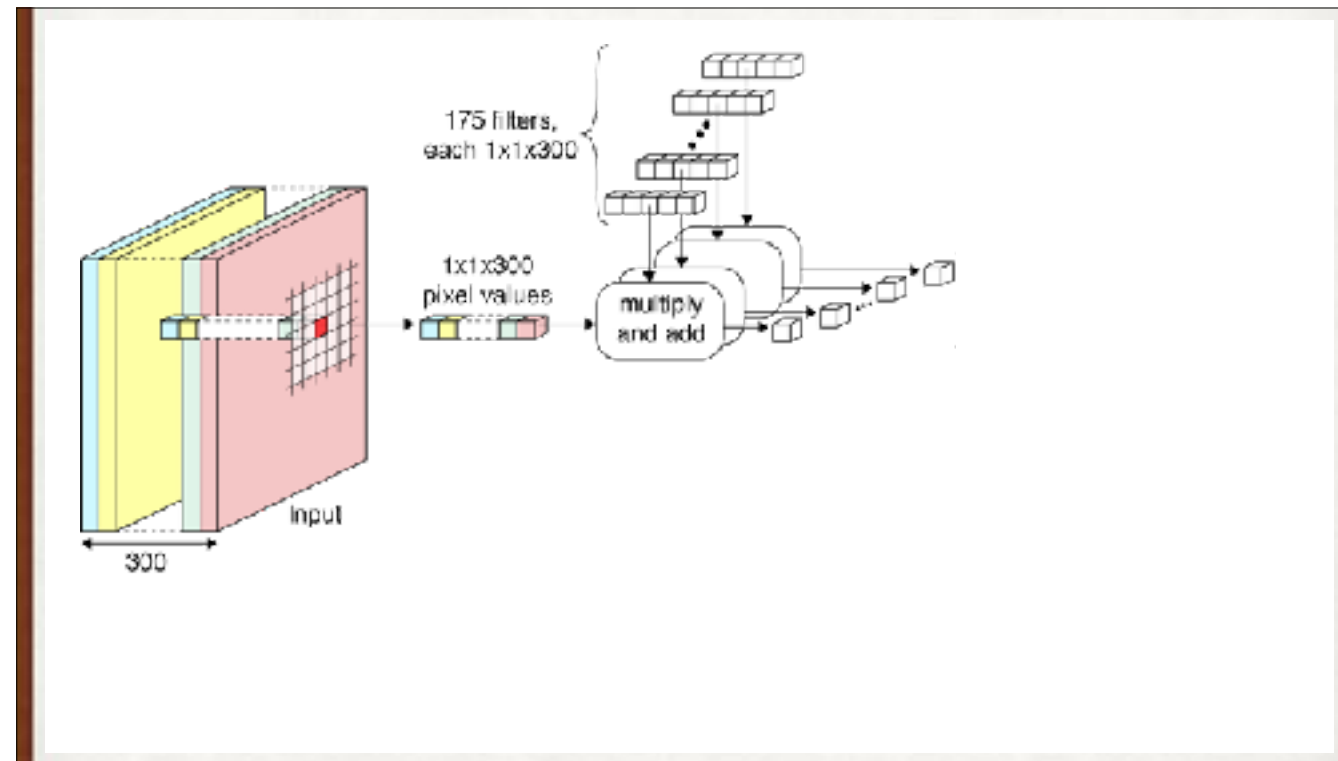
We're not limited to RGB inputs with 3 layers. The input here has 300 layers, so each "core sample" is 1 by 1 by 300. So each filter also has 300 values. We multiply and add, or convolve, the input and filter to get a single number. Here we have 175 filters, so we'll get back 175 numbers. Those get stored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



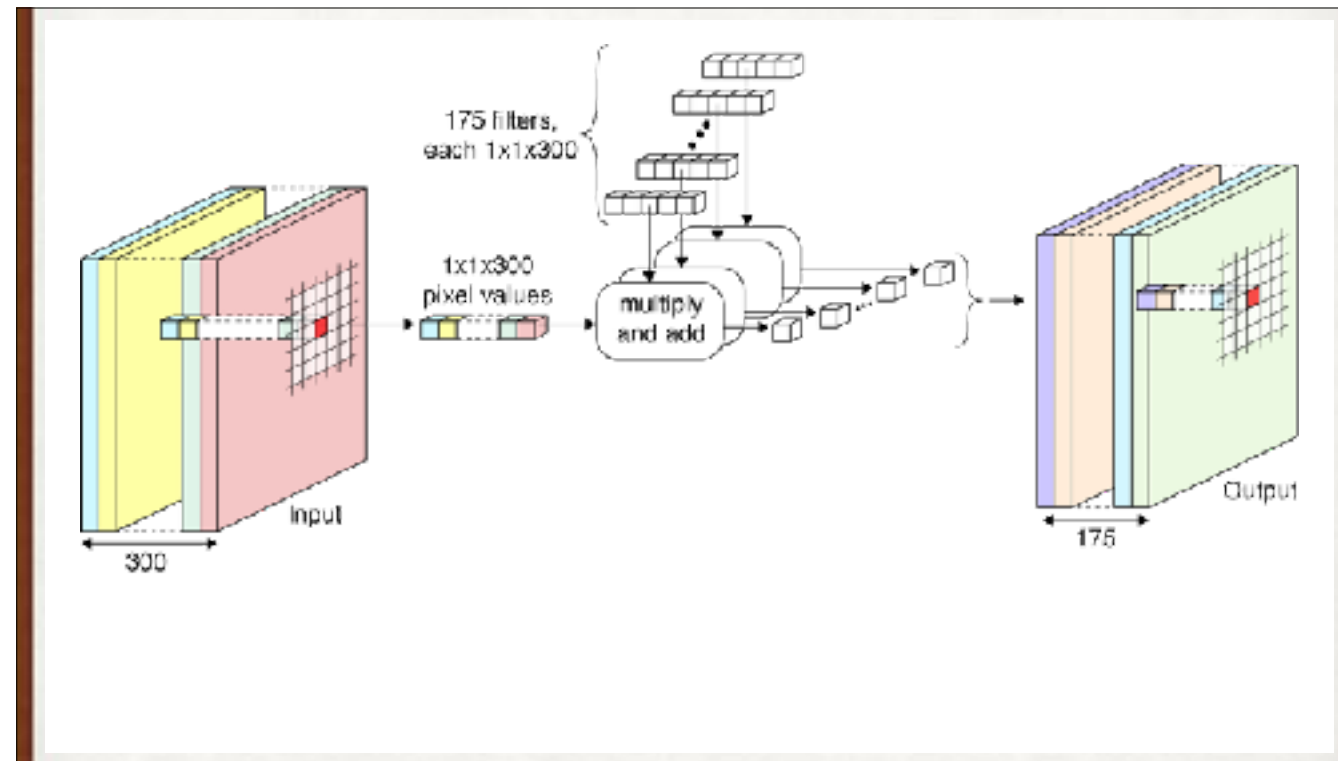
We're not limited to RGB inputs with 3 layers. The input here has 300 layers, so each "core sample" is 1 by 1 by 300. So each filter also has 300 values. We multiply and add, or convolve, the input and filter to get a single number. Here we have 175 filters, so we'll get back 175 numbers. Those get stored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



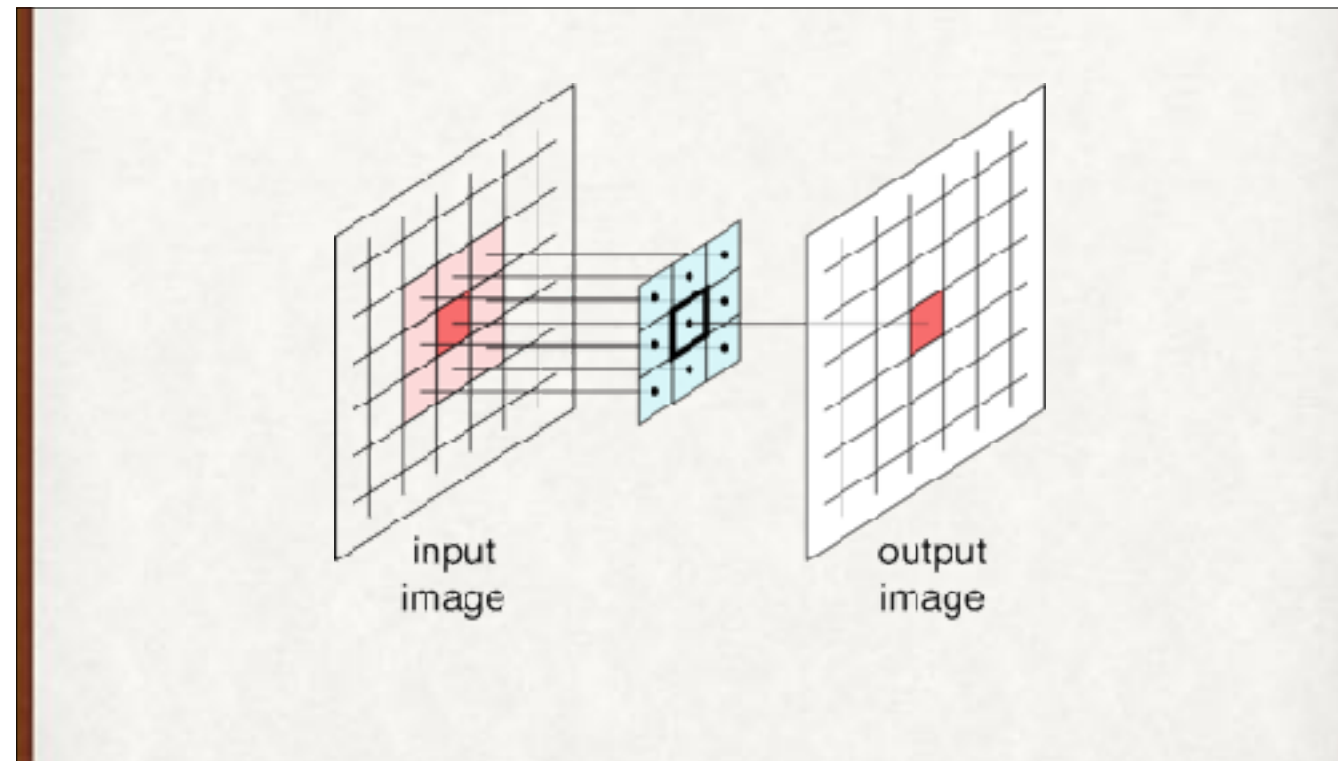
We're not limited to RGB inputs with 3 layers. The input here has 300 layers, so each "core sample" is 1 by 1 by 300. So each filter also has 300 values. We multiply and add, or convolve, the input and filter to get a single number. Here we have 175 filters, so we'll get back 175 numbers. Those get stored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



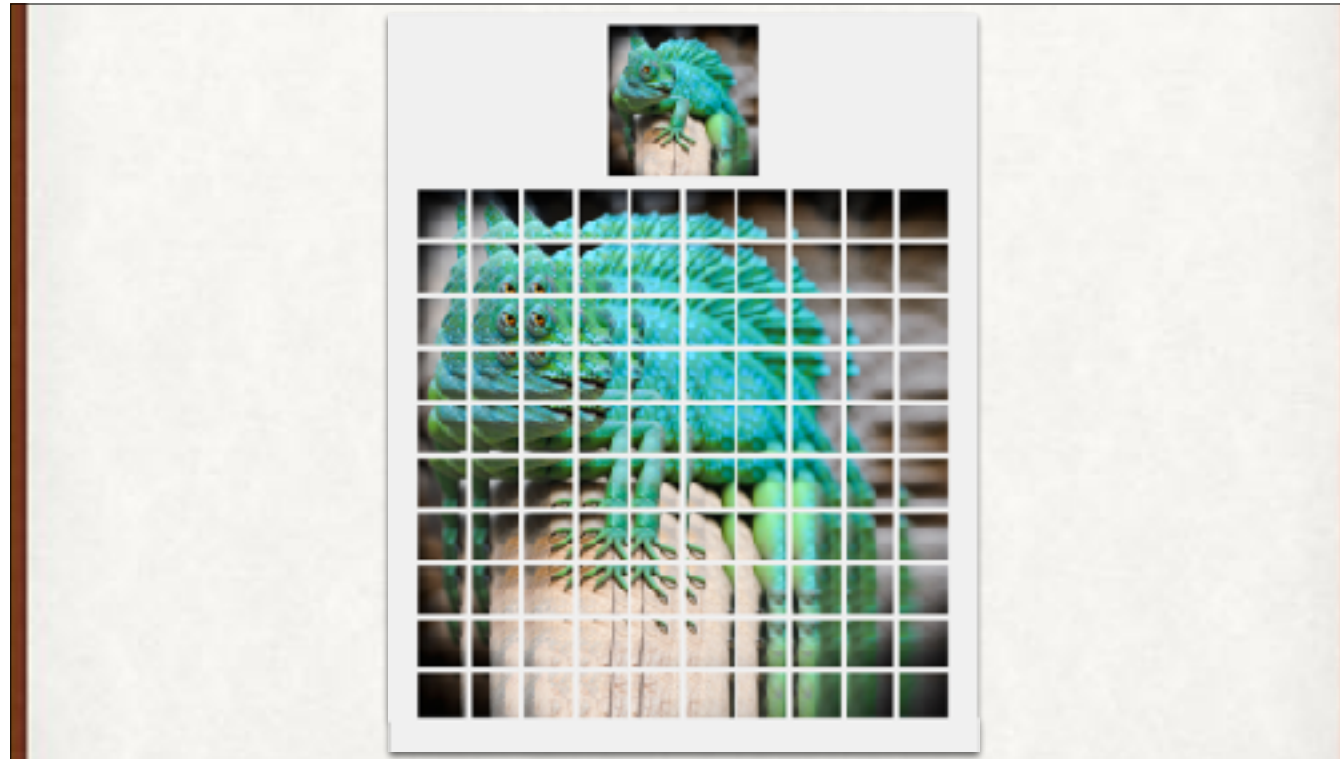
We're not limited to RGB inputs with 3 layers. The input here has 300 layers, so each "core sample" is 1 by 1 by 300. So each filter also has 300 values. We multiply and add, or convolve, the input and filter to get a single number. Here we have 175 filters, so we'll get back 175 numbers. Those get stored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



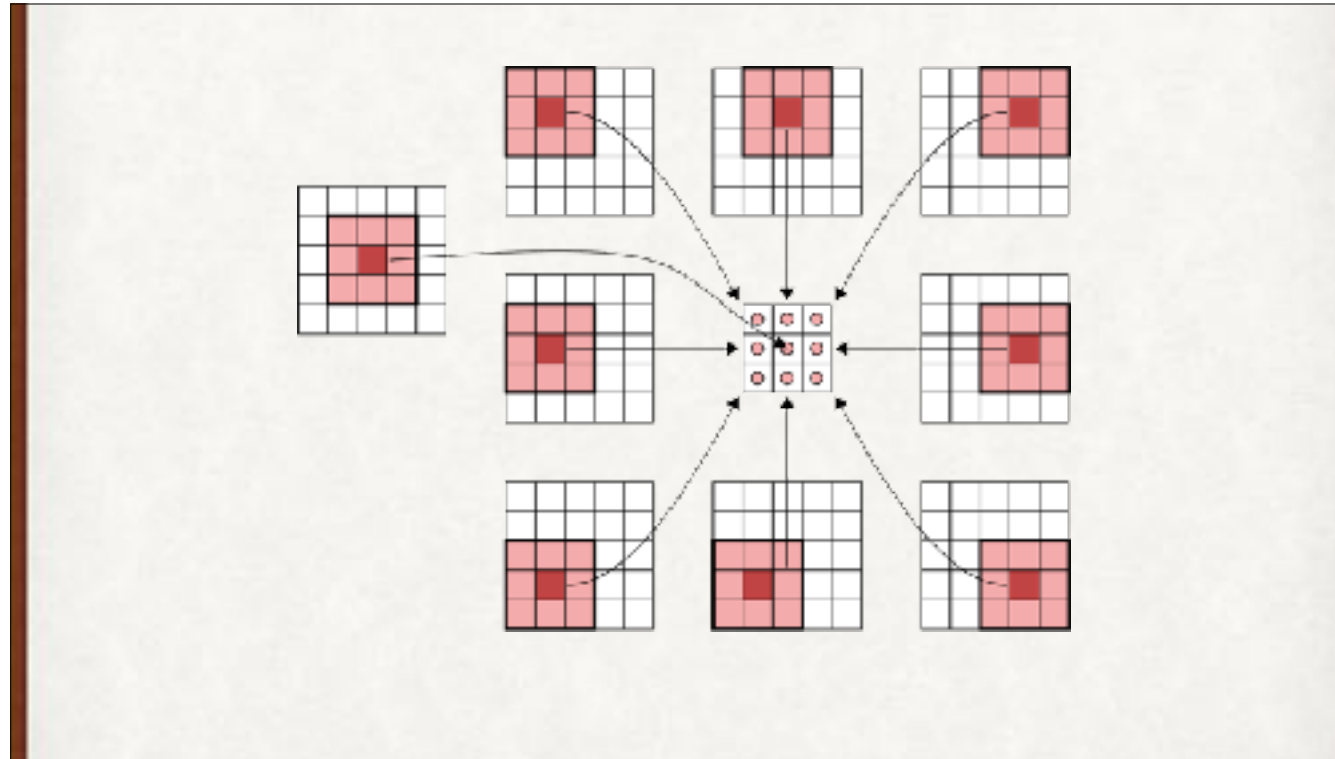
We're not limited to RGB inputs with 3 layers. The input here has 300 layers, so each "core sample" is 1 by 1 by 300. So each filter also has 300 values. We multiply and add, or convolve, the input and filter to get a single number. Here we have 175 filters, so we'll get back 175 numbers. Those get stored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



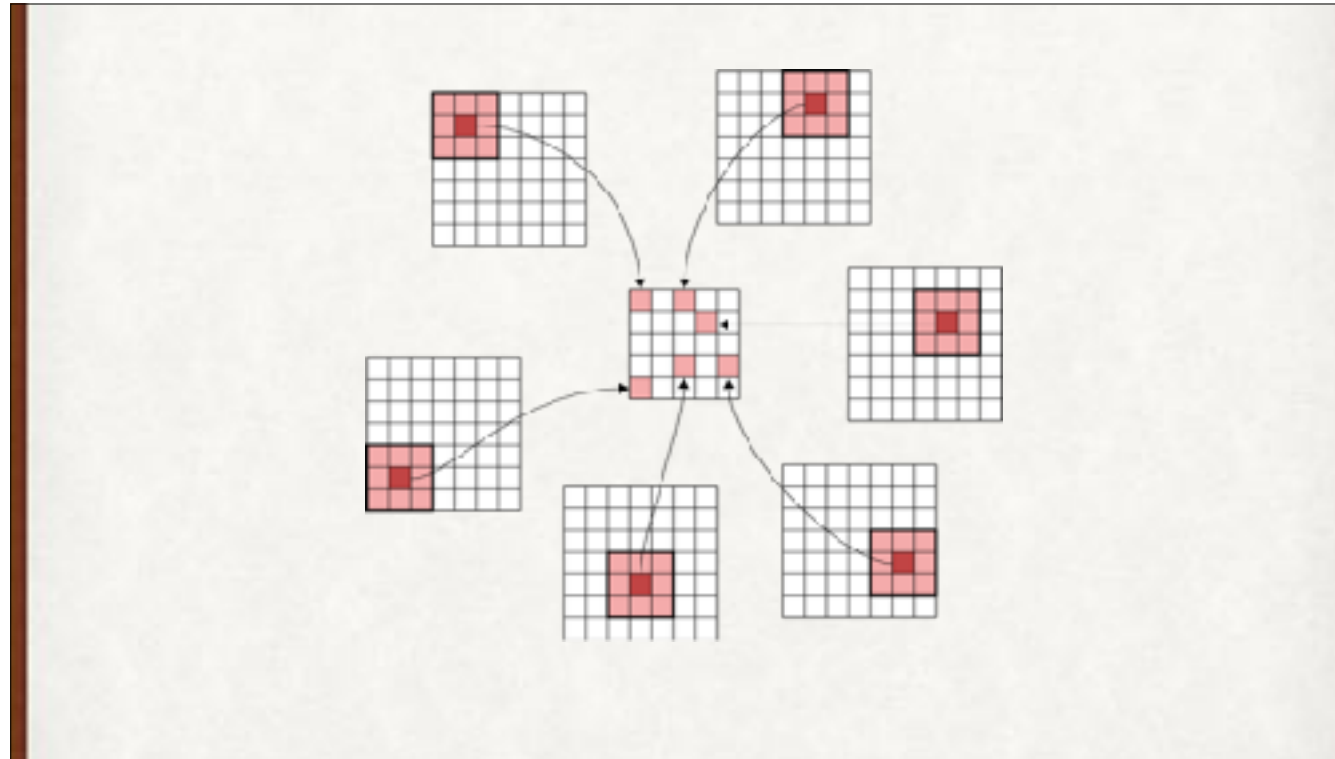
Each filter has a “footprint,” or the region it covers in its input. The cyan filter’s footprint is 3 by 3.



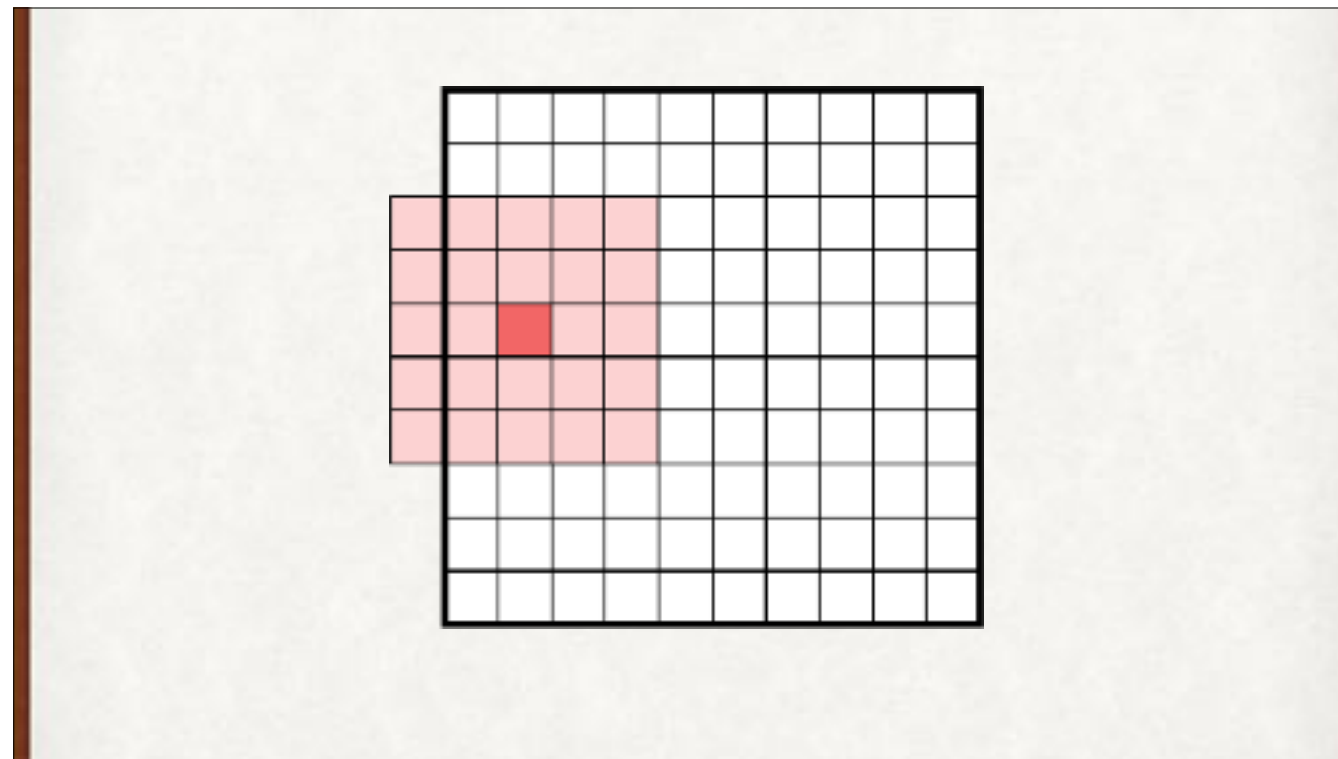
Another way to think of convolution. Split up the input image into multiple overlapping pieces, and then apply the filter to each piece.



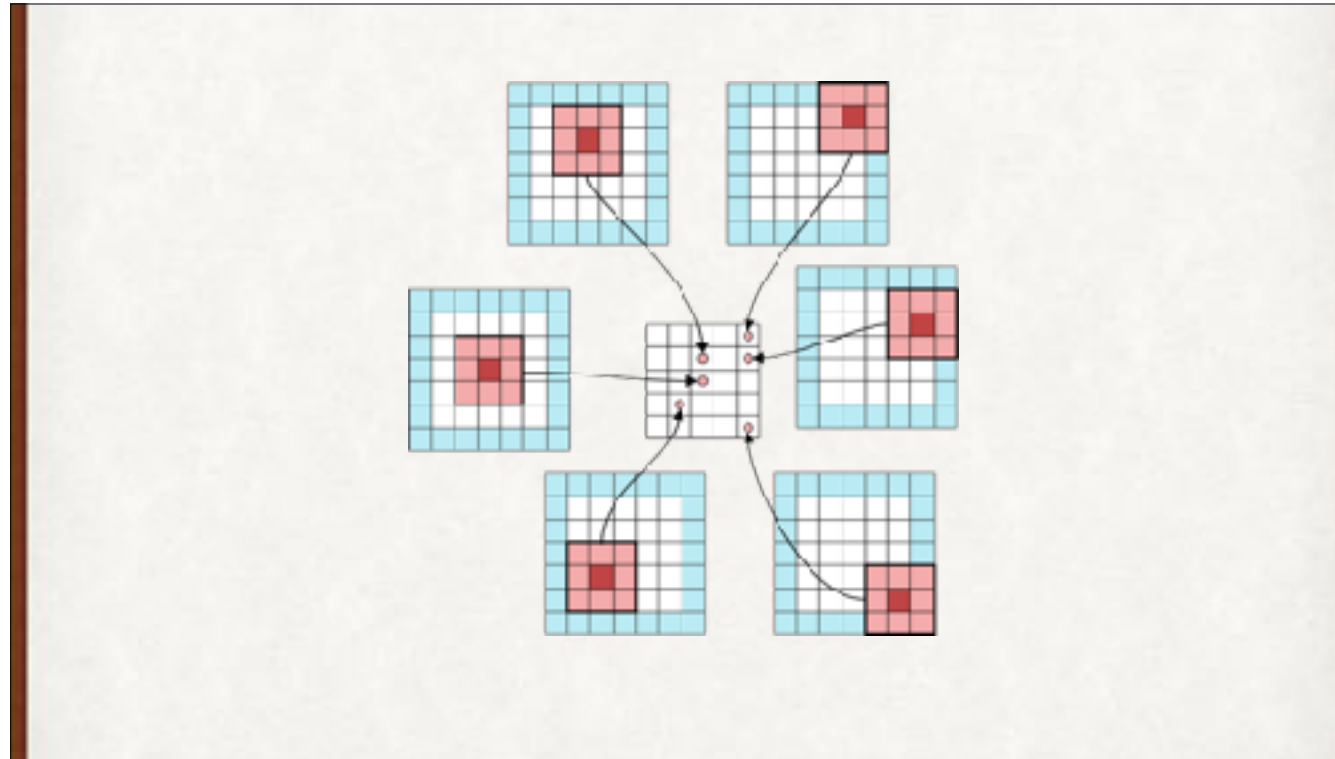
A 5 by 5 input and a 3 by 3 filter. There are only 9 places for the filter to go without falling off the edge, producing a 3 by 3 output.



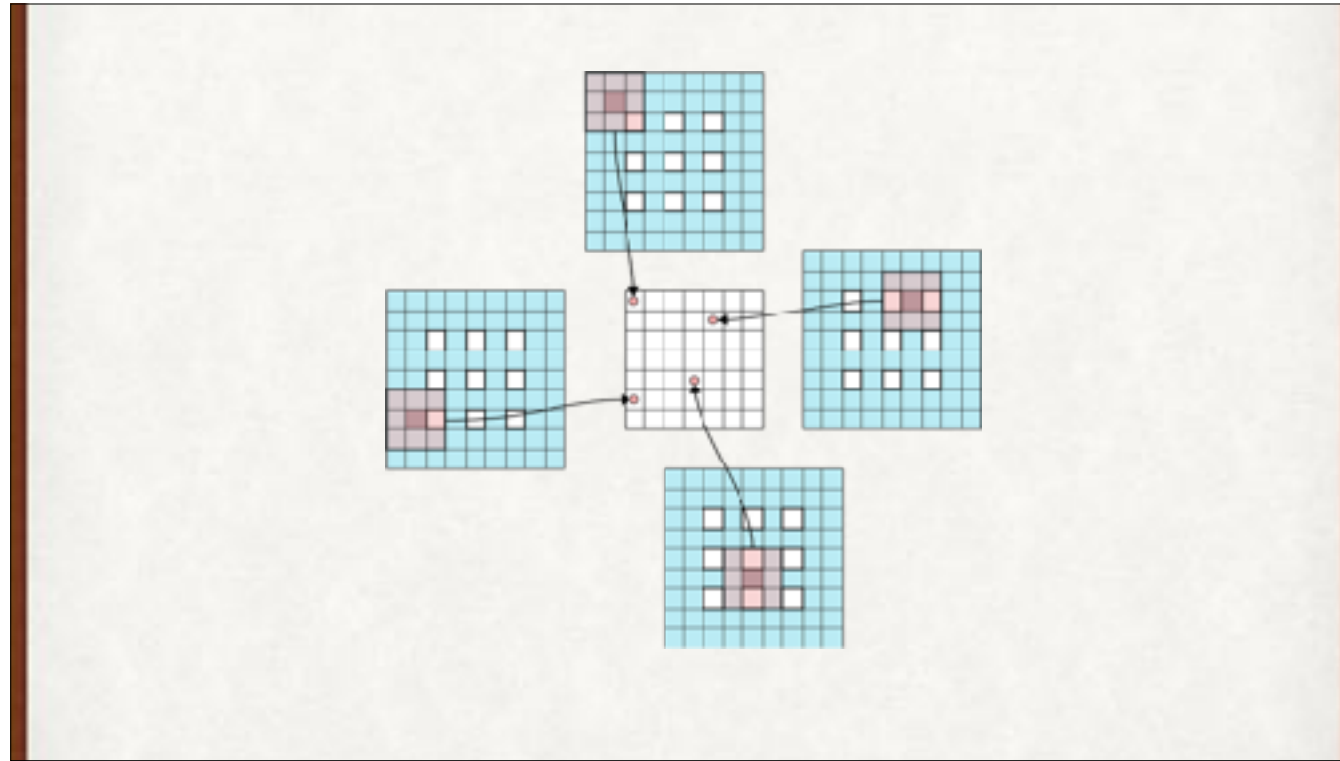
A 7 by 7 input and a 3 by 3 filter. This time there are 25 places where the filter can completely fit, and our output is 5 by 5. It would be nice to make the output have the same size as the input, so the original data doesn't shrink on every step.



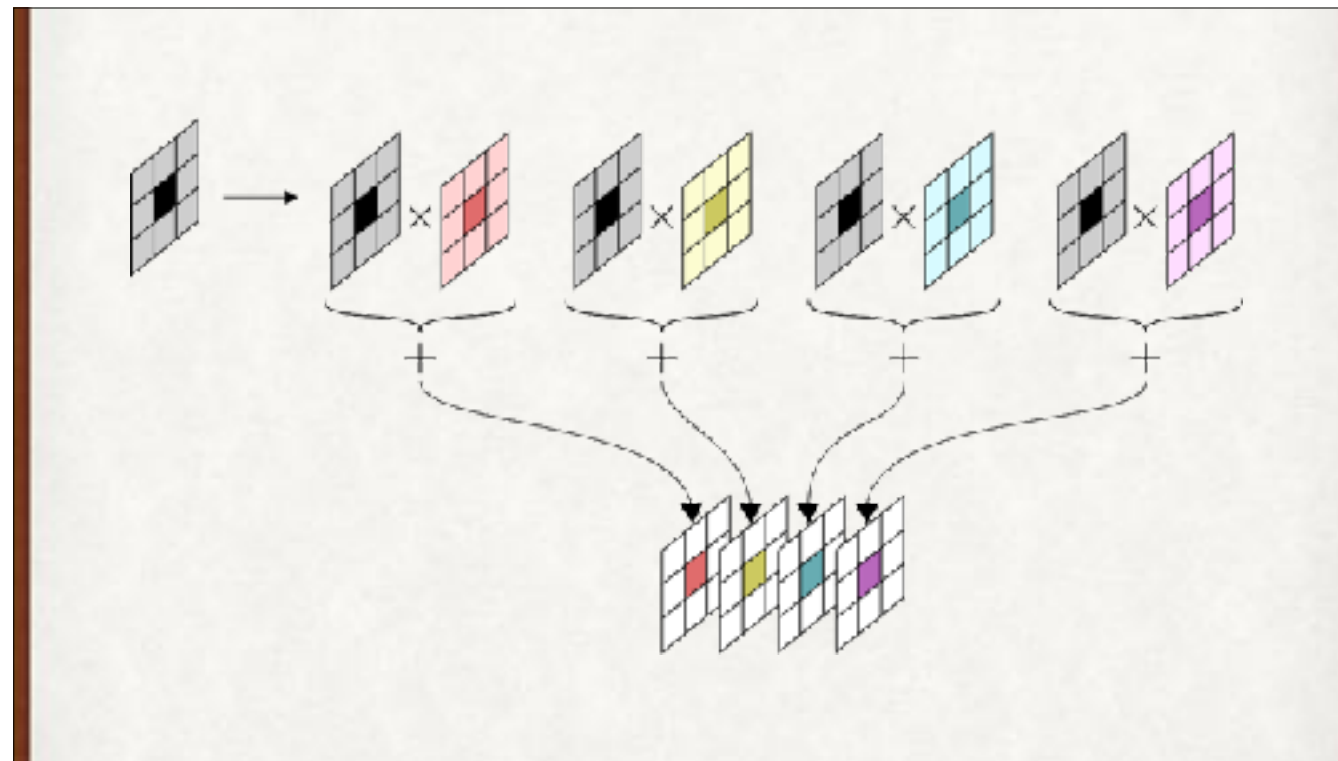
This is why we've been shrinking: we haven't let the filter fall off the edge. Is there a way to make this a valid convolution?



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.

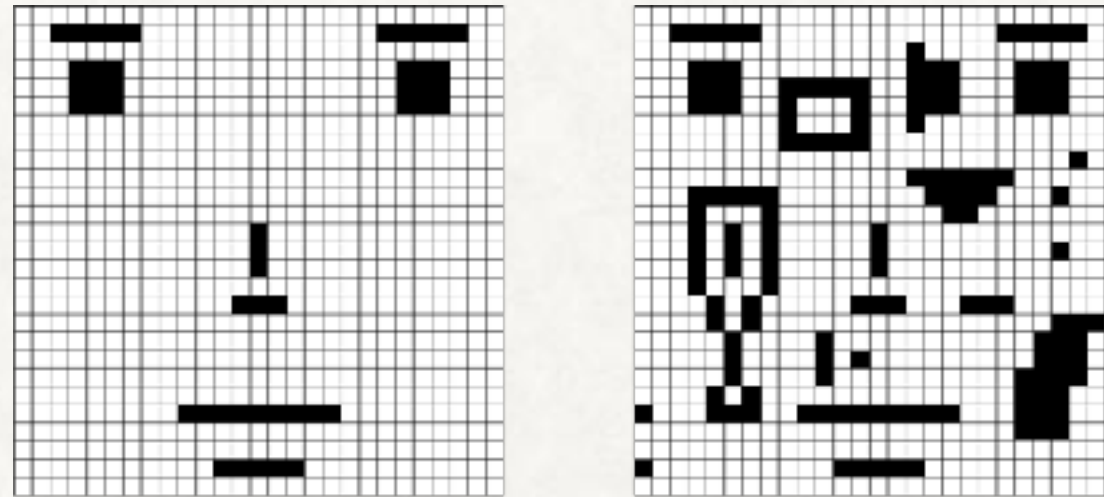


We can even make the output bigger than the input. The input here is 3 by 3. Surround it with two rings of 0's, and also place a row and column of 0's between each input element. The result is a field that's 9 by 9, but mostly zeros. Moving the filter only where it doesn't fall off the edge gives us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

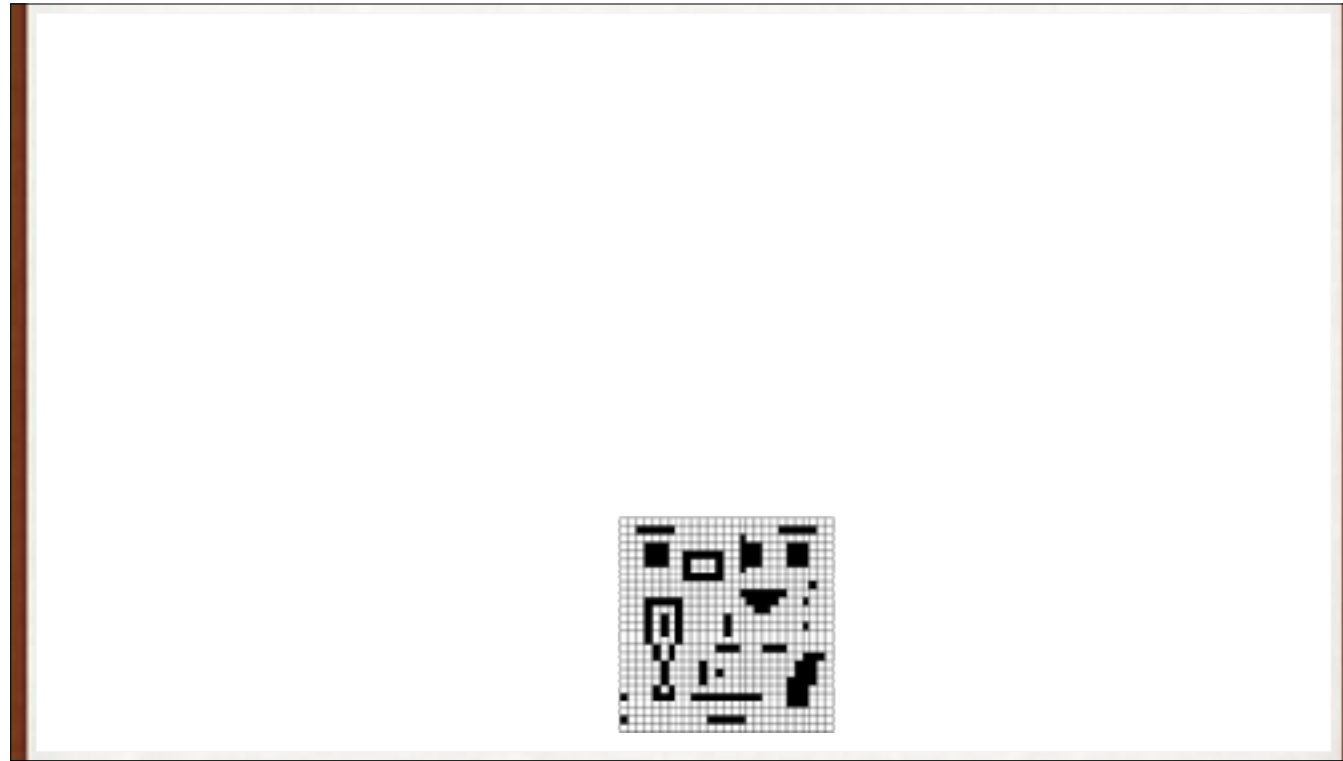


If we have multiple filters, they each produce an output for the same input. This process is simultaneous and independent. Hey, that sounds like something a GPU could do in parallel! It is!

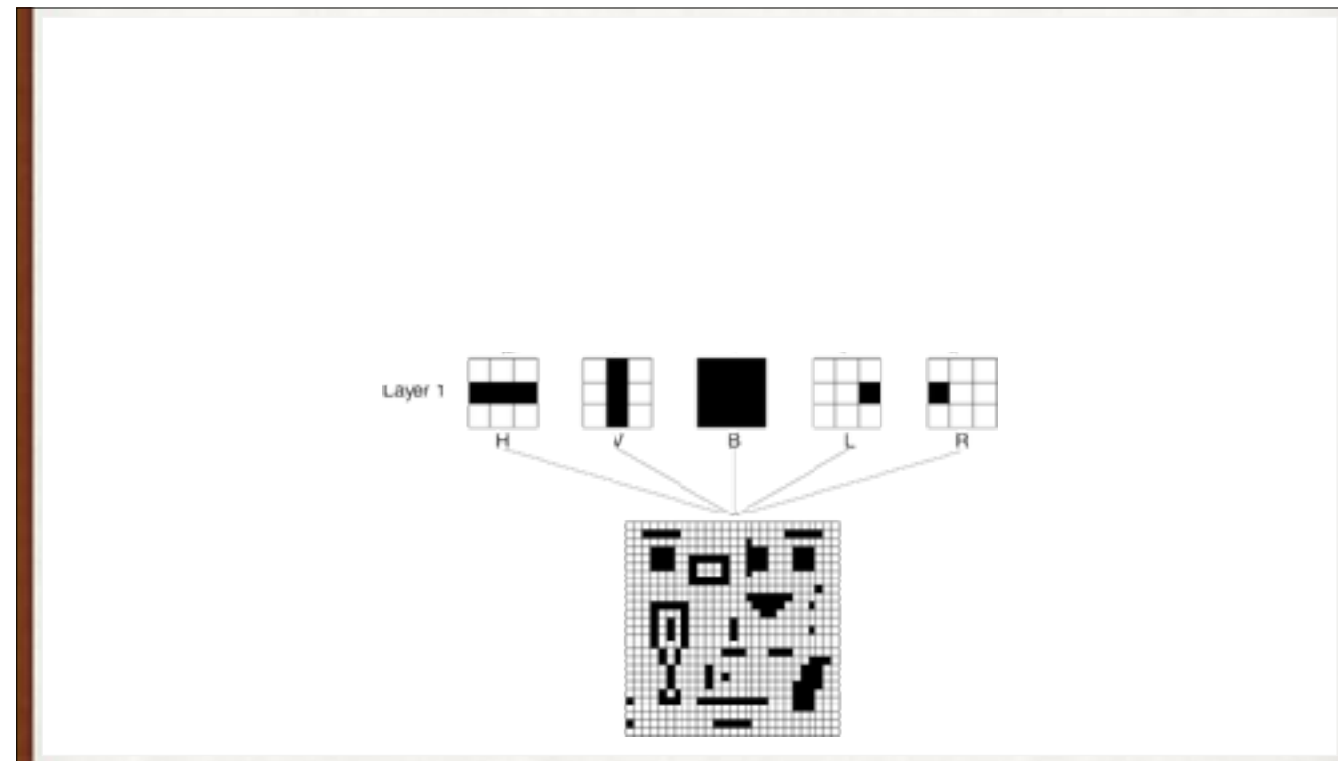
Hierarchy of Filters



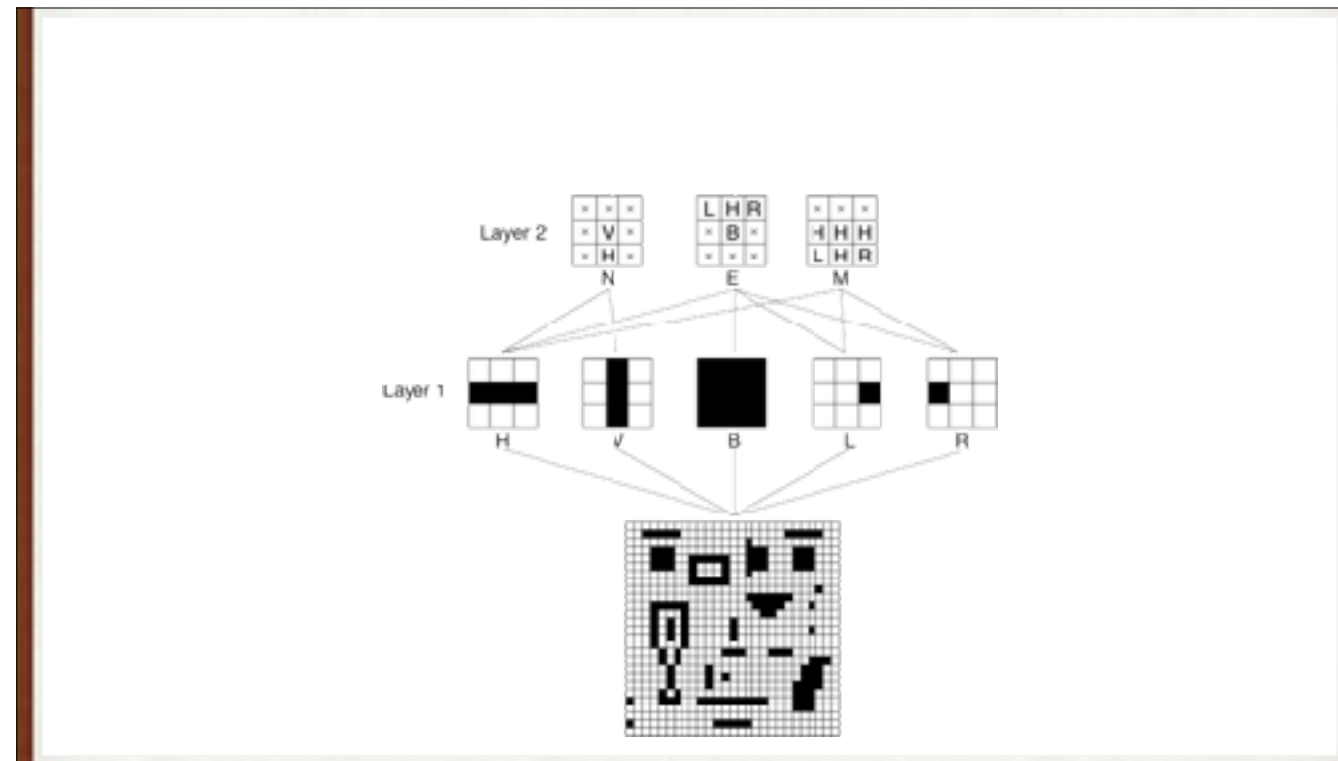
Is the face on the left contained in the mask on the right? We can tell by just checking every pixel, but let's look at how to answer the question with convolution. Then we can use that technique for harder questions.



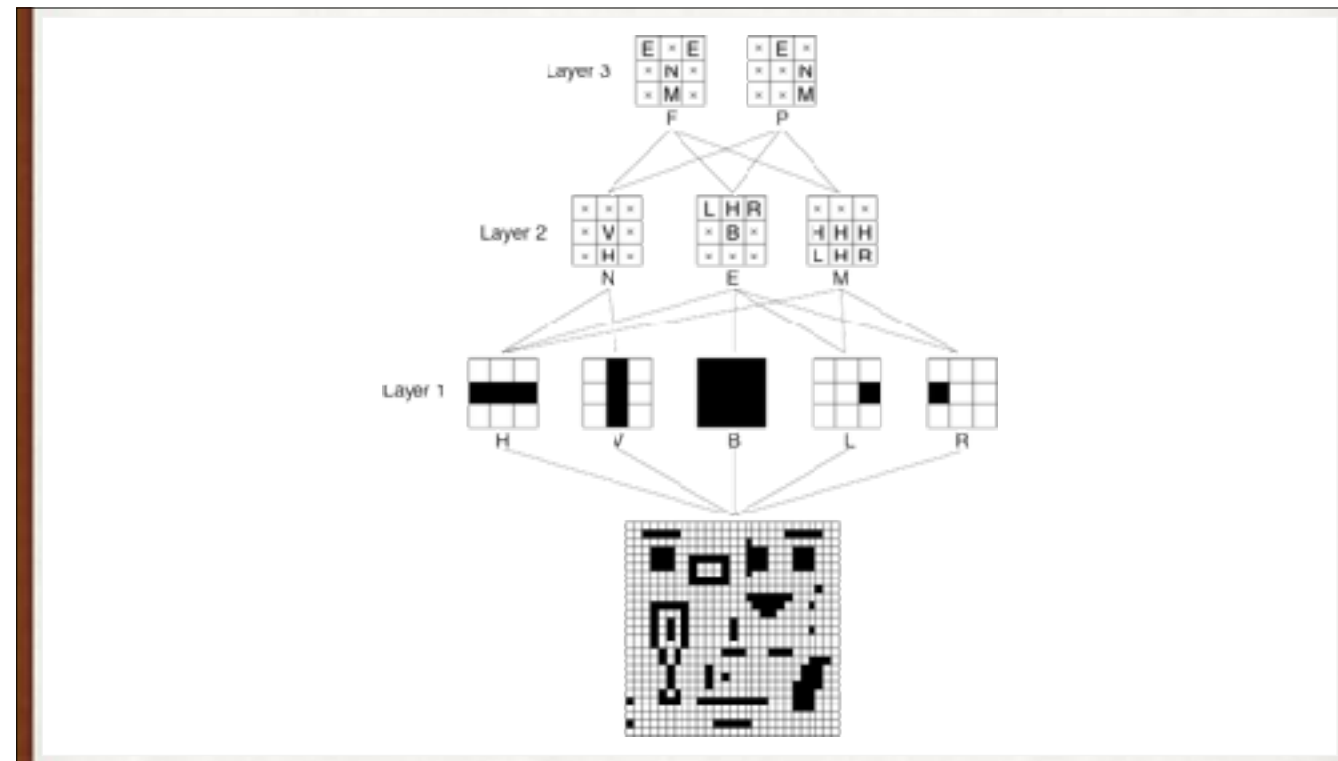
Layer 1 has 5 filters of 3 by 3 each (Horizontal, Vertical, Black, Left, Right). Layer 2 has three filters of 3 by 3, each looking for some arrangement of the Layer 1 filters (Nose, Eye, Mouth) Layer 3, in turn, is looking for arrangements of Layer 2 filters (Forward, Profile). We have a hierarchy of scales.



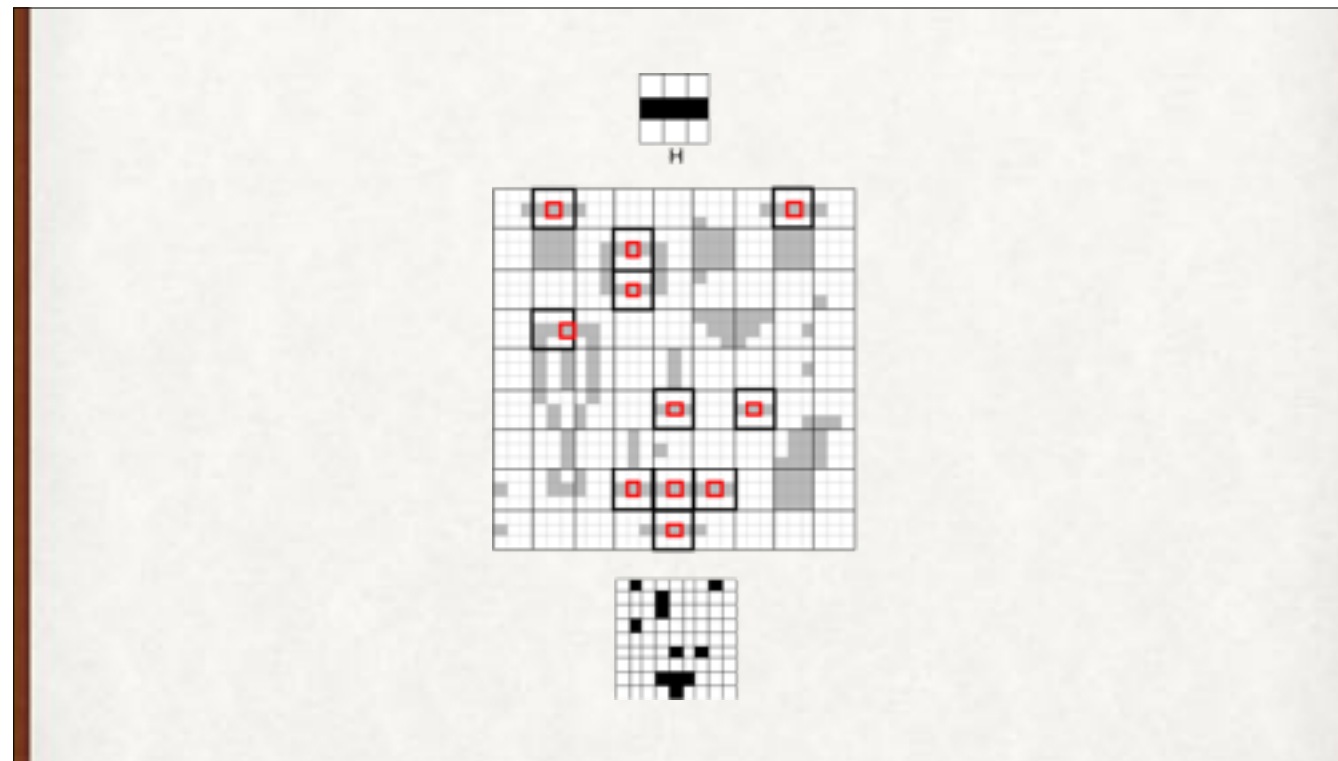
Layer 1 has 5 filters of 3 by 3 each (Horizontal, Vertical, Black, Left, Right). Layer 2 has three filters of 3 by 3, each looking for some arrangement of the Layer 1 filters (Nose, Eye, Mouth) Layer 3, in turn, is looking for arrangements of Layer 2 filters (Forward, Profile). We have a hierarchy of scales.



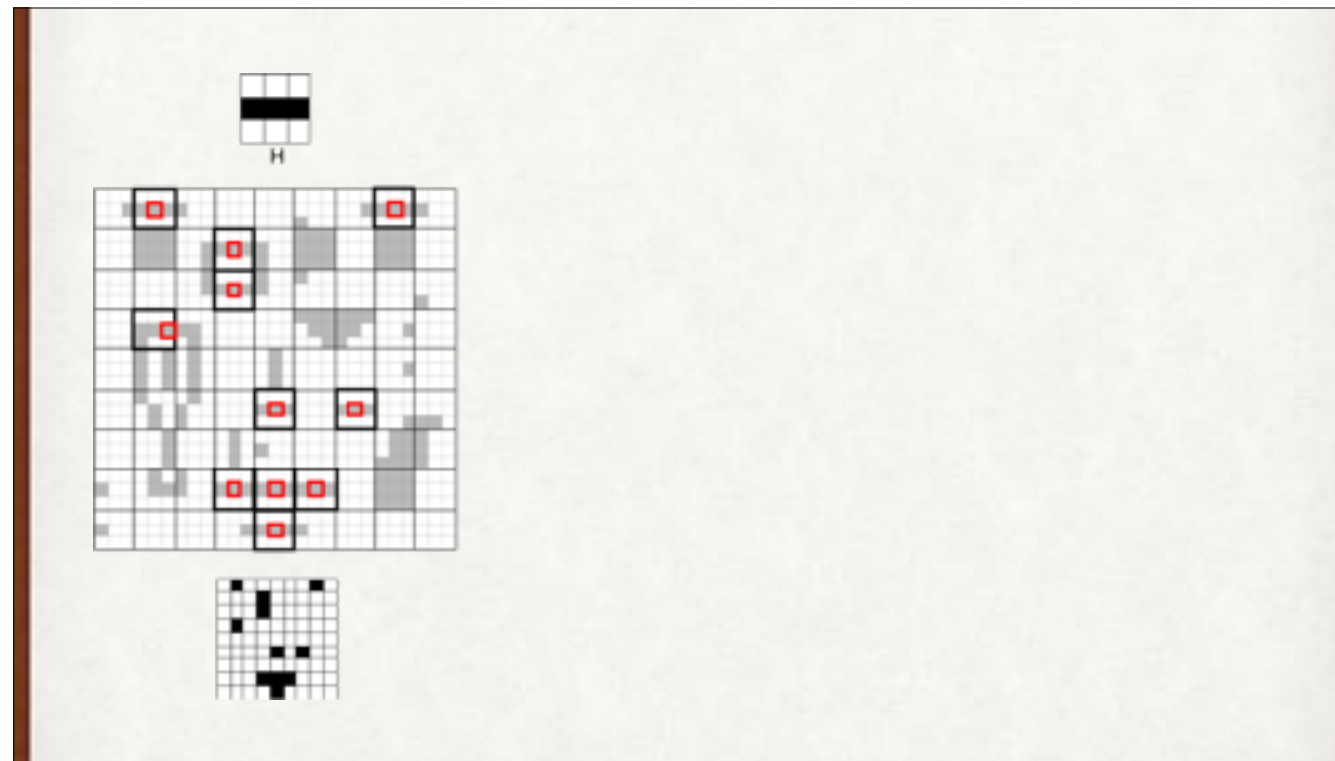
Layer 1 has 5 filters of 3 by 3 each (Horizontal, Vertical, Black, Left, Right). Layer 2 has three filters of 3 by 3, each looking for some arrangement of the Layer 1 filters (Nose, Eye, Mouth) Layer 3, in turn, is looking for arrangements of Layer 2 filters (Forward, Profile). We have a hierarchy of scales.



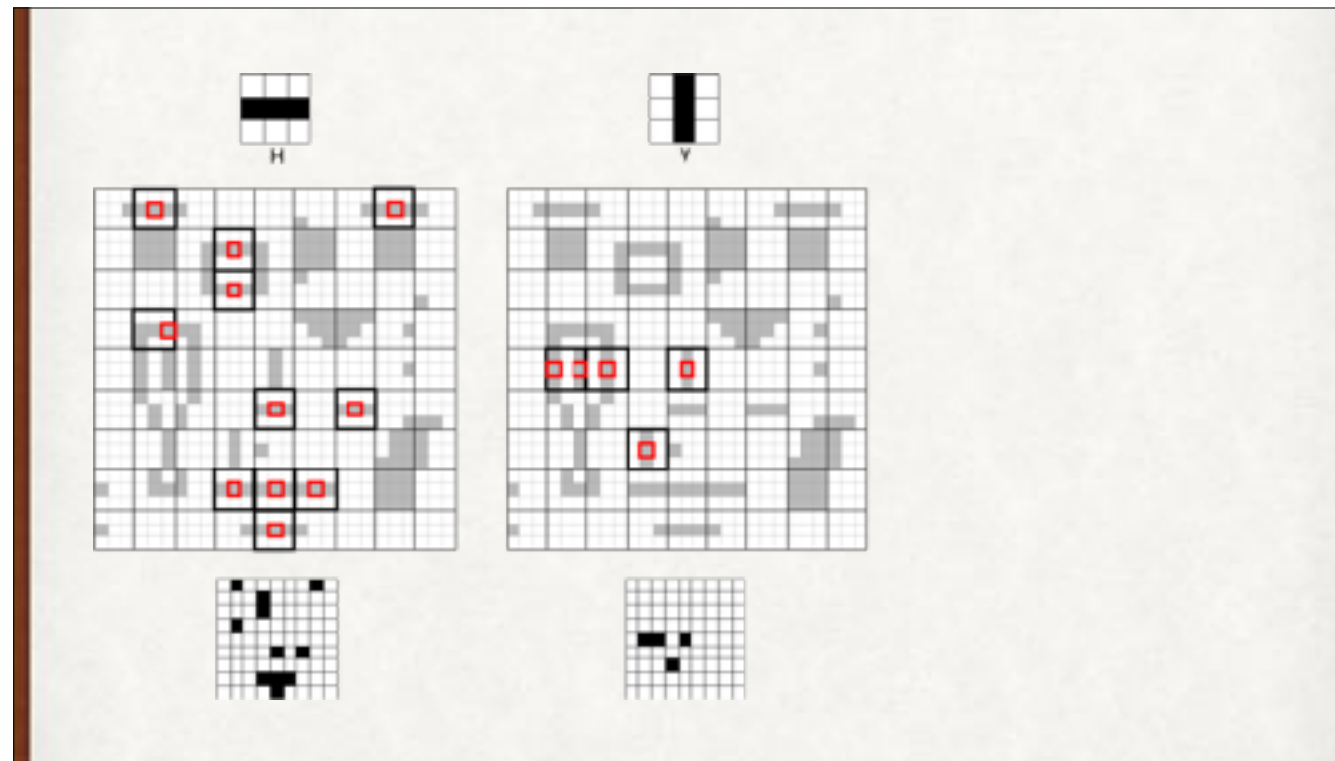
Layer 1 has 5 filters of 3 by 3 each (Horizontal, Vertical, Black, Left, Right). Layer 2 has three filters of 3 by 3, each looking for some arrangement of the Layer 1 filters (Nose, Eye, Mouth) Layer 3, in turn, is looking for arrangements of Layer 2 filters (Forward, Profile). We have a hierarchy of scales.



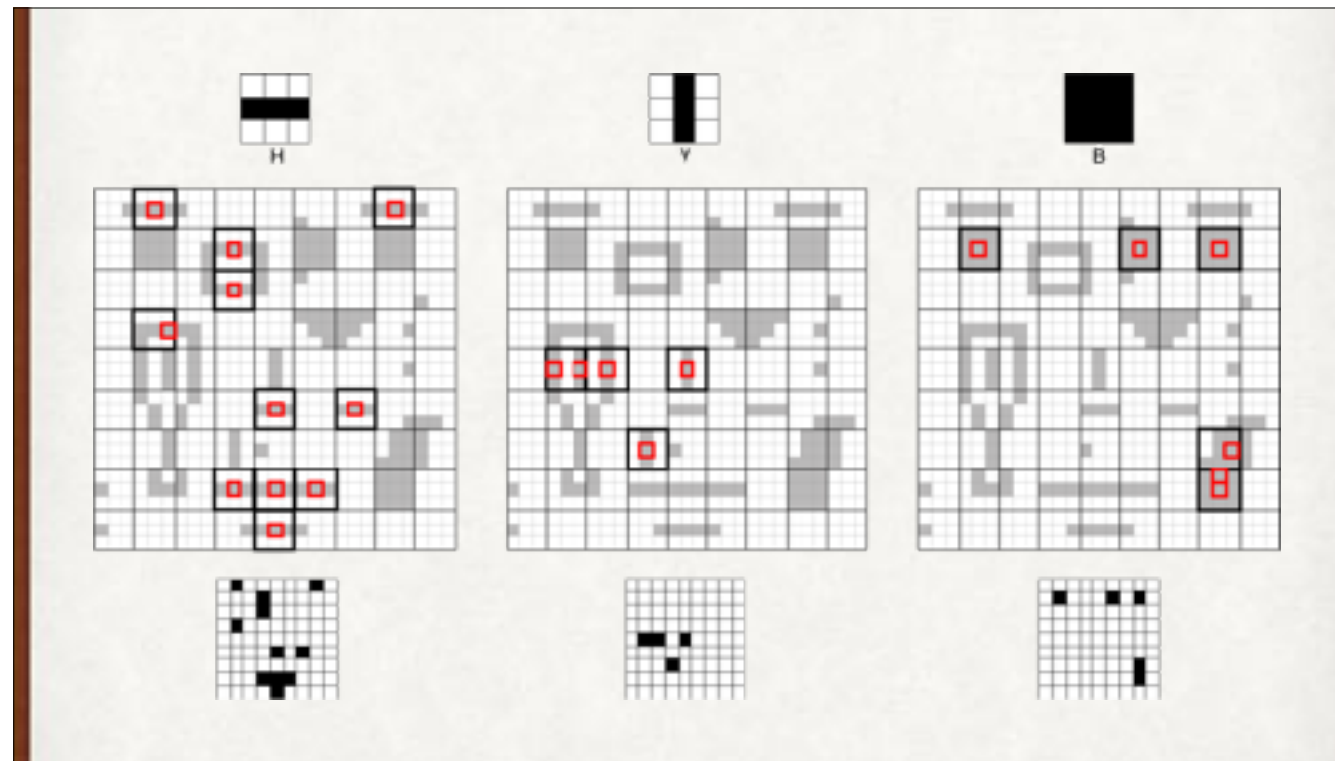
Applying the H, V, B filters to the mask. Red squares are matches. Then we downsample by 3 in each dimension. Blocks with a red square in them are highlighted, and are black in the output (bottom).



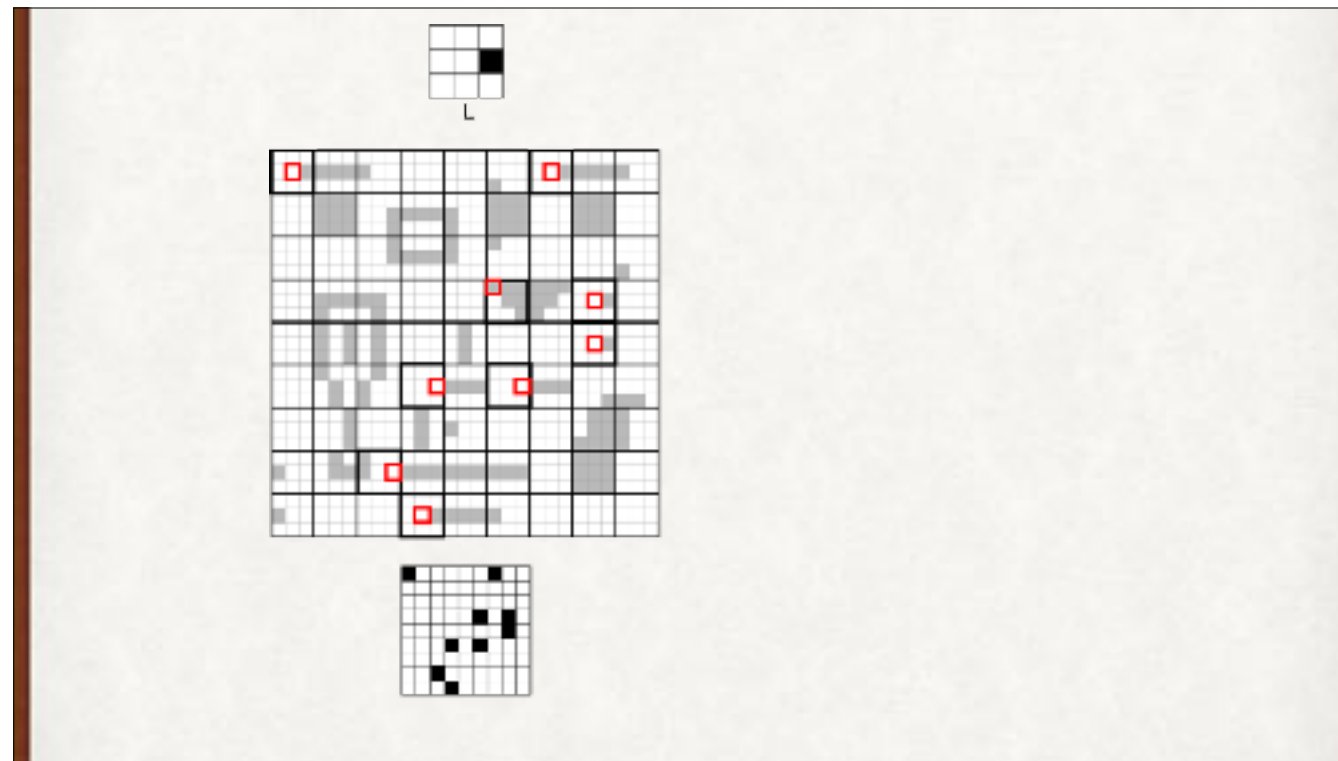
Applying the H, V, B filters to the mask. Red squares are matches. Then we downsample by 3 in each dimension. Blocks with a red square in them are highlighted, and are black in the output (bottom).



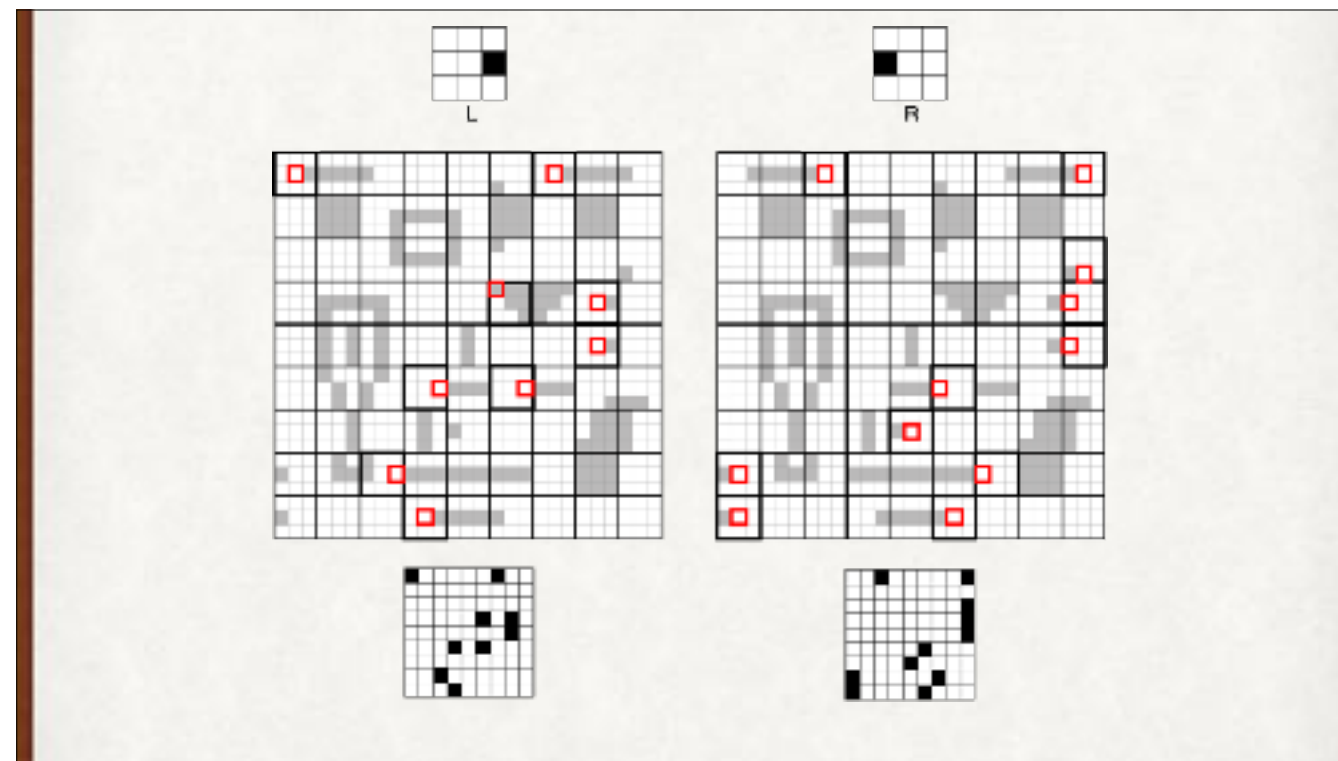
Applying the H, V, B filters to the mask. Red squares are matches. Then we downsample by 3 in each dimension. Blocks with a red square in them are highlighted, and are black in the output (bottom).



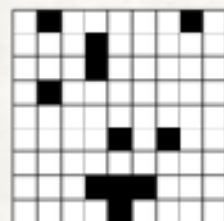
Applying the H, V, B filters to the mask. Red squares are matches. Then we downsample by 3 in each dimension. Blocks with a red square in them are highlighted, and are black in the output (bottom).



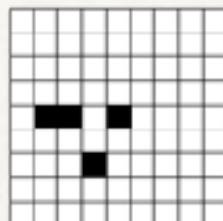
Applying the L and R filters.



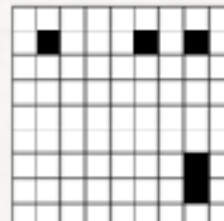
Applying the L and R filters.



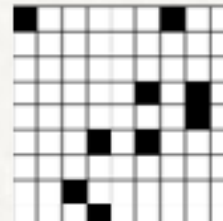
H



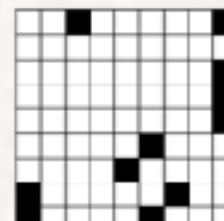
V



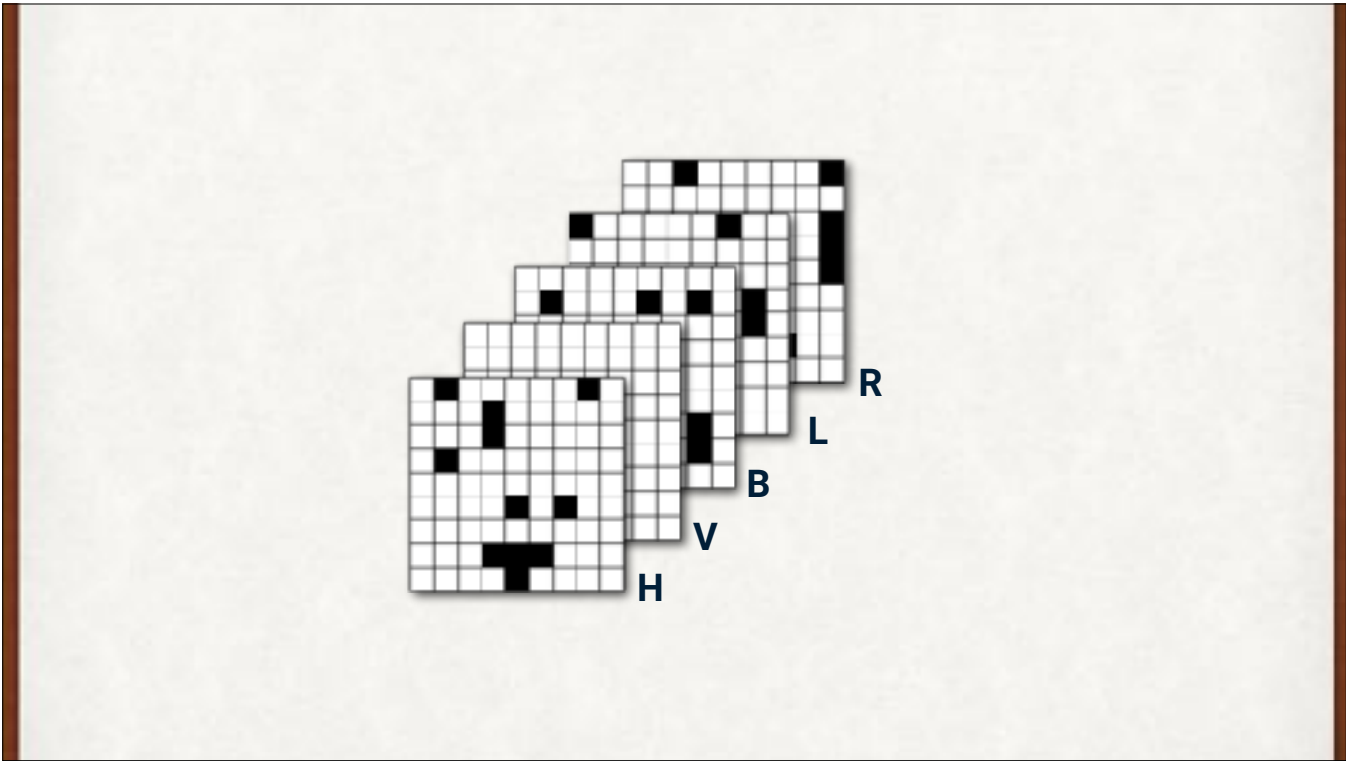
B

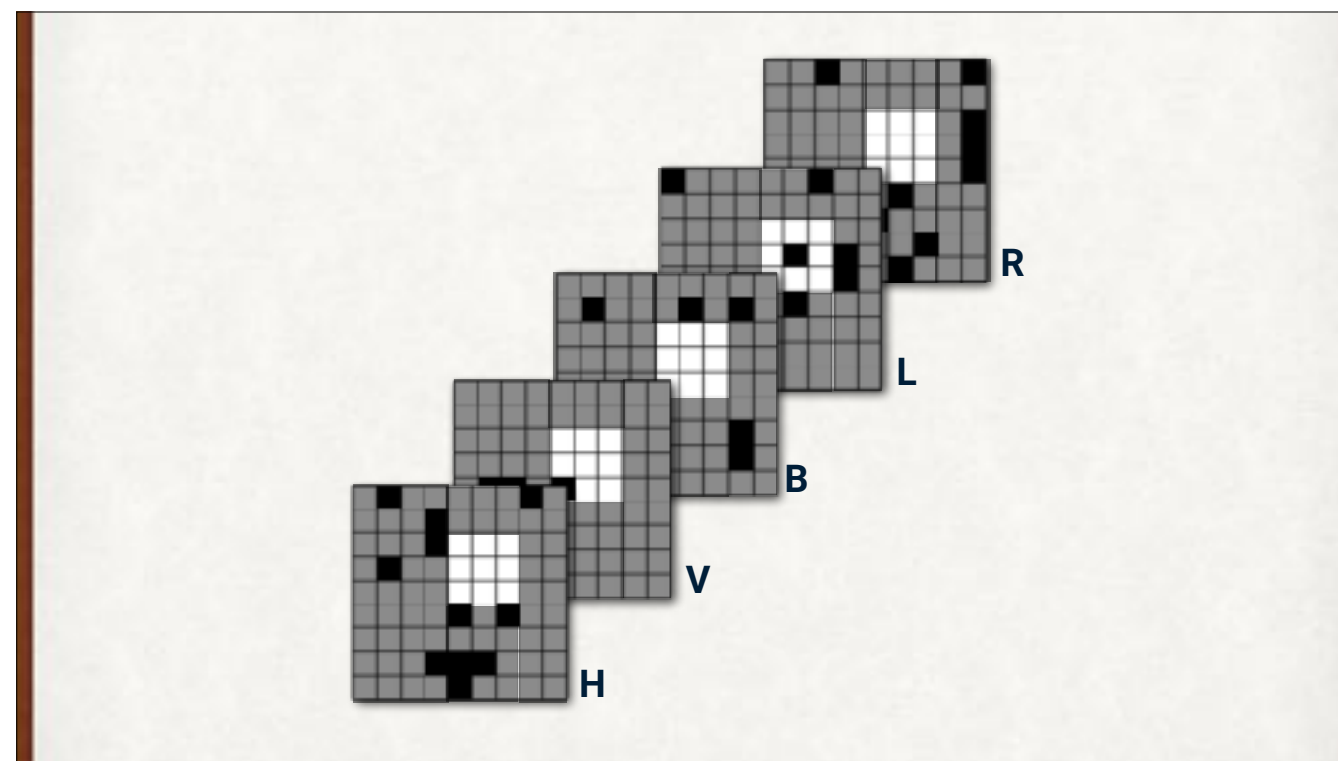


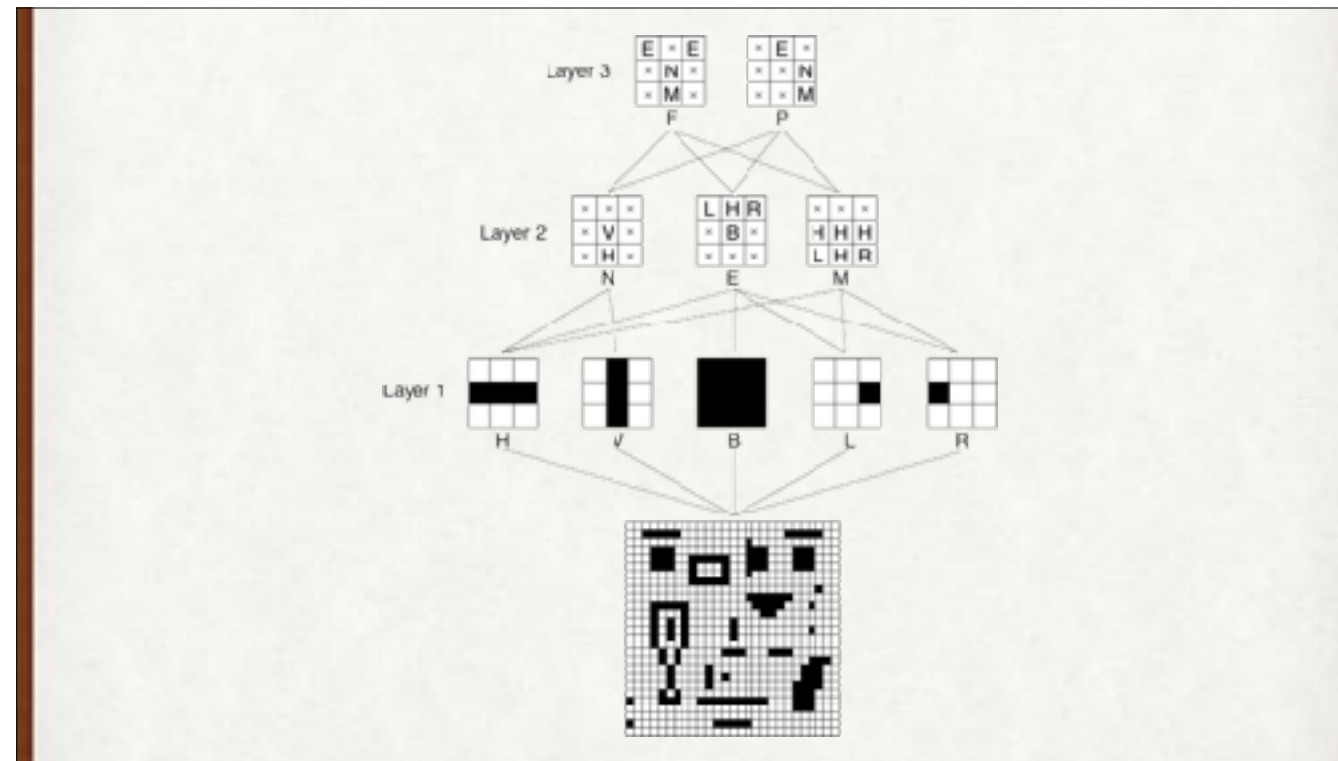
L



R







Layer 1 has 5 filters of 3 by 3 each (Horizontal, Vertical, Black, Left, Right). Layer 2 has three filters of 3 by 3, each looking for some arrangement of the Layer 1 filters (Nose, Eye, Mouth) Layer 3, in turn, is looking for arrangements of Layer 2 filters (Forward, Profile). We have a hierarchy of scales.

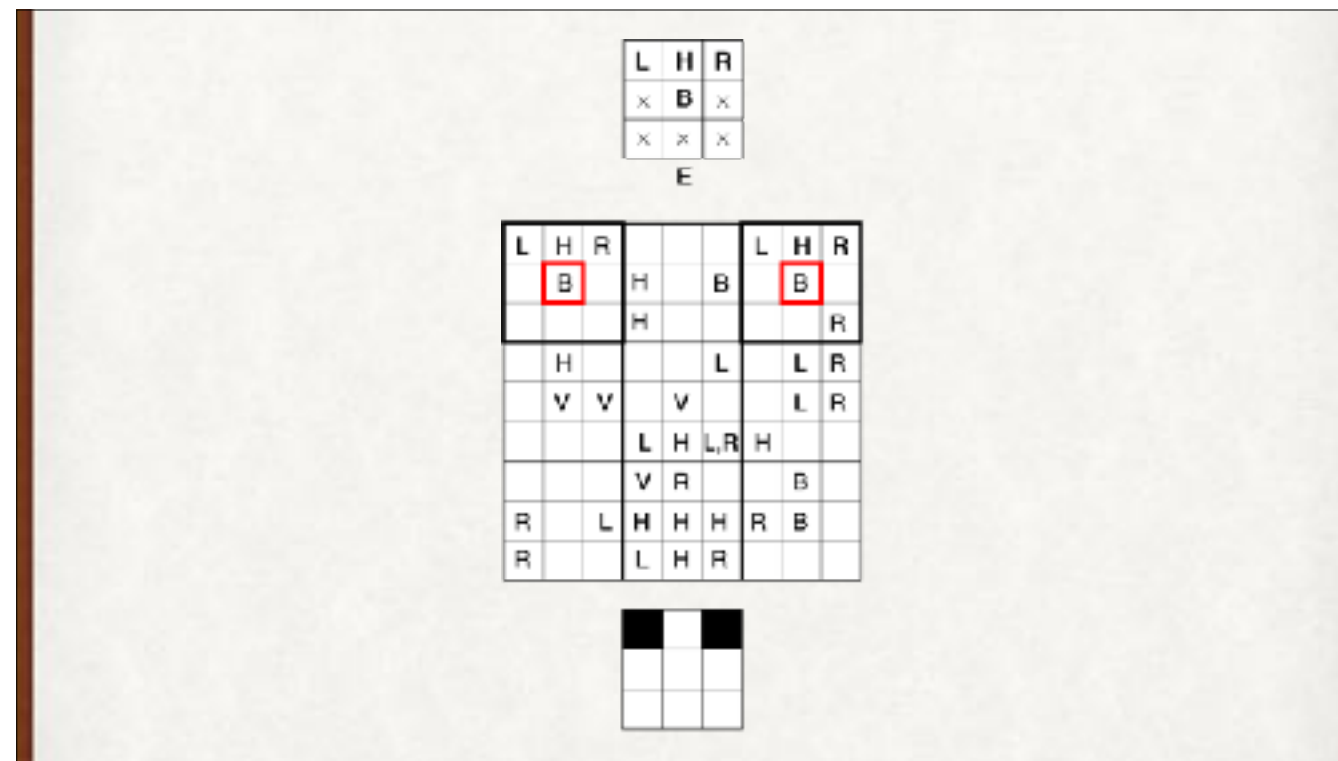
L	H	R
x	B	x
x	x	x
E		

Applying the Layer 2 filters to the Layer 1 output. The new results are just 3 by 3.

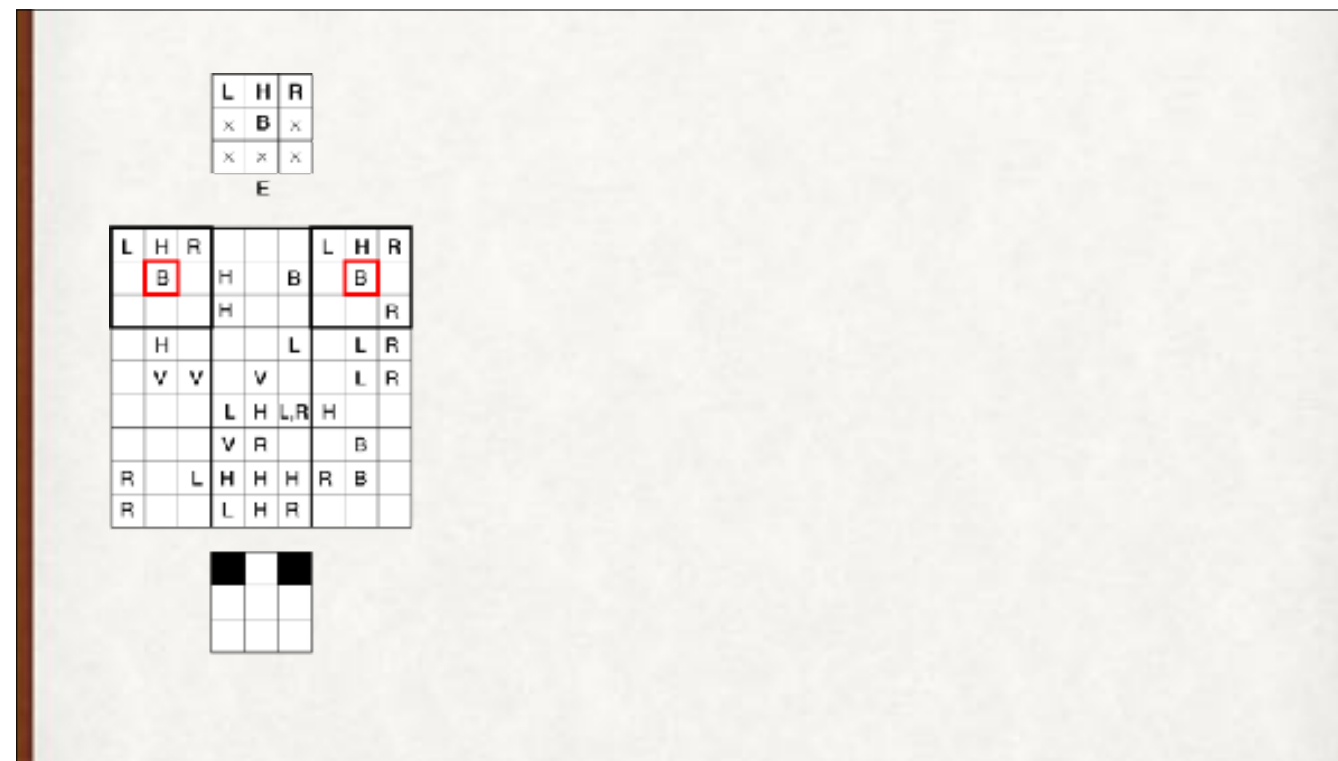
L	H	R
x	B	x
x	x	x
E		

L	H	R				L	H	R
	B		H		B		B	
			H					R
	H				L		L	R
	V	V		V			L	R
			L	H	L,R	H		
			V	R			B	
R		L	H	H	H	R	B	
R			L	H	R			

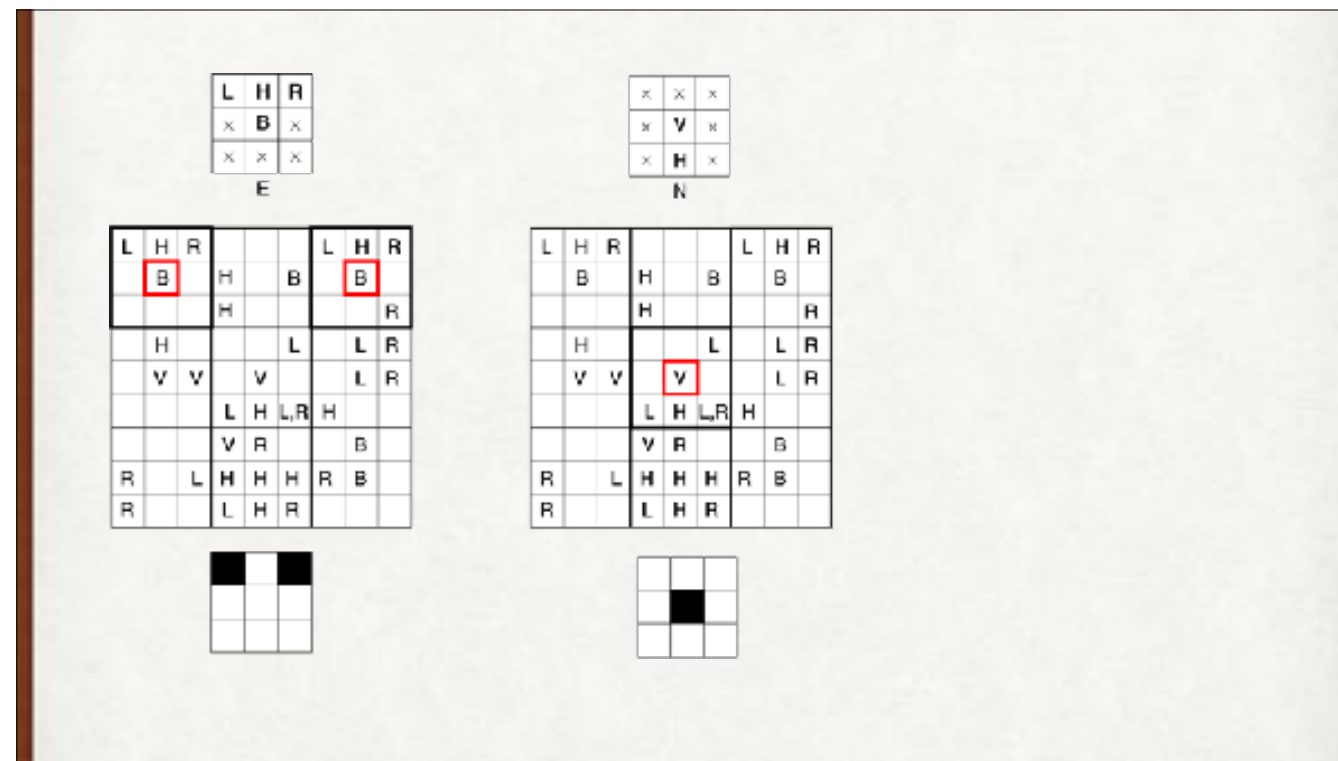
Applying the Layer 2 filters to the Layer 1 output. The new results are just 3 by 3.



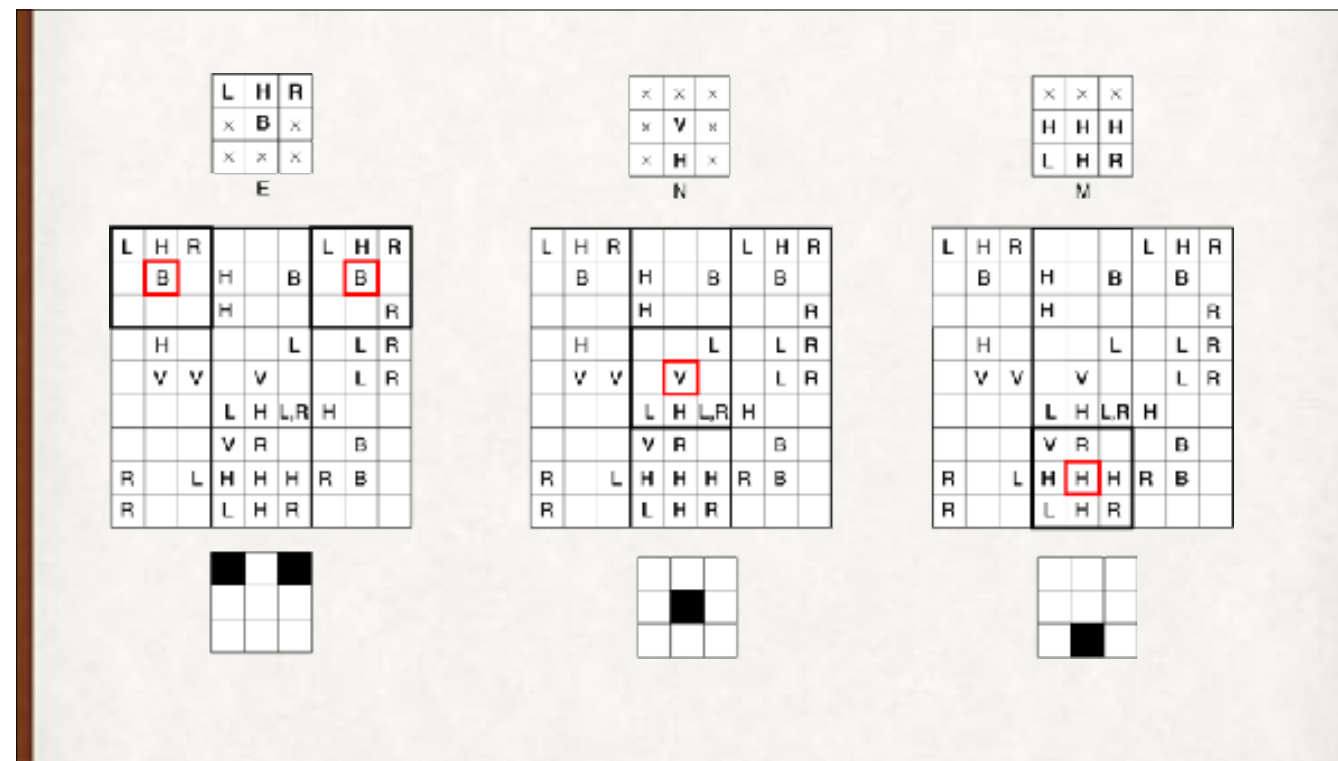
Applying the Layer 2 filters to the Layer 1 output. The new results are just 3 by 3.



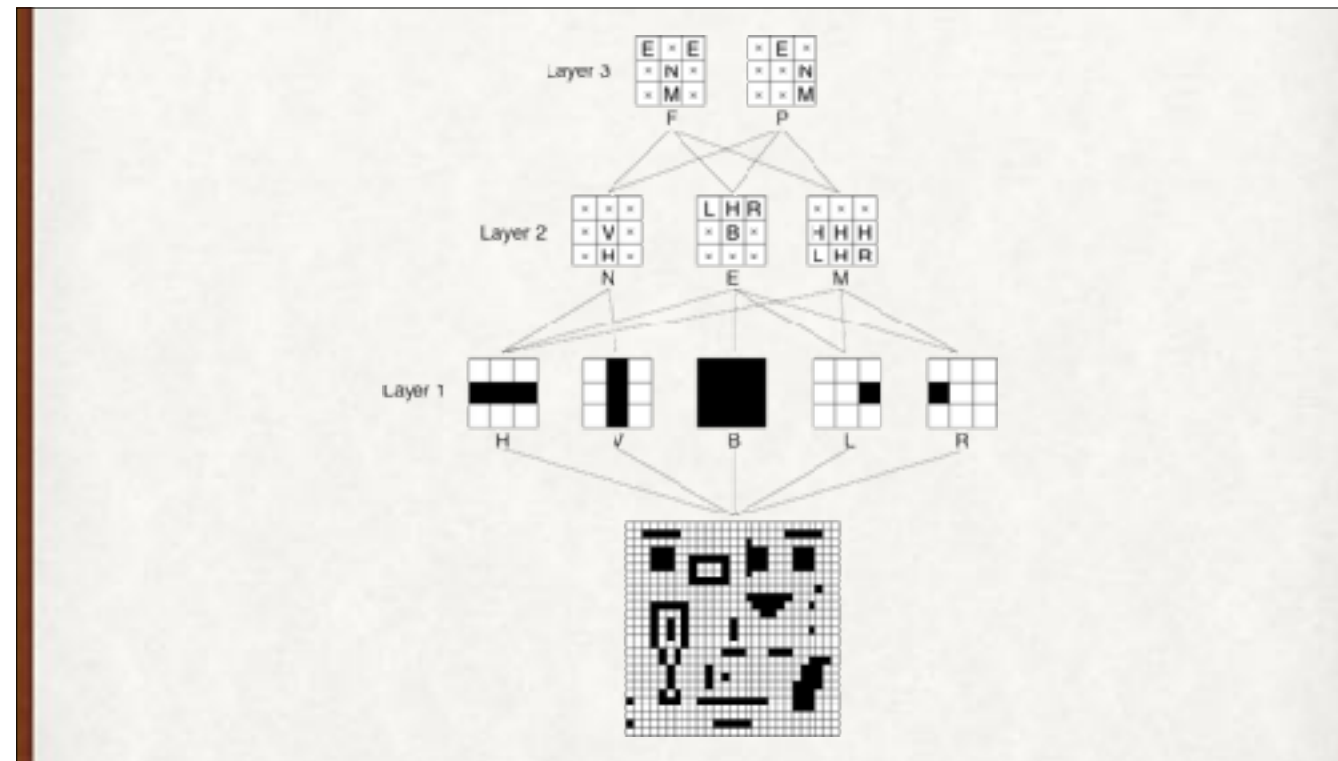
Applying the Layer 2 filters to the Layer 1 output. The new results are just 3 by 3.



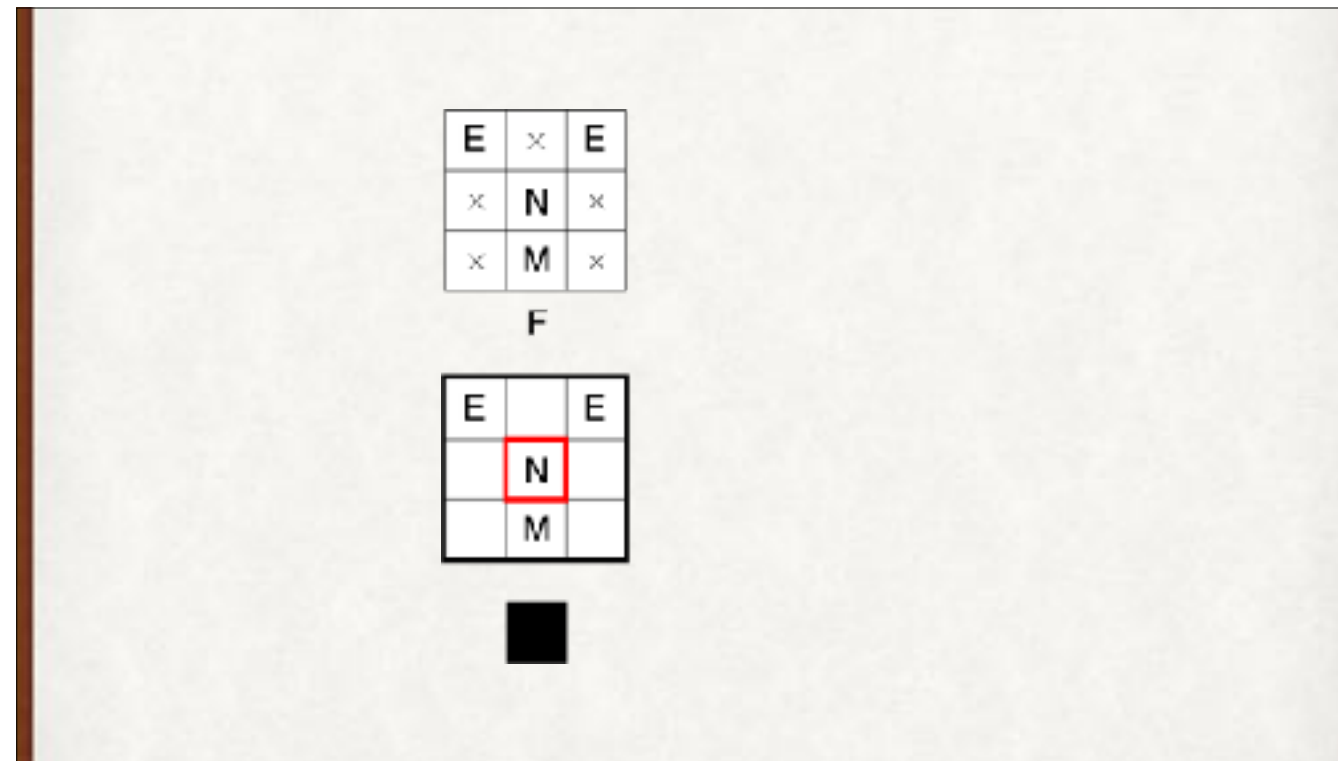
Applying the Layer 2 filters to the Layer 1 output. The new results are just 3 by 3.



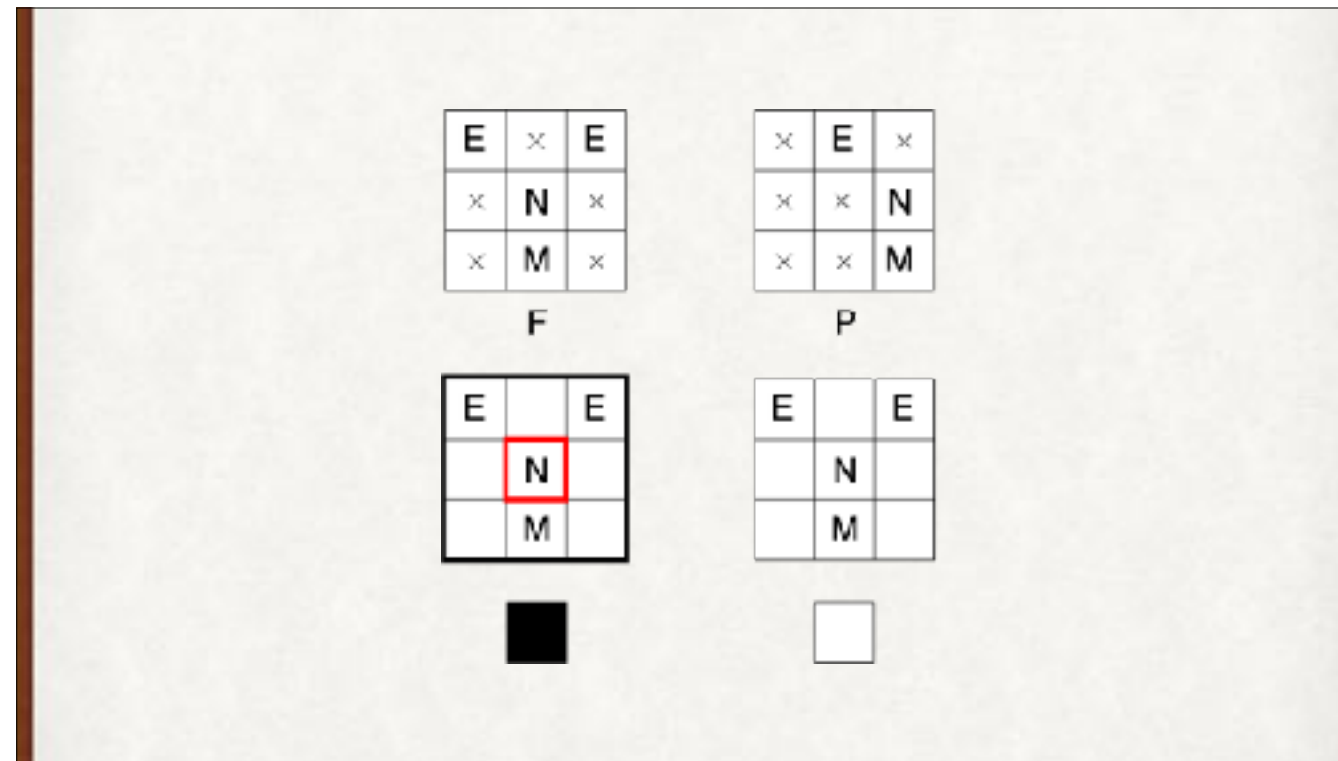
Applying the Layer 2 filters to the Layer 1 output. The new results are just 3 by 3.



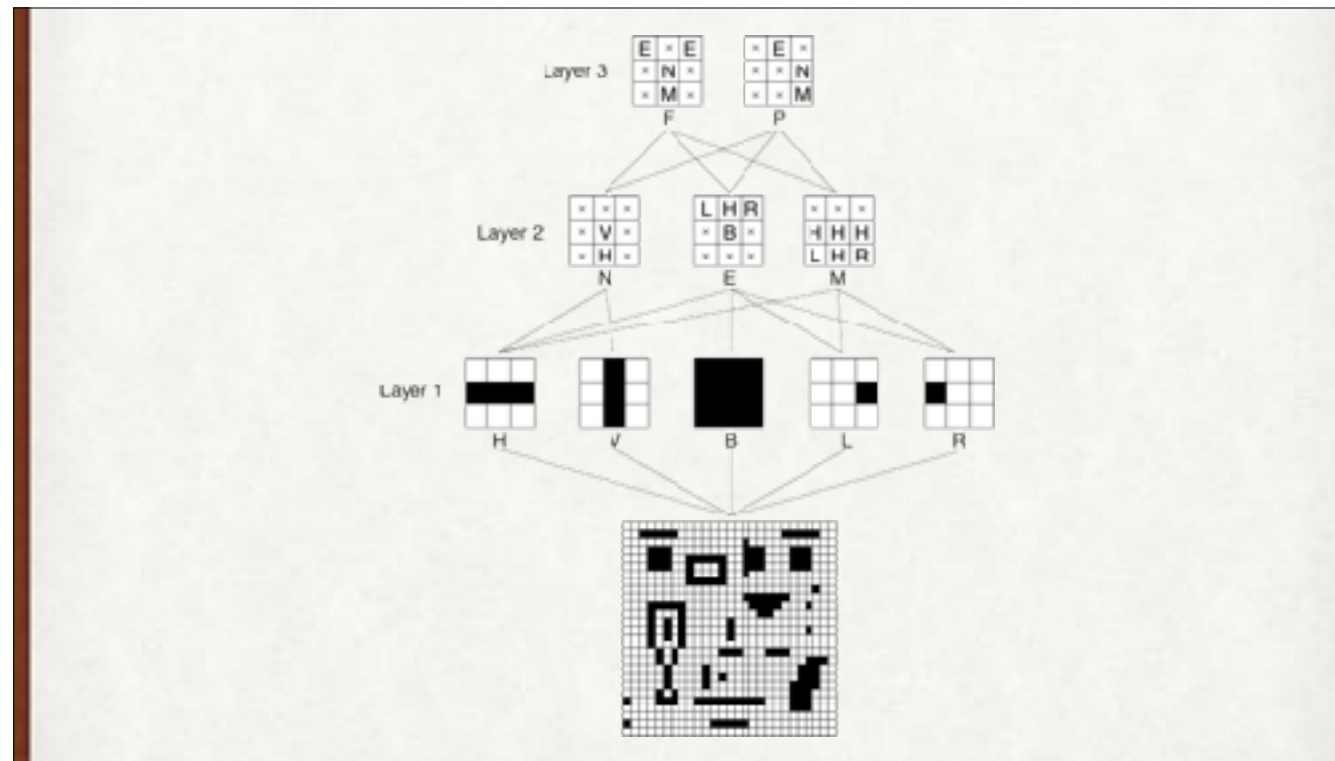
Layer 1 has 5 filters of 3 by 3 each (Horizontal, Vertical, Black, Left, Right). Layer 2 has three filters of 3 by 3, each looking for some arrangement of the Layer 1 filters (Nose, Eye, Mouth) Layer 3, in turn, is looking for arrangements of Layer 2 filters (Forward, Profile). We have a hierarchy of scales.



Looking for a forward (F) or profile (P) face in the mask. We find a forward. We've used convolution to find something we cared about. In this example we designed the filters by hand. The beauty of this method is that we can let the system start with random values for the filters, and learn for itself what numbers to use so that it detects those things we want it to find.

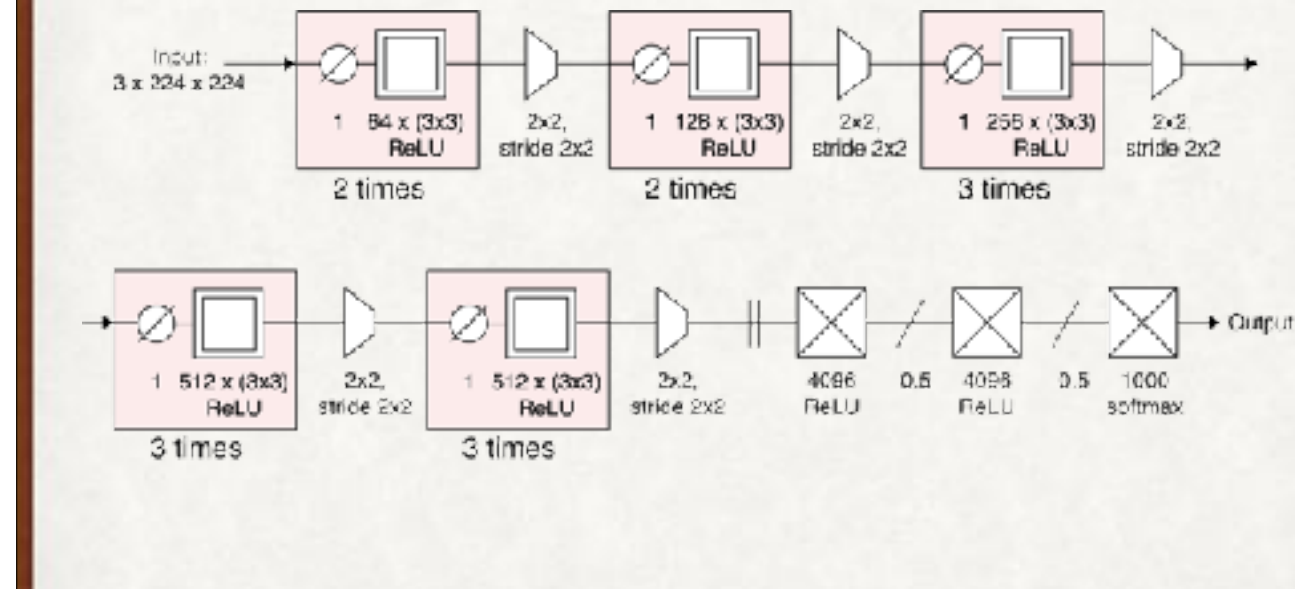


Looking for a forward (F) or profile (P) face in the mask. We find a forward. We've used convolution to find something we cared about. In this example we designed the filters by hand. The beauty of this method is that we can let the system start with random values for the filters, and learn for itself what numbers to use so that it detects those things we want it to find.



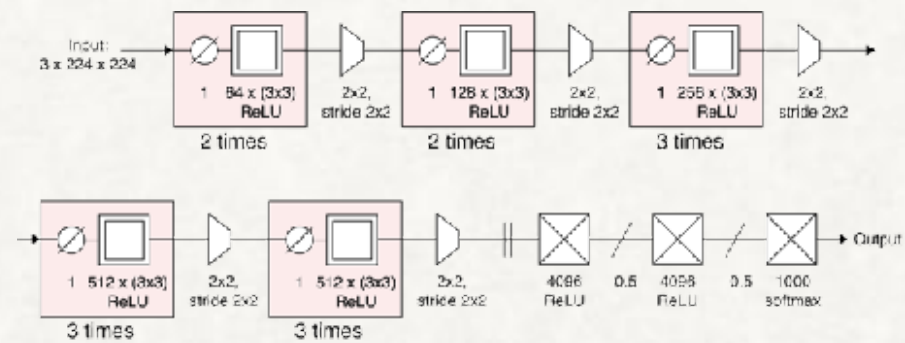
Layer 1 has 5 filters of 3 by 3 each (Horizontal, Vertical, Black, Left, Right). Layer 2 has three filters of 3 by 3, each looking for some arrangement of the Layer 1 filters (Nose, Eye, Mouth) Layer 3, in turn, is looking for arrangements of Layer 2 filters (Forward, Profile). We have a hierarchy of scales. We did this all by hand. Imagine if our image was 1000 by 1000, in color, and it could be any of 1000 different objects. We'd never be able to hand-build all the different filters that would let us determine what object was in a photo. But if our network has enough power (that is, enough neurons arranged in an appropriate way), it can **learn** all those filters for us. And that's exactly what we do.

VGG16



A famous, simple, and versatile convolutional neural network: VGG16. Its weights can be found online, trained on a dataset of 1.2 million images, labeled by their contents into 1000 categories (hence the 1000 neurons in the final fully-connected layer). Sizes: 224, 112, 56, 28, 14. Dense input: 100,352 elements. Top-5 accuracy of 90%

VGG16

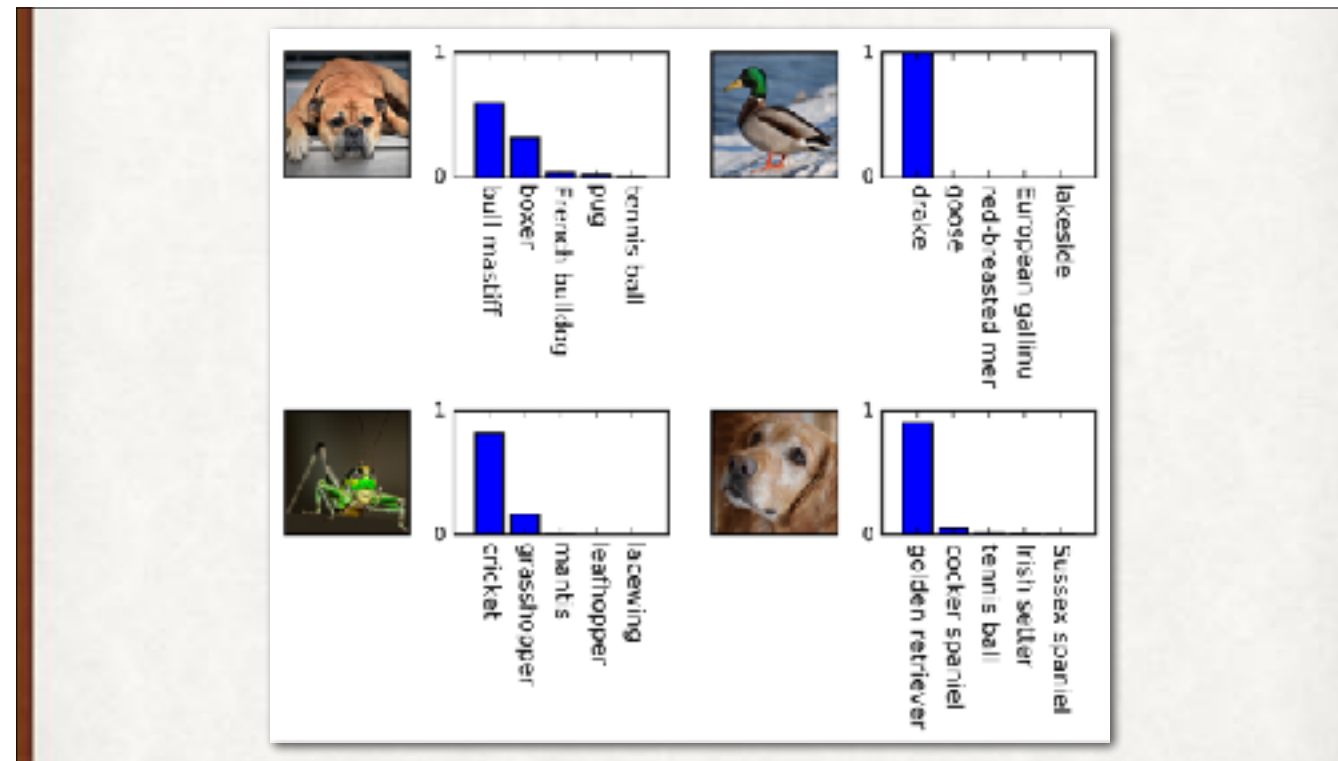


~4,200 filters

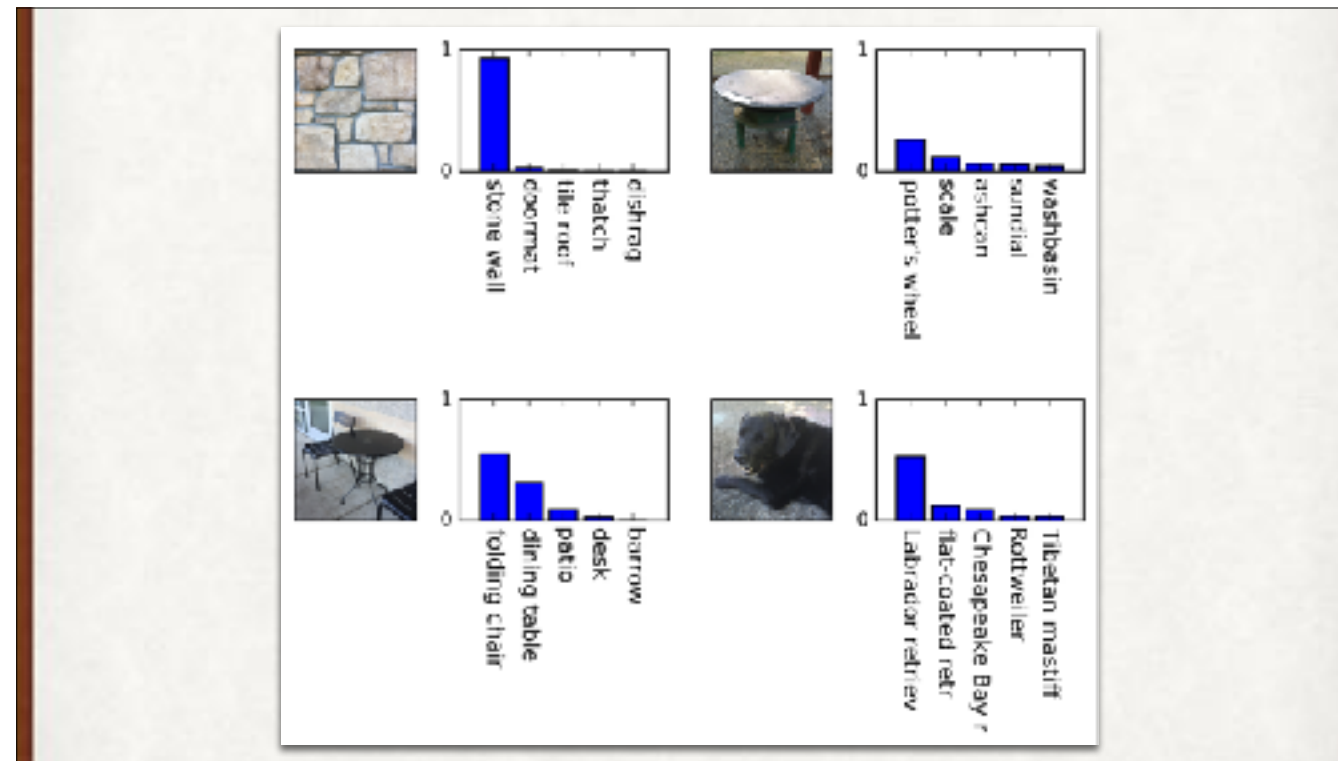
~38,000 filter weights

~145,000 total weights

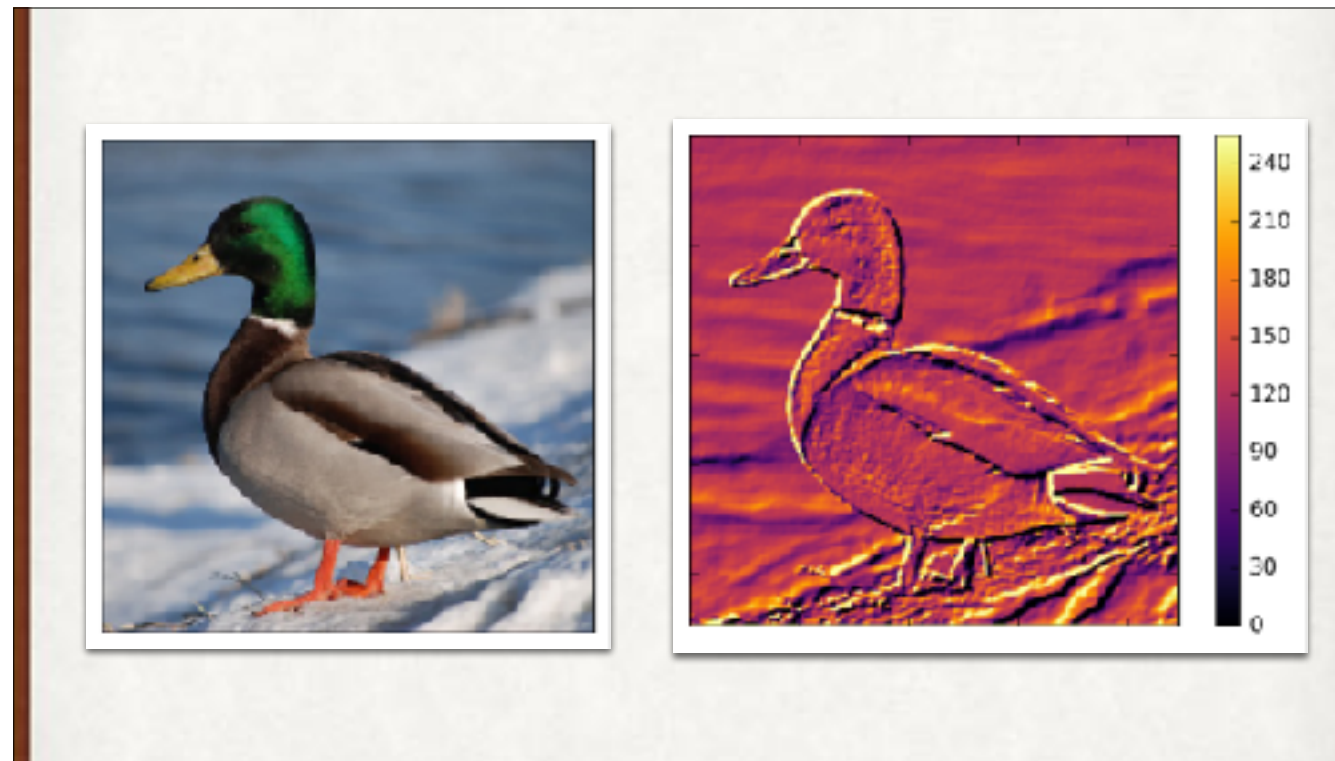
A famous, simple, and versatile convolutional neural network: VGG16. It can be found online trained on a dataset of 1.2 million images, labeled by their contents into 1000 categories (hence the 1000 neurons in the final fully-connected layer). Sizes: 224, 112, 56, 28, 14. Dense input: 100,352 elements. Top-5 accuracy of 90%



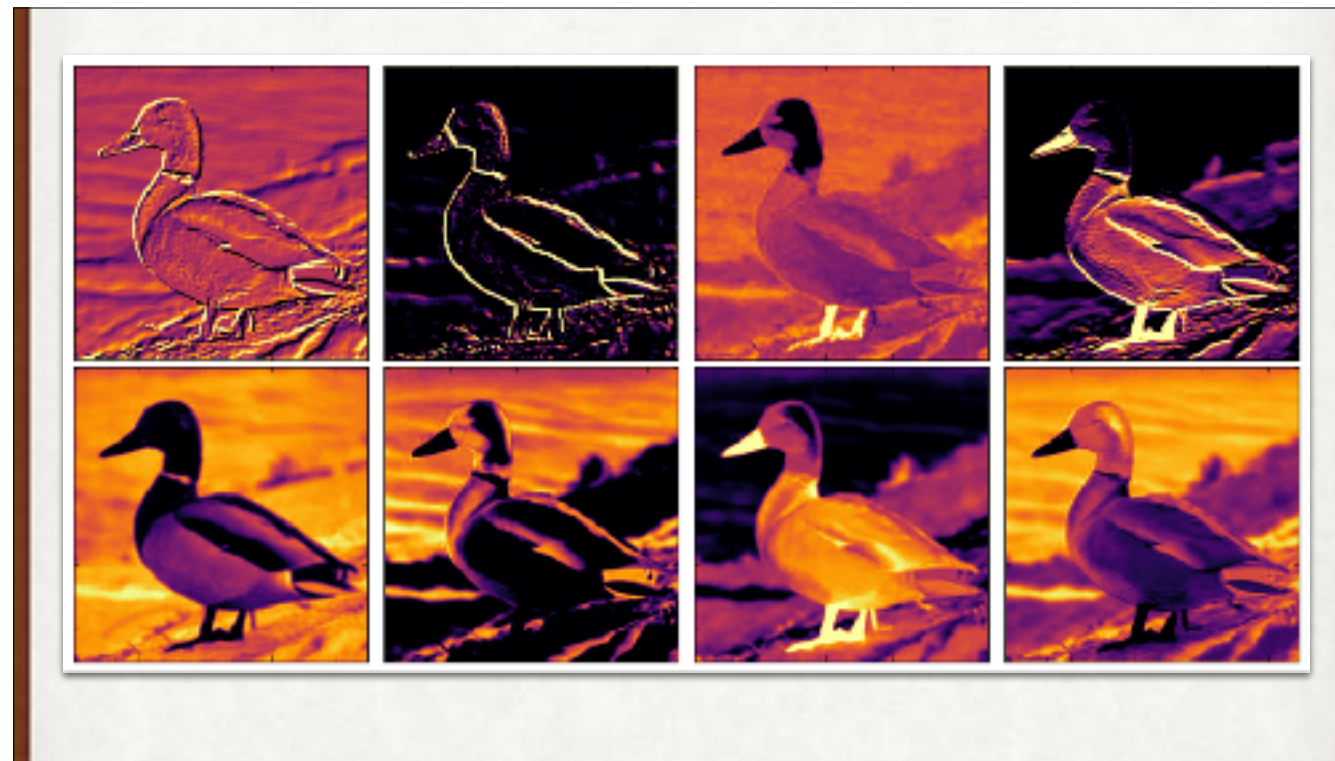
VGG16 classifications for online photographs, so they're completely new to the system. It did great.



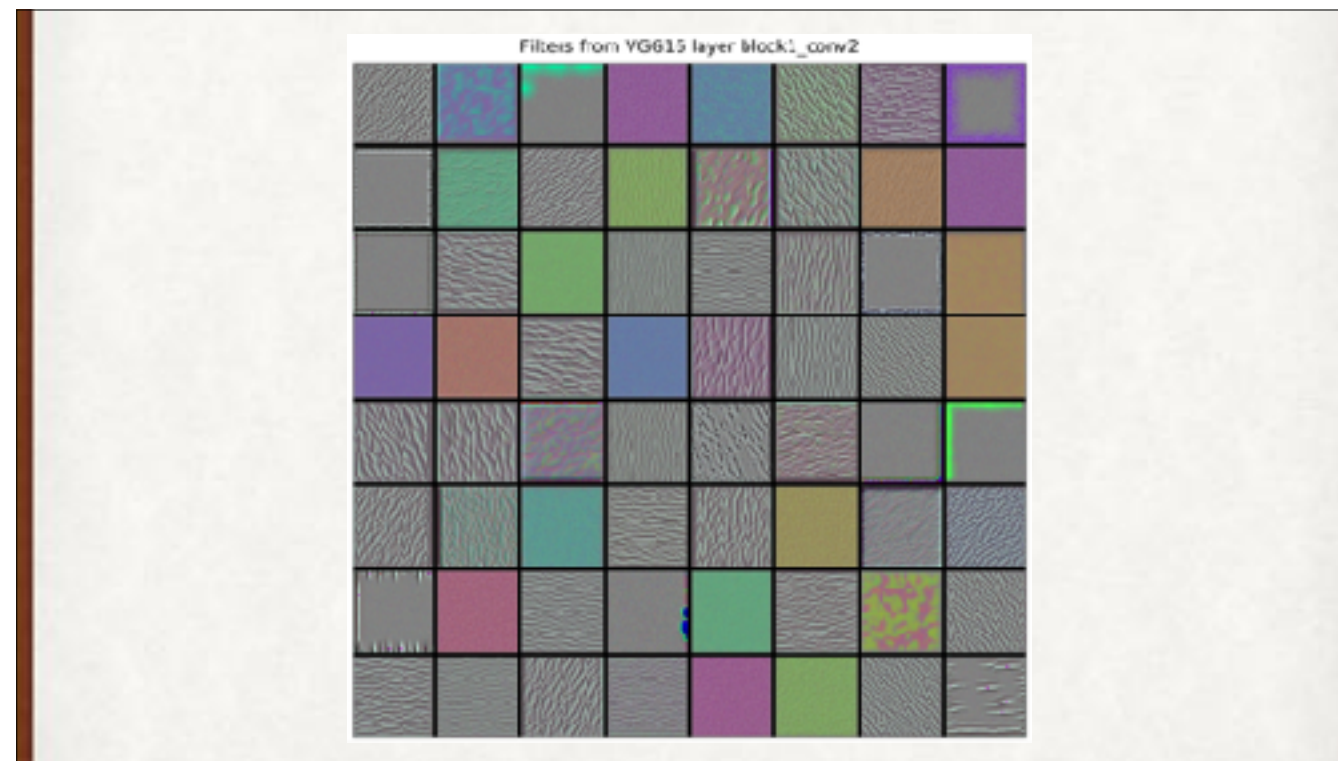
VGG16 classifications for my own photographs, so they're completely new to the system. It did great.



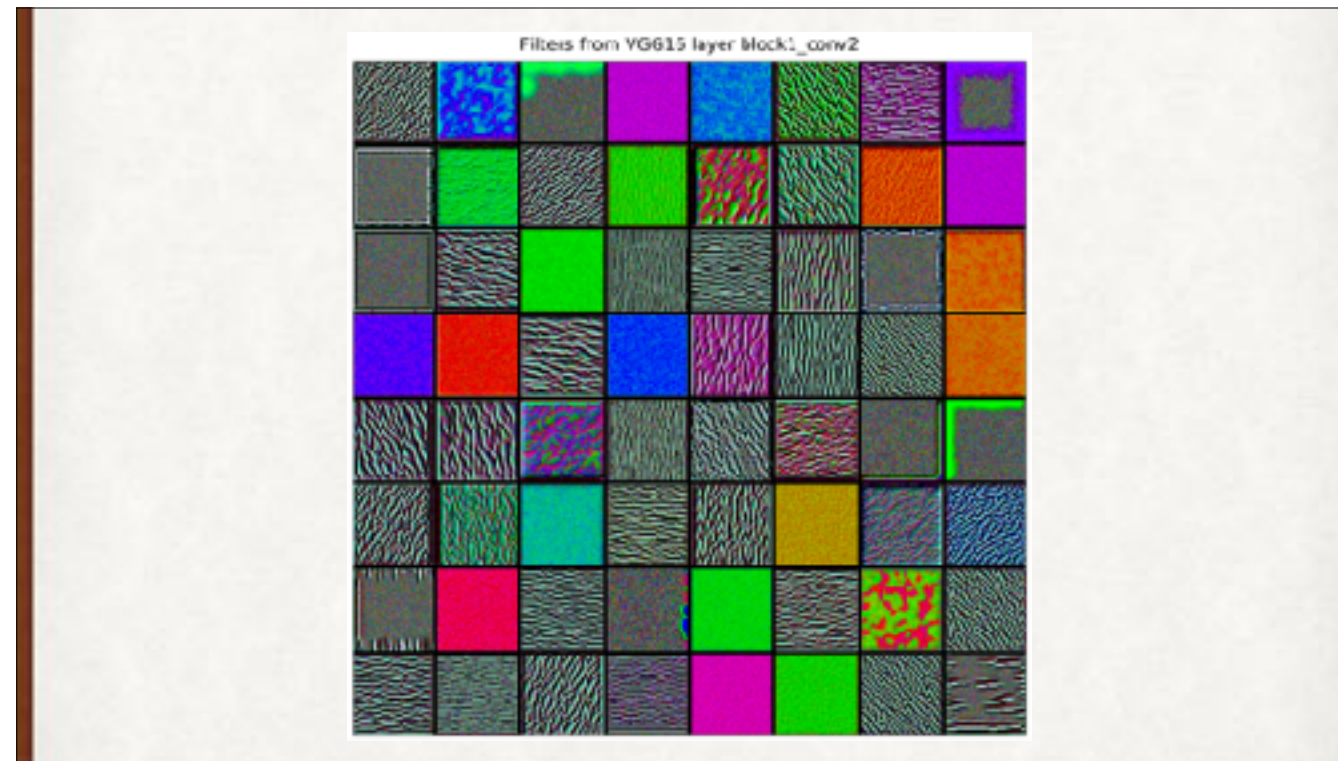
Left, an input image of a duck. Right, the response of filter 0 in the first layer of VGG16. This filter appears to respond to strong edges. Where there's no clear edge, we get middling value.



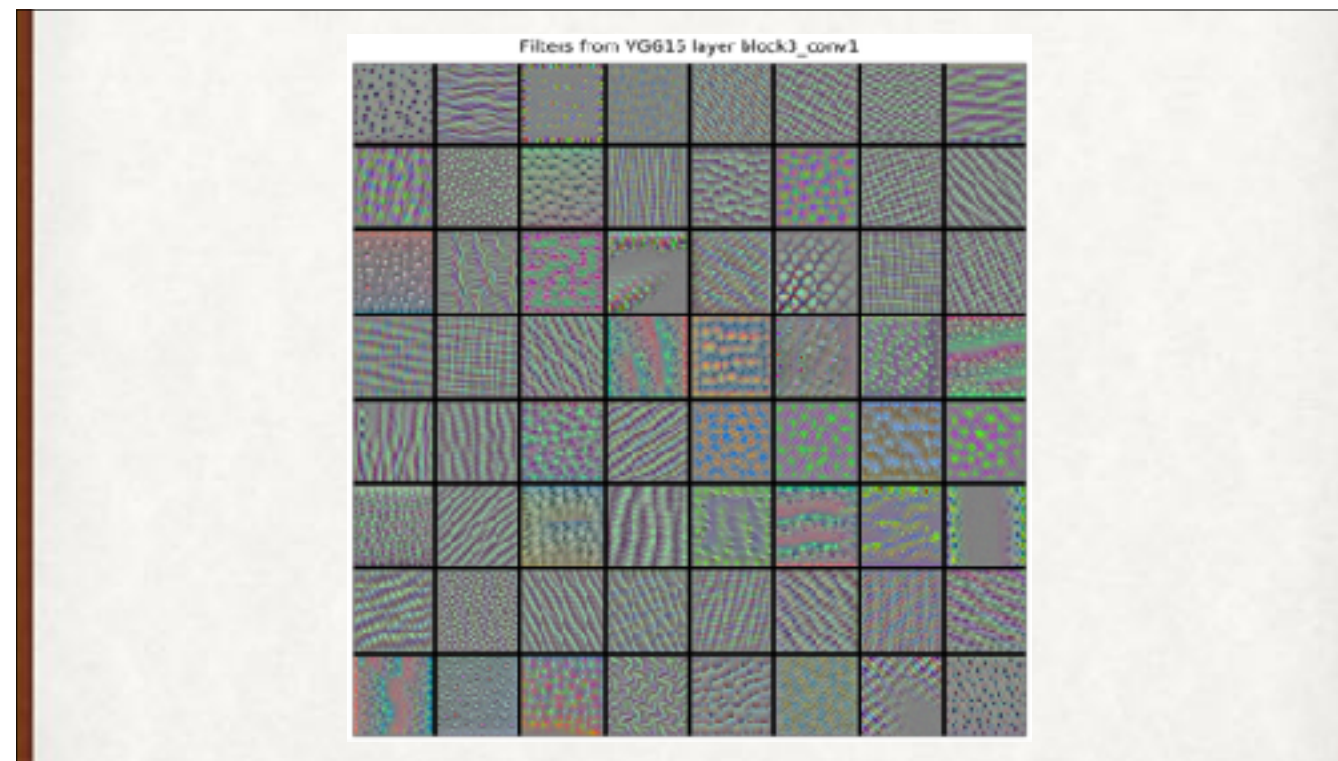
The responses of 8 hand-picked filters on the first layer of VGG16 to the duck. These different filters have learned to “look” for different types of patterns in the image.



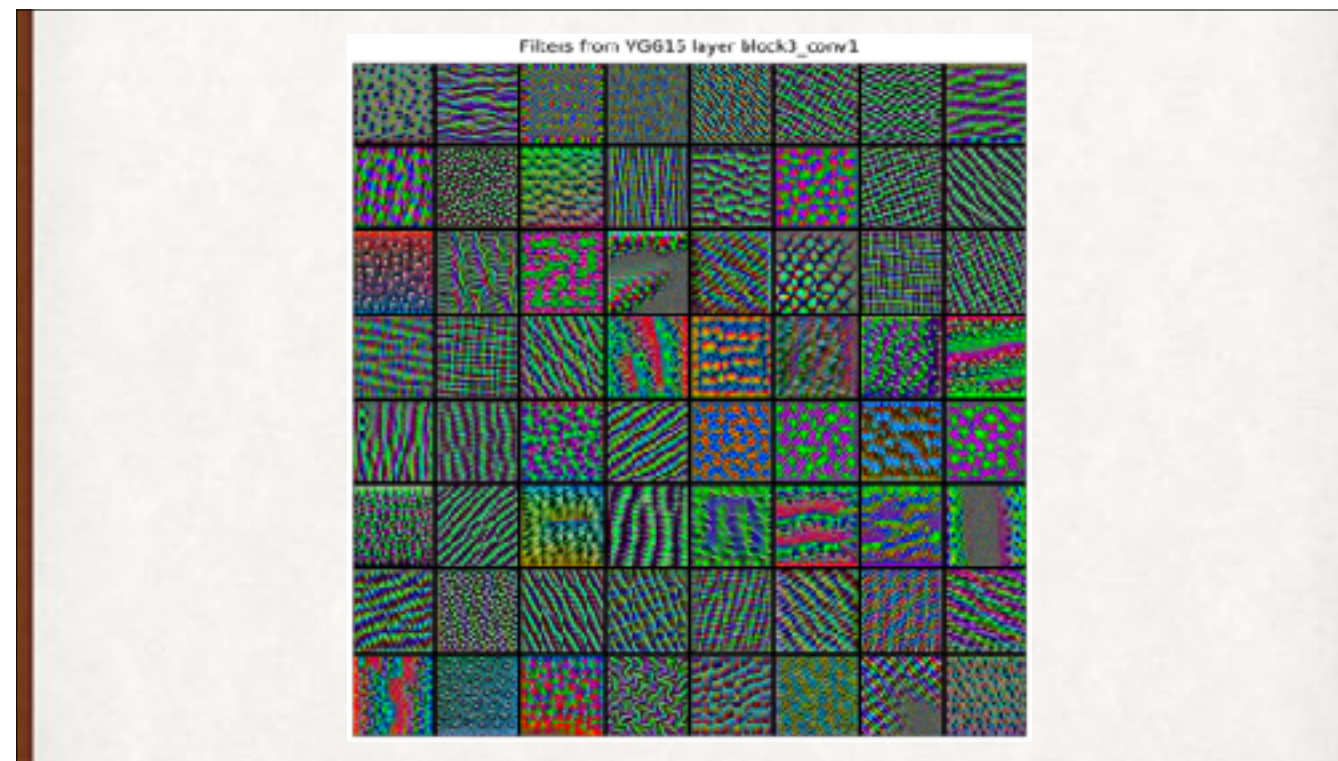
Visualizing layer patterns using noise as an input. We send random noise into the system, and measure how strongly each layer responds by adding up all of its output values. We use that as the “error”, and then modify the noisy input pixel values to make that “error” as large as possible - that is, we’re 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.”



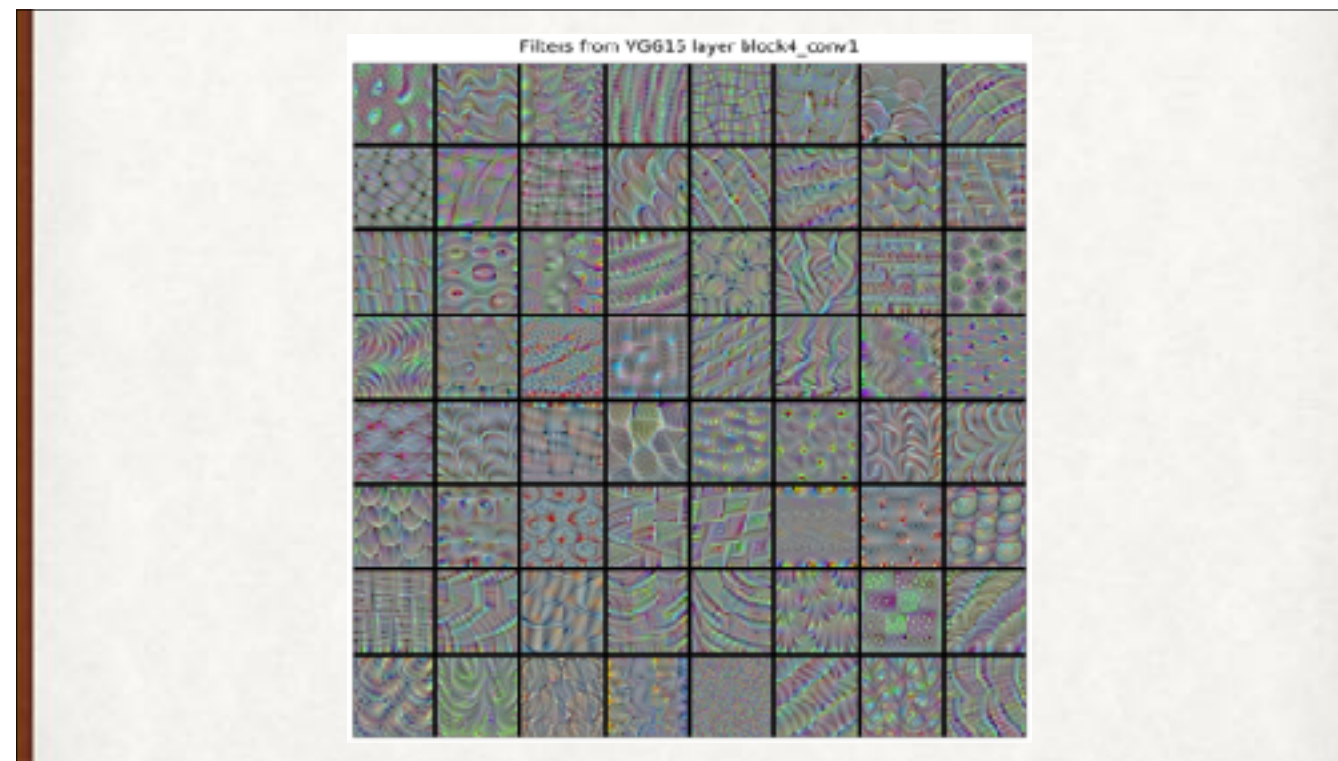
Visualizing layer patterns using noise as an input. We send random noise into the system, and measure how strongly each layer responds by adding up all of its output values. We use that as the “error”, and then modify the noisy input pixel values to make that “error” as large as possible - that is, we’re 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.”



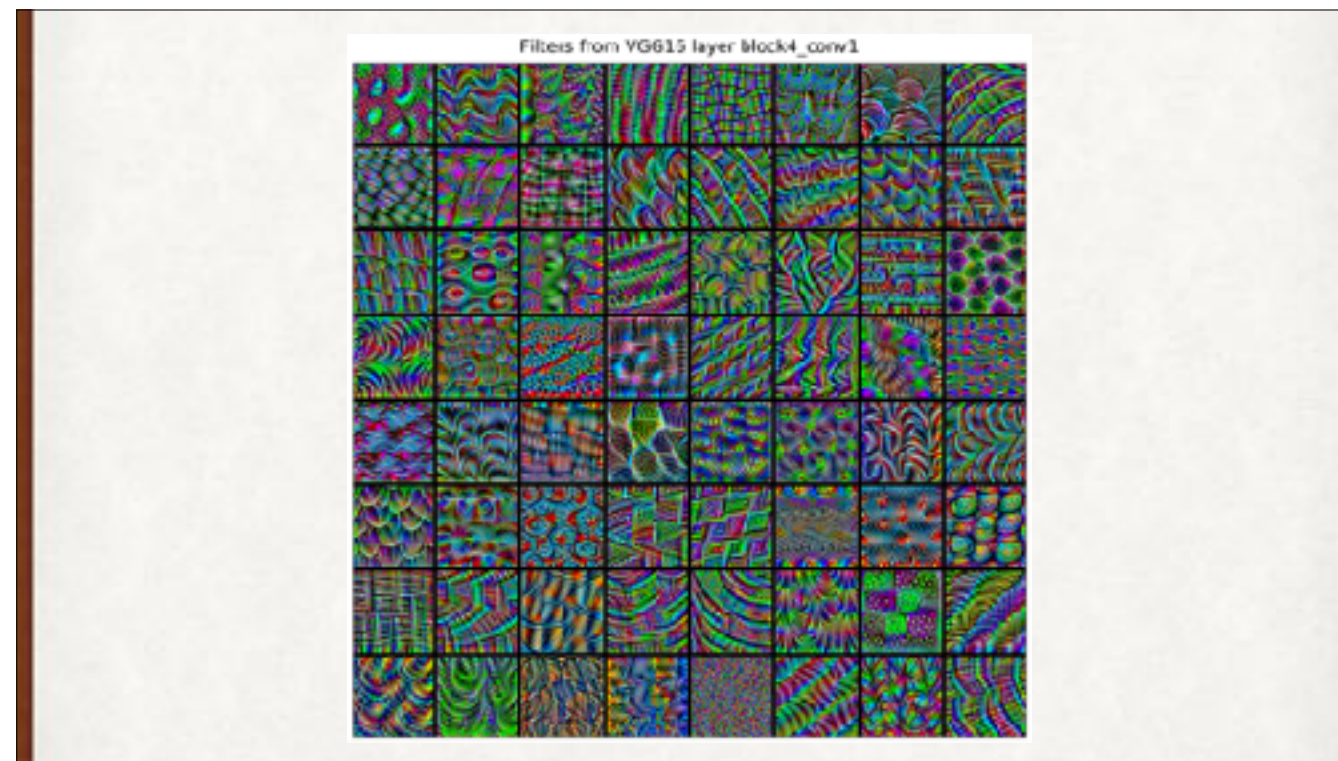
Visualizing filters from the third block of VGG16.



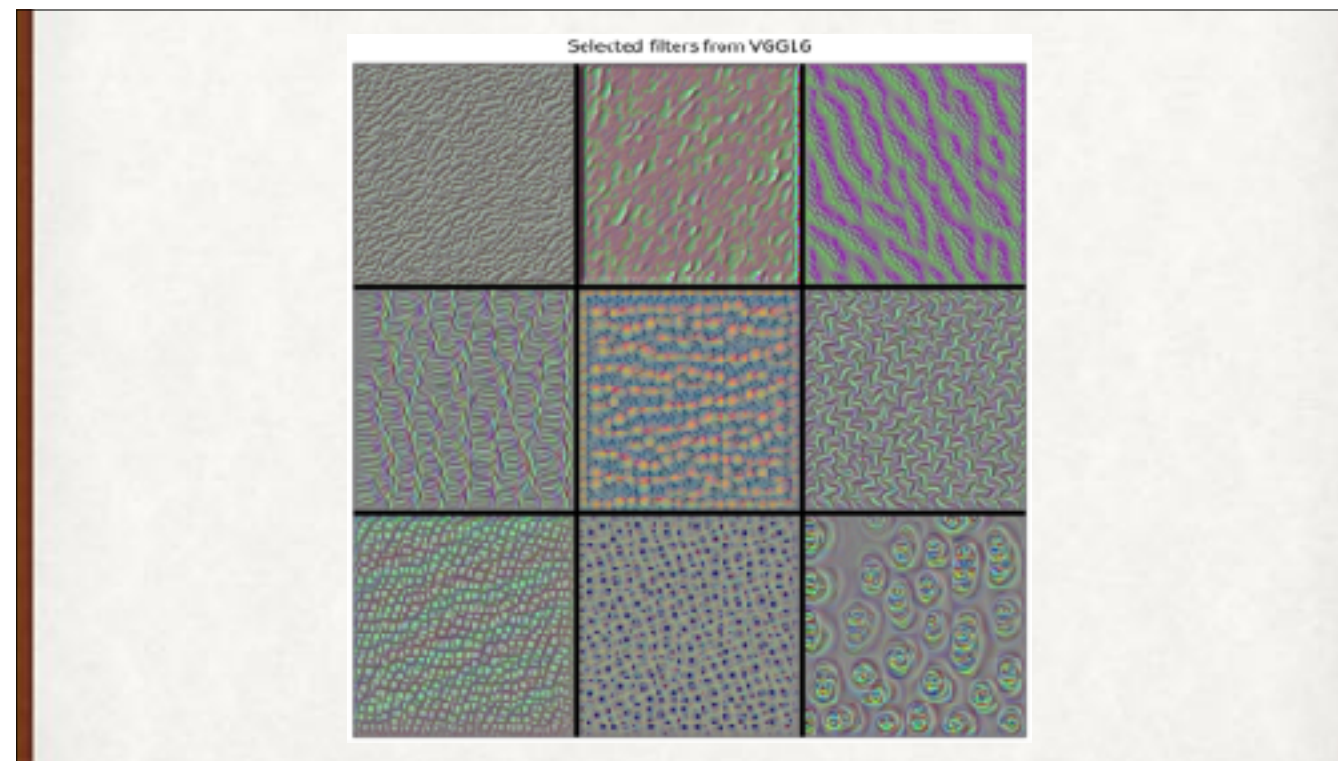
Visualizing filters from the third block of VGG16.



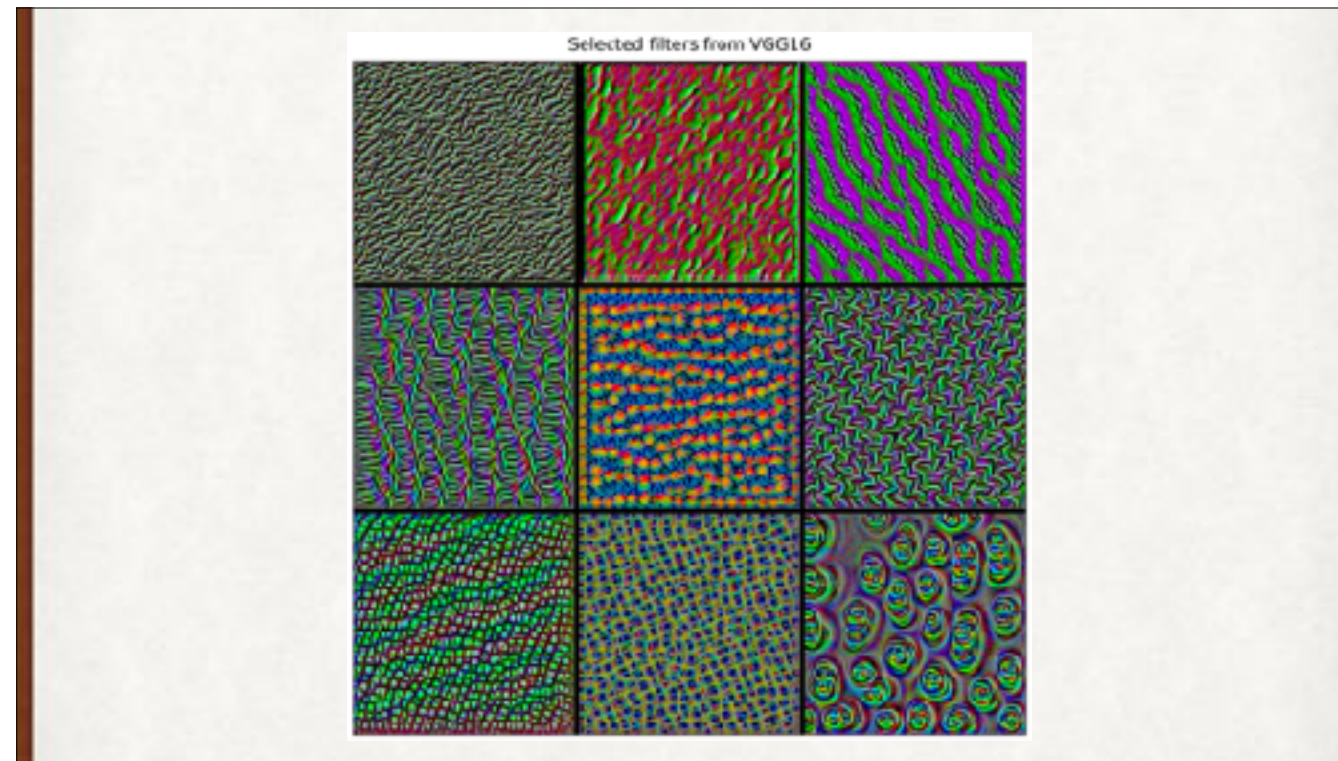
Visualizing filters from the fourth block of VGG16.



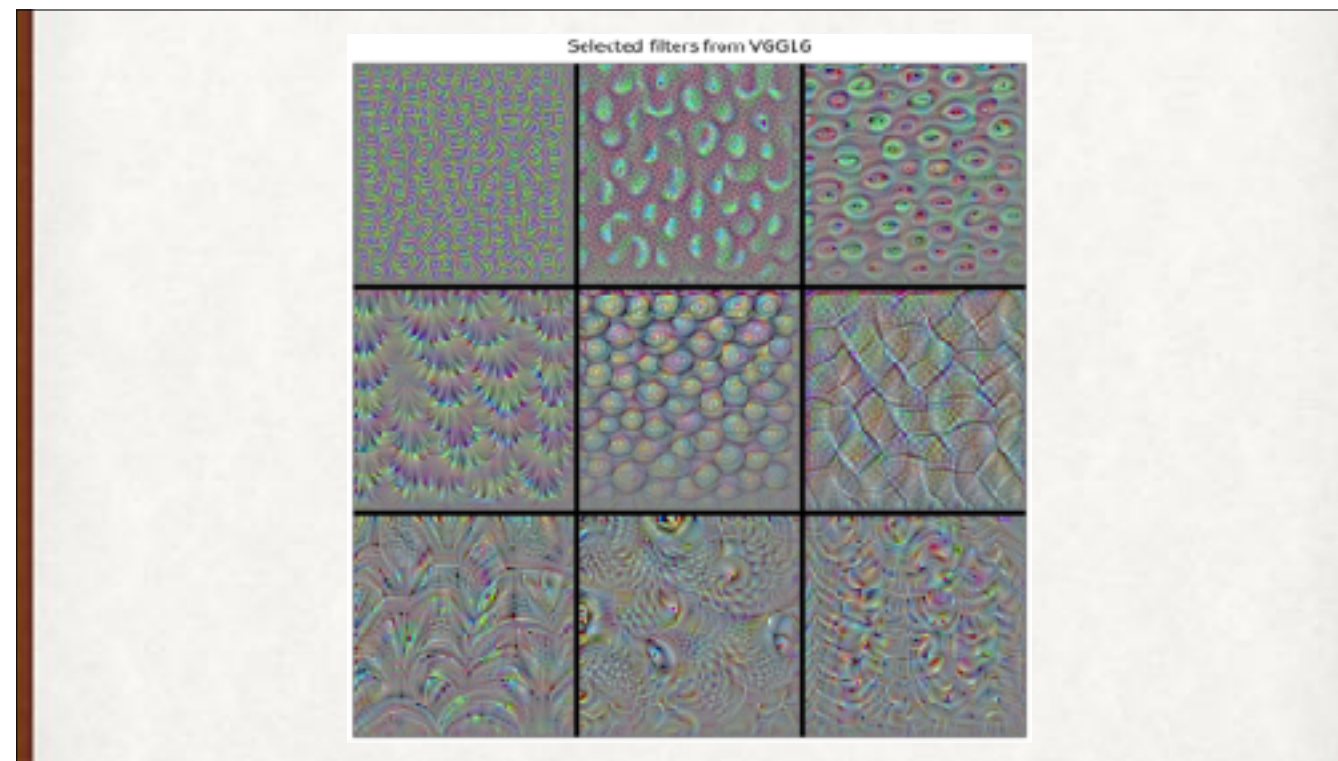
Visualizing filters from the fourth block of VGG16.



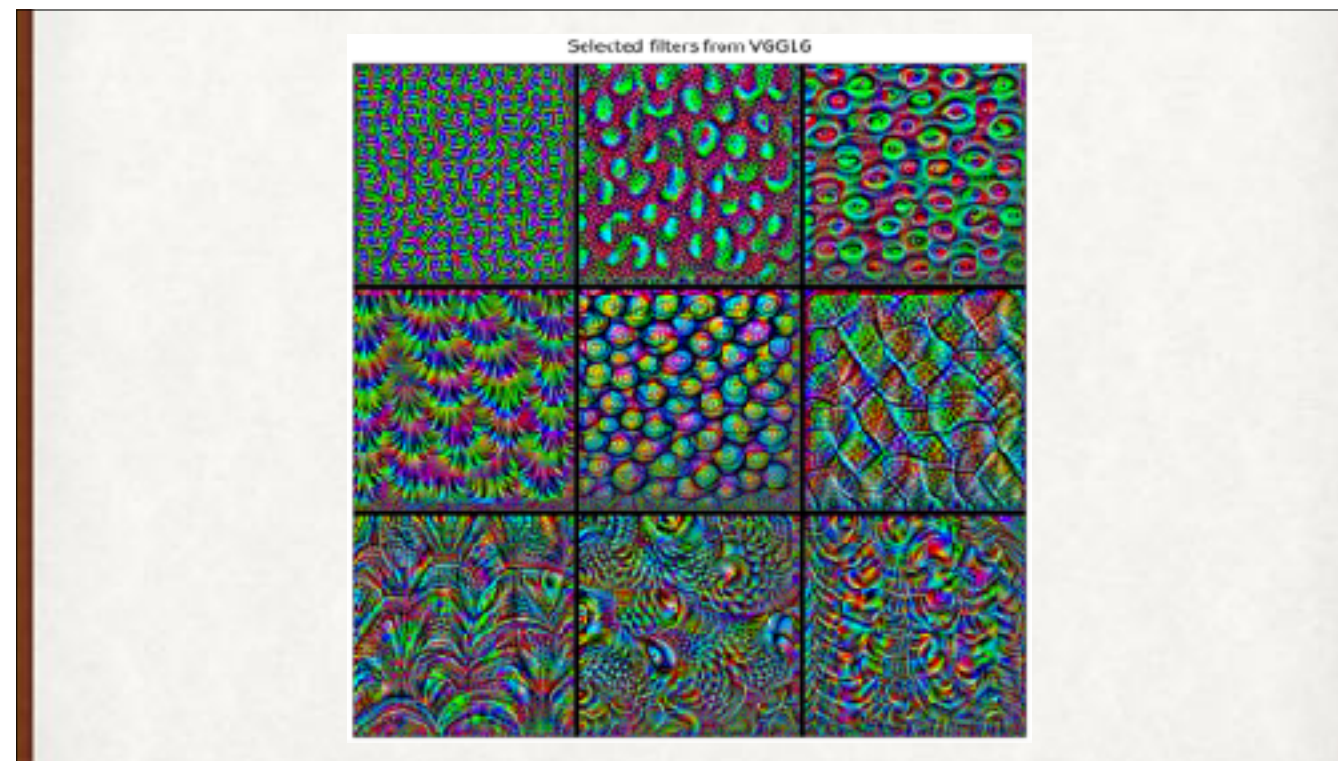
Some filter responses I chose by hand from the complete set because they looked cool.



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



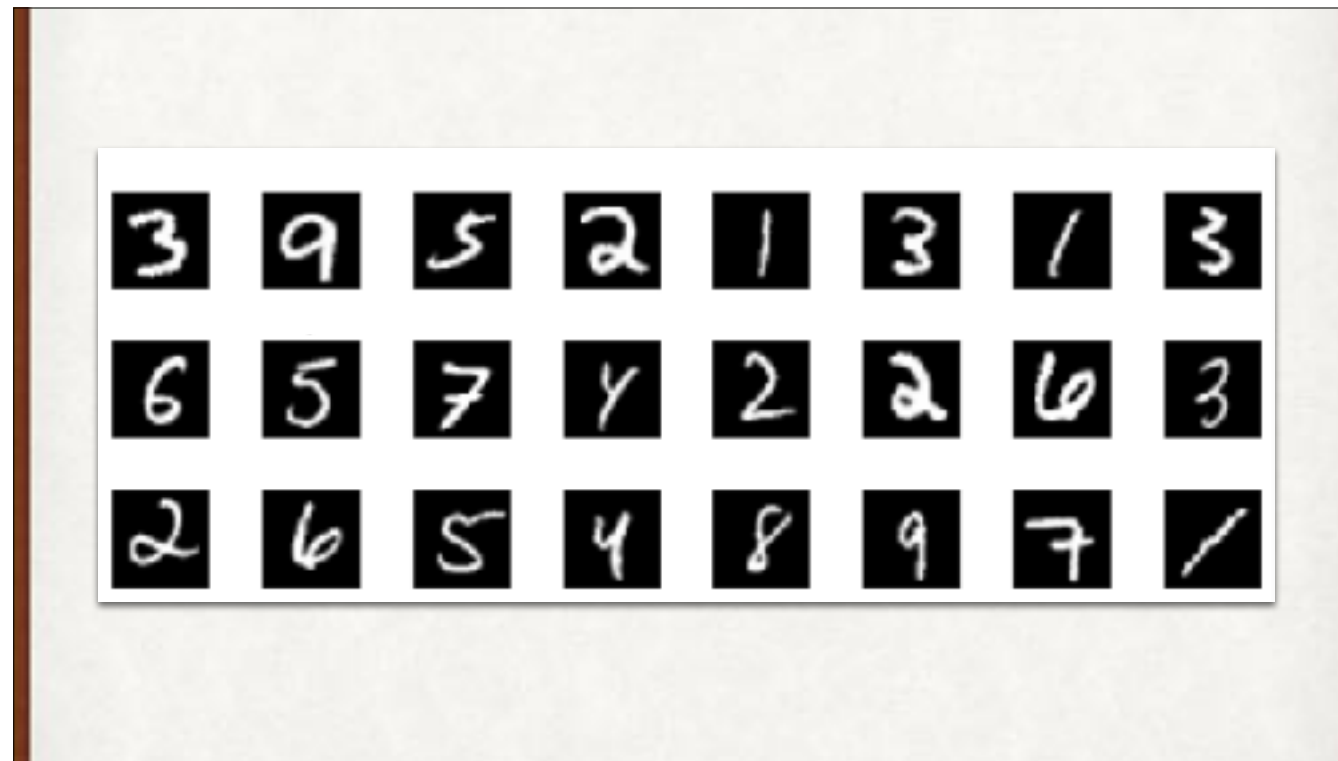
Some more cool filter responses I picked by hand.



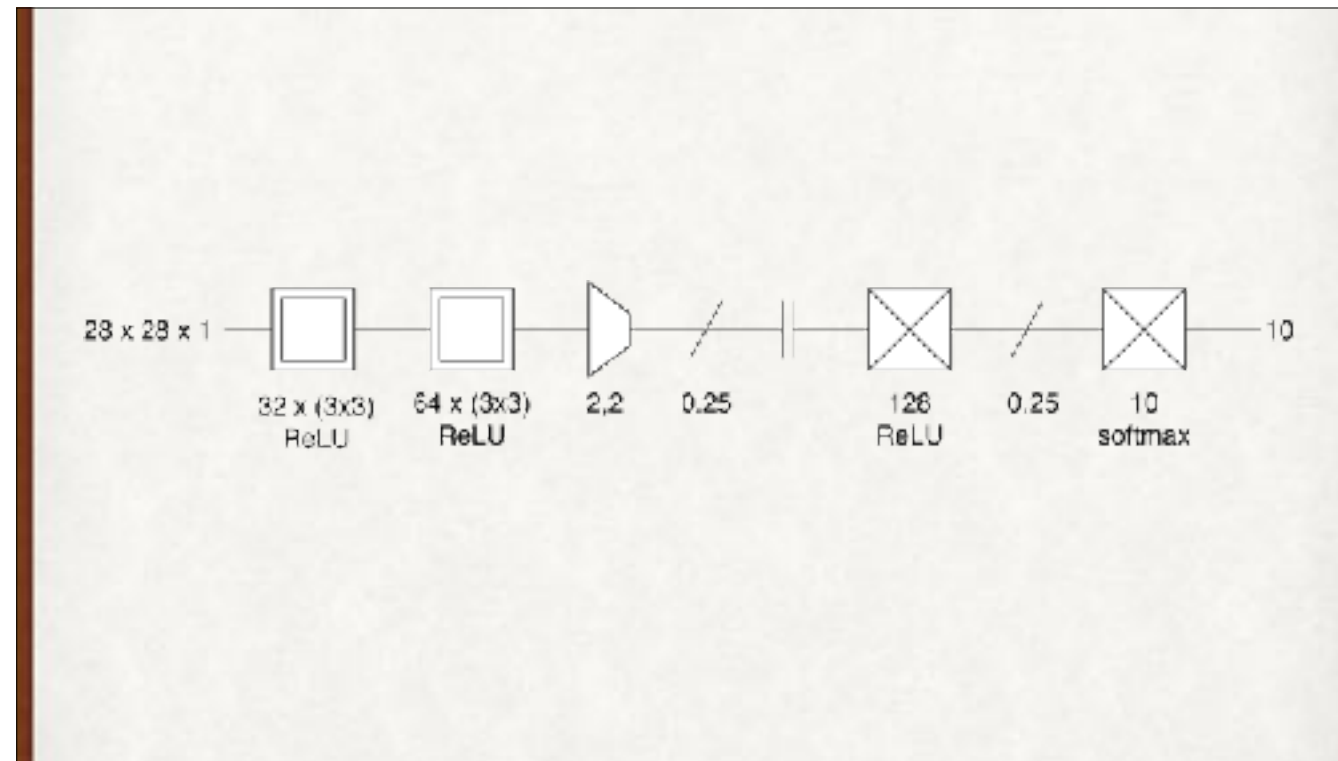
Some more cool filter responses I picked by hand.

Back to MNIST

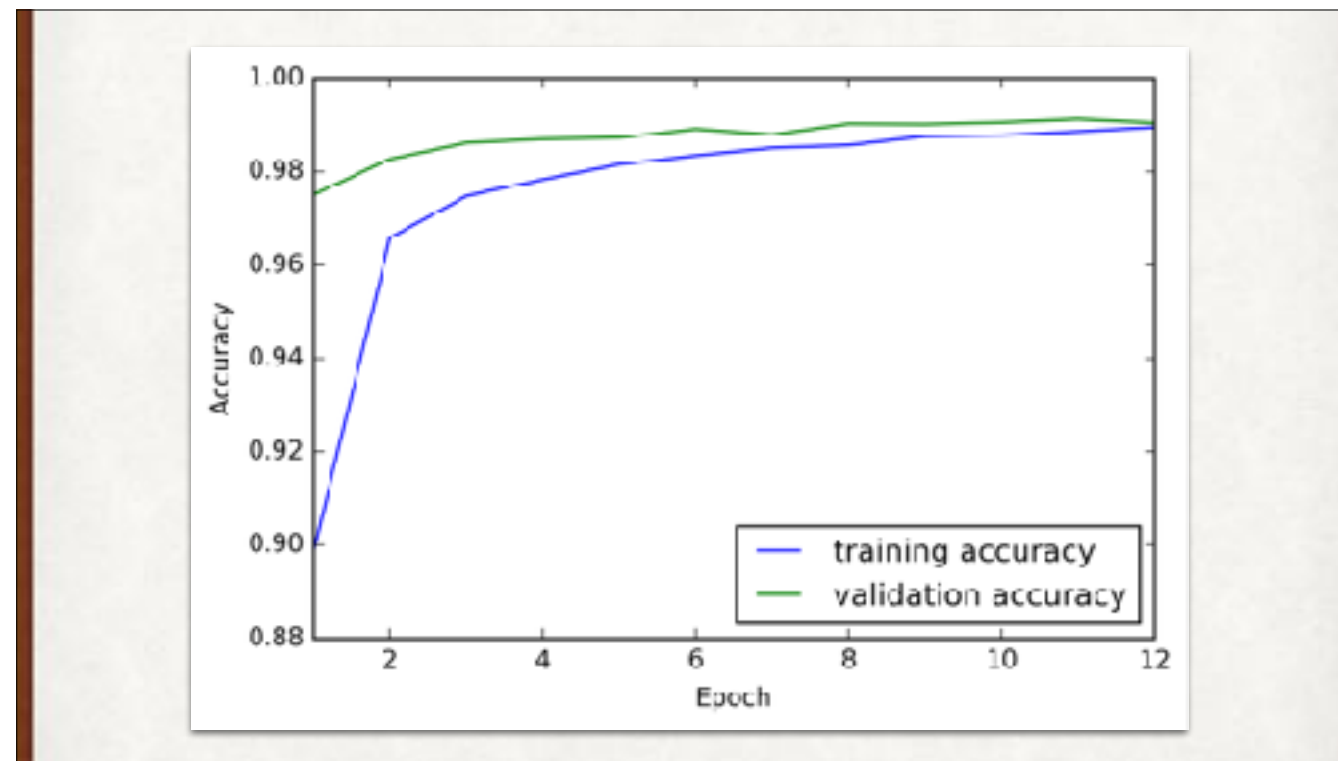
Back to our original problem. Let's use convolution to identify handwritten digits.



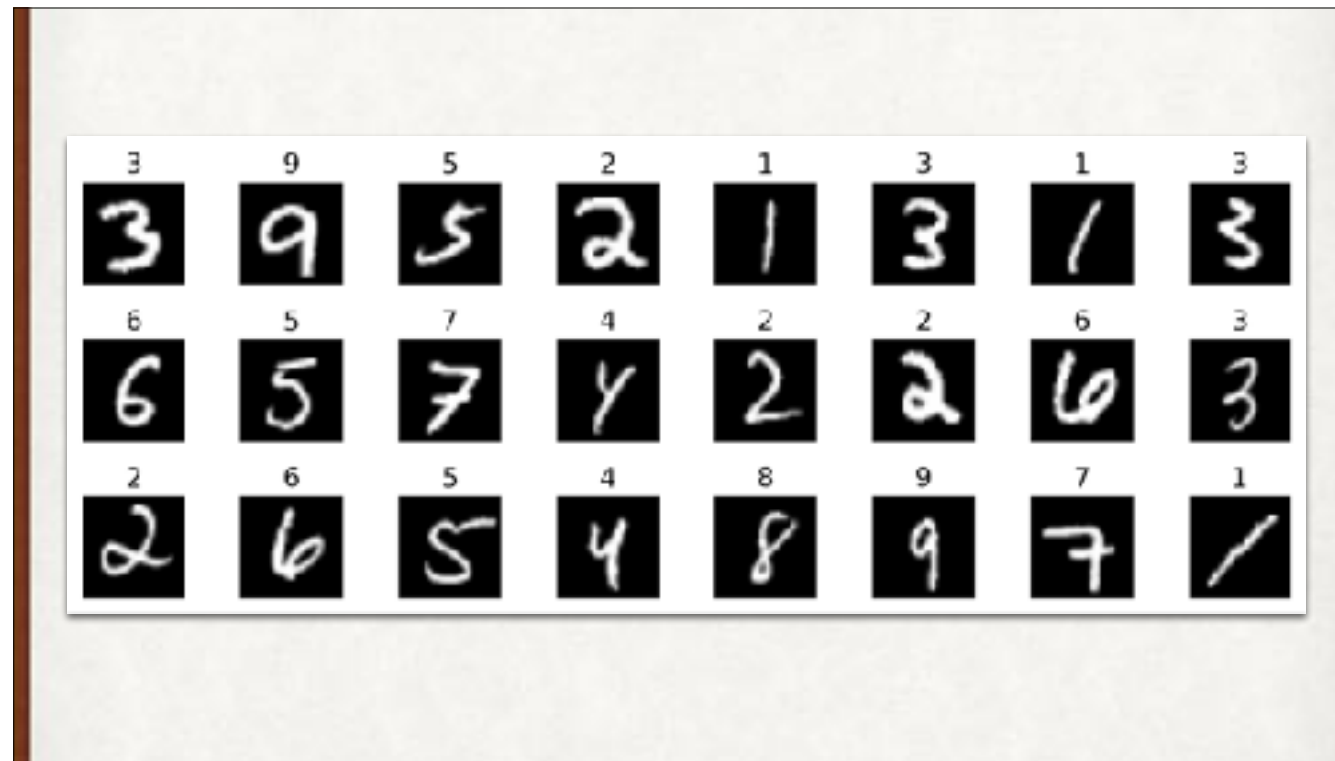
Back to our original problem. Let's use convolution to identify handwritten digits.



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.



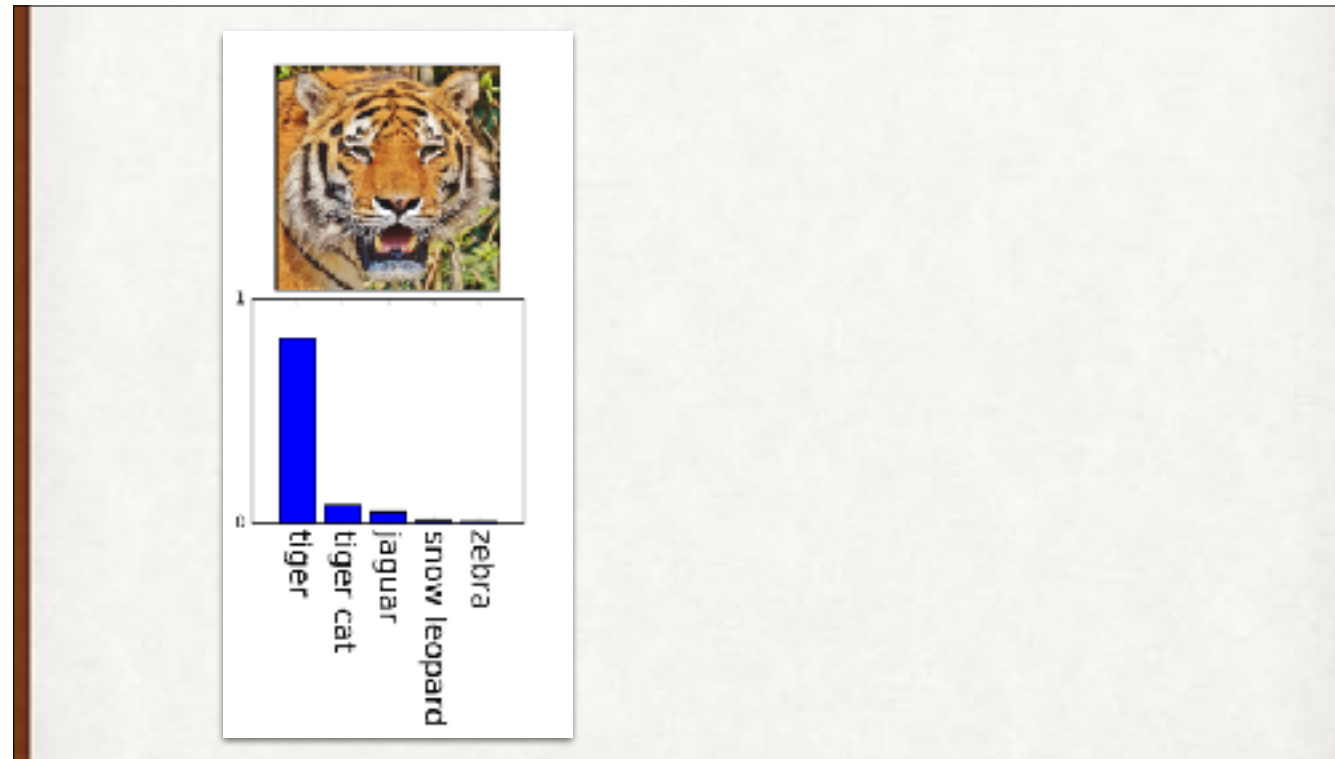
Performance of our CNN. Just about 99% accuracy on the MNIST test data after 12 epochs. Pretty great!



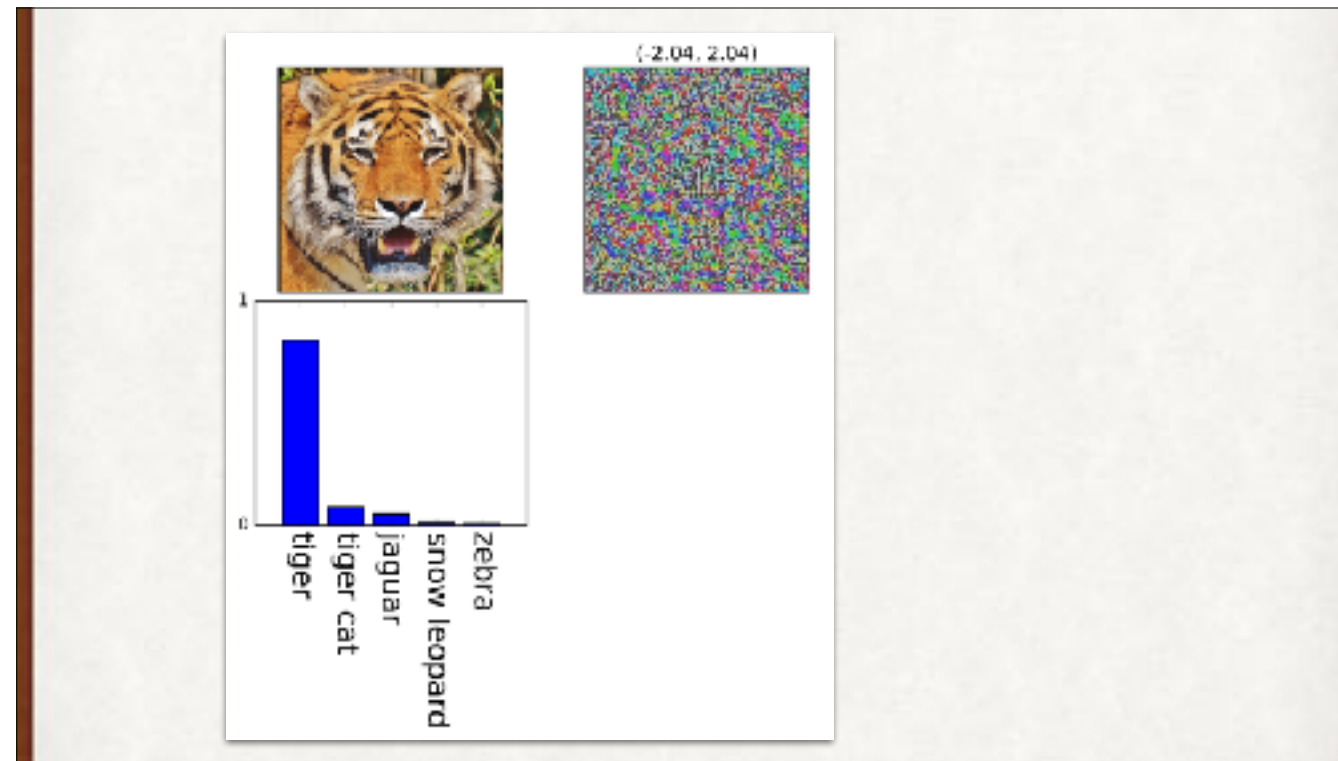
MNIST results from our CNN. All correct!

Adversaries

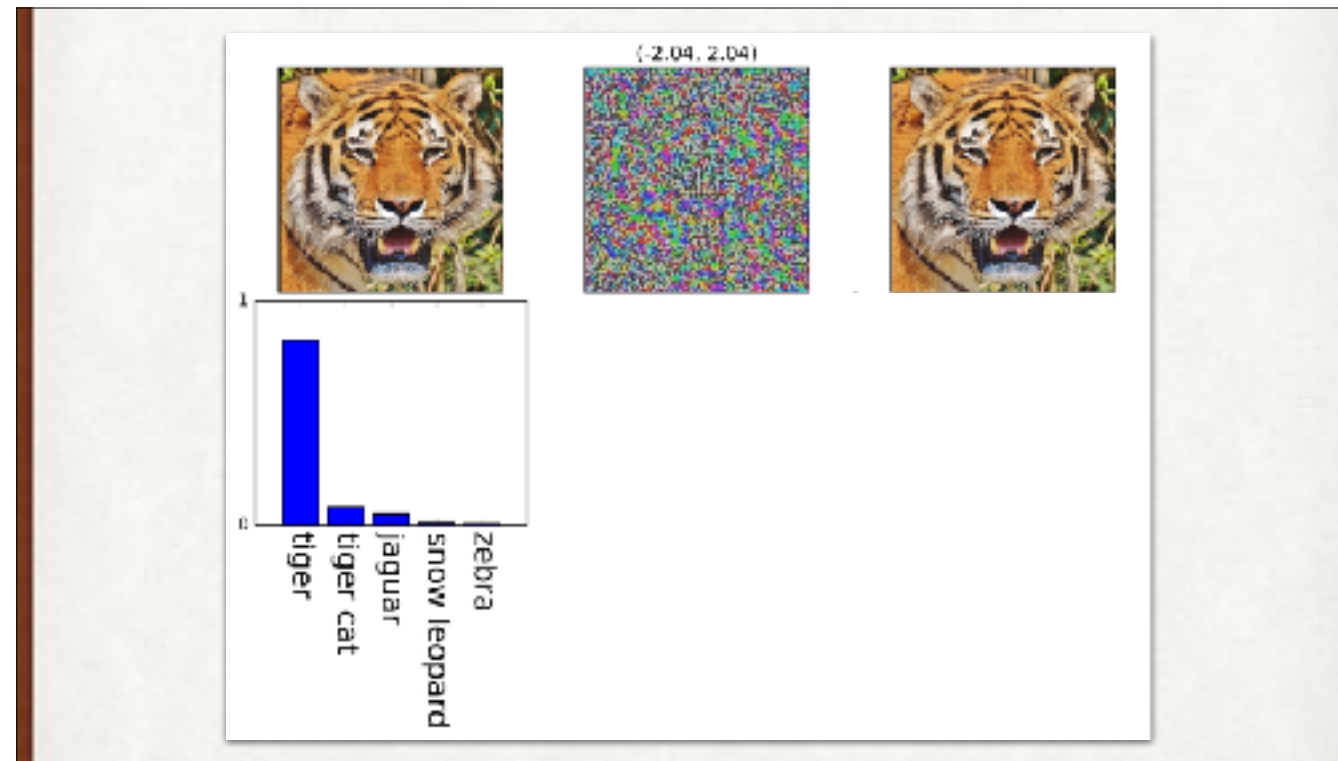
A CNN can be tricked by adding just a little bit of carefully-craft perturbations, called an adversary, to an image. To our eye, the image is unchanged. But the CNN thinks it's a different object.



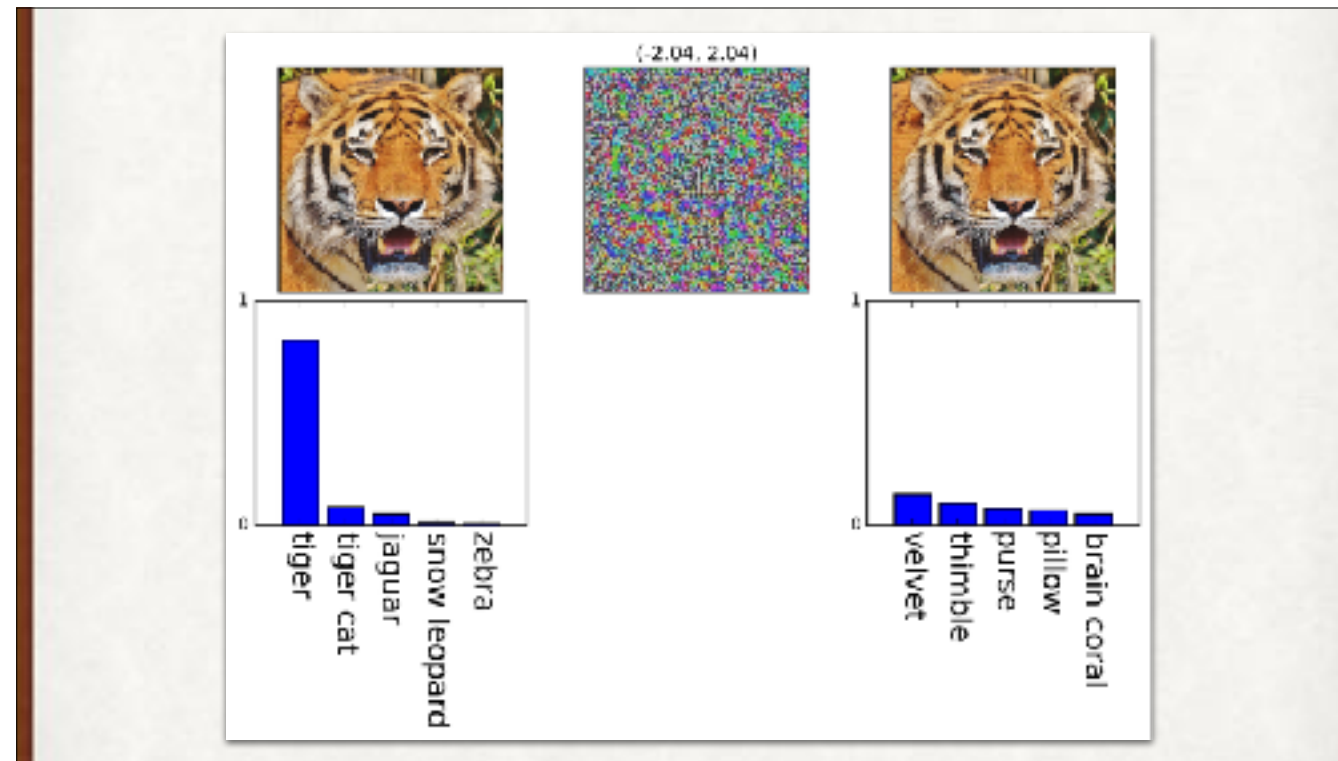
A tiger (left) is correctly identified, but add some carefully constructed noise (middle) in the range of about $(-2,2)$, and VGG16's predictions (right) are... weird.



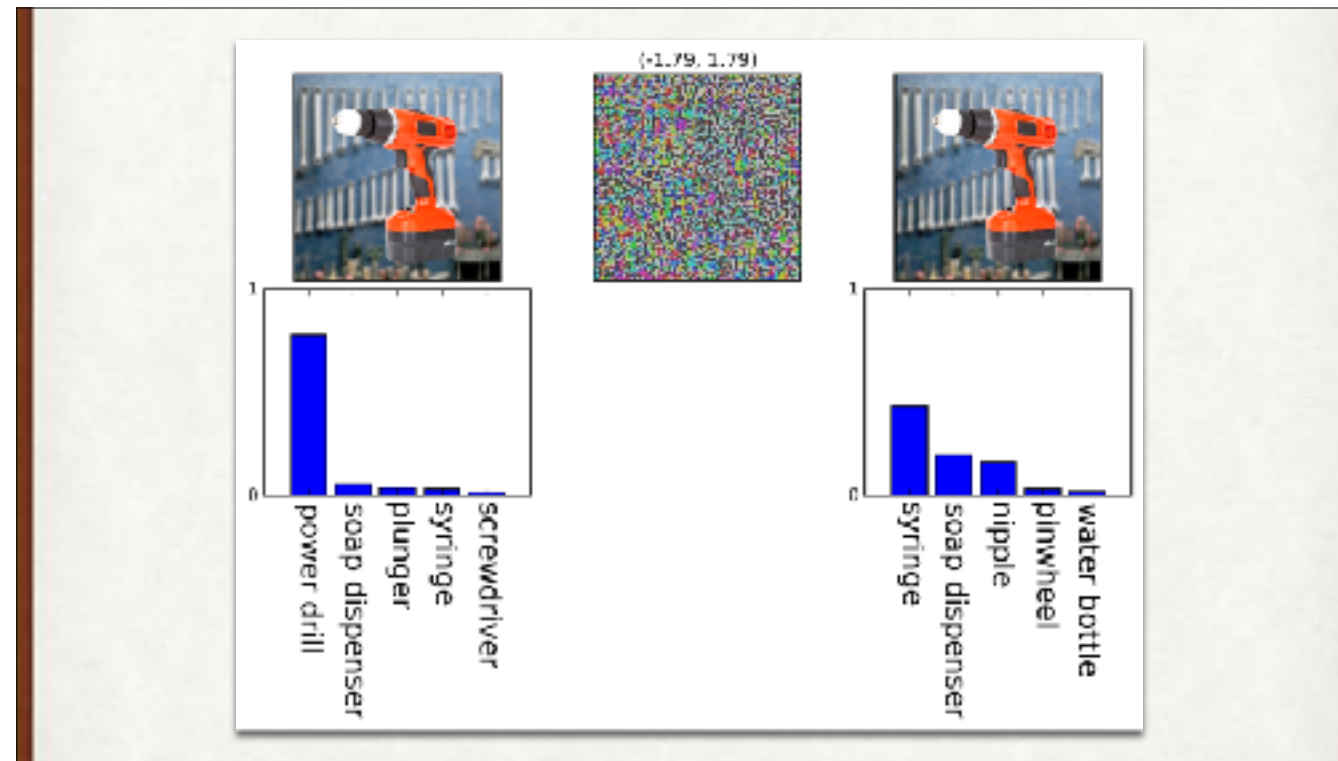
A tiger (left) is correctly identified, but add some carefully constructed noise (middle) in the range of about $(-2,2)$, and VGG16's predictions (right) are... weird.



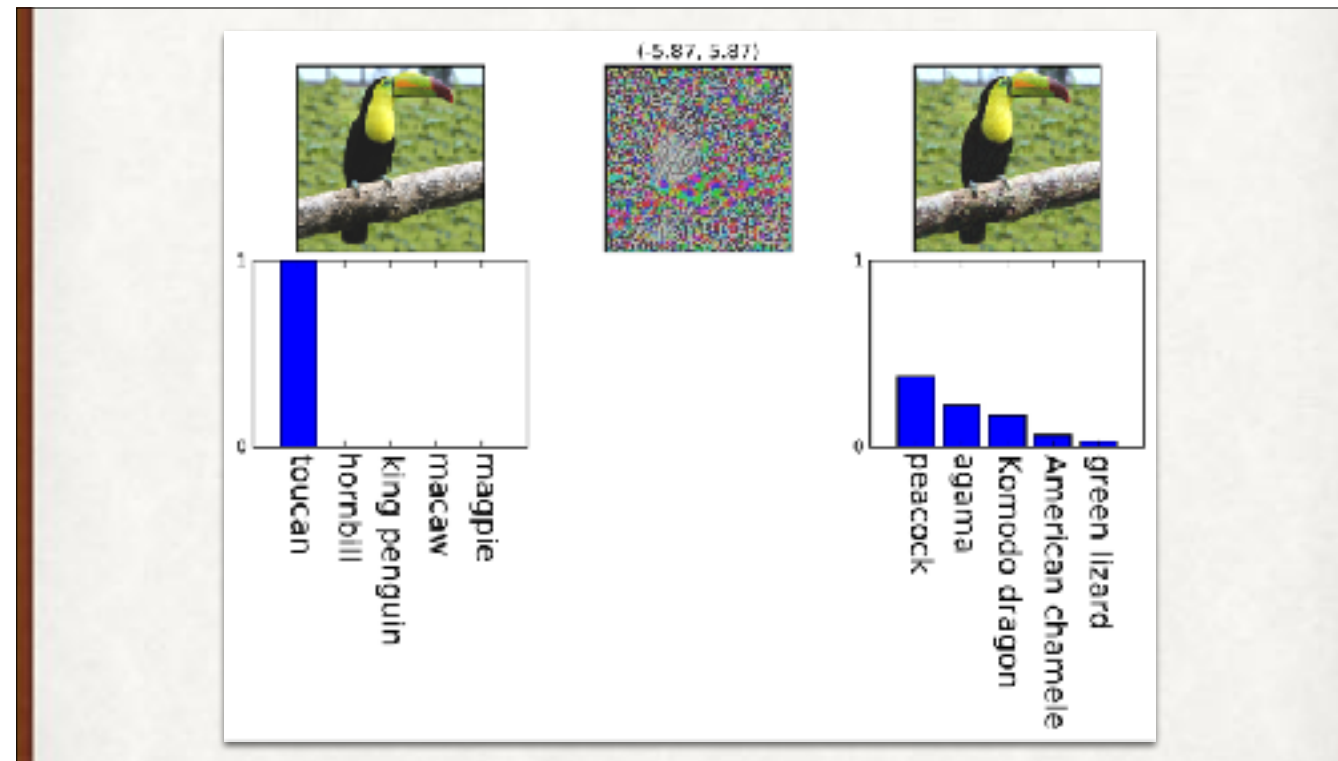
A tiger (left) is correctly identified, but add some carefully constructed noise (middle) in the range of about $(-2,2)$, and VGG16's predictions (right) are... weird.



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?



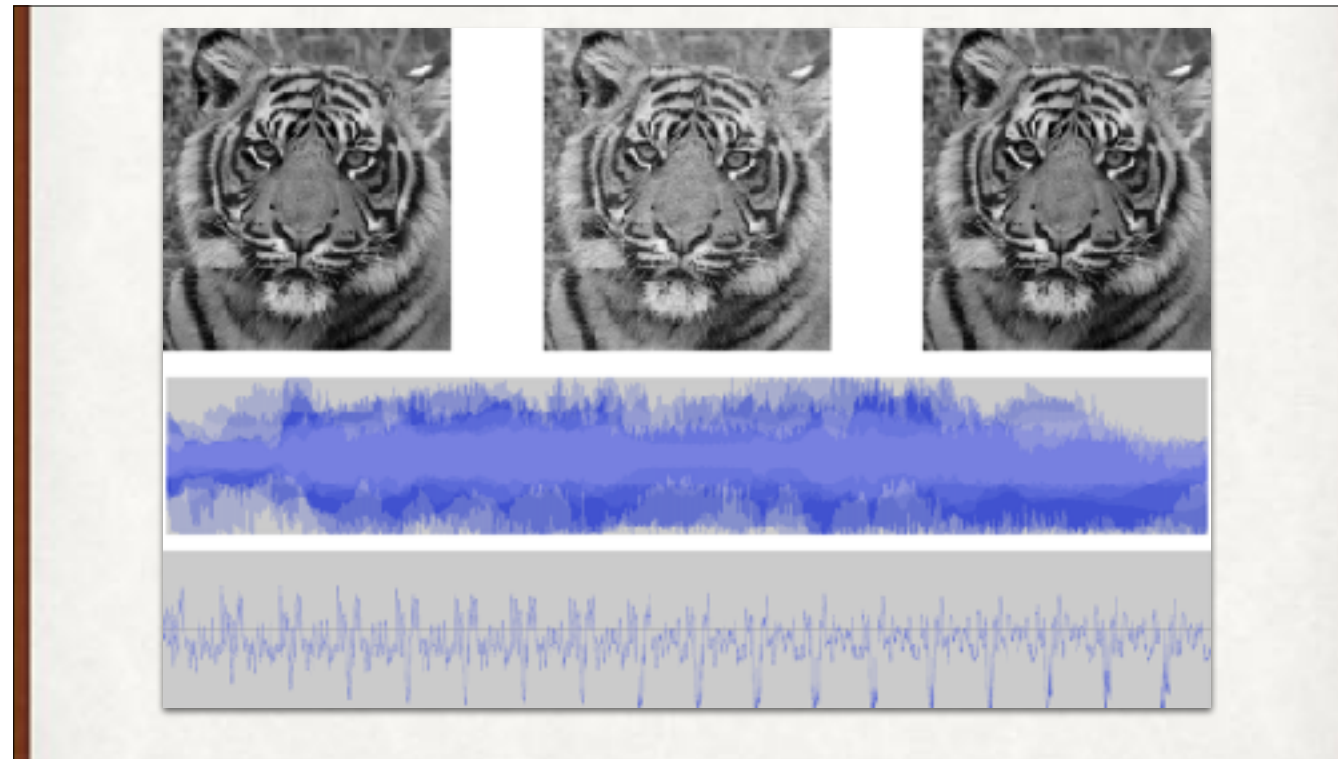
This is absolutely a toucan, but then probably a peacock, or one of several different lizards.



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

Autoencoders

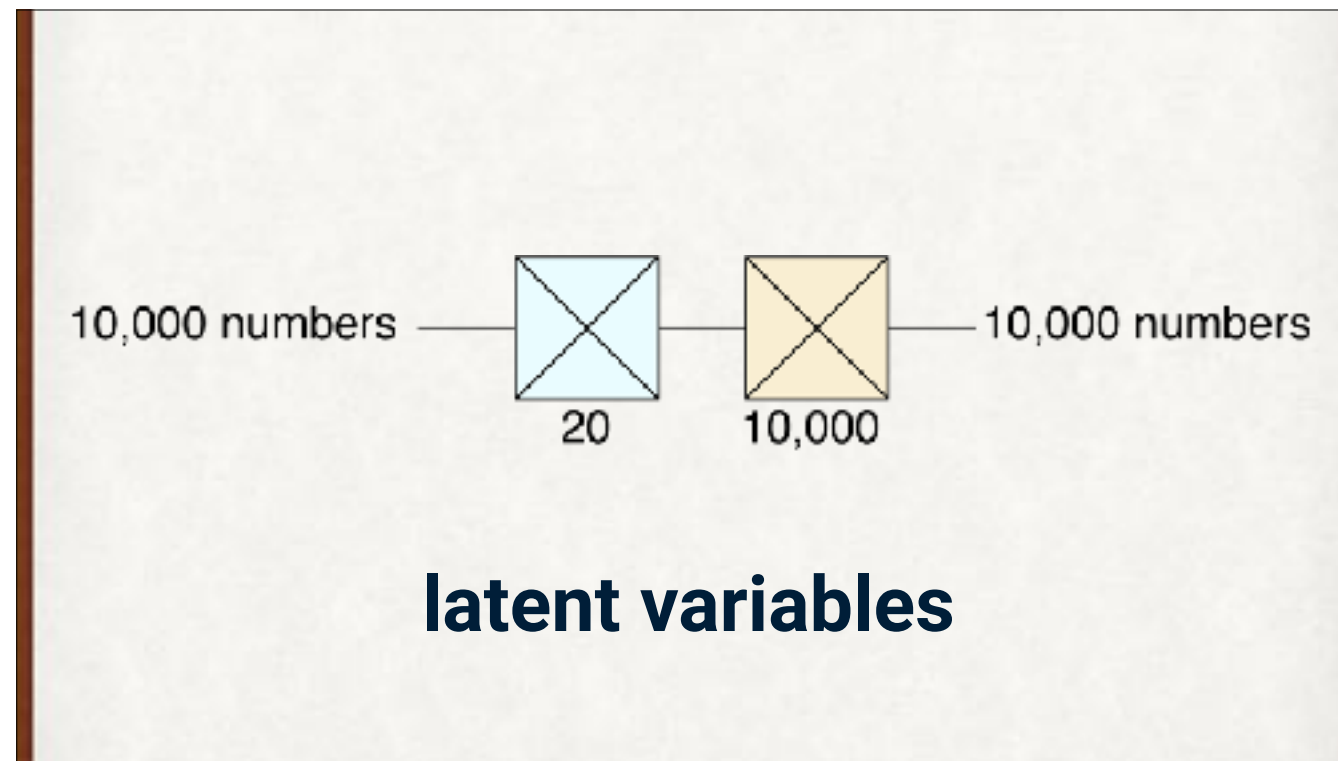
An autoencoder learns how to “automatically” compress, or “encode,” an input into a smaller size that can later be decoded into a version of the original that is acceptable for a given use.



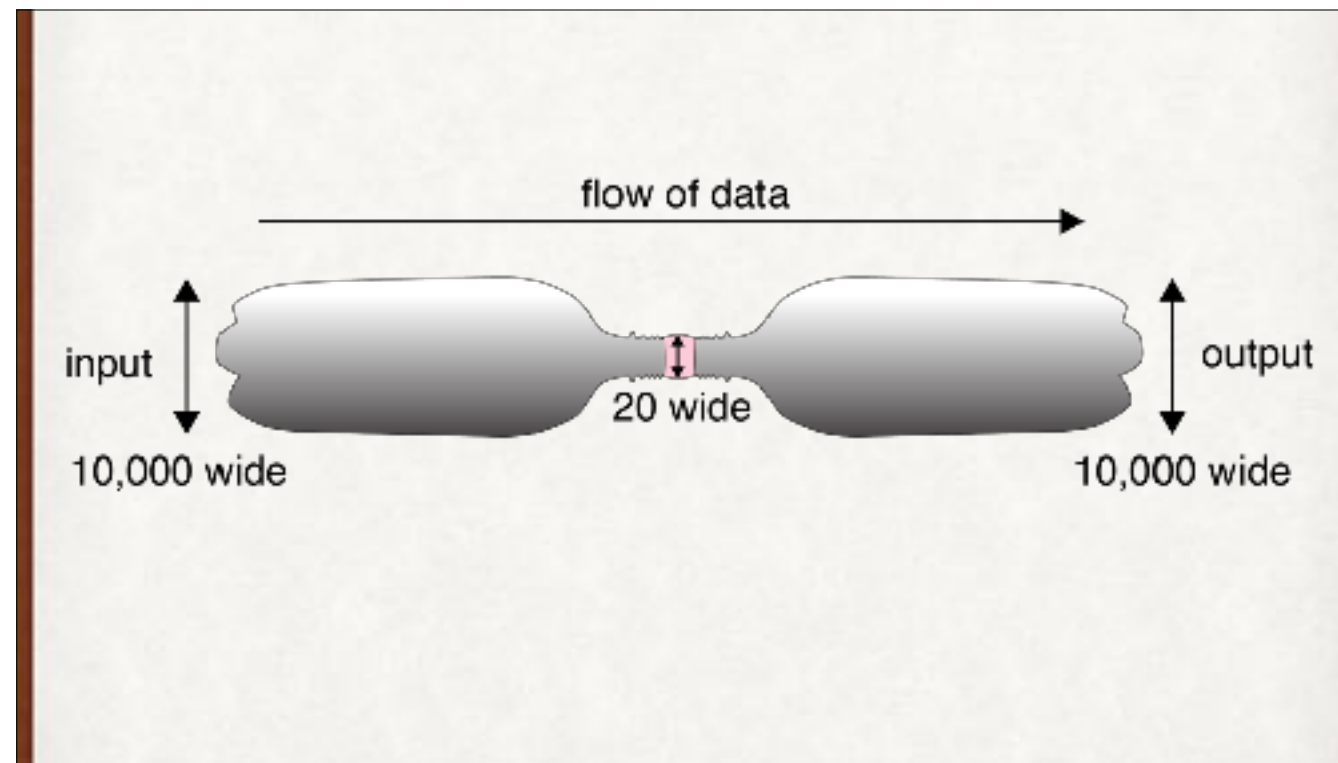
Compression of an image. Upper-left: original tiger. Upper-middle: tiger data compressed with MP3 and then decompressed. Upper-right: compressed by JPG. Below, the original tiger data in WAV format. Bottom: Close-up of the start of the waveform. The dips correspond to the black tip of the ear in the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



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.



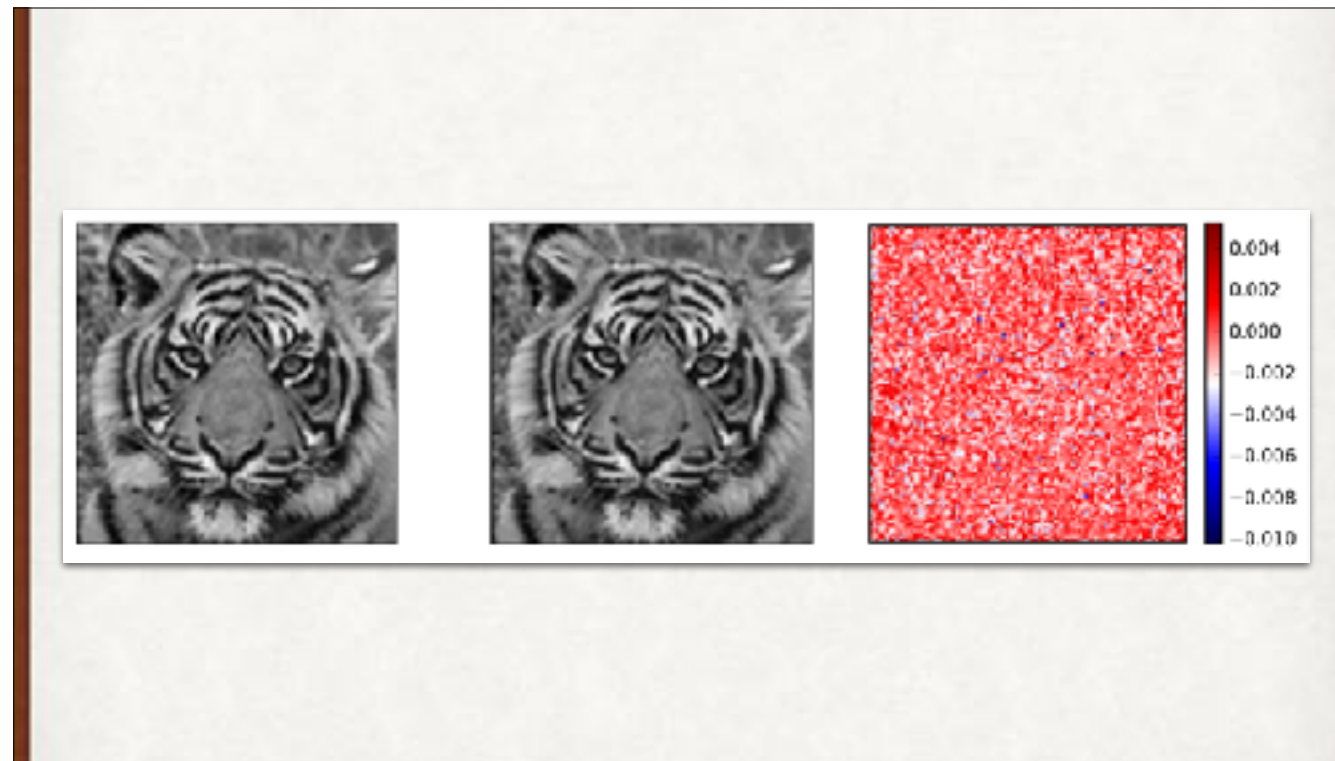
Many autoencoders have a “bottleneck” where the data is compressed to a minimum.



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.



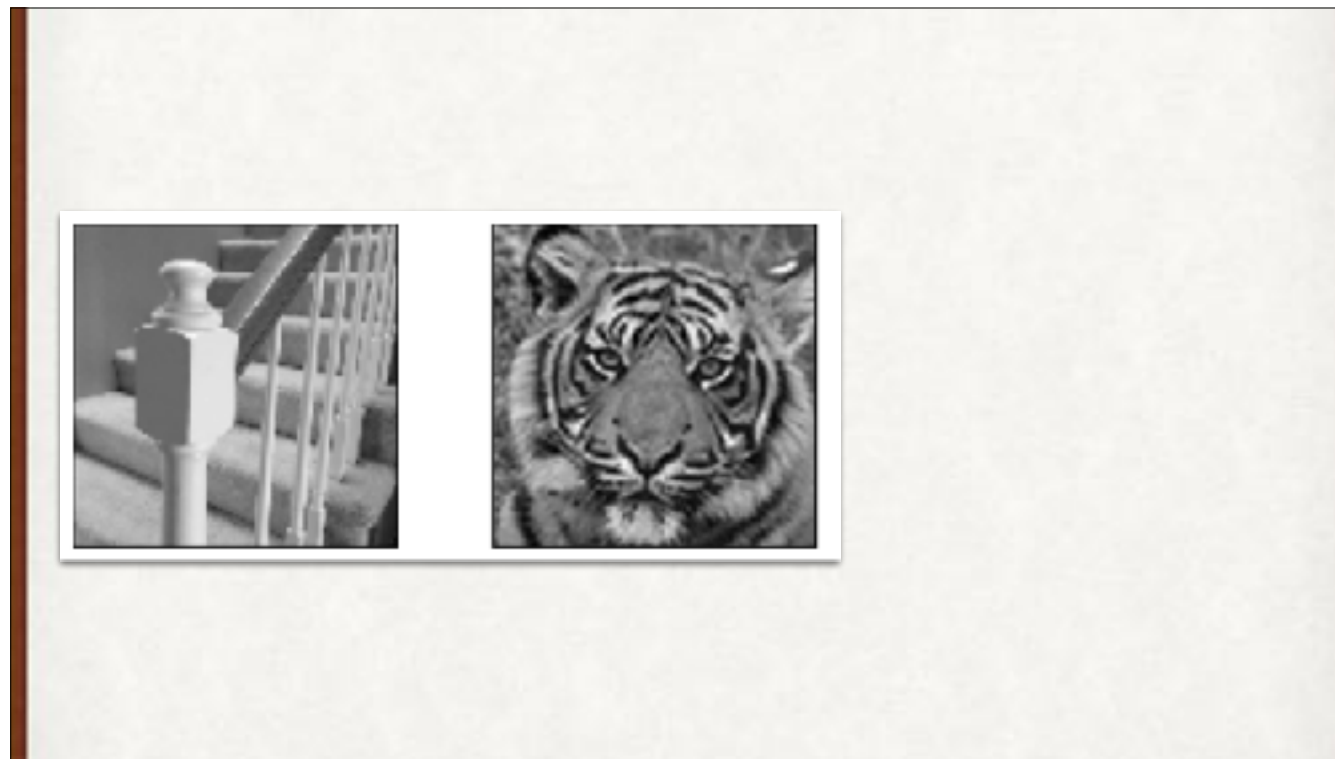
We trained the autoencoder on the picture of the tiger. Here are the results. Left: the input, 100x100. Middle: the autoencoder's output. Right: The pixel-by-pixel differences between input and output. The autoencoder seems to have found an amazing way to compress images super well with only 20/10,000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



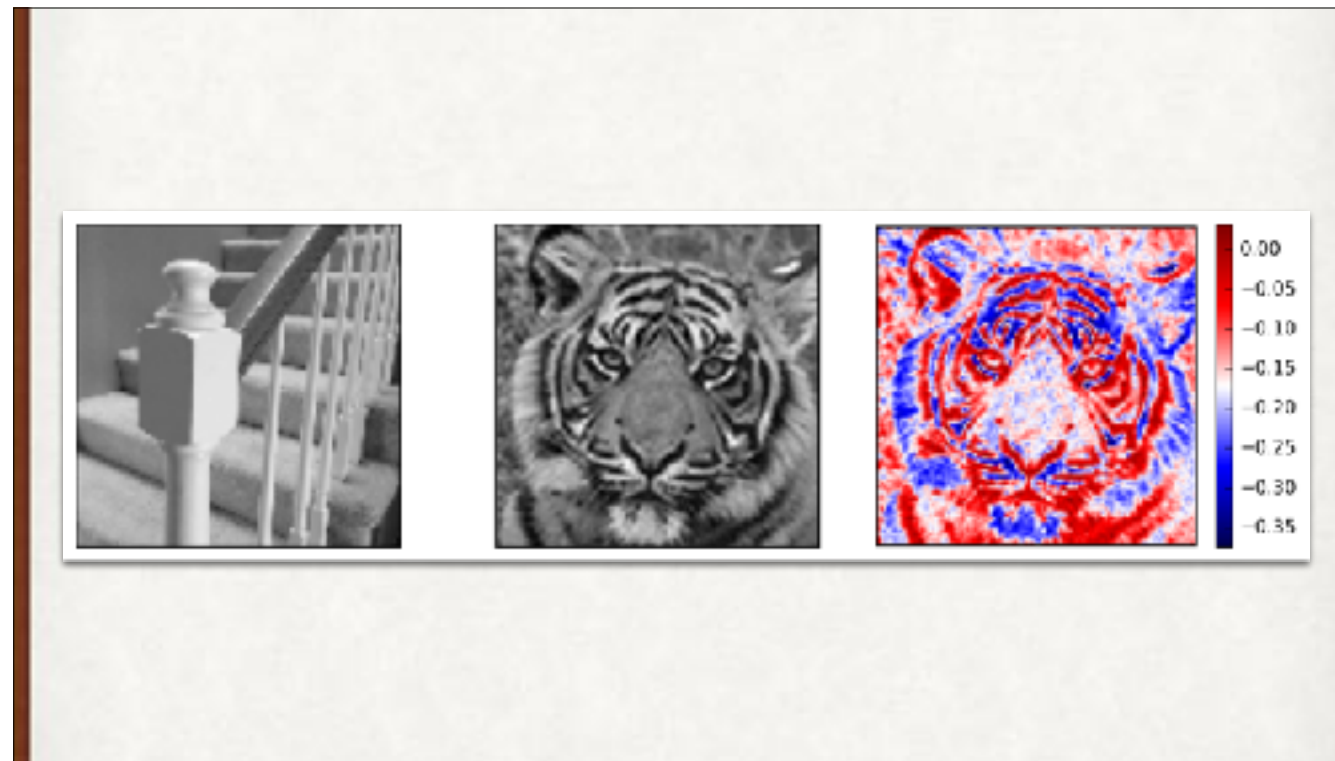
We trained the autoencoder on the picture of the tiger. Here are the results. Left: the input, 100x100. Middle: the autoencoder's output. Right: The pixel-by-pixel differences between input and output. The autoencoder seems to have found an amazing way to compress images super well with only 20/10,000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



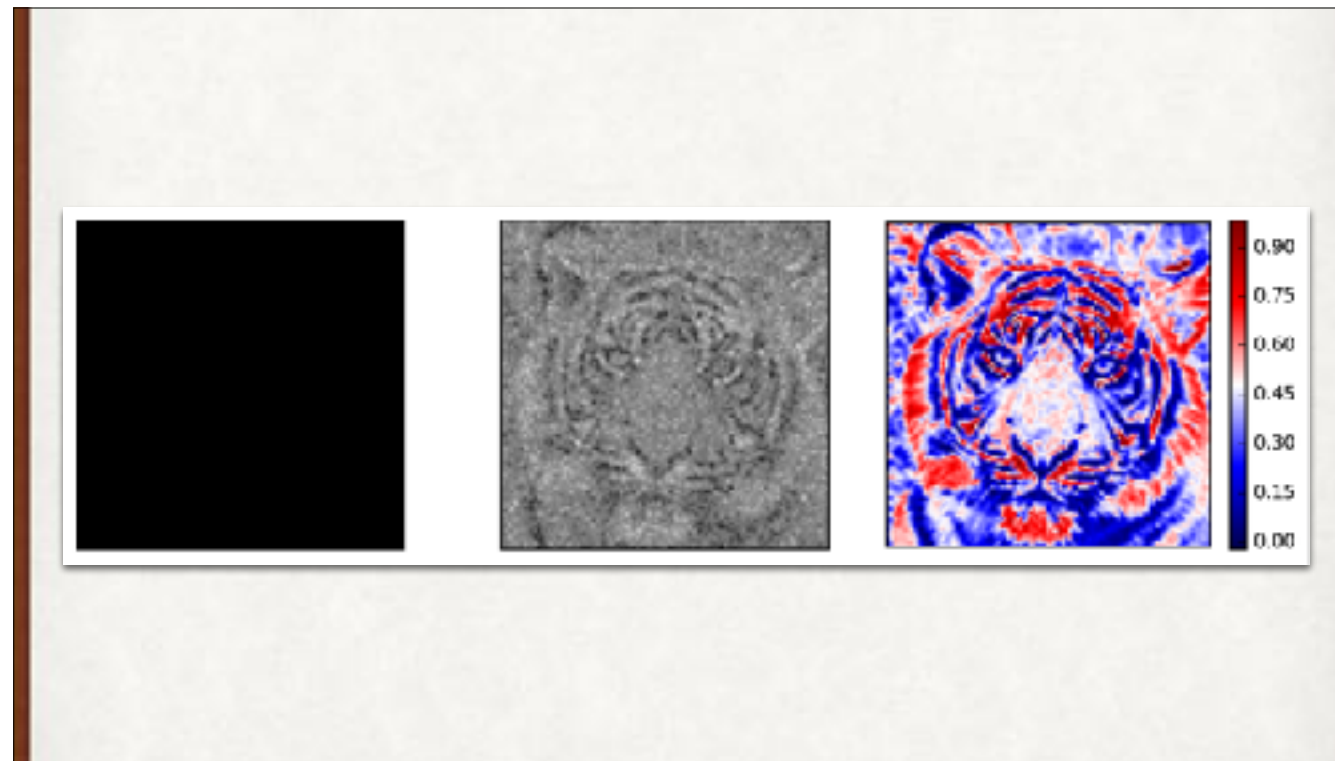
Let's try feeding the autoencoder a picture of a bannister. Left: input. Middle: the output is the tiger! What? Right: pixel differences.



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.

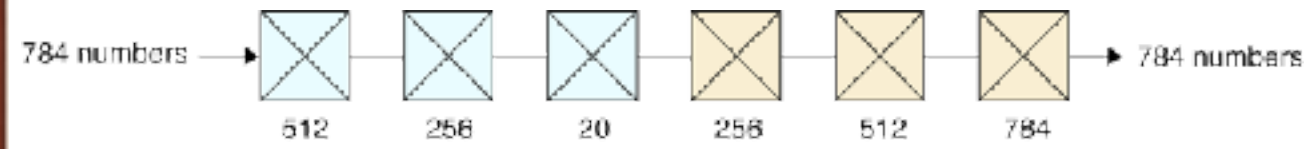


A black input (left) still produces the tiger (middle). Differences on the far right. So this network isn't compressing anything, really, it's just memorized how to produce a tiger image. The bias values in the neurons have "remembered" values that cause the system to produce a tiger.



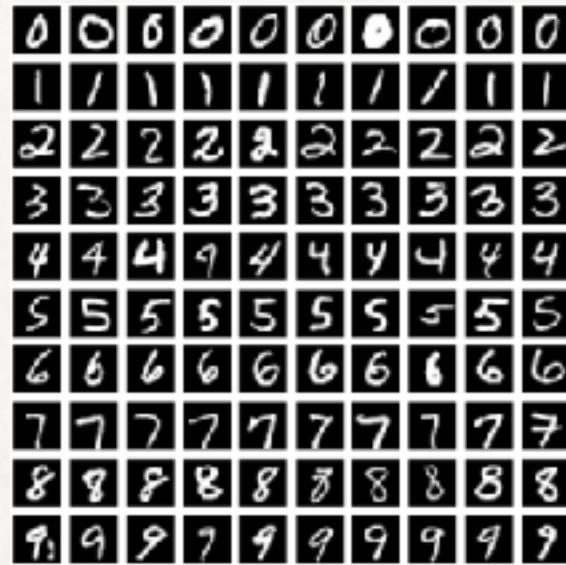
Not so amazing after all! When trained on diverse images, turning a whole picture into just 20 numbers isn't going to let us get back much of anything. Left: input. Middle: output. Right: pixel differences.

MNIST Autoencoder

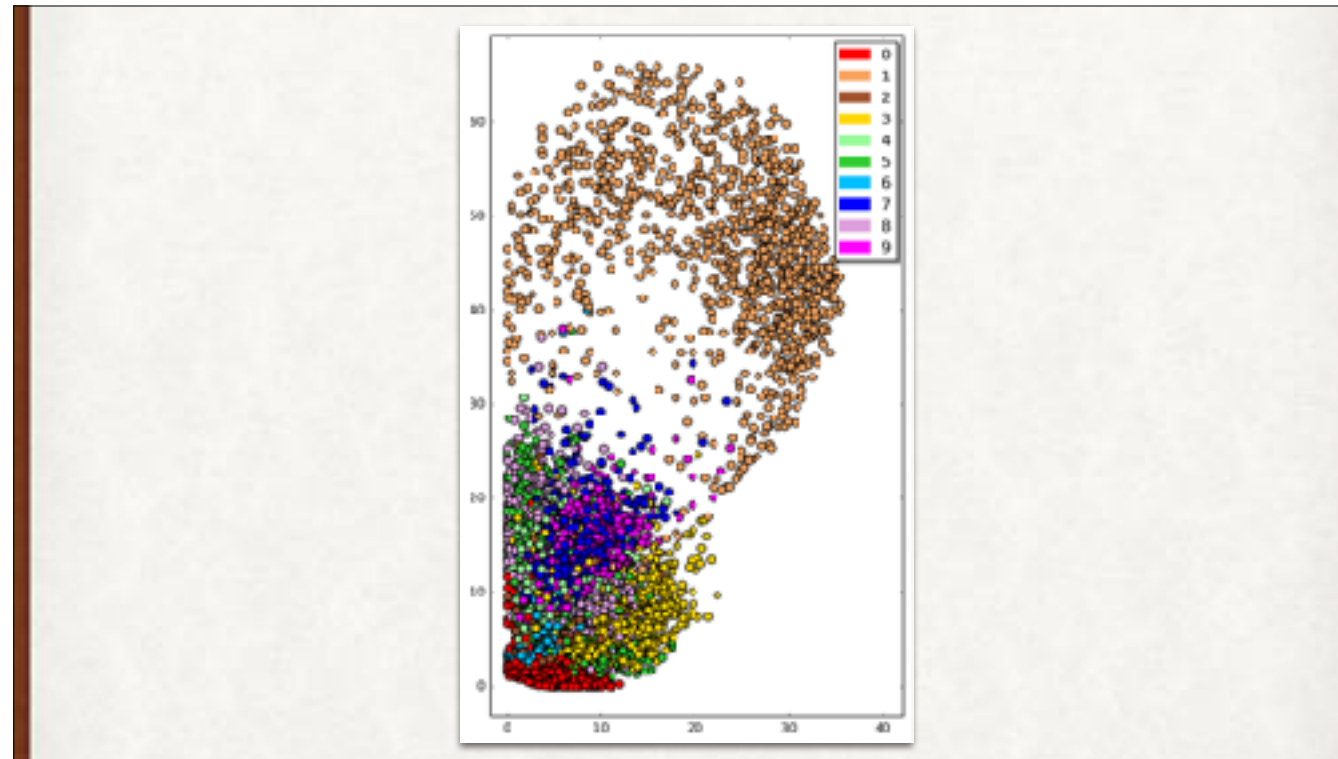


An autoencoder for MNIST data, with more fully-connected layers.

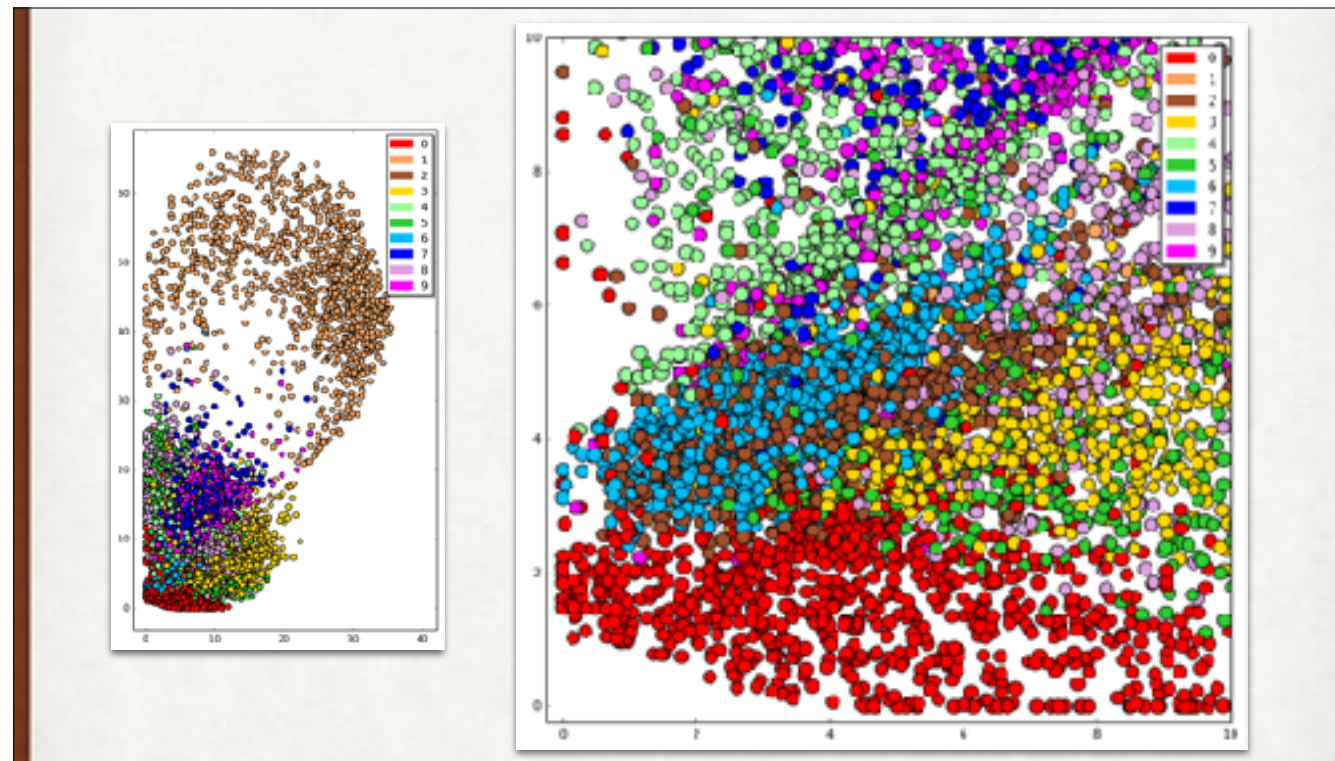
MNIST Input



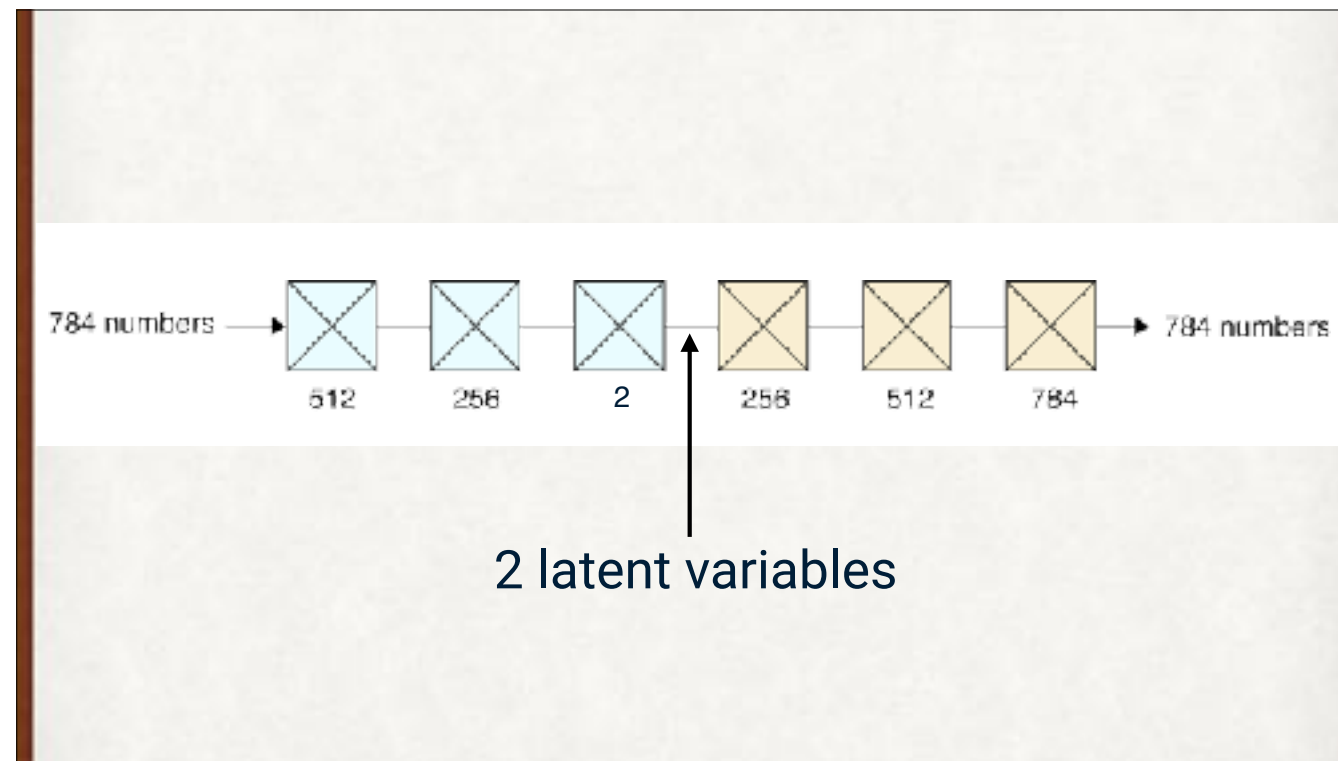
Now let's try the MNIST data, which is way easier than the tiger.



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.

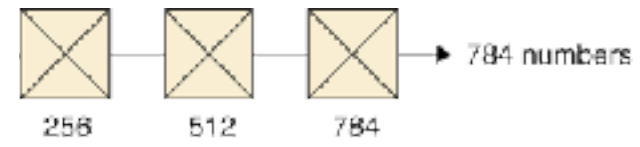


A close-up of the bottom left.



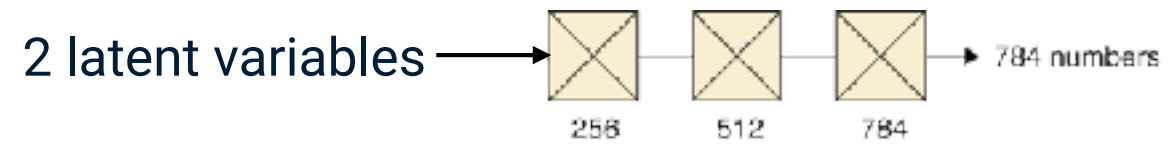
Let's reduce this down to just 2 latent variables.

Just the decoder

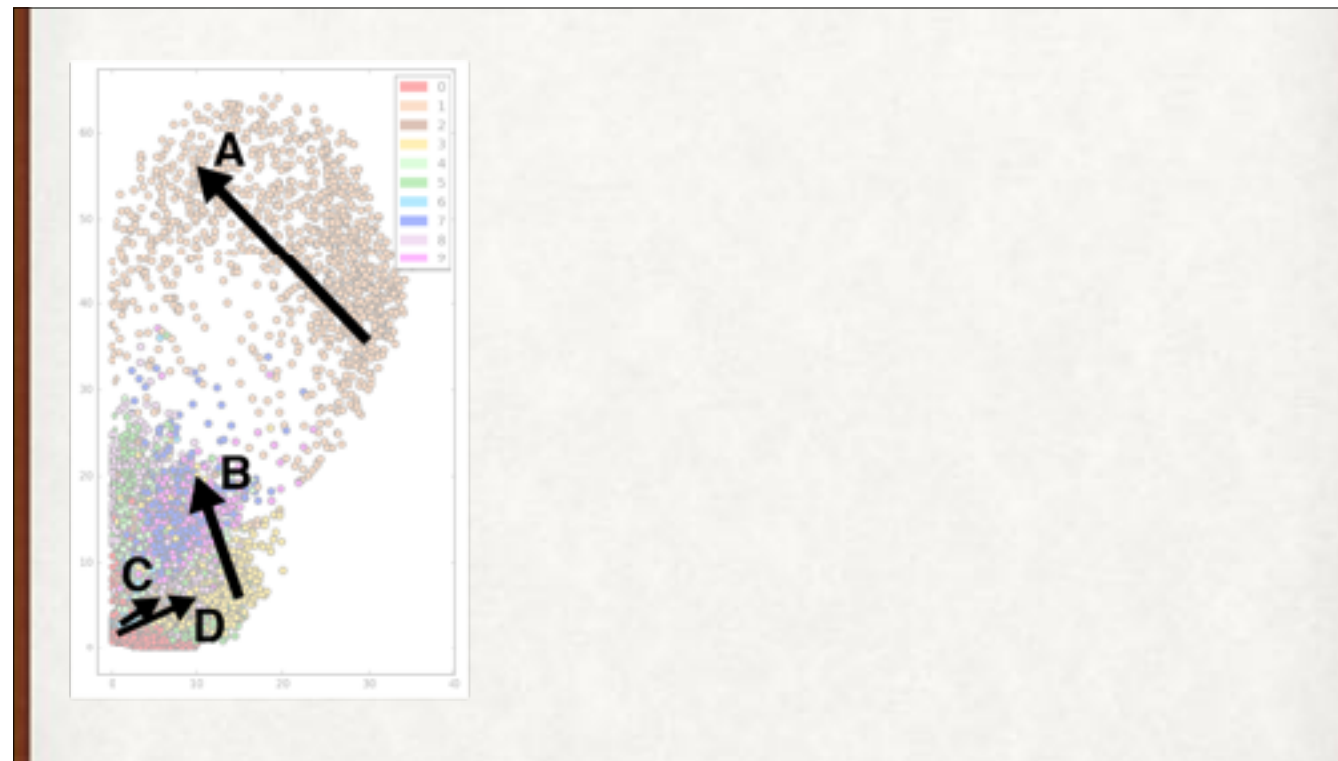


Let's reduce this down to just 2 latent variables.

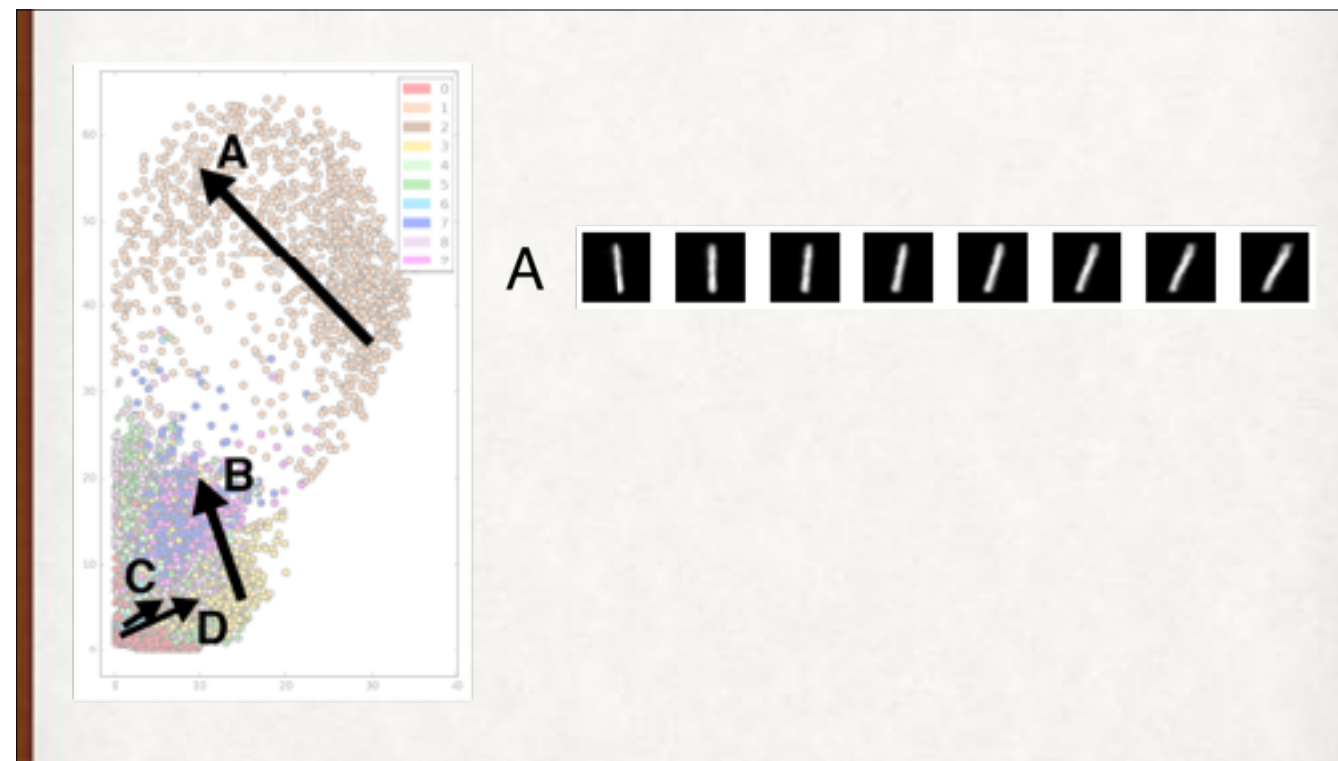
Just the decoder



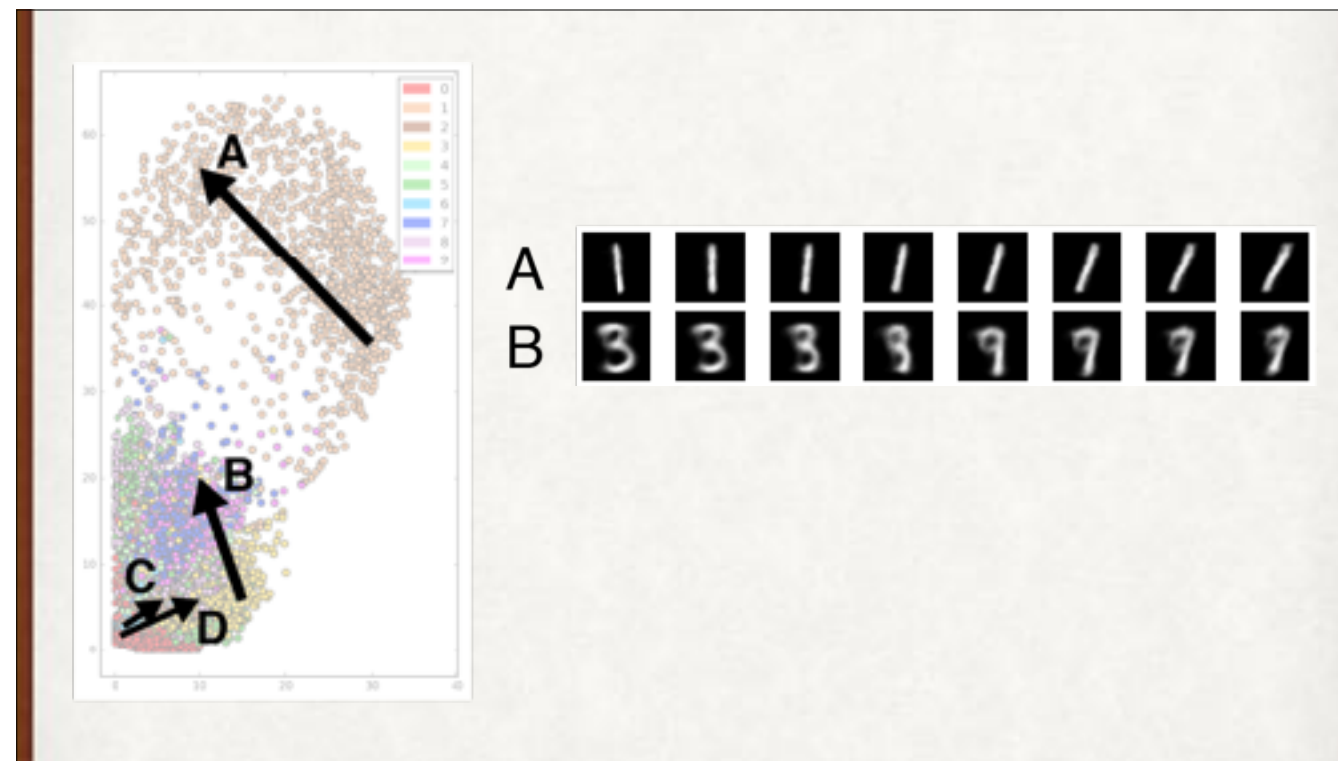
Let's reduce this down to just 2 latent variables.



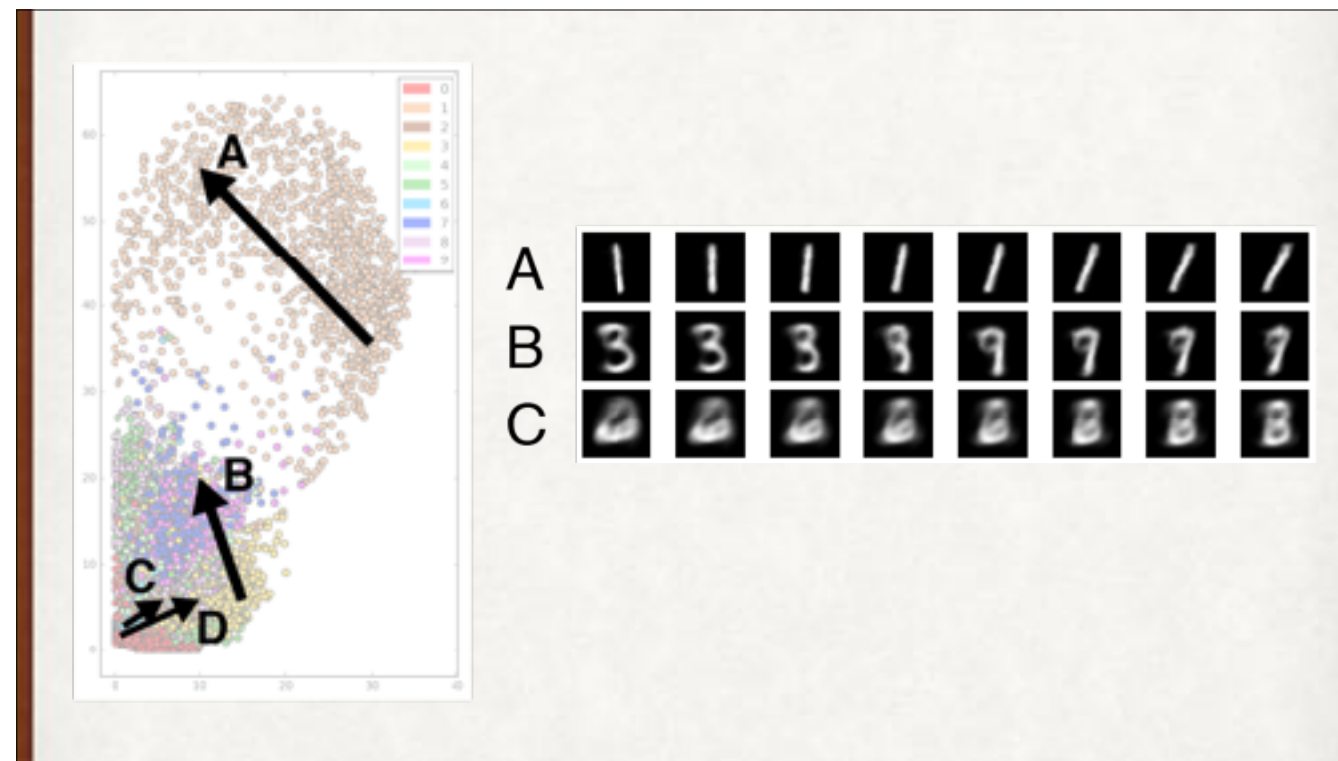
Blending in latent space. Each row shows equally-spaced steps along the corresponding arrow. Interpolated “digits” aren’t great, but they’re not random nonsense.



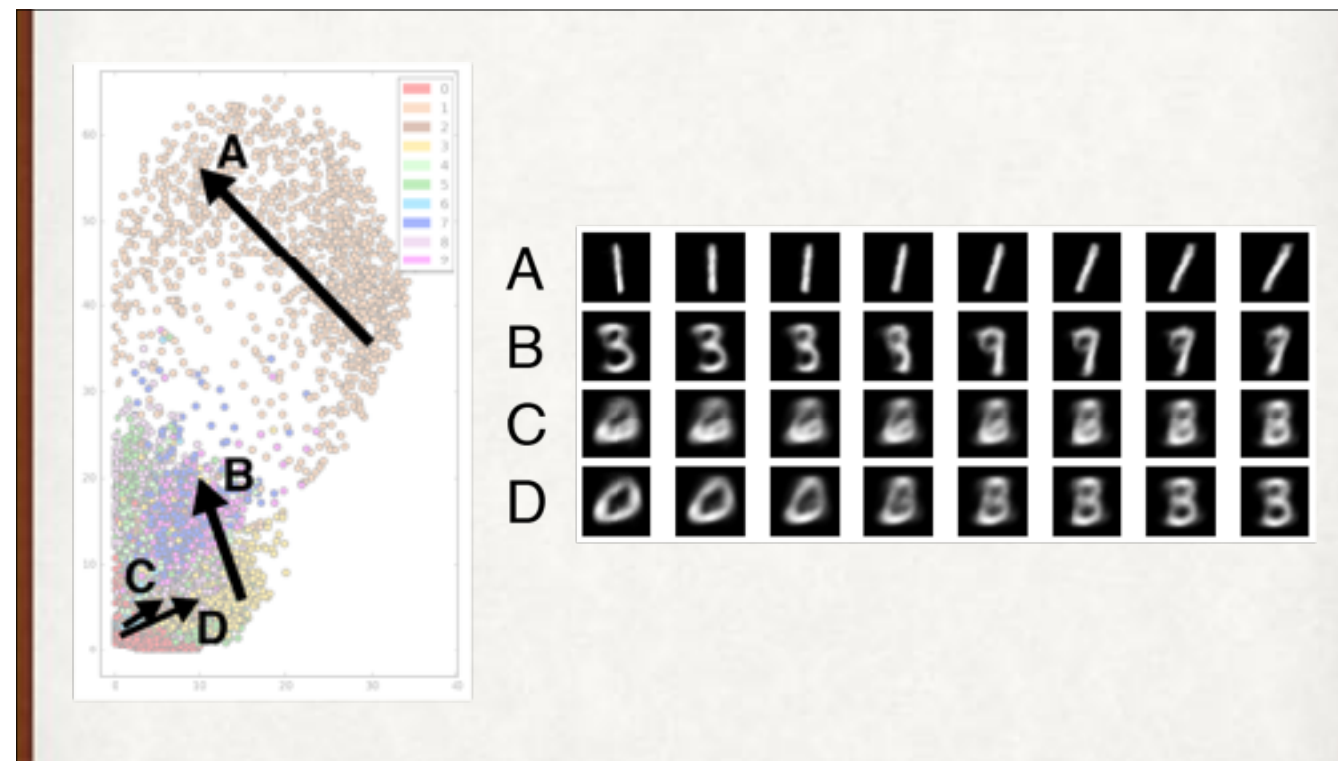
Blending in latent space. Each row shows equally-spaced steps along the corresponding arrow. Interpolated “digits” aren’t great, but they’re not random nonsense.



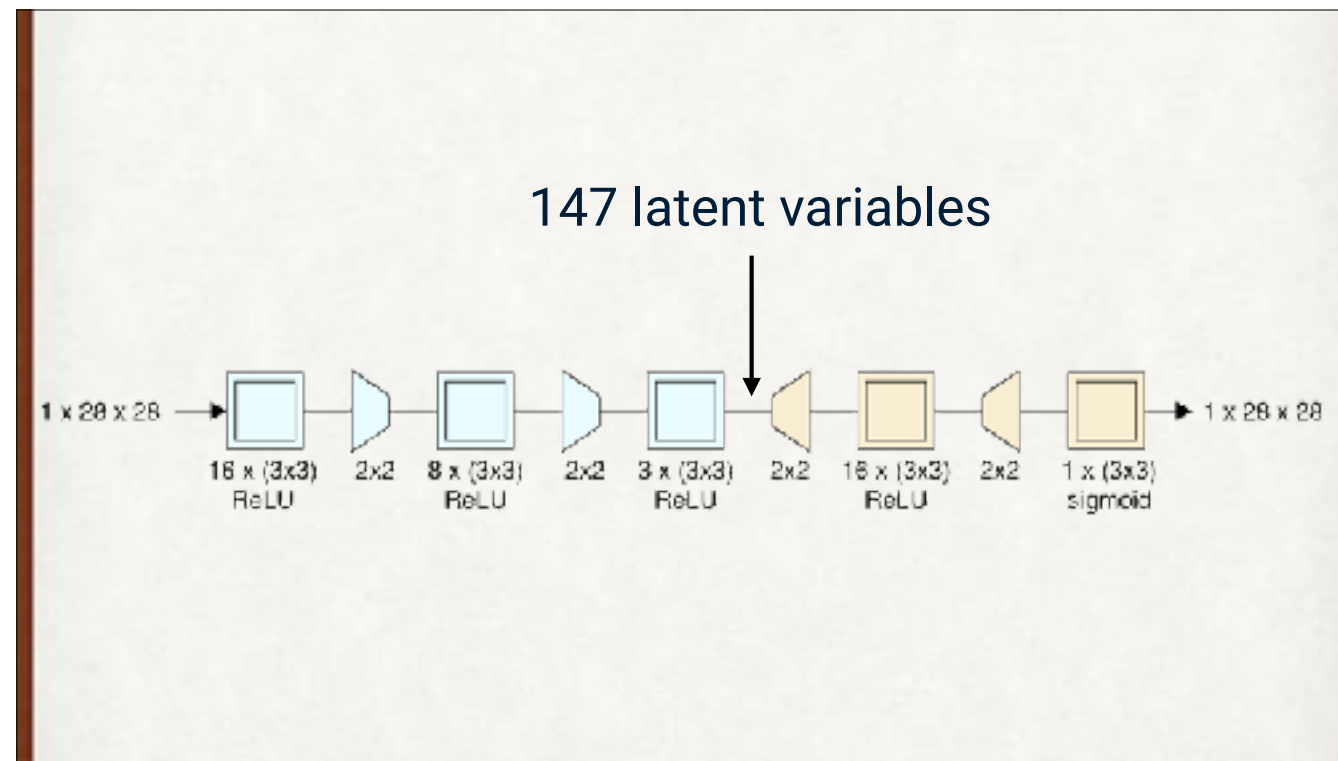
Blending in latent space. Each row shows equally-spaced steps along the corresponding arrow. Interpolated “digits” aren’t great, but they’re not random nonsense.



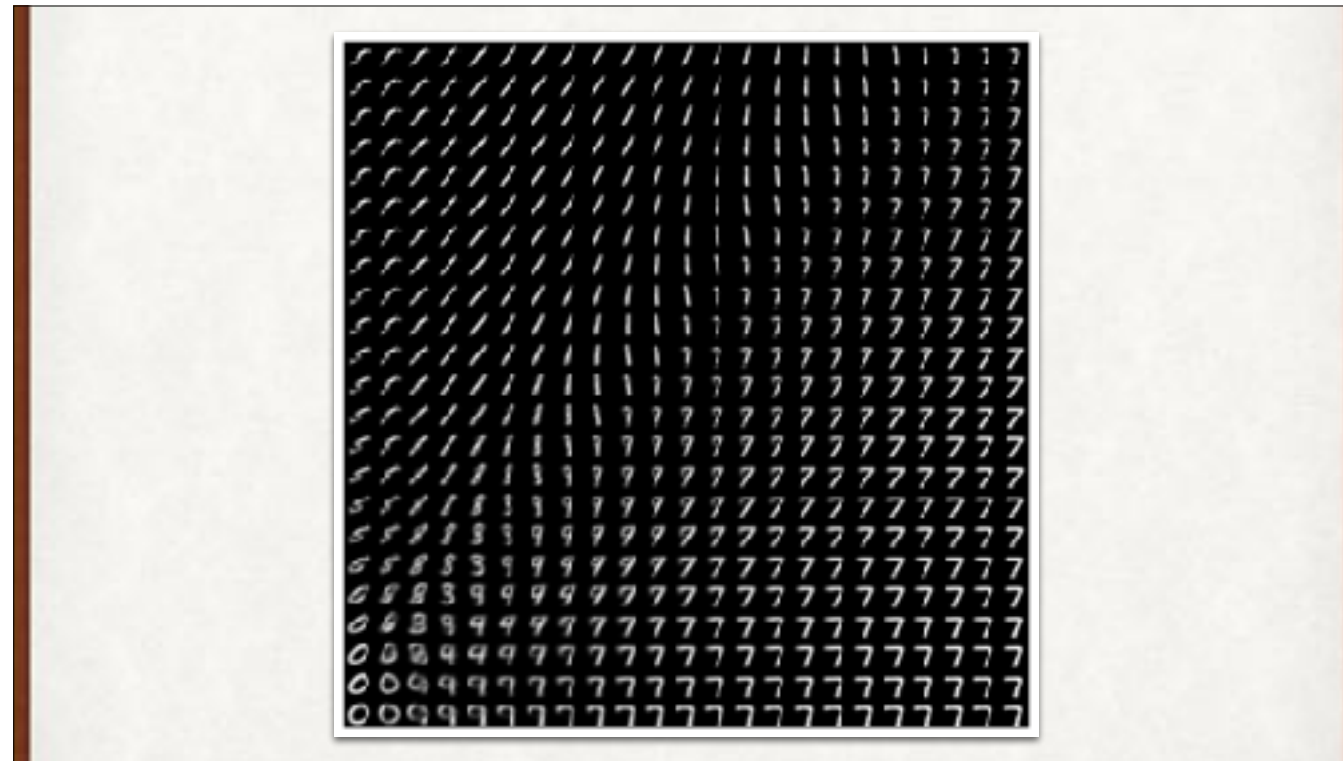
Blending in latent space. Each row shows equally-spaced steps along the corresponding arrow. Interpolated “digits” aren’t great, but they’re not random nonsense.



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.



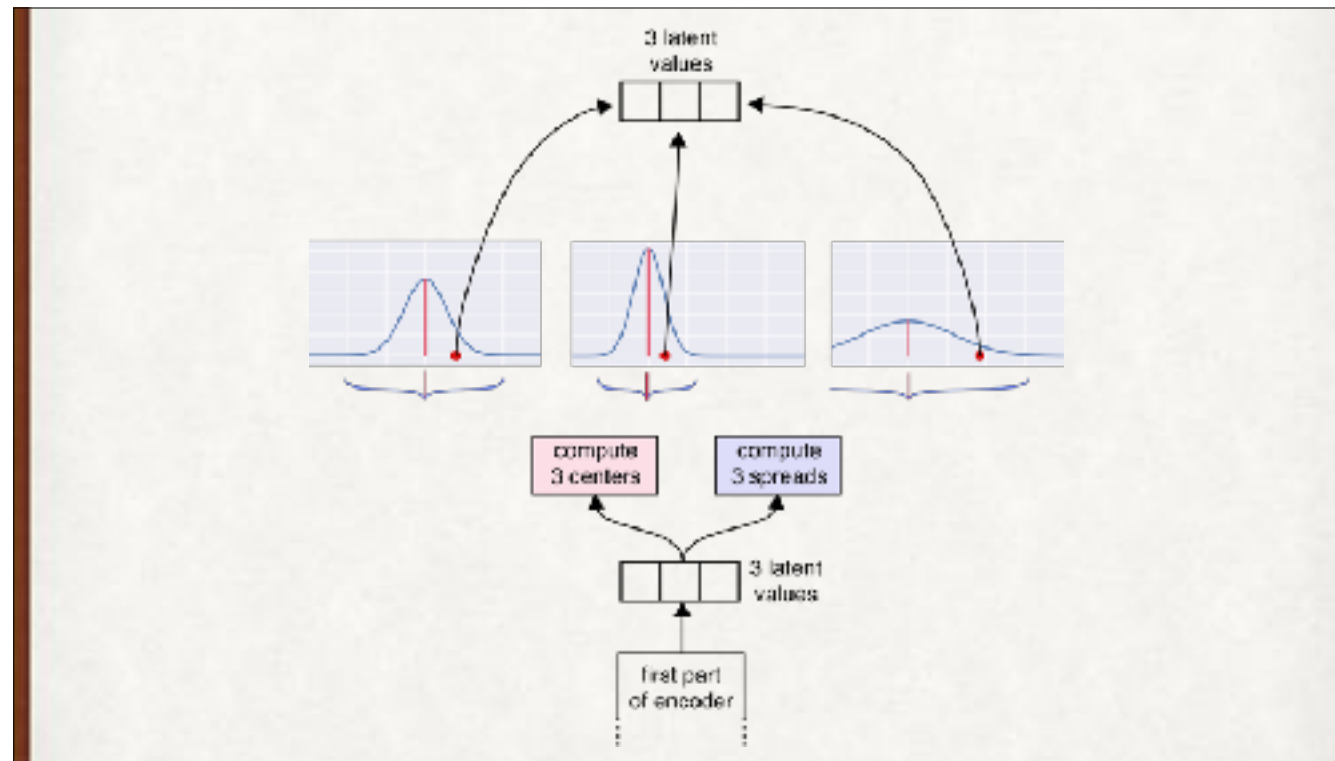
A convolutional autoencoder for MNIST digits. The bottleneck has $3 \times 7 \times 7 = 147$ values, rather than the $28 \times 28 = 784$ in the input, about 18%



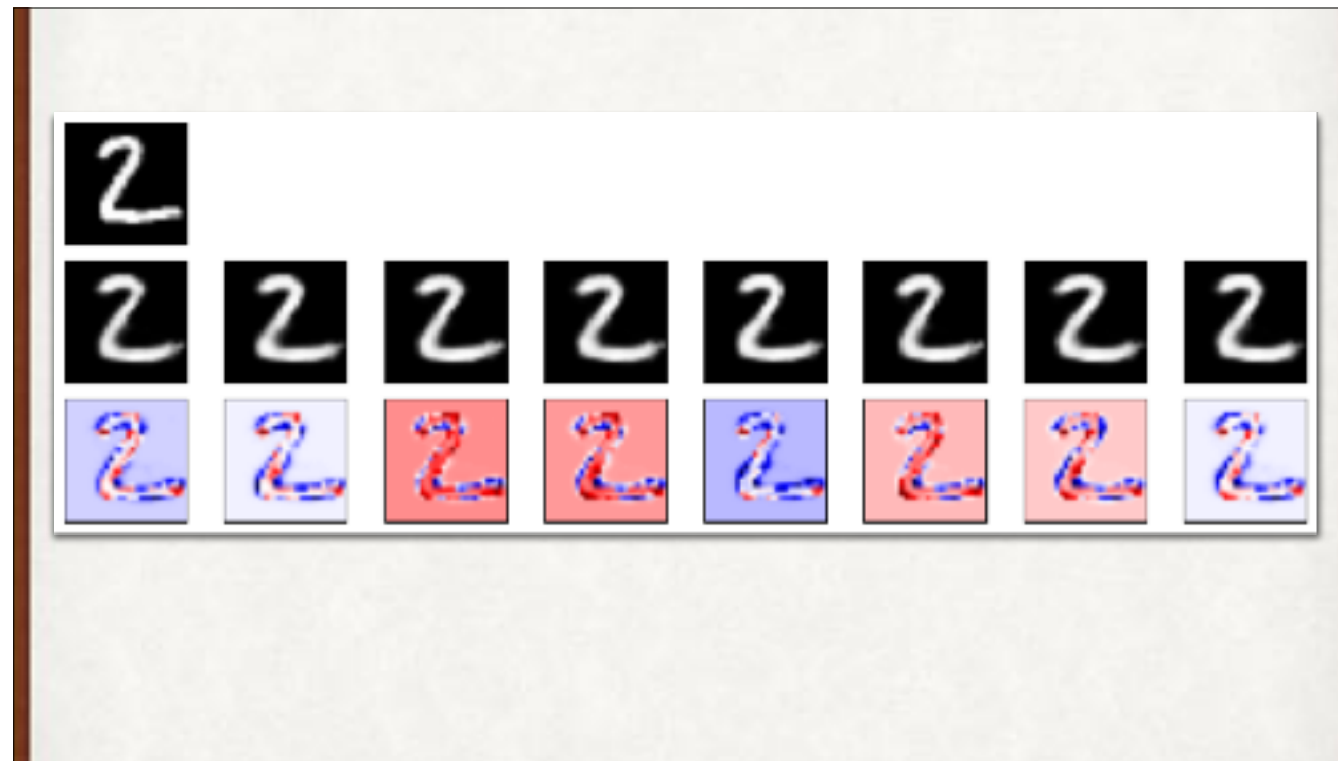
Sampling a grid over the 2D latent variables, and drawing the result for each pair of values. The 1s and 7s take up most of the space, but we can spot the other digits in the bottom-left corner.

Variational Autoencoder

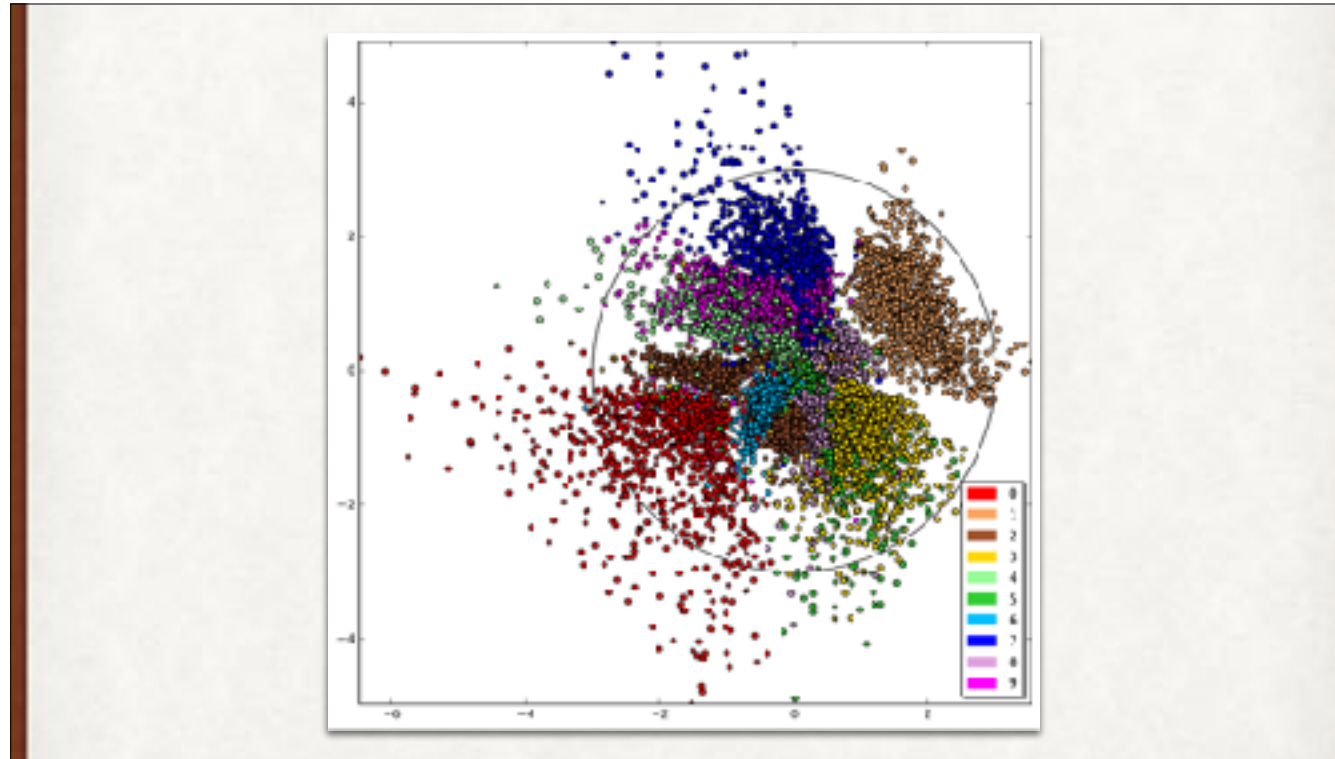
Another type of autoencoder. It's more complicated, but produces nice results.



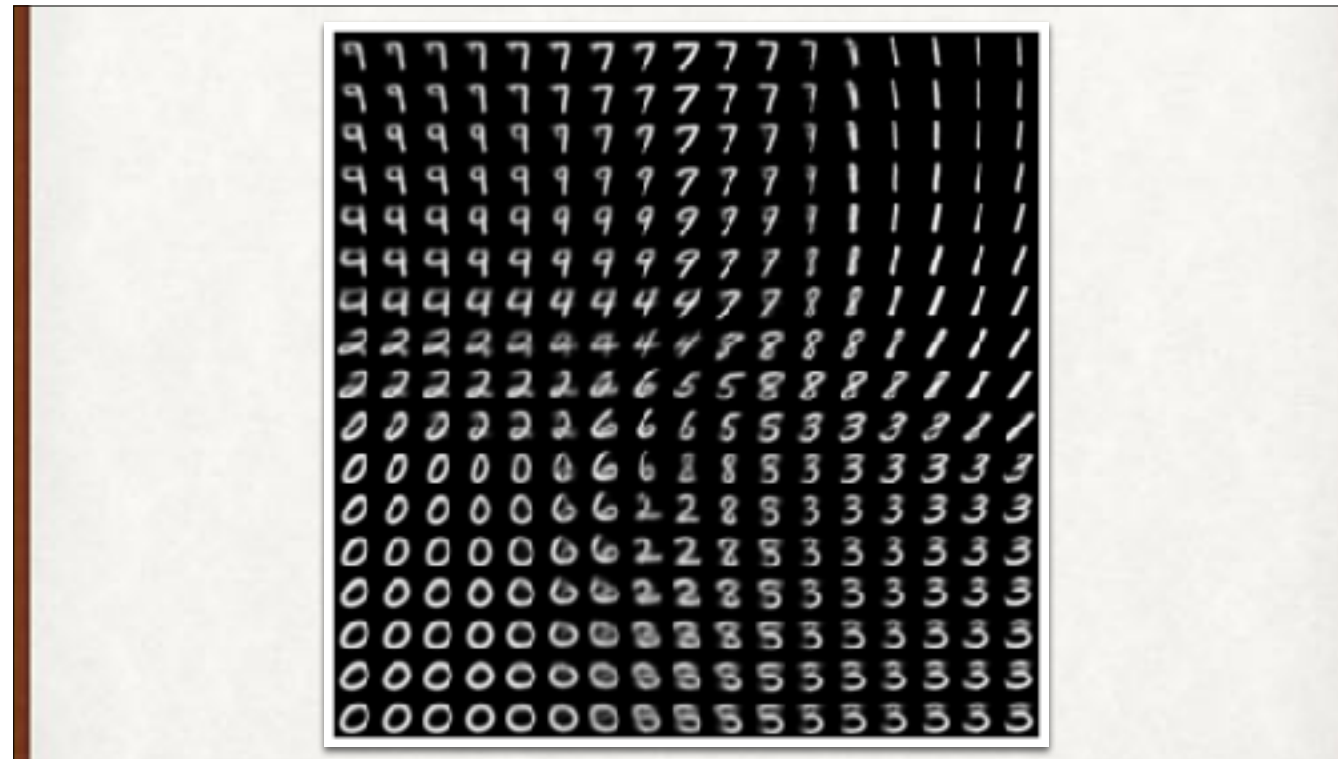
A schematic view of the heart of a variational autoencoder (VAE). It generates gaussian bumps that are then sampled by random variables, which then are the latent values.



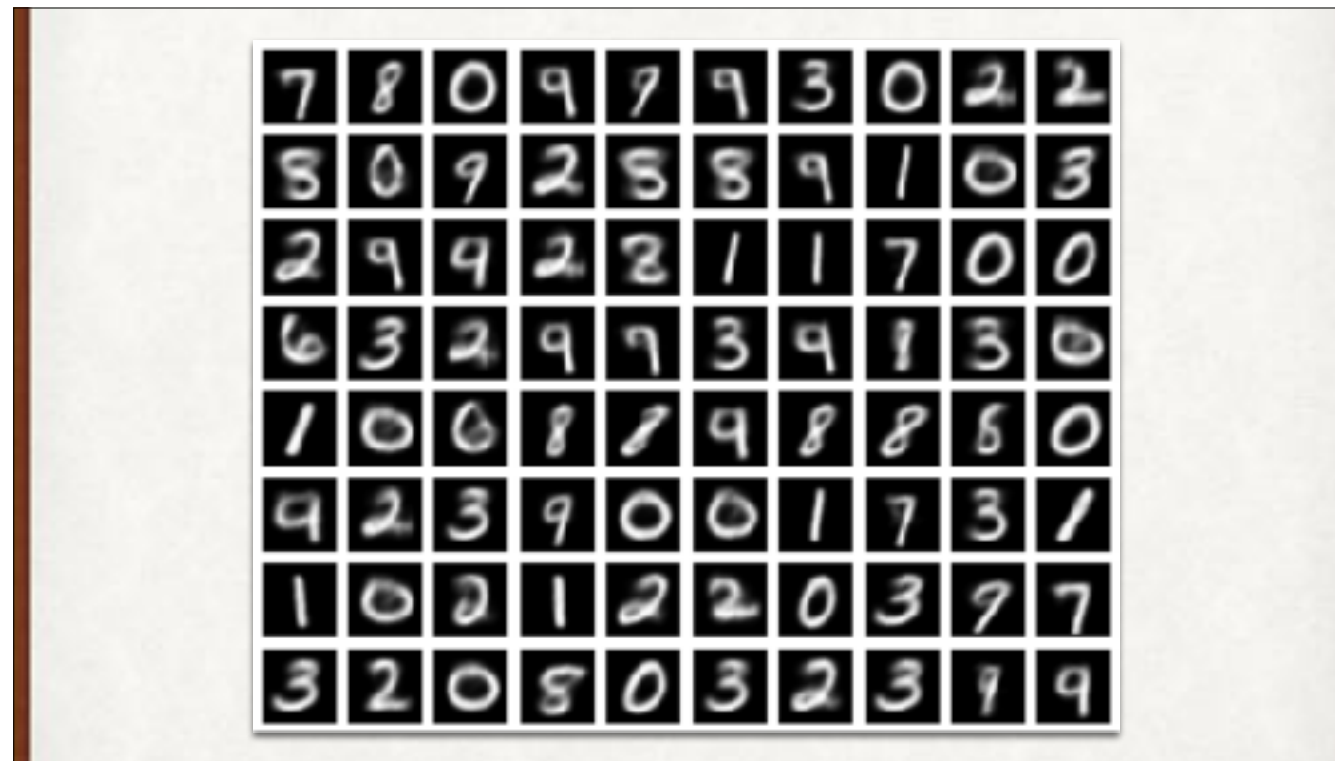
Thanks to its random nature, each output of a VAE is different, even from the same input. The top 2 was fed to a trained VAE repeatedly. The images below are the outputs. The images in the bottom row are the difference between the input and output.



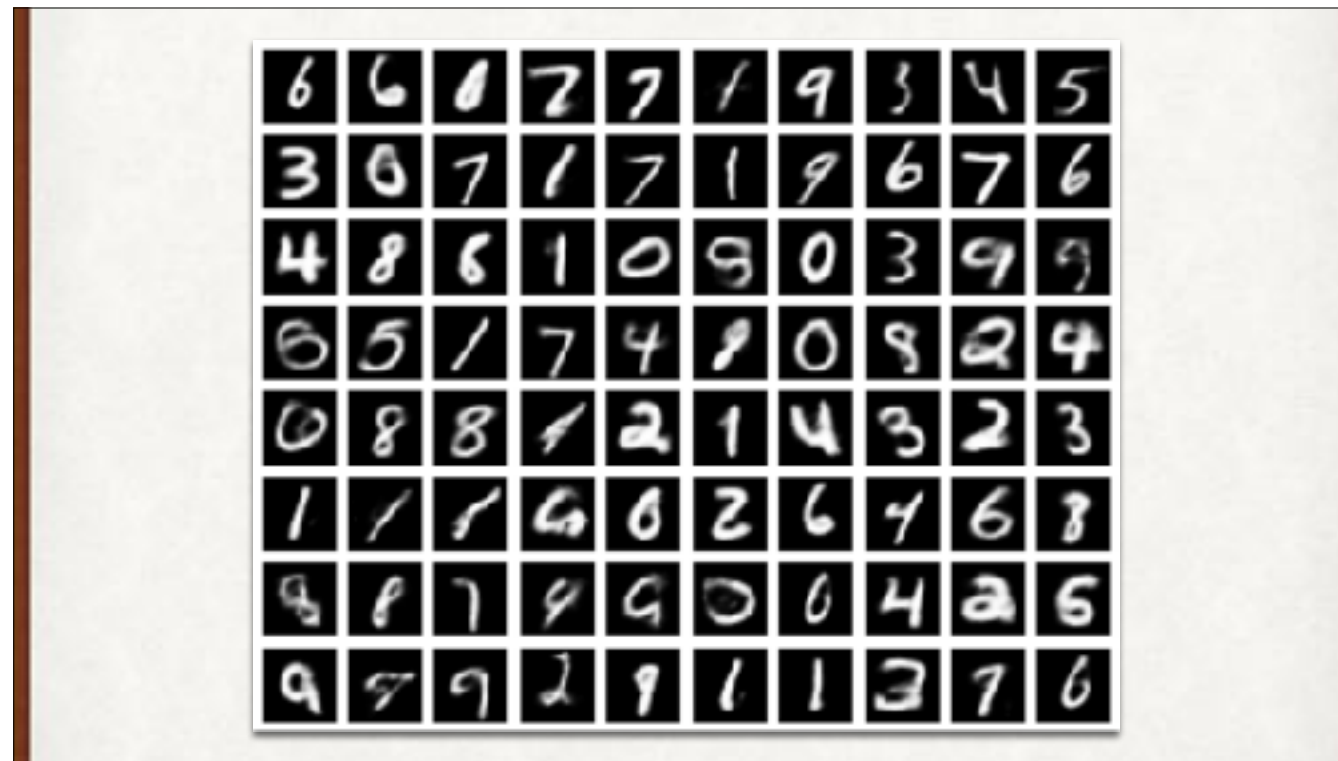
Visualizing the 2D latent variable space from a VAE. The digits are much better isolated and grouped than with our earlier autoencoder.



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.



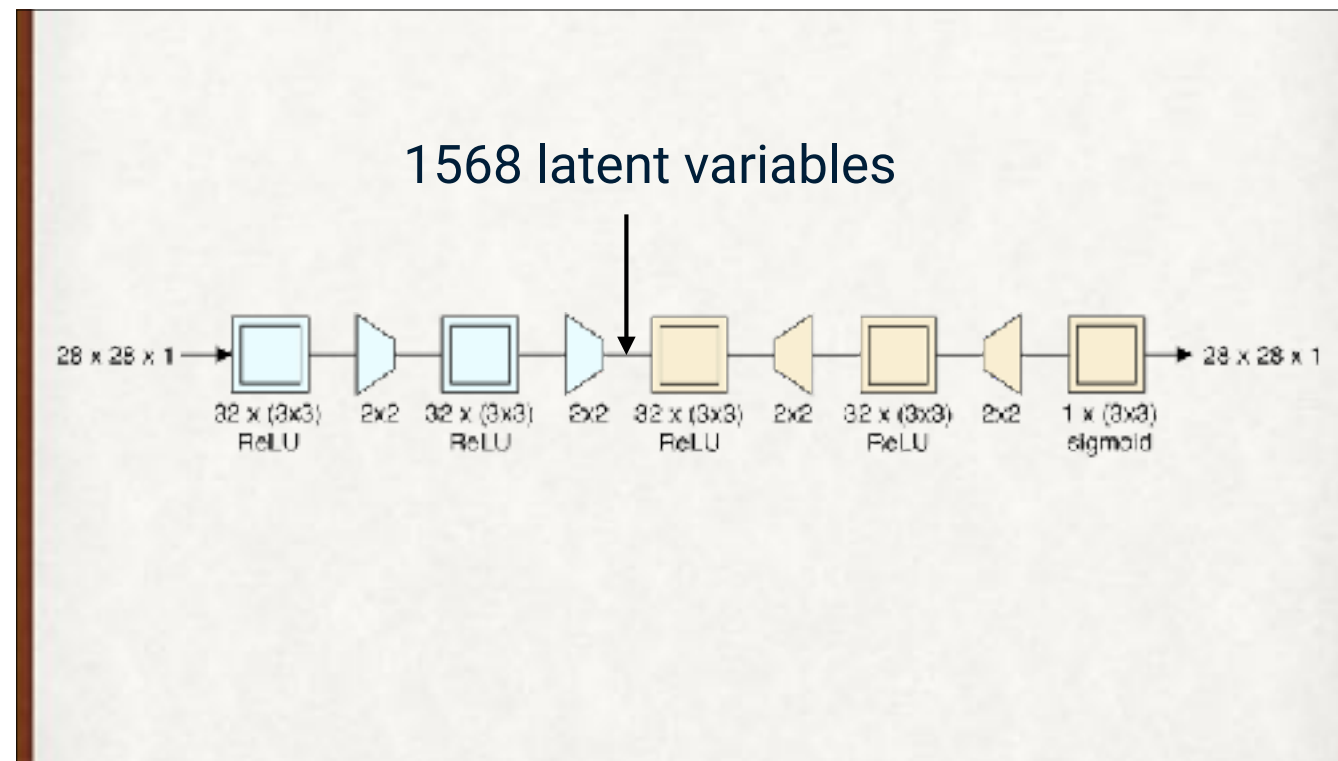
80 images generated by a VAE with 2 latent variables. Each image is the result of 2 random variables used as input.



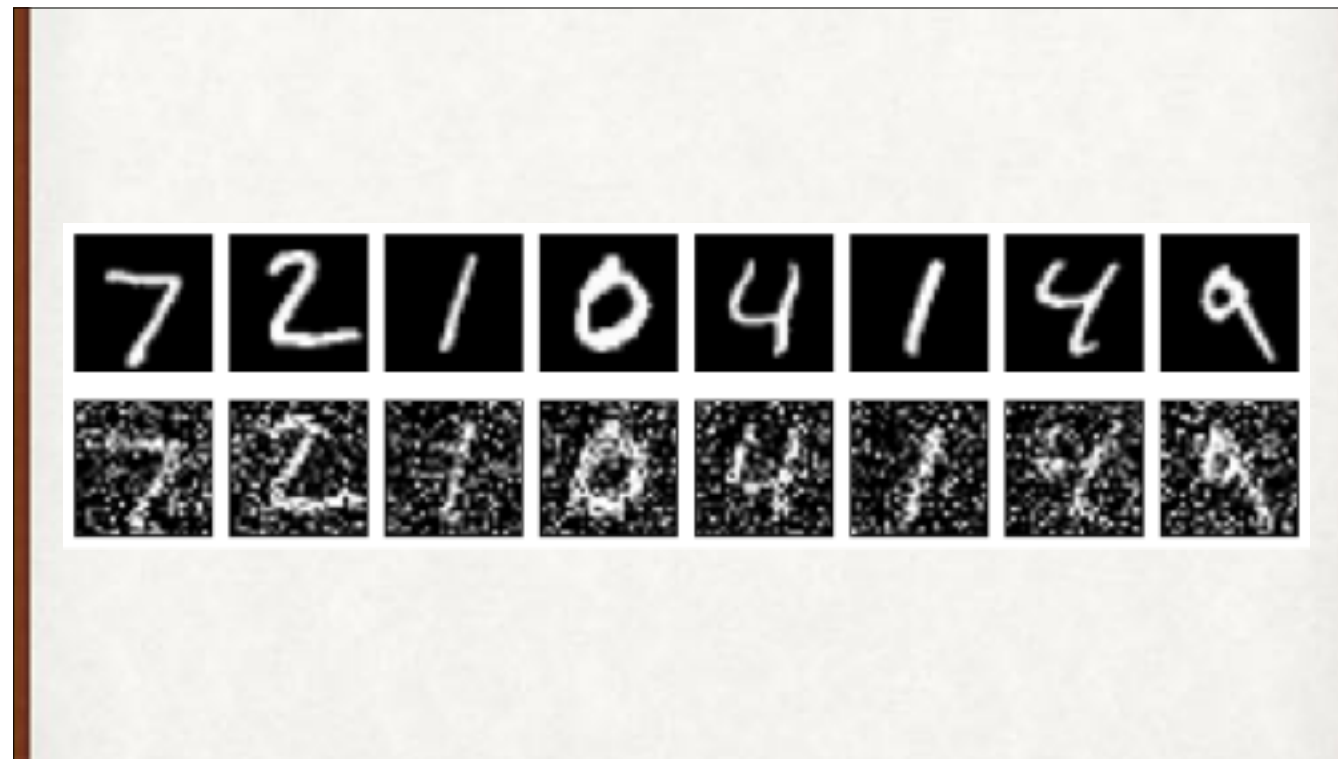
More random digits, only now we're using a VAE with 50 latent variables. So the input to the VAE was a list of 50 random numbers, and these images popped out at the end.

Denoising

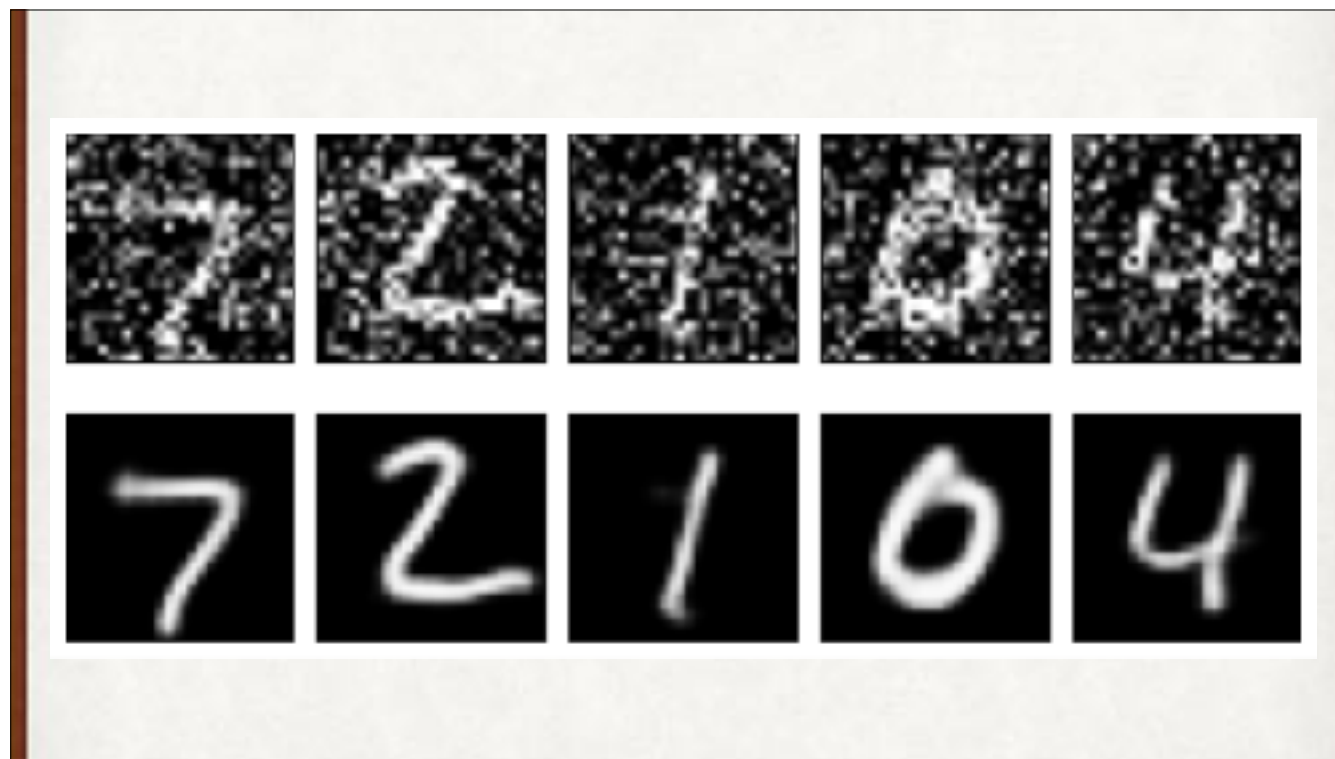
Autoencoders are great for removing noise.



An autoencoder for denoising. There are $32 \times 7 \times 7 = 1568$ 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.

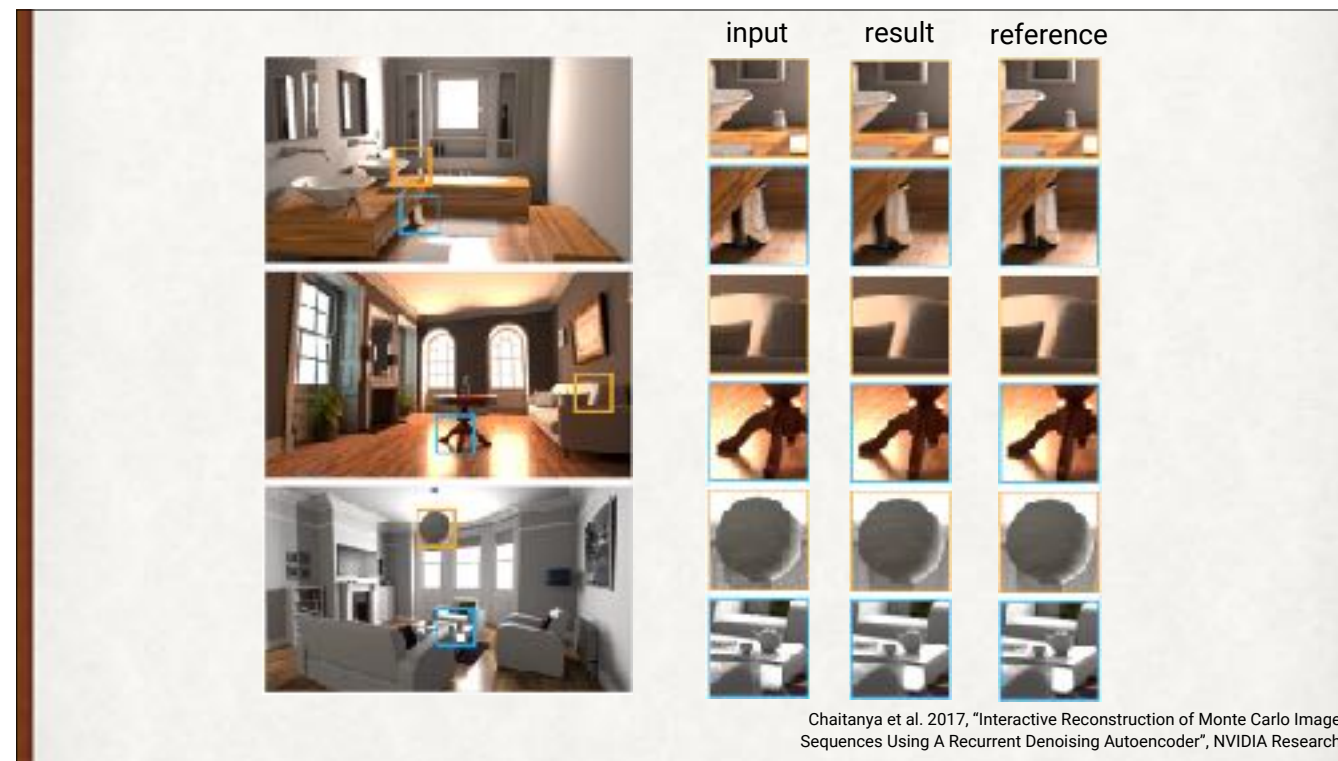
Interactive Reconstruction of Monte Carlo Image Sequences Using A Recurrent Denoising Autoencoder

Chaitanya et al. 2017, NVIDIA Research

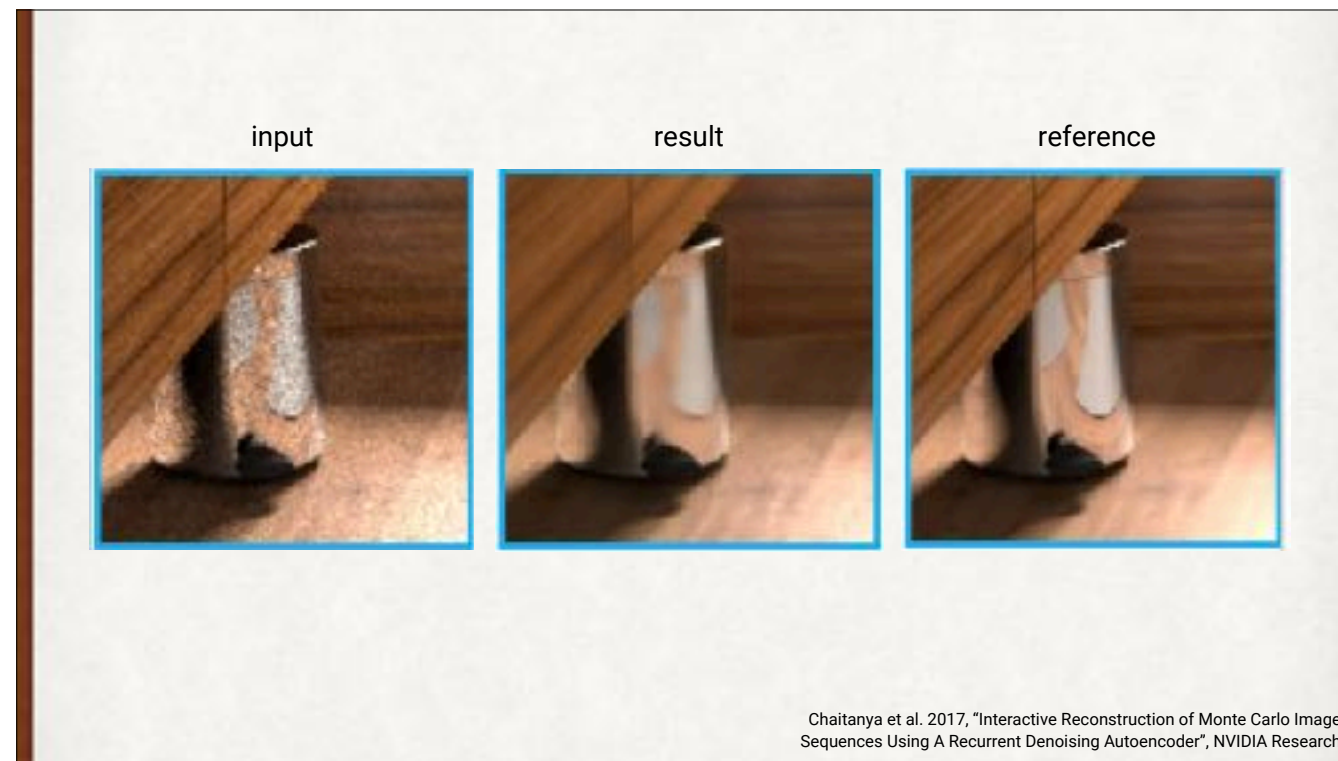
http://research.nvidia.com/sites/default/files/publications/dnn_denoise_author.pdf



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.



More reconstructions from the autoencoder.



Close-up reconstructions from the autoencoder.



Take a breath, enjoy some palette cleanser.

Reinforcement Learning

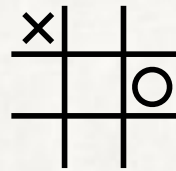
Another way to learn. An “agent” takes actions in an environment. The environment tells the agent how good each action is. The agent seeks to find actions that earn big rewards.



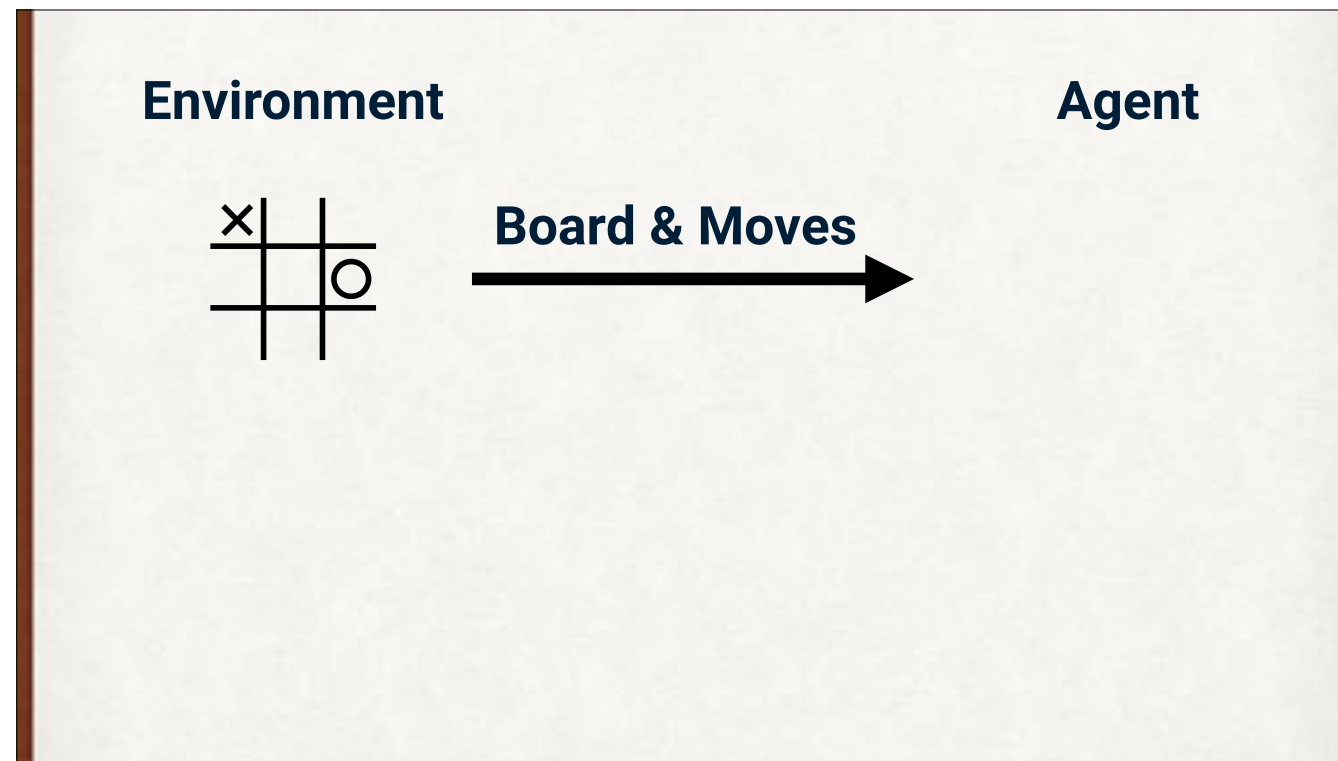
Learning to play tic-tac-toe. The agent (blue) chooses moves. The environment (yellow) does everything else, and tells the agent how good (or not) each move is. Good moves get the agent closer to winning. Great moves are those that win the game.

Environment

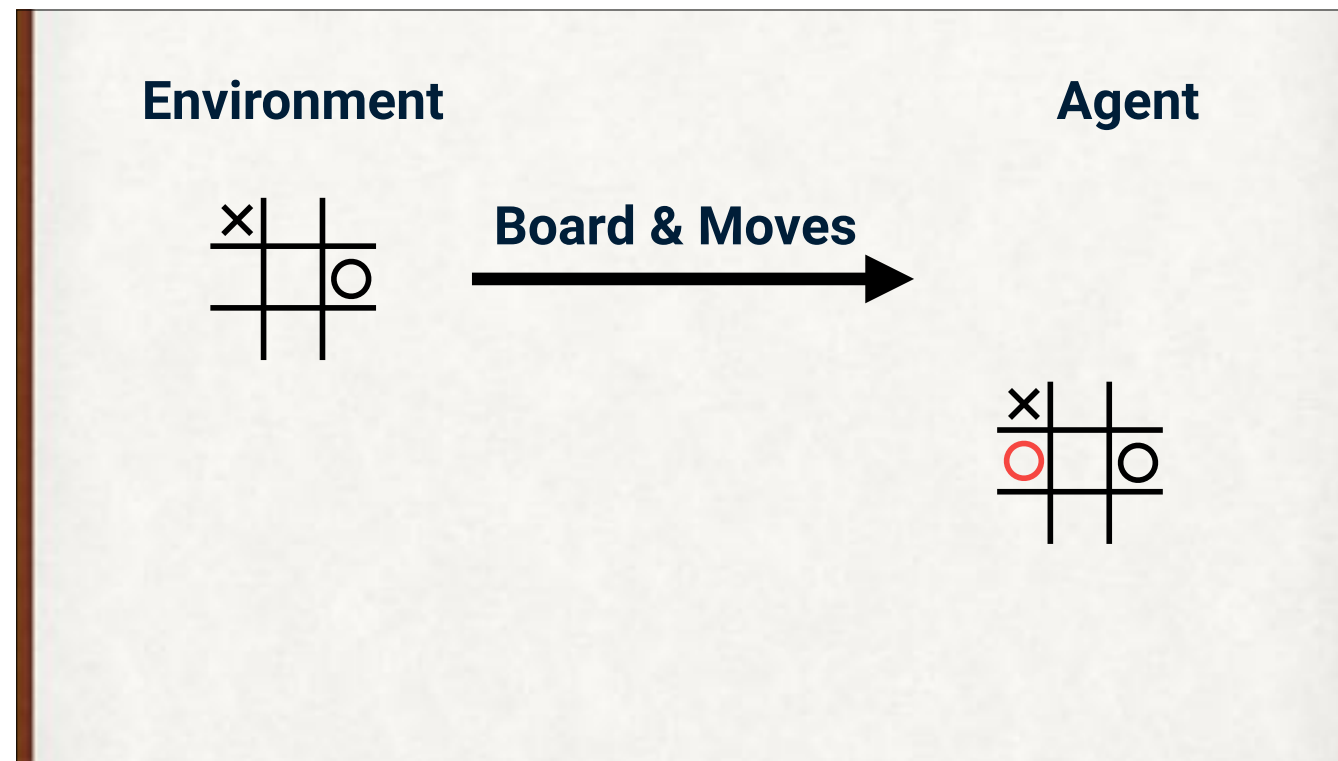
Agent



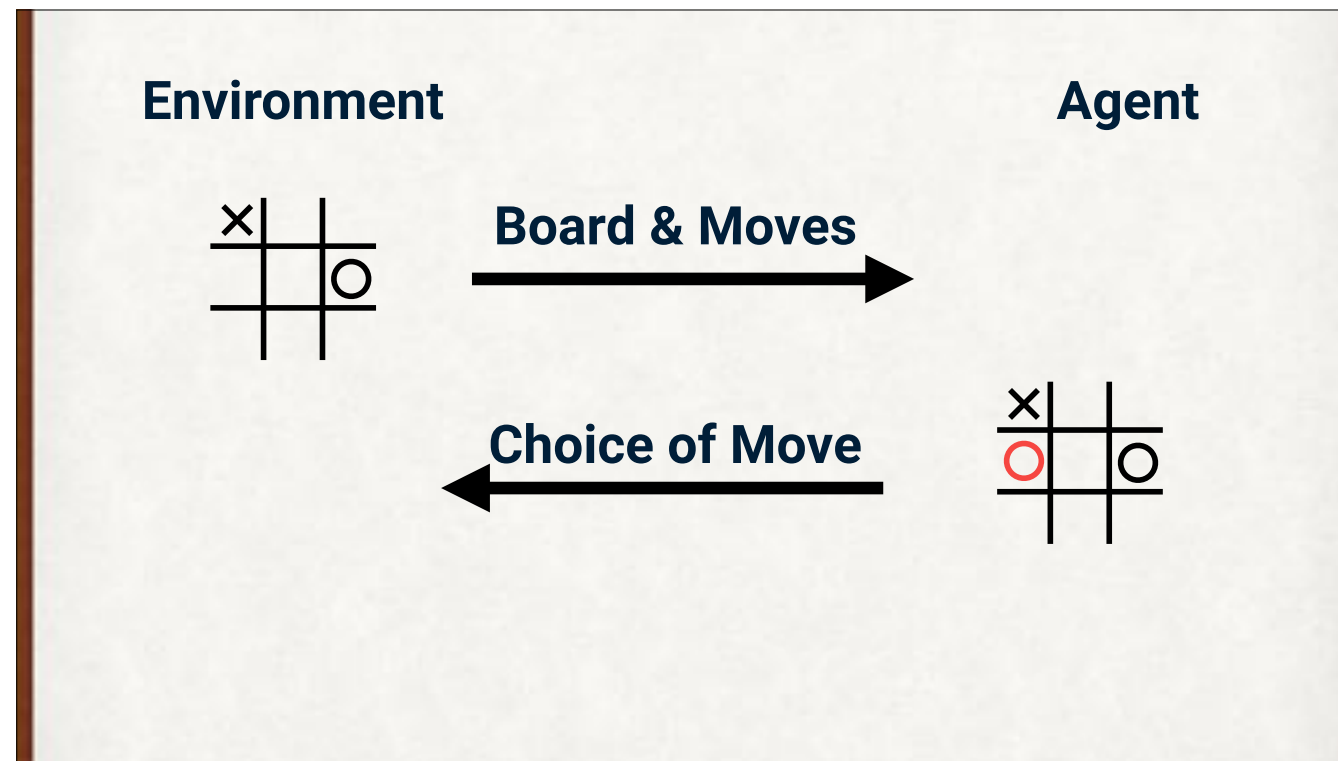
Learning to play tic-tac-toe. The agent chooses moves. The environment does everything else, and tells the agent how good (or not) each move is. Good moves get the agent closer to winning. Great moves are those that win the game.



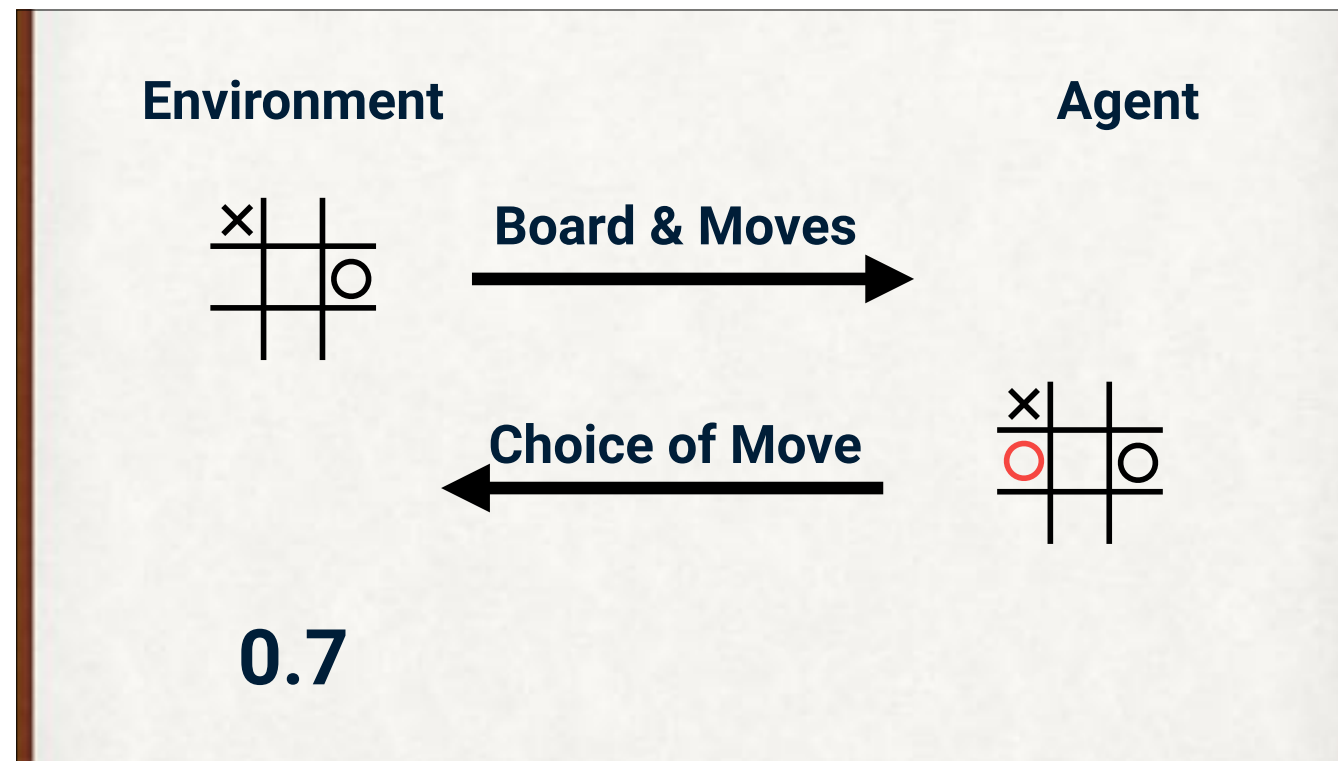
Another way to learn. An “agent” takes actions in an environment. The environment tells the agent how good each action is. The agent seeks to find actions that earn big rewards.



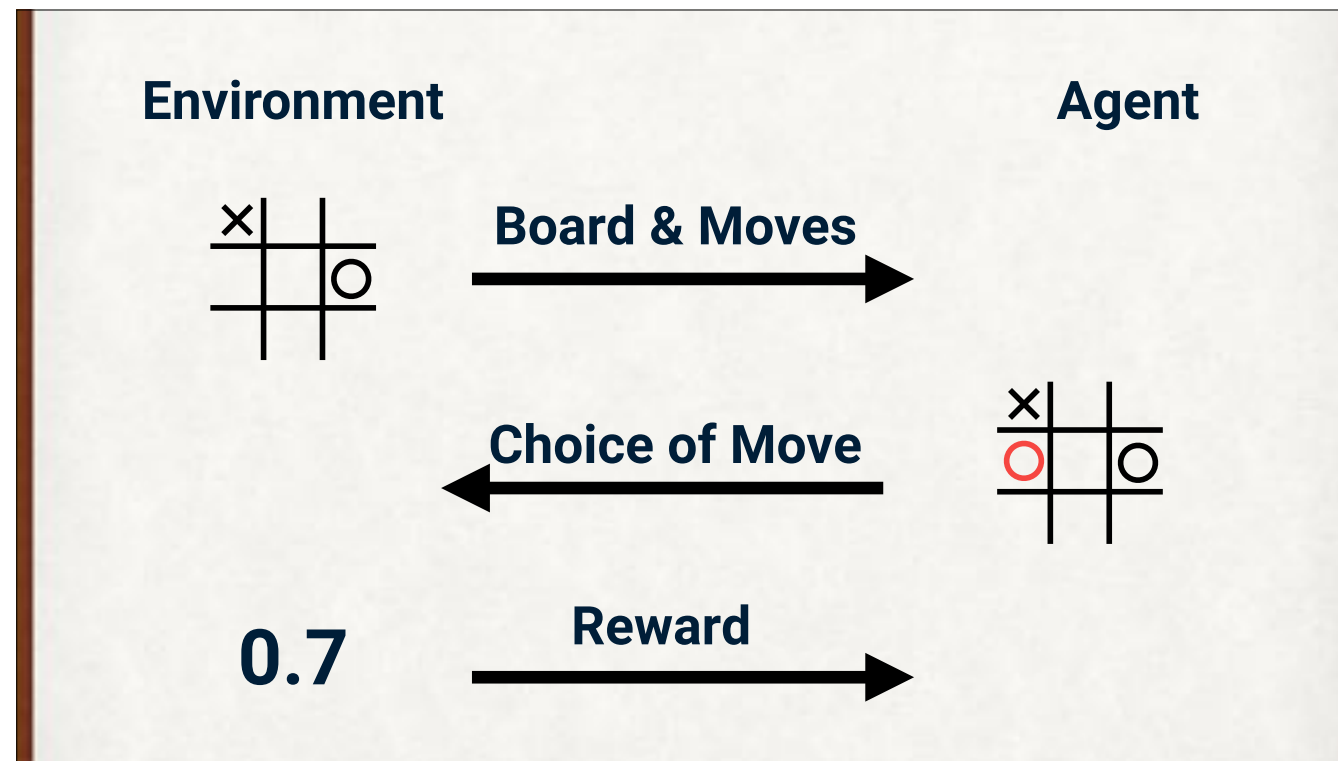
Another way to learn. An “agent” takes actions in an environment. The environment tells the agent how good each action is. The agent seeks to find actions that earn big rewards.



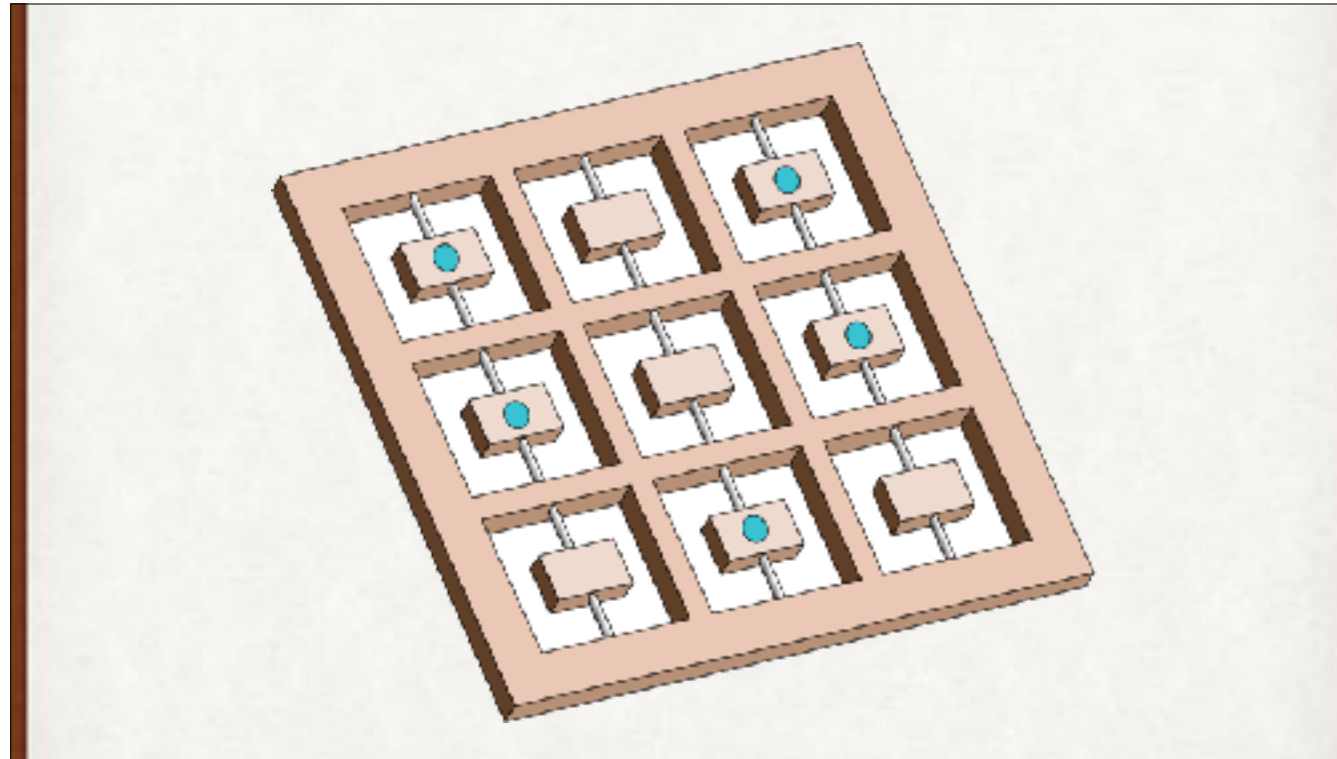
Another way to learn. An “agent” takes actions in an environment. The environment tells the agent how good each action is. The agent seeks to find actions that earn big rewards.



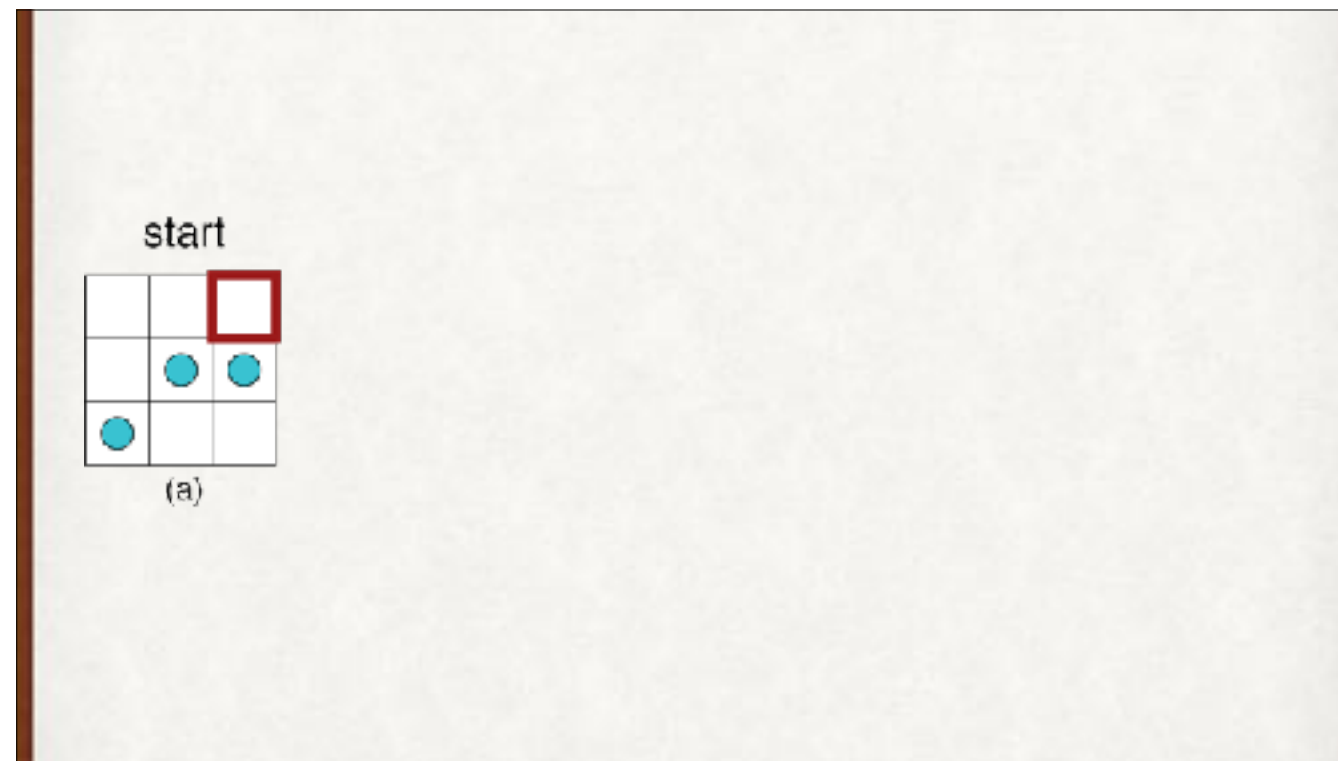
Another way to learn. An “agent” takes actions in an environment. The environment tells the agent how good each action is. The agent seeks to find actions that earn big rewards.



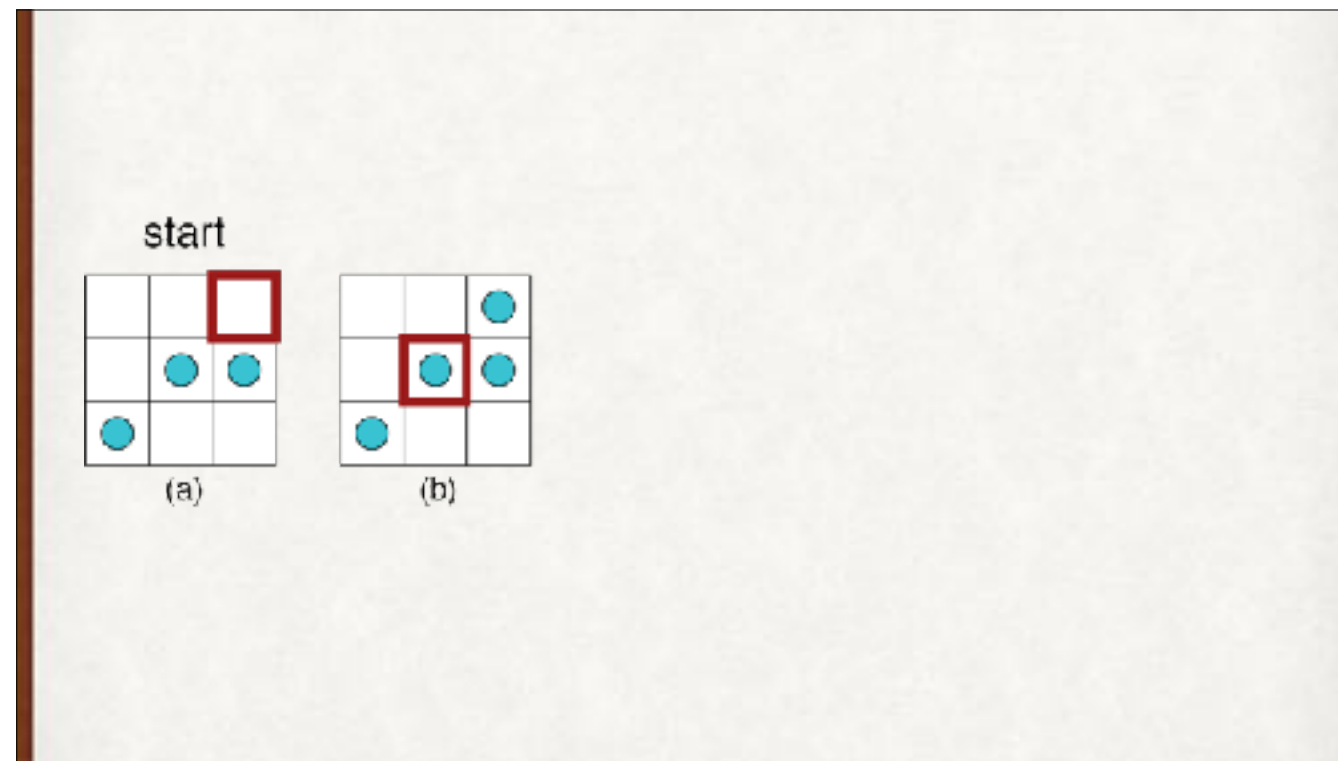
Another way to learn. An “agent” takes actions in an environment. The environment tells the agent how good each action is. The agent seeks to find actions that earn big rewards.



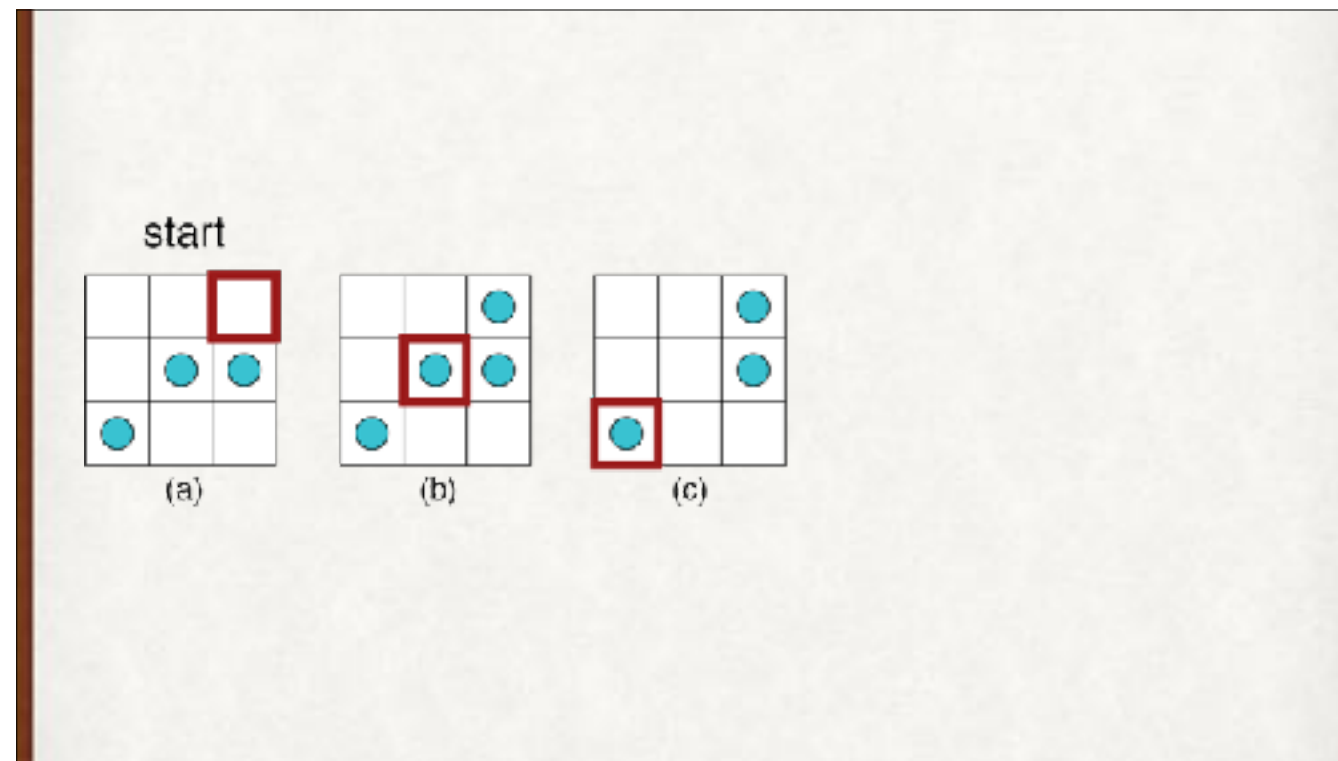
The absurd game of solitaire tic-tac-toe, called Flippers. Each tile has a blue dot on only one side. We want to get three dots horizontal or vertical, and blank tiles everywhere else. A move consists of flipping one tile.



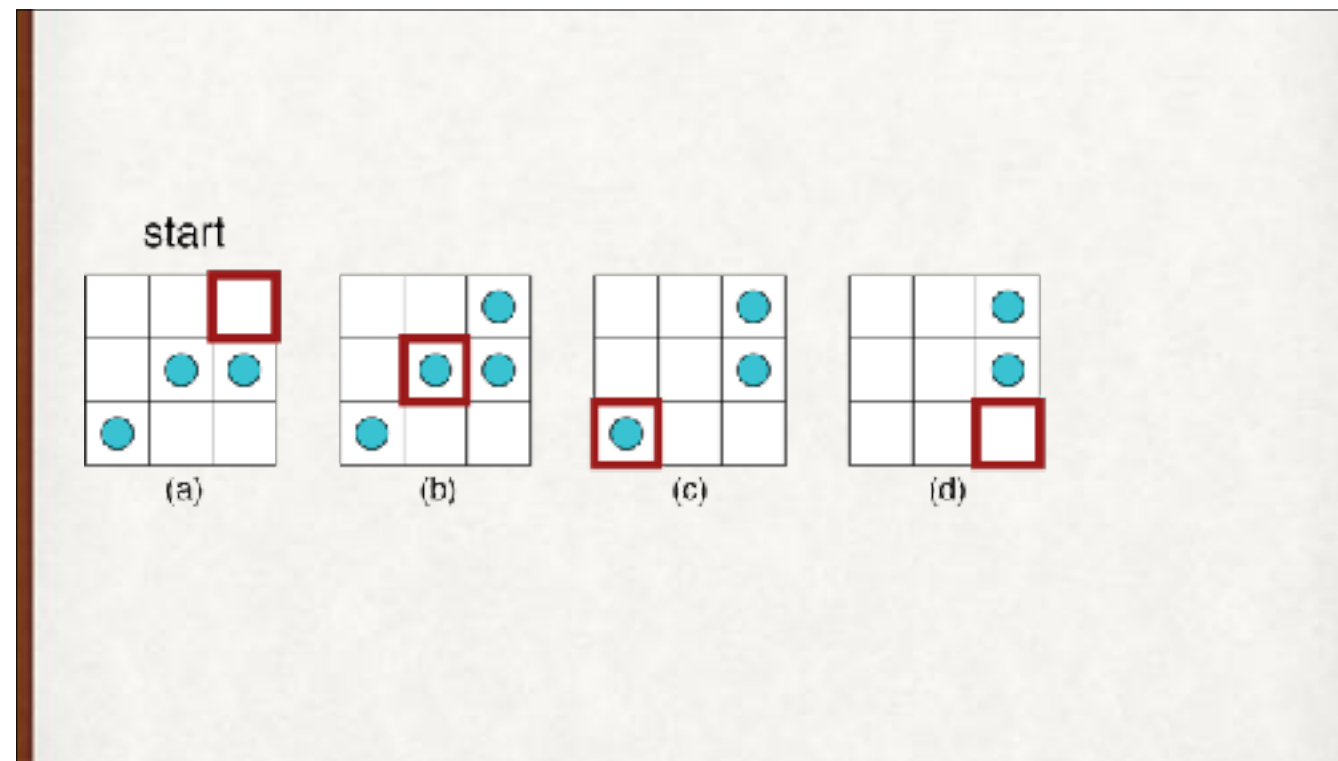
Playing Flippers. Red cells are where we're going to flip next.



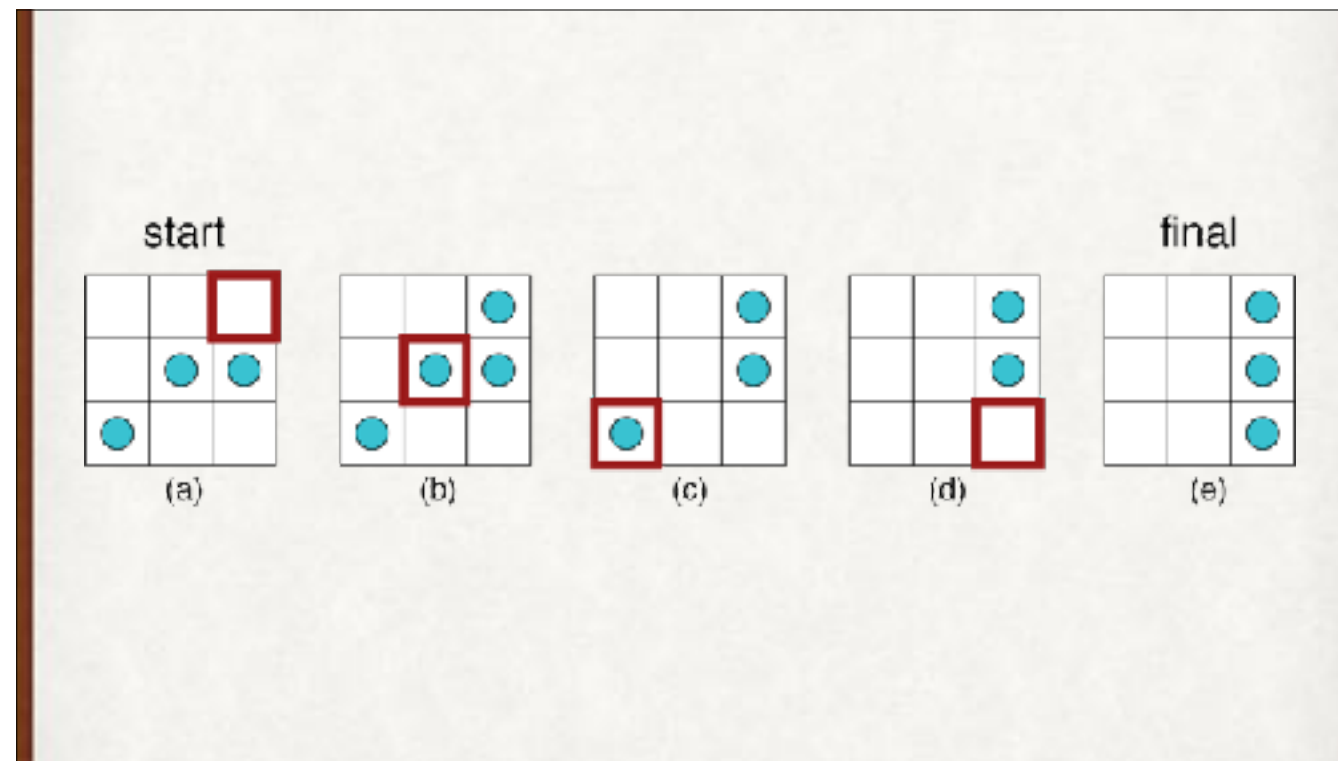
Playing Flippers. Red cells are where we're going to flip next.



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

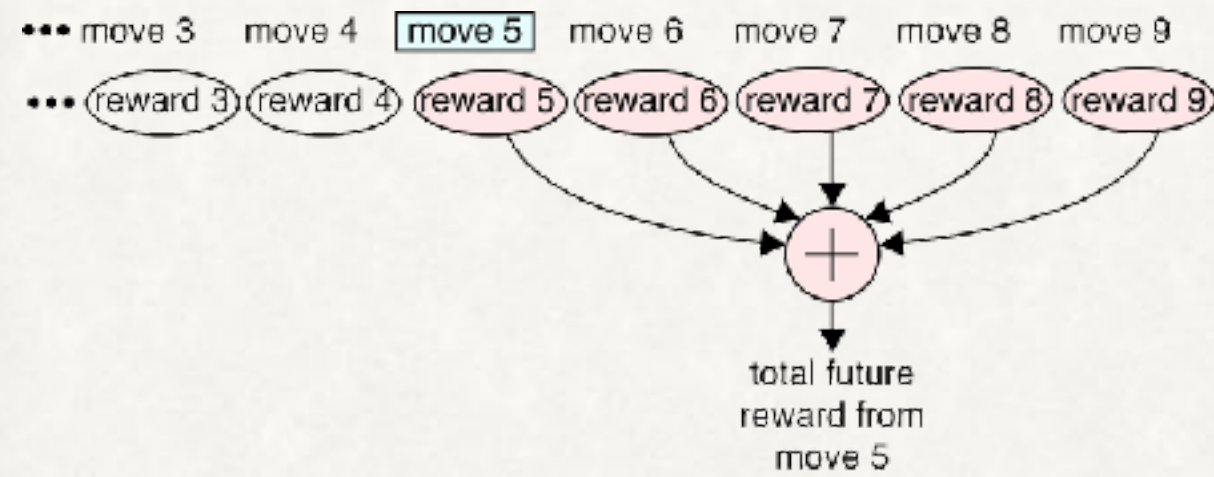


Playing Flippers. Red cells are where we're going to flip next.

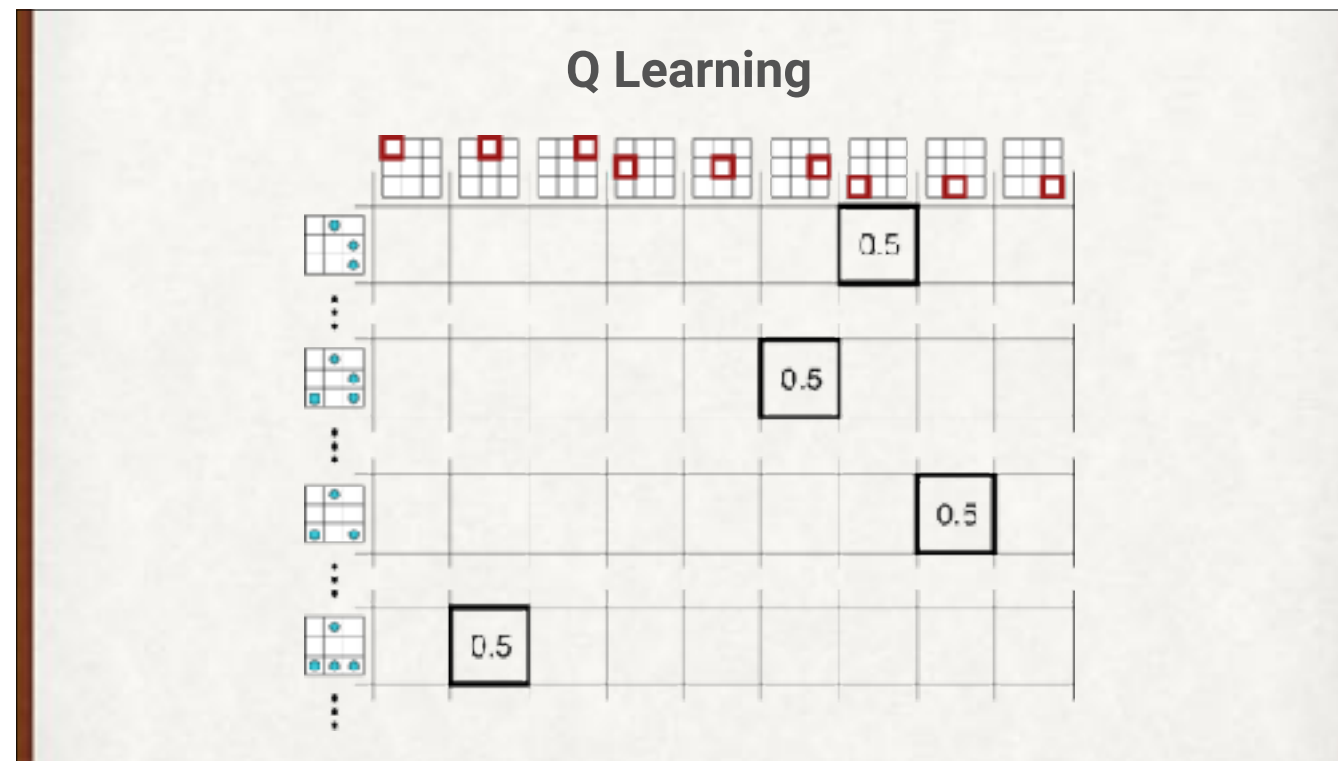


Playing Flippers. Red cells are where we're going to flip next.

Total Future Reward (TFR)



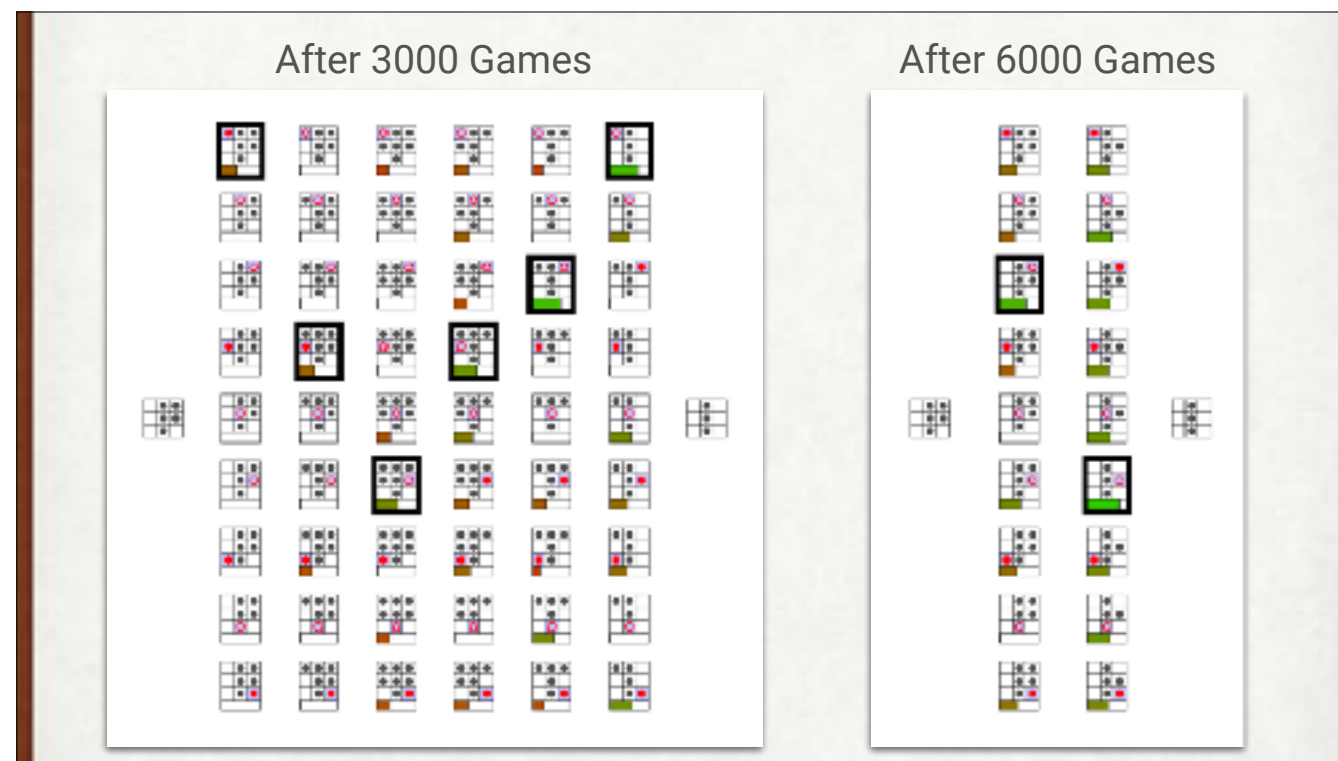
The Total Future Reward tells us how much reward we'll get from all moves starting with this one.



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



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.

Reinforcement Learning + Computer Graphics

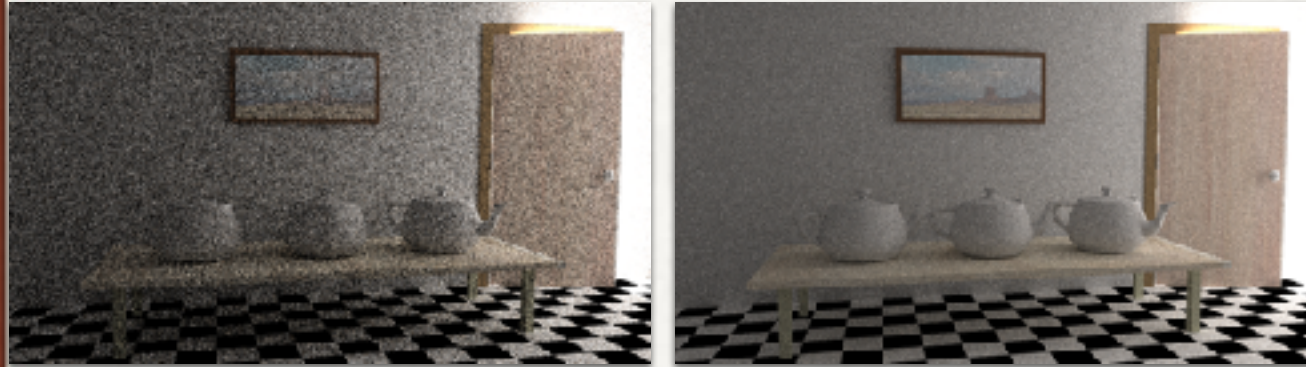
Using RL to create images.

Learning Light Transport the Reinforced Way

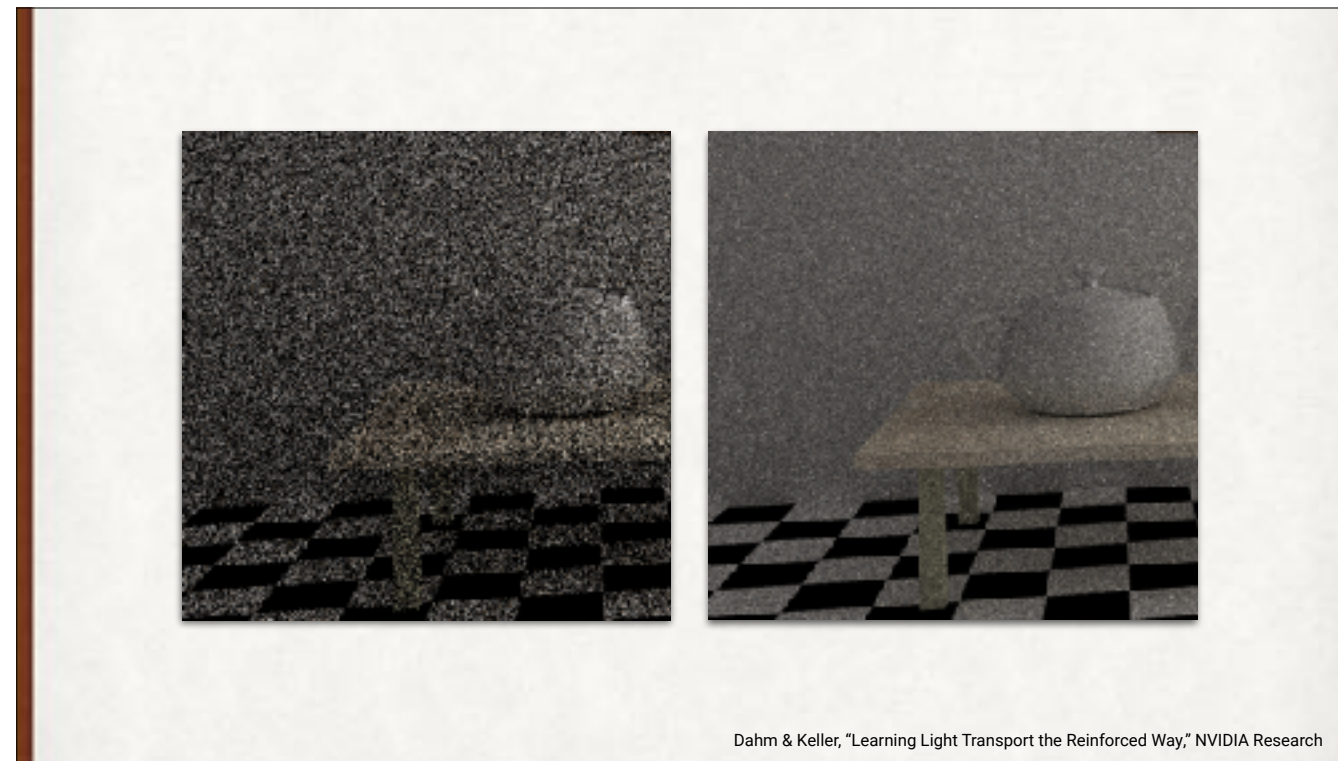
Ken Dahm and Alexander Keller

NVIDIA Research

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



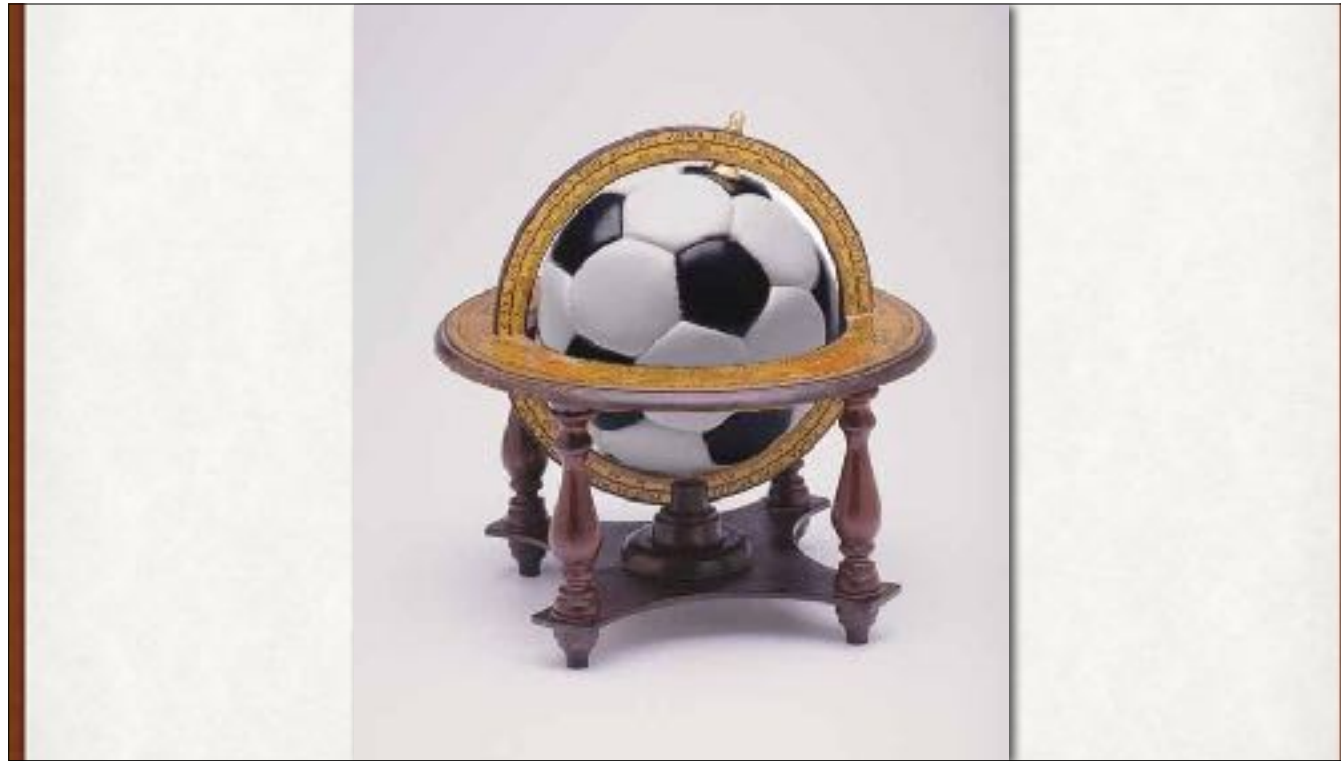
Left: Results from a simple path tracer, 1024 samples per pixel. It's still very noisy. Right: the same number of samples, only their colors are computed using Q learning and BSDF weighting.



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.



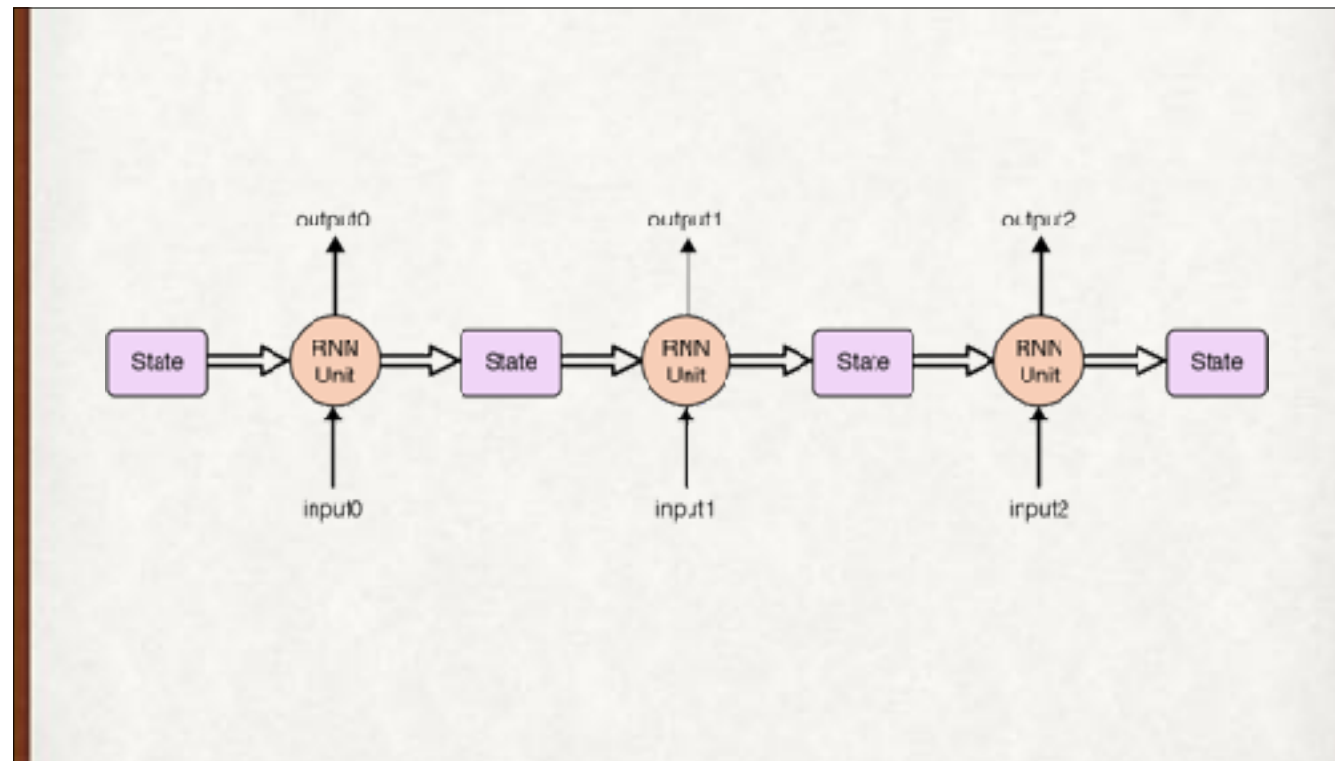
Sun and sky illumination at 32 paths per pixel. Left: original scene (cropped). Middle: Importance sampling of the sky. Right: Importance sampling with learned importance.



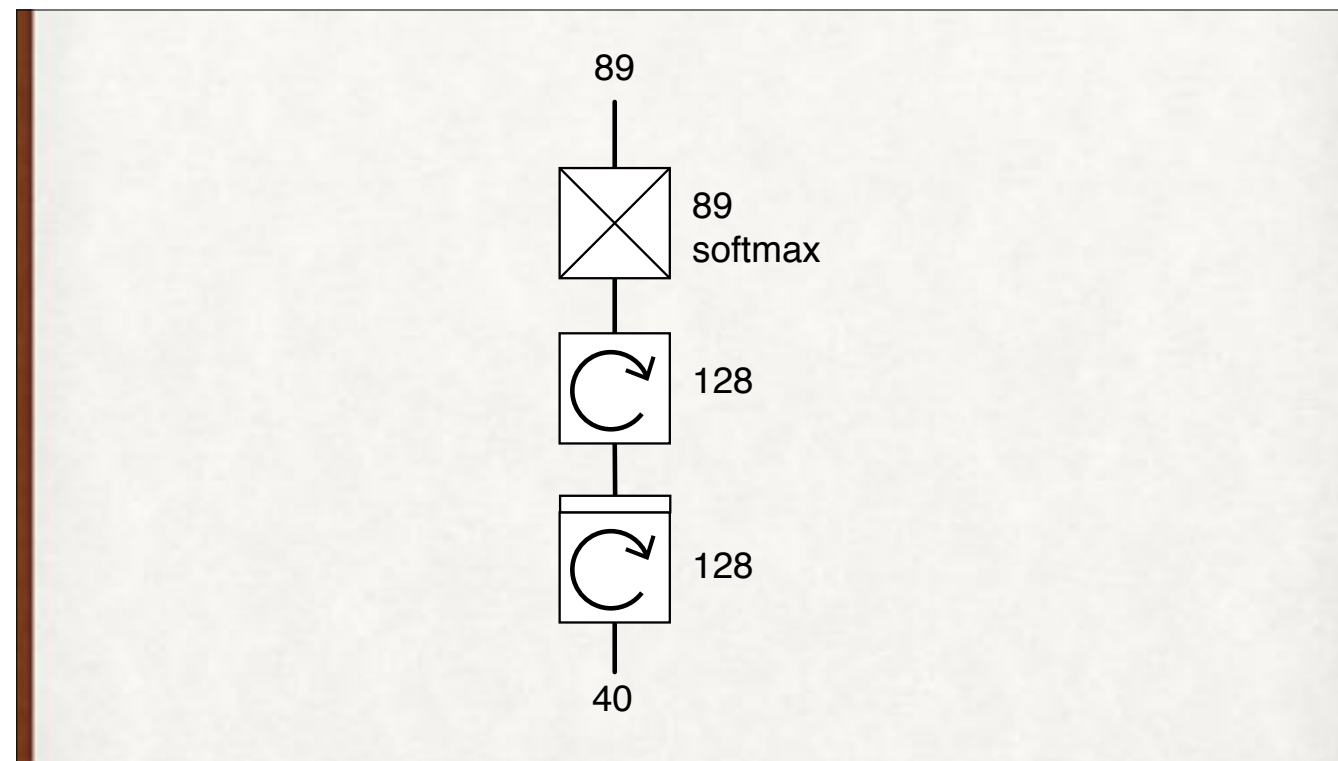
Another moment of rest for another change of topic.

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).*

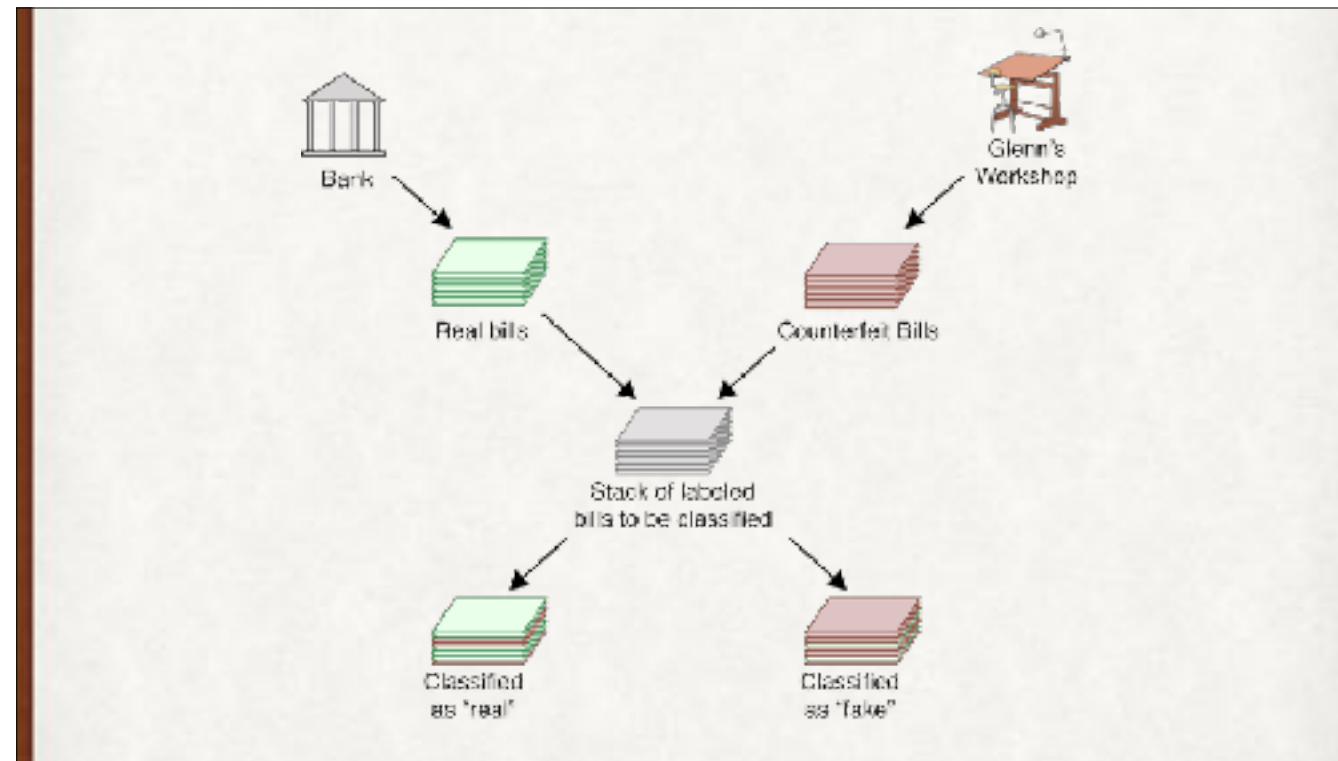
Short phrases are more fun!



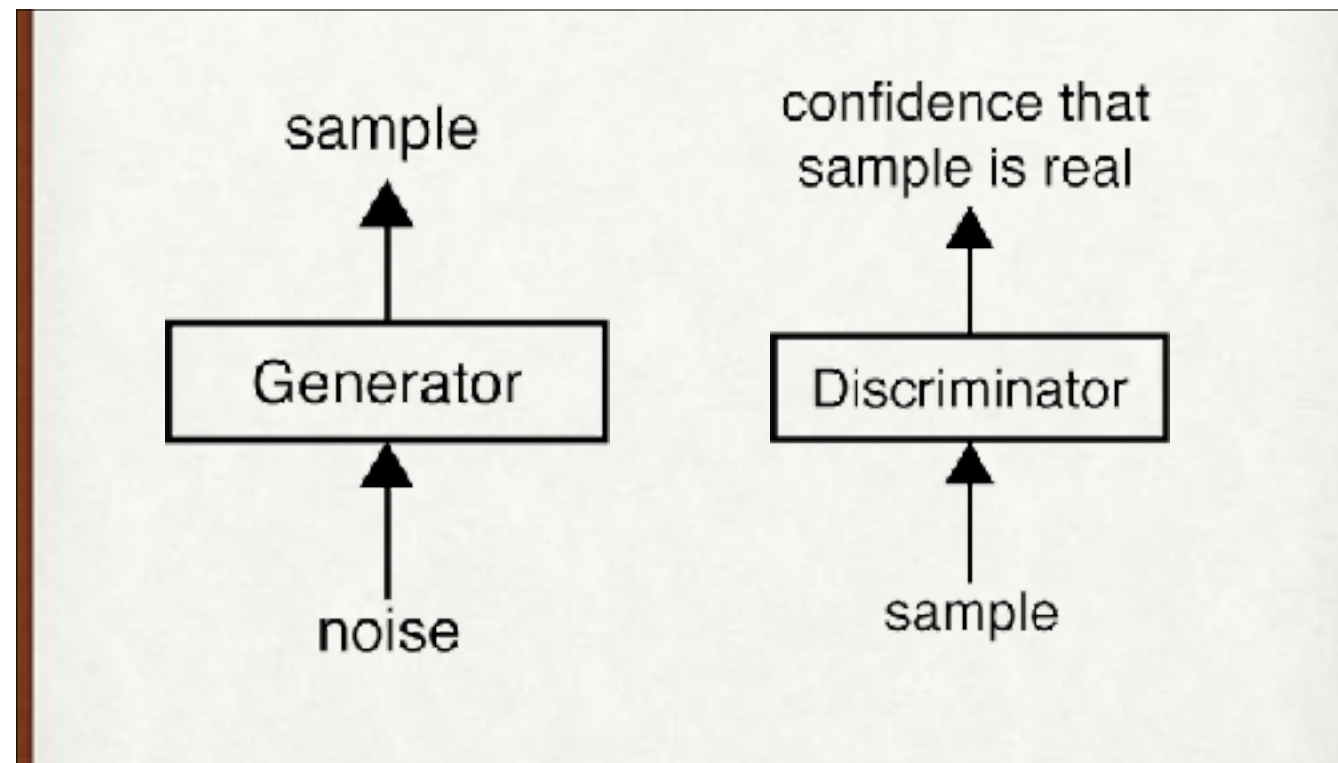
A moment of relaxation before our topic.

Generative Adversarial Networks

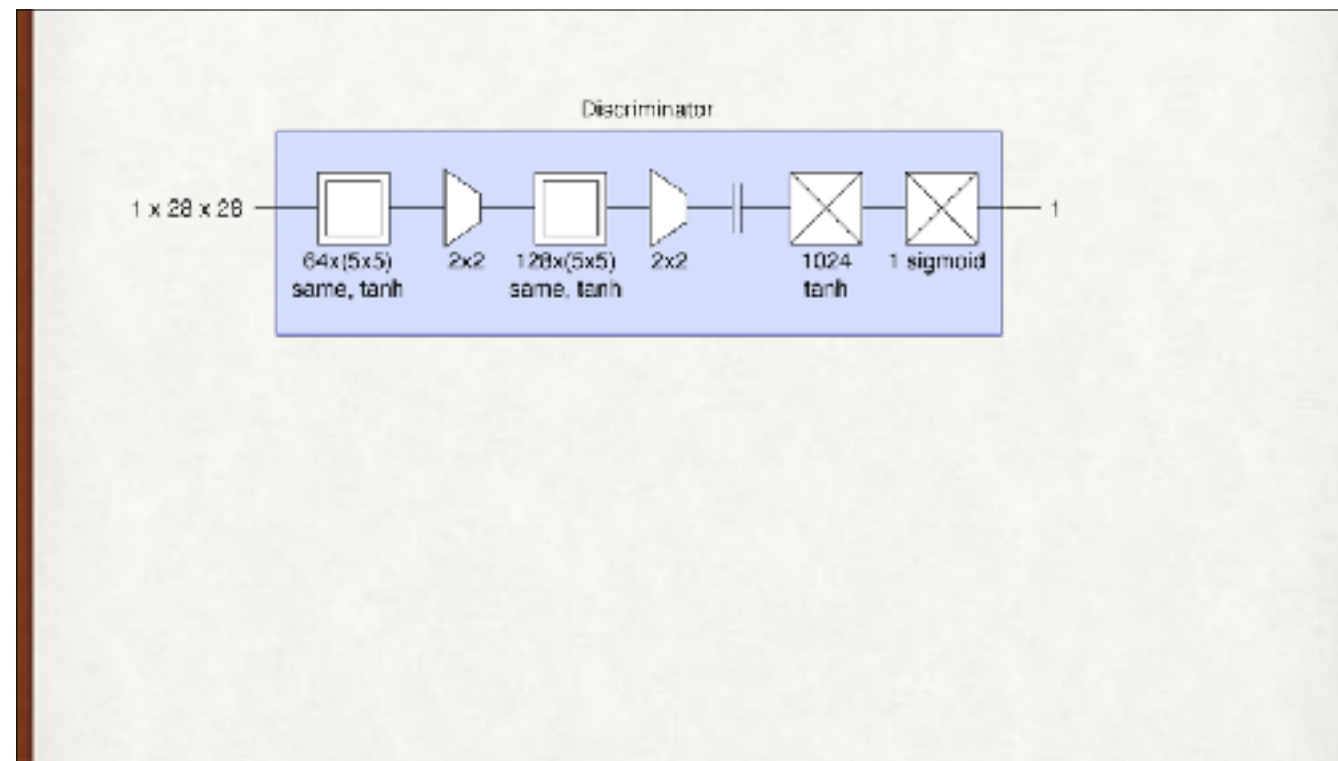
GANs learn how to create new data by running a competition between a data generator and a detector that distinguishes between original input data and generated data.



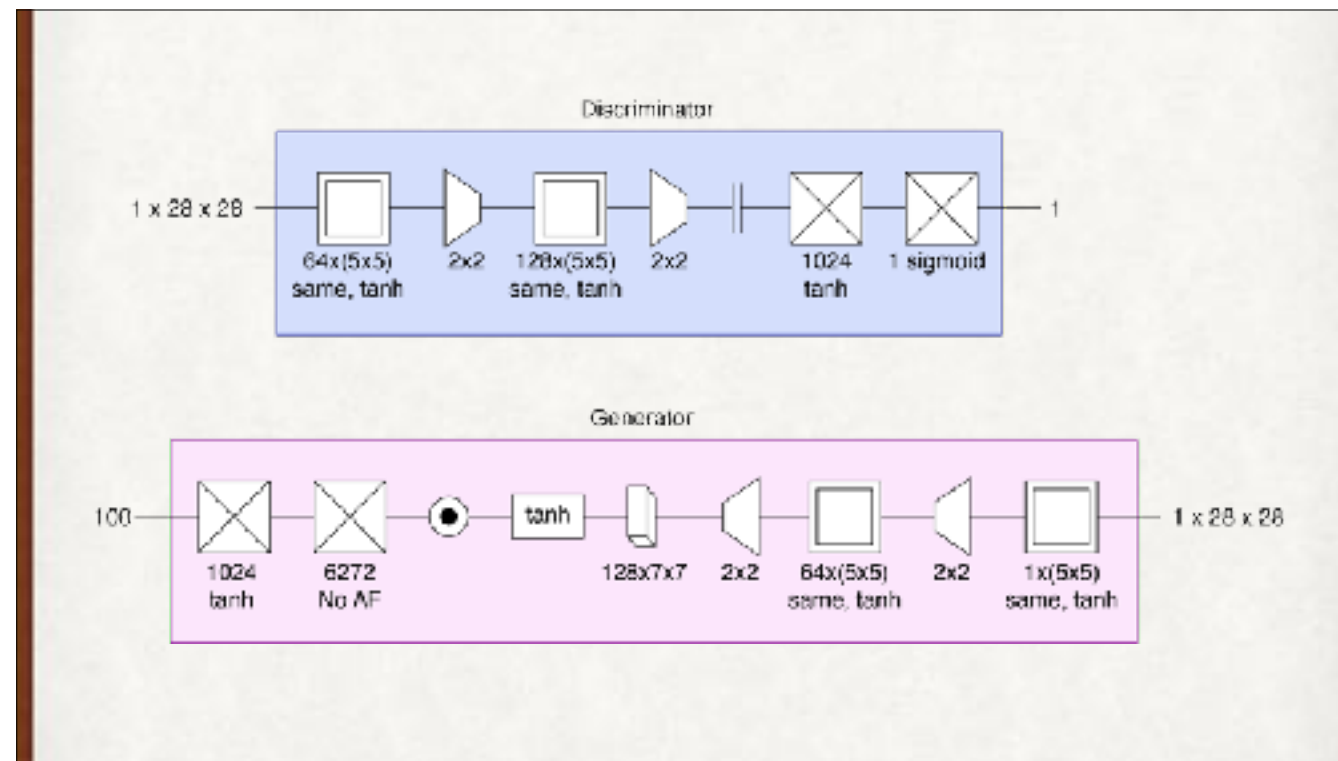
Glenn wants to make counterfeit money. Dawn takes each day's output, plus some real money from the bank, and tries to tell which bills are real and which are counterfeits. Glenn is the "generator," and Dawn is the "detector." Each tries to become as skilled as possible at their task, and that simultaneous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



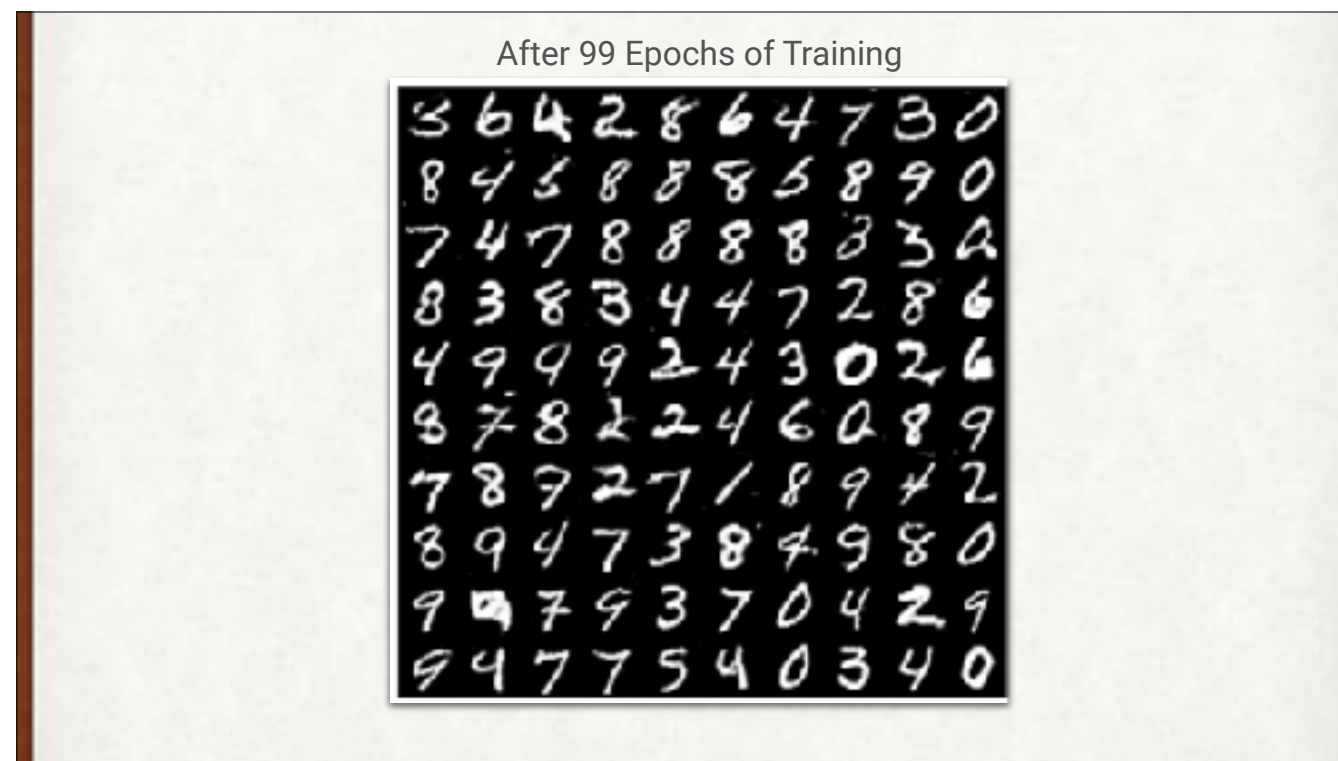
A discriminator and generator for a GAN using MNIST data.



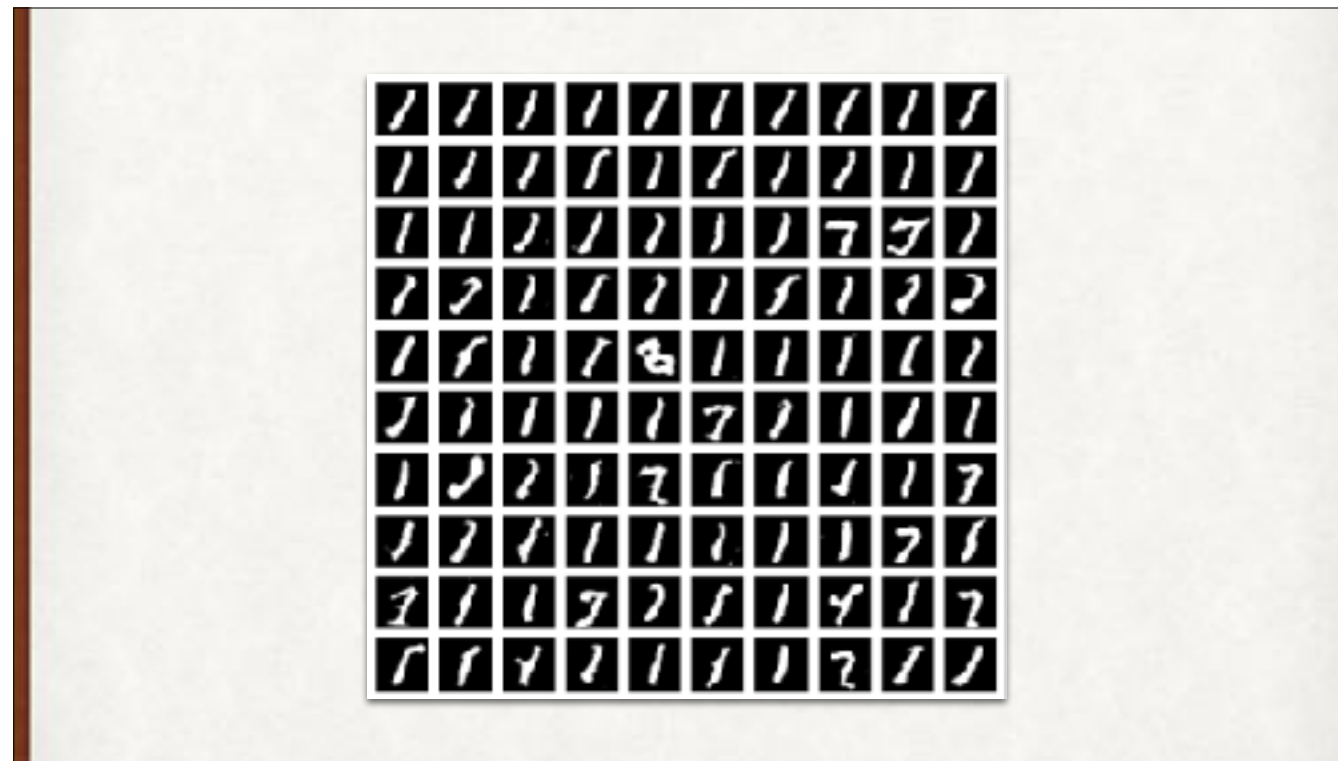
A discriminator and generator for a GAN using MNIST data. The circle with a dot is a “batchnorm” operation, which is a form of regularization. We place this between each neuron’s summation and activation function, so the second fully-connected layer has no AF, then batchnorm, and then a tanh AF.



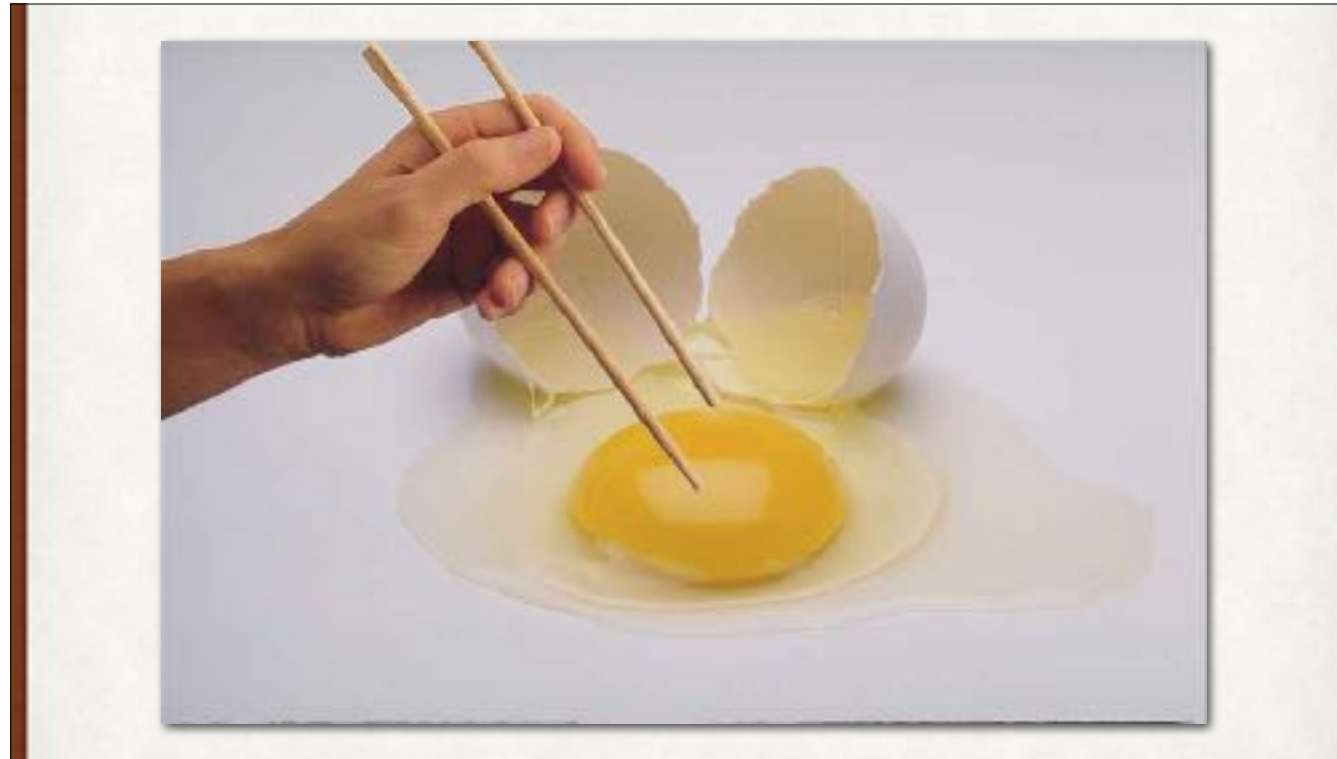
After 1 epoch, we get MNIST-y splotches. This is encouraging!



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



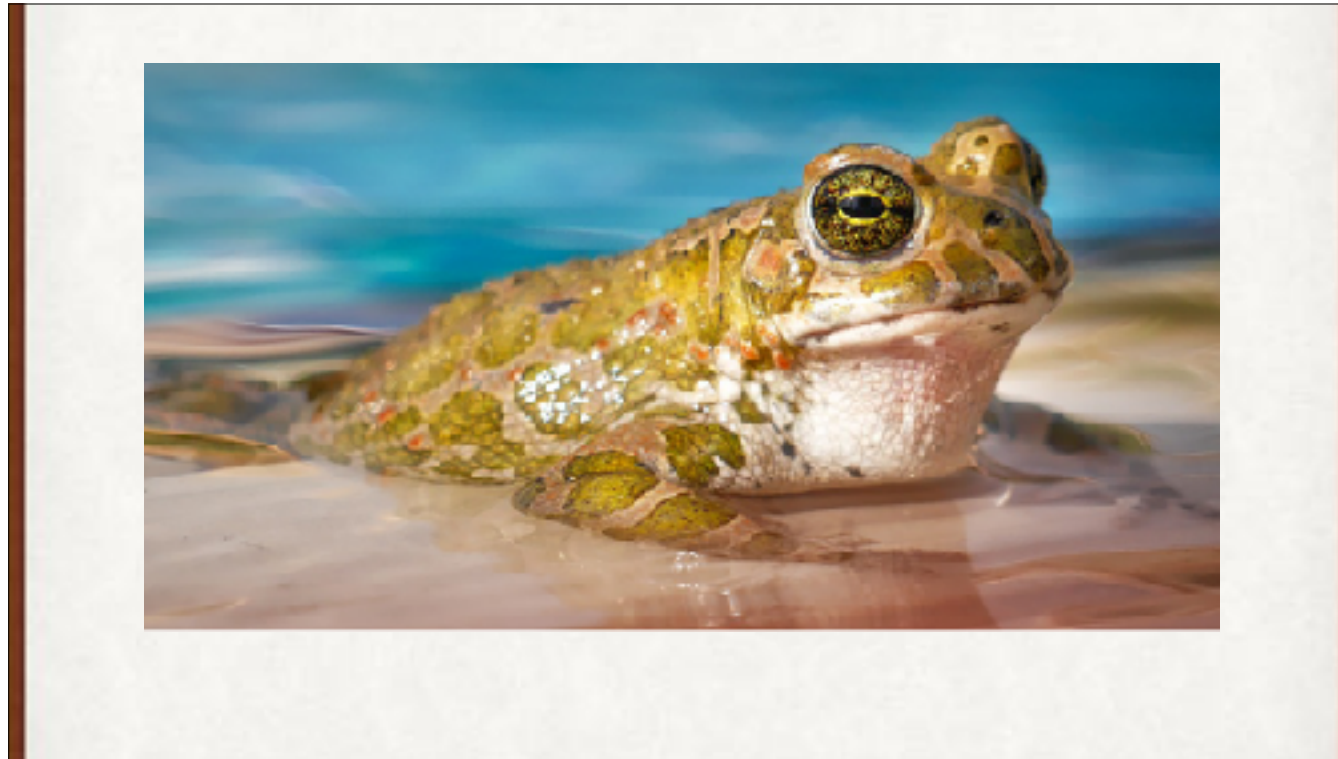
Beware modal collapse! Another sneaky way for AI to do what we asked for, not what we want. If this image of a 1 passes the detector, the GAN could just output that every time (or almost every time) and pass the test. This GAN is on its way to doing just that. We need to take extra steps to prevent this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



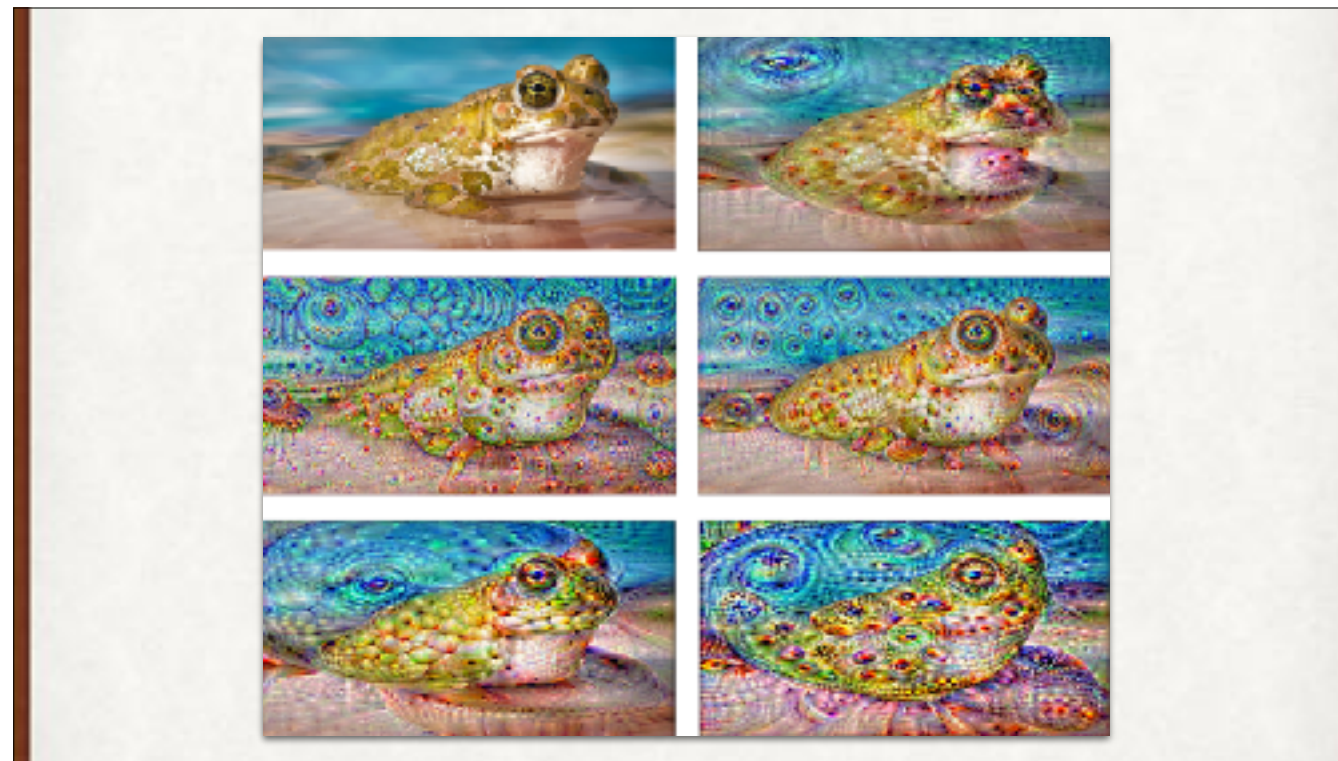
That's it for the main technical stuff. Let's see some fun applications.

Creative Application: Deep Dreaming

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

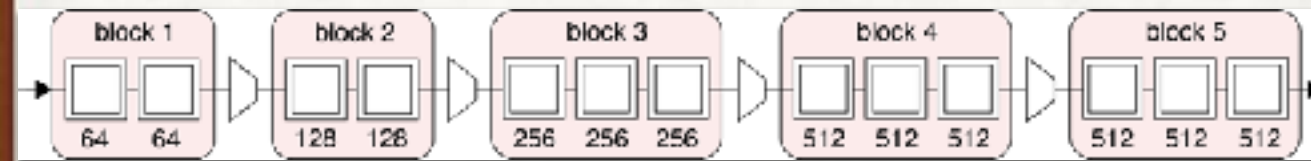


A spectacular frog. This frog is 100% frog. This frog is all frogs. Frogs are so cool.

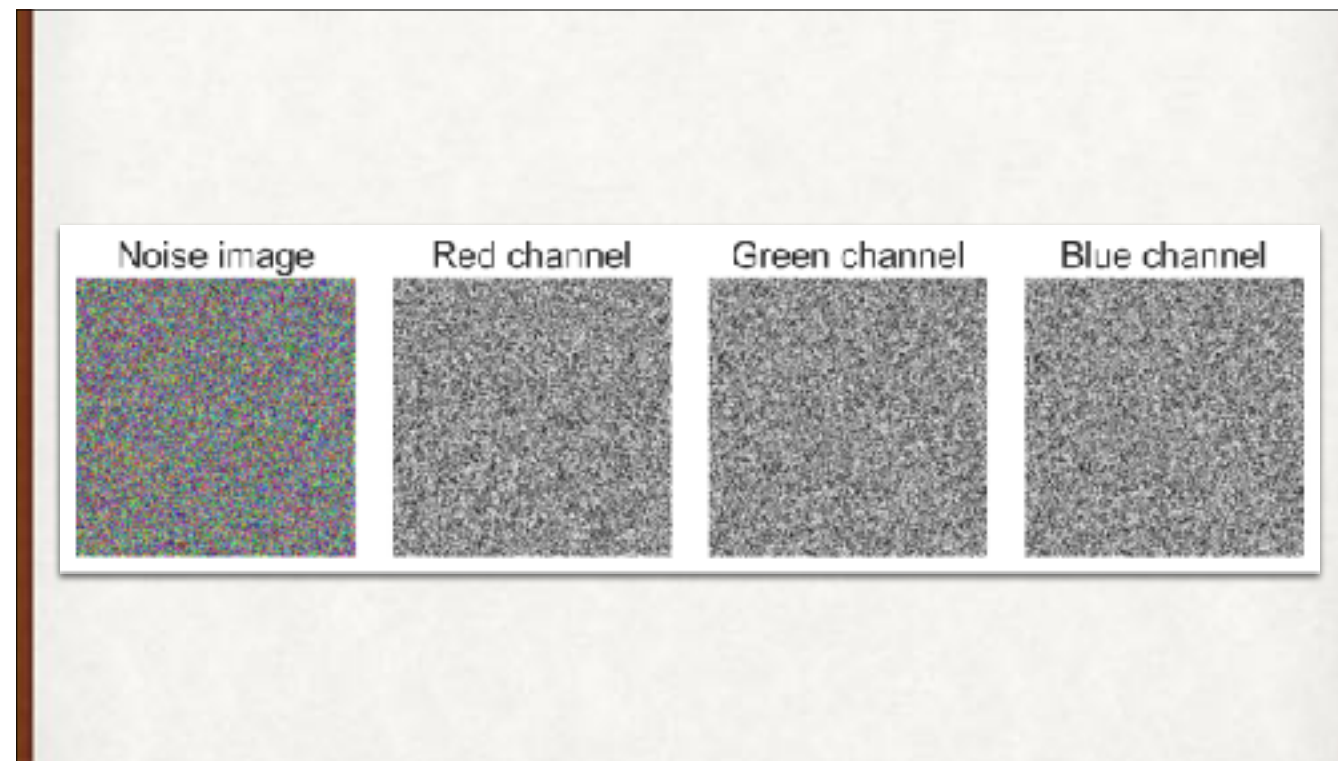


This is where we're going: deep dreaming from the frog.

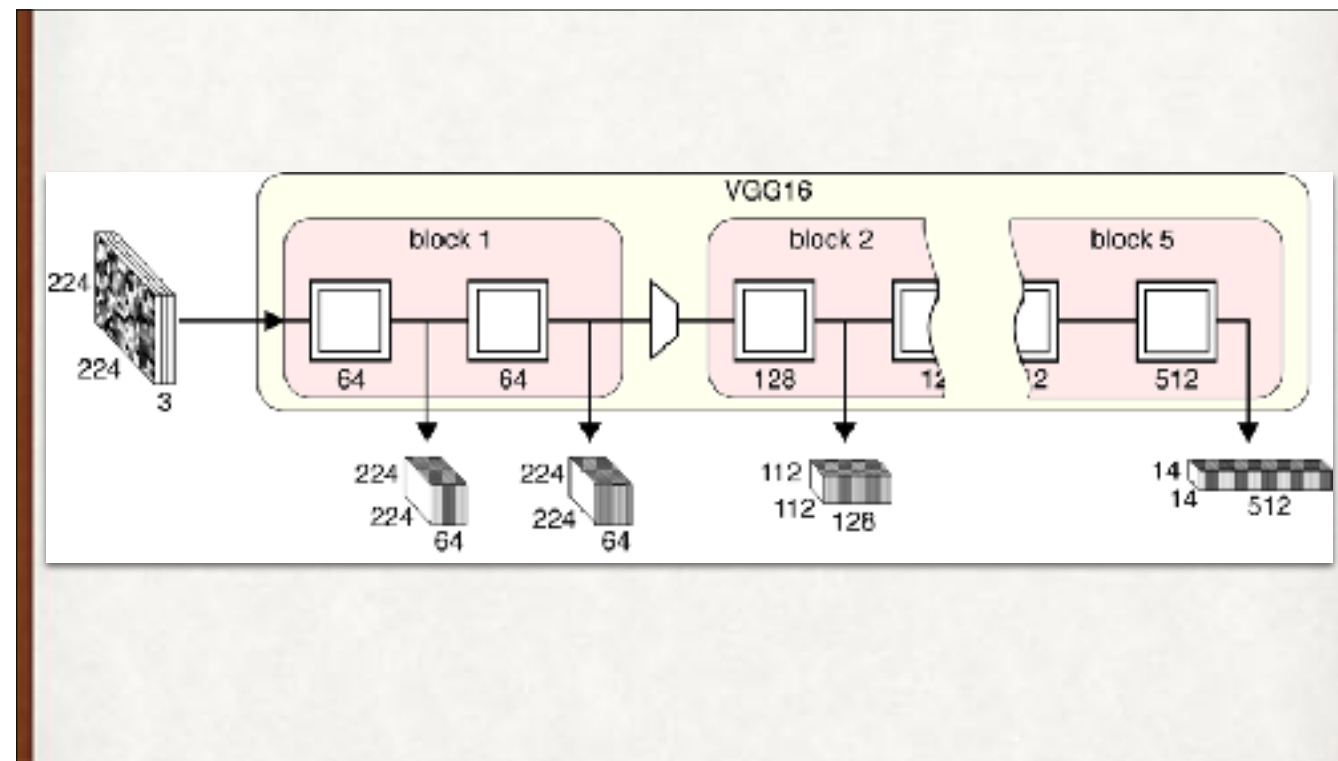
Simplified VGG16 Drawing



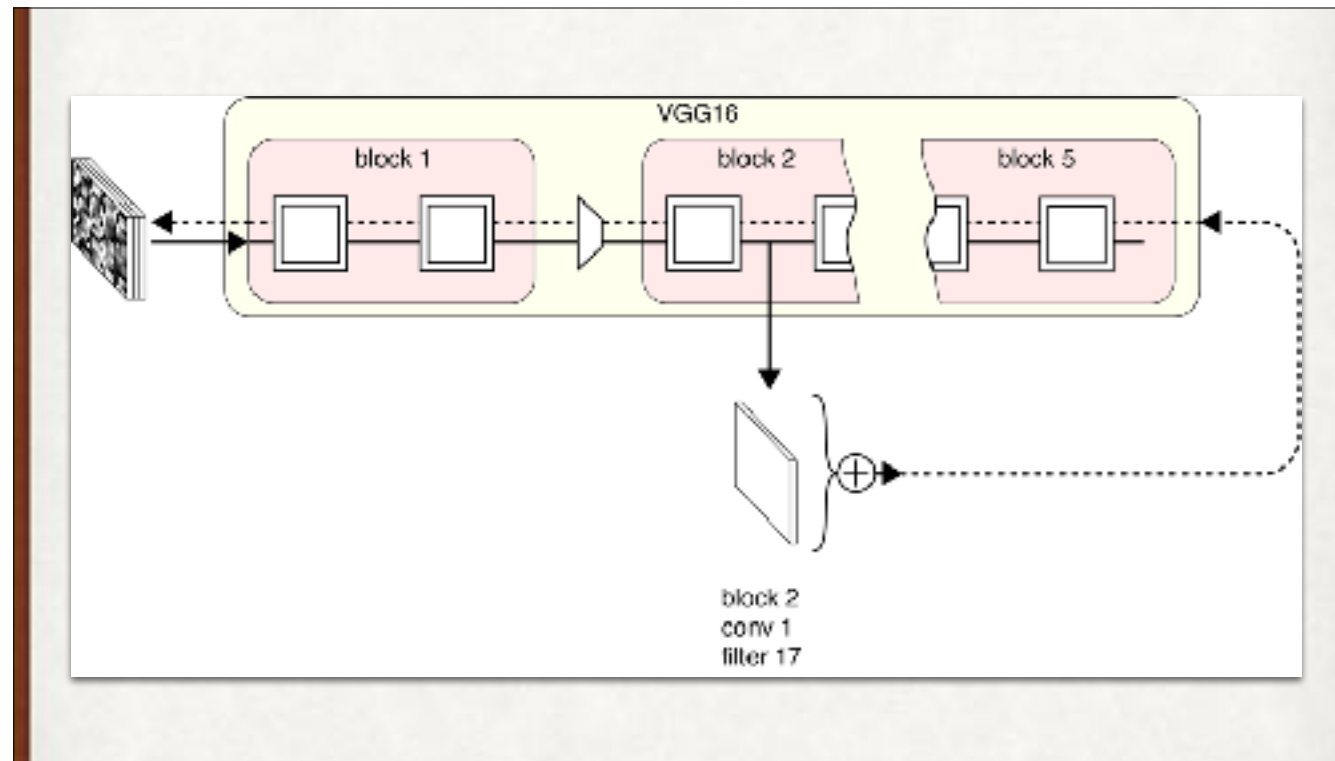
We'll simplify the drawing of VGG16 (but not the network!) to show just the convolution and downsampling layers.



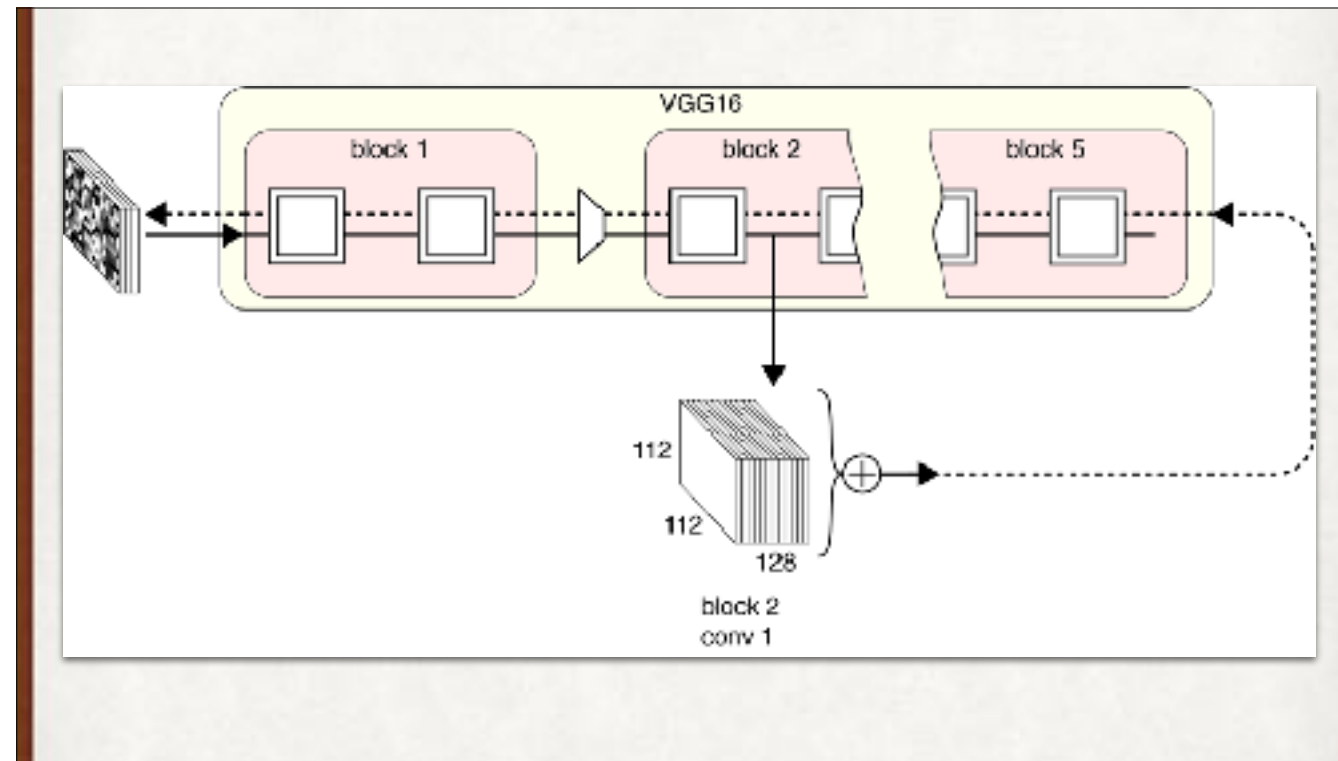
Let's feed VGG16 a random, noisy image as input.



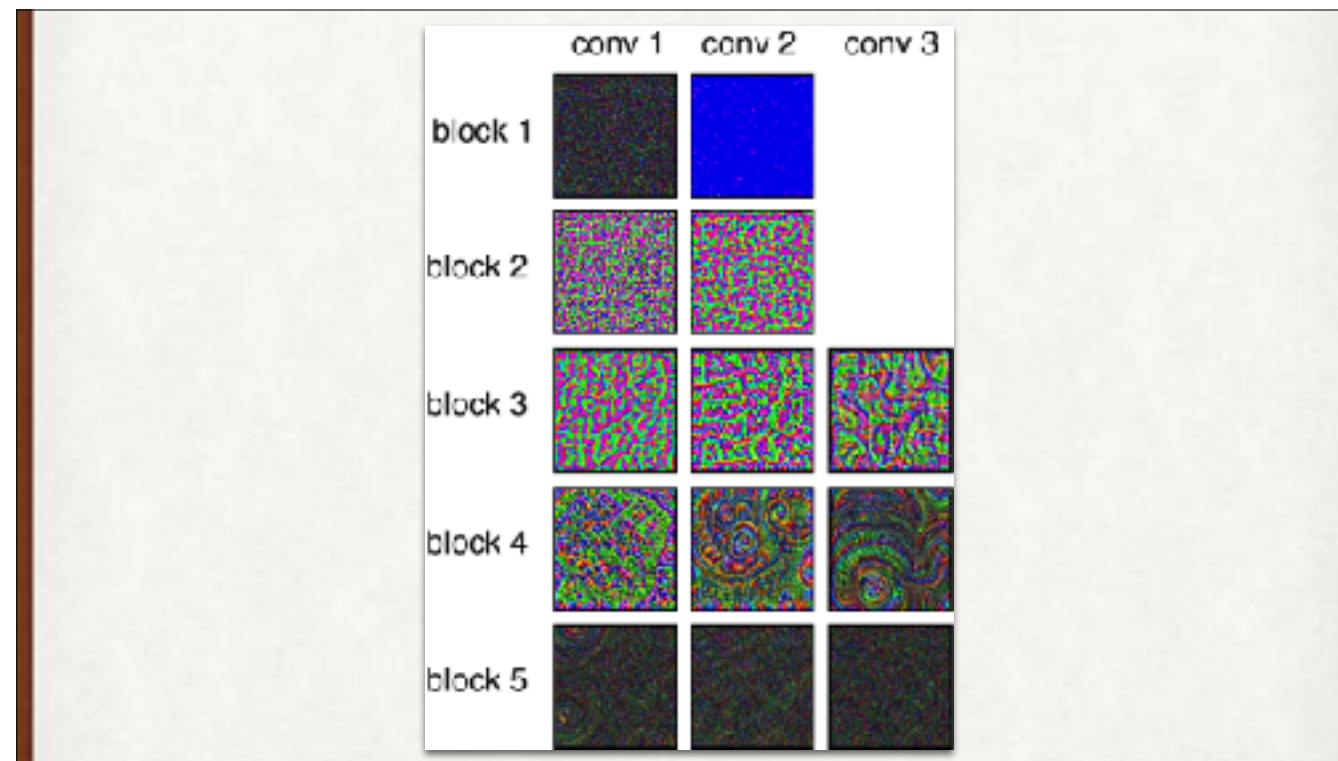
A picture of the tensor coming out of some of the layers. The VGG16 filters are already trained to look for image features. Since the input is noise, the filters are likely to fire a little bit here and there, as random splotches of pixels approximate the features they were trained to look for.



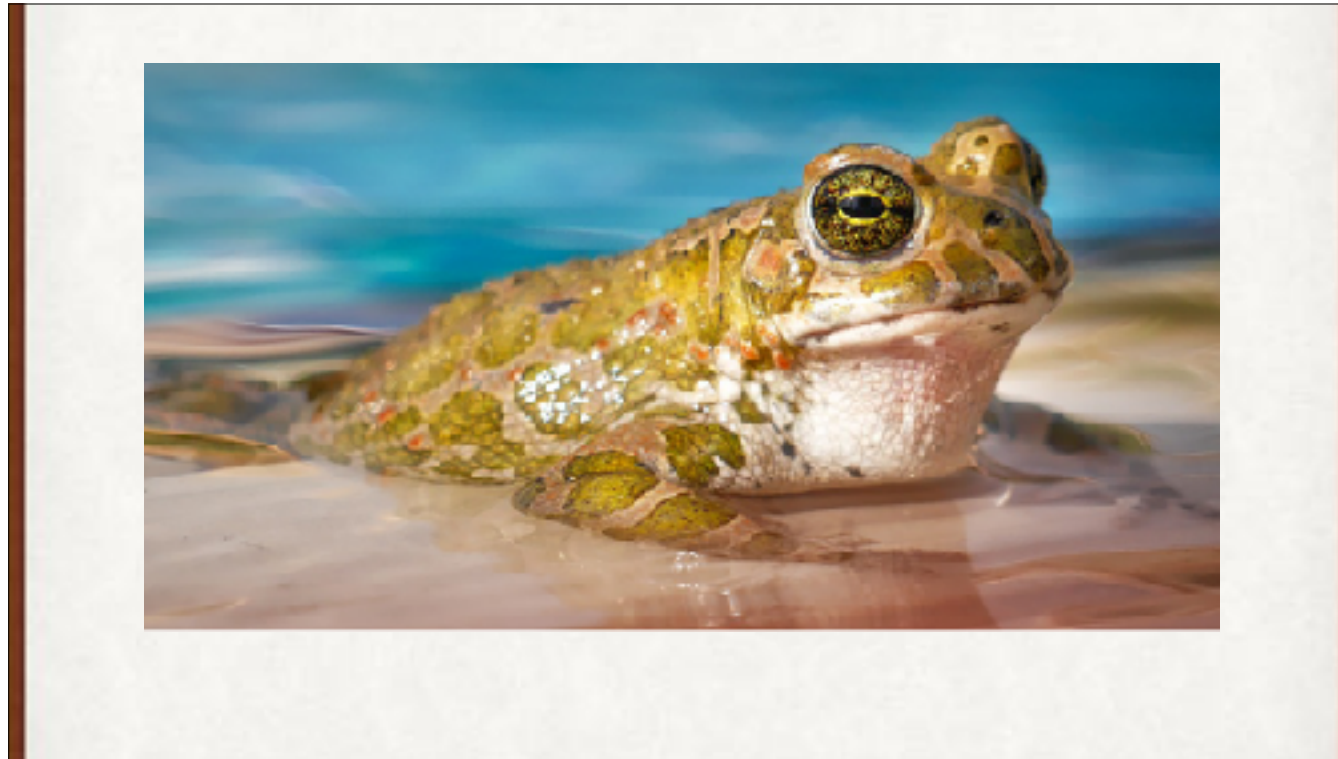
We can take the summed output of one filter on one block and use that as our “error”. We’ll push the error gradient backwards through the network using backprop, but we won’t update the network! The network is “frozen” and fixed. Instead, we’ll push the gradient all the way back to before the first layer. That will tell us how to change the pixel values to reduce the “error”. But we want to make the filter respond strongly, so instead of minimizing this “error”, we’ll change the pixel values in order to maximize it. This is how we visualized layers earlier, looking for an input that makes the layer respond as strongly as possible.



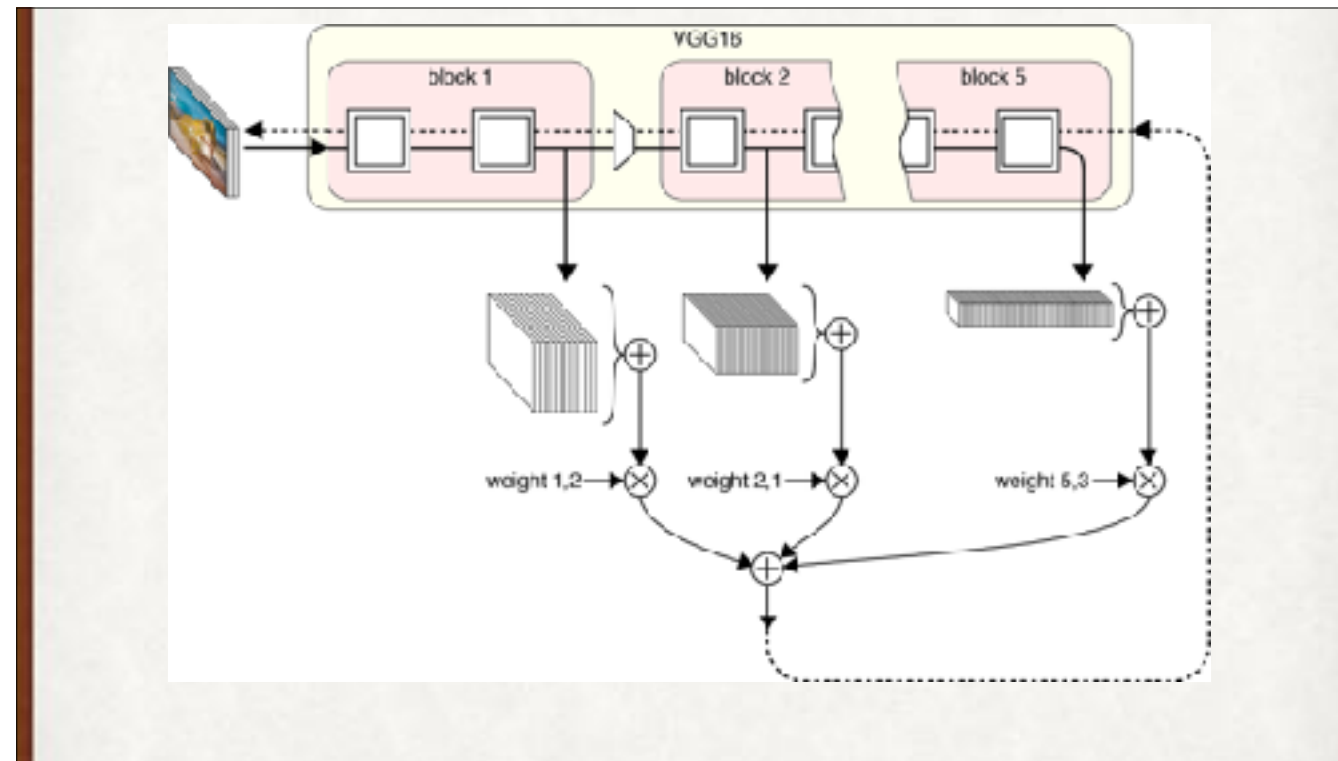
Let's sum up the responses of all the filters after a layer, and use that as our error. Again, the network doesn't change in any way. But we do modify the pixel values in the noise image so that they will cause a bigger stimulation to the outputs of this layer.



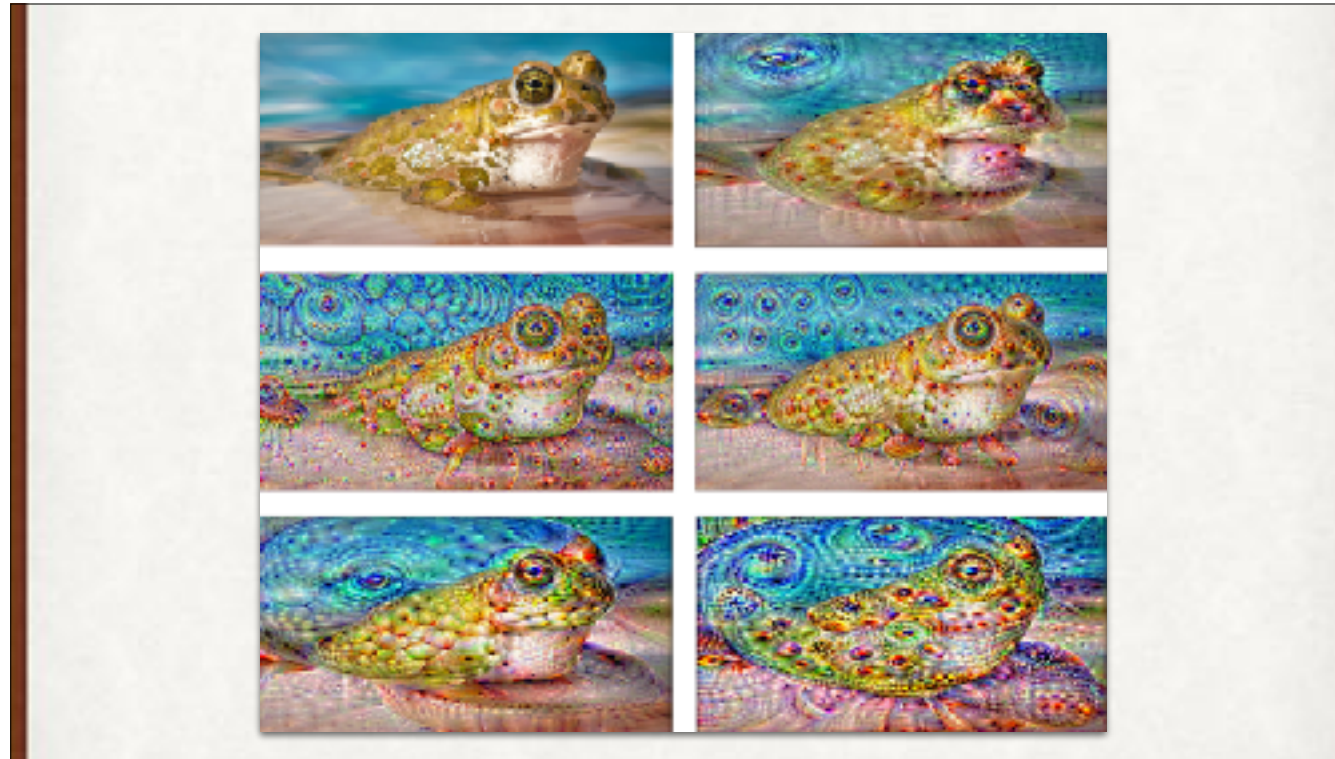
Here are the images that the layers reacted to most strongly. Since the input was noise, every time we make a picture like this the results will be different, but they should be roughly similar. Notice that blocks 3 and 4 seem to be pushing our noise into forms with a lot of structure.



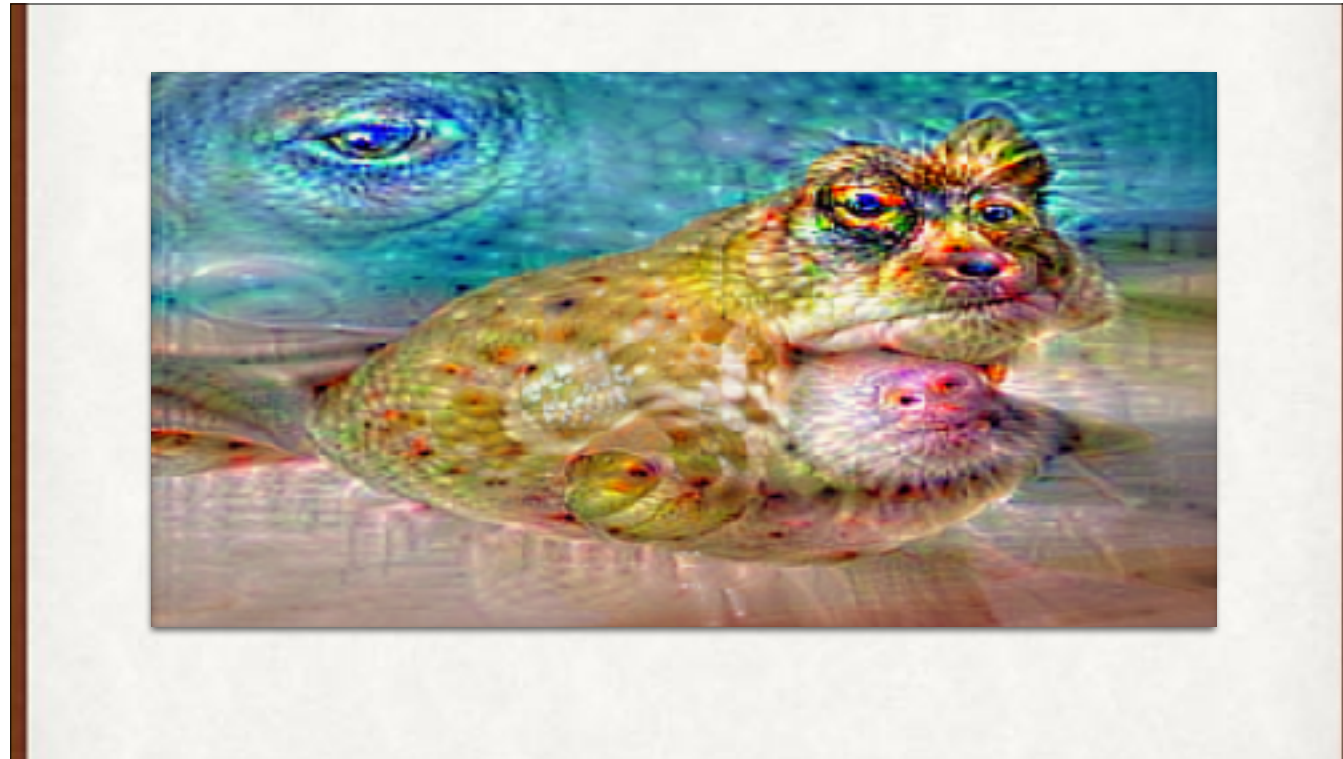
A spectacular frog. This frog is 100% frog. This frog is all frogs. Frogs are so cool.



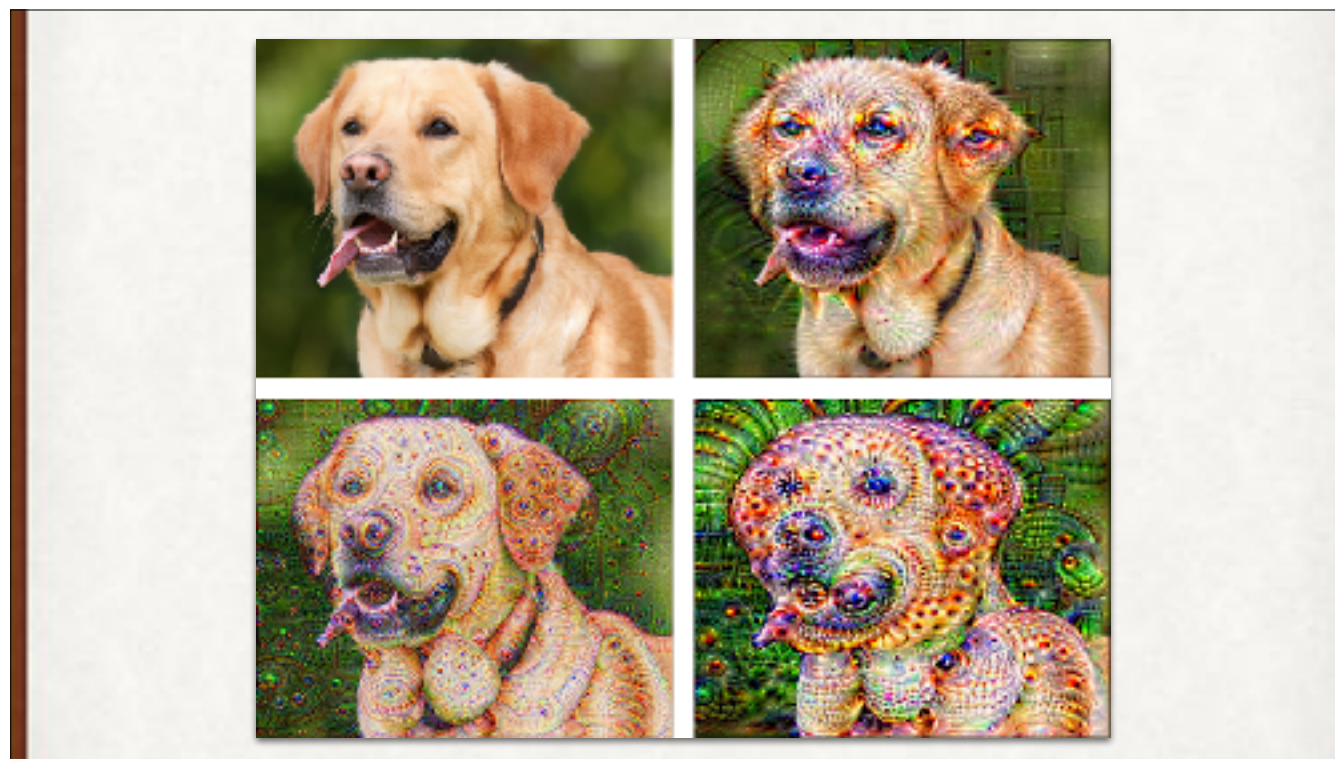
We'll feed the frog to VGG16, instead of noise, and get the summed responses of different layers. We'll scale those sums, add them together, that's our new "error". We'll then modify the pixel values in the frog image to drive up this "error". We'll run the modified frog through the network again, compute 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.



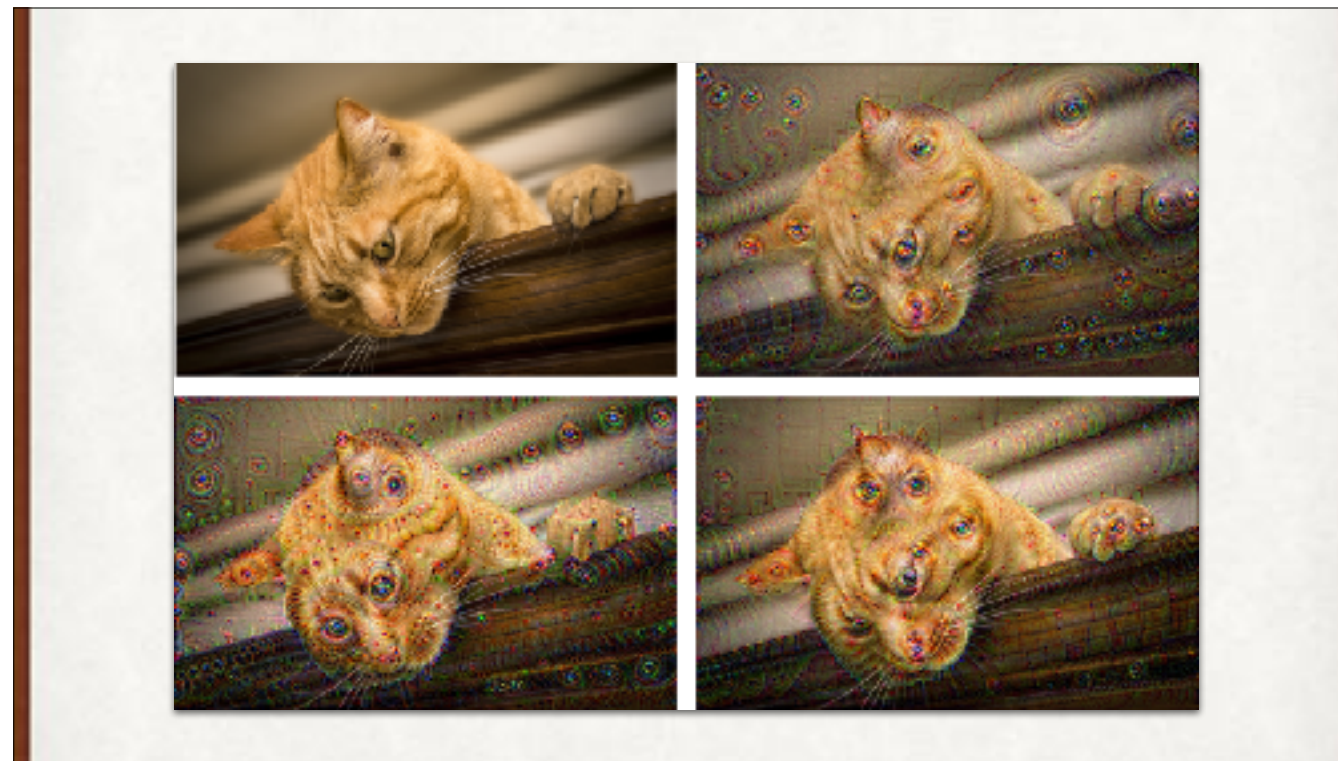
Obviously (hah!) this is what we get back. Originally called inceptionism, it's usually called deep dreaming. The filters are responding to little variations in the input, and the loop we've built causes those variations to get amplified and enhanced, causing the filters to respond more strongly, causing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



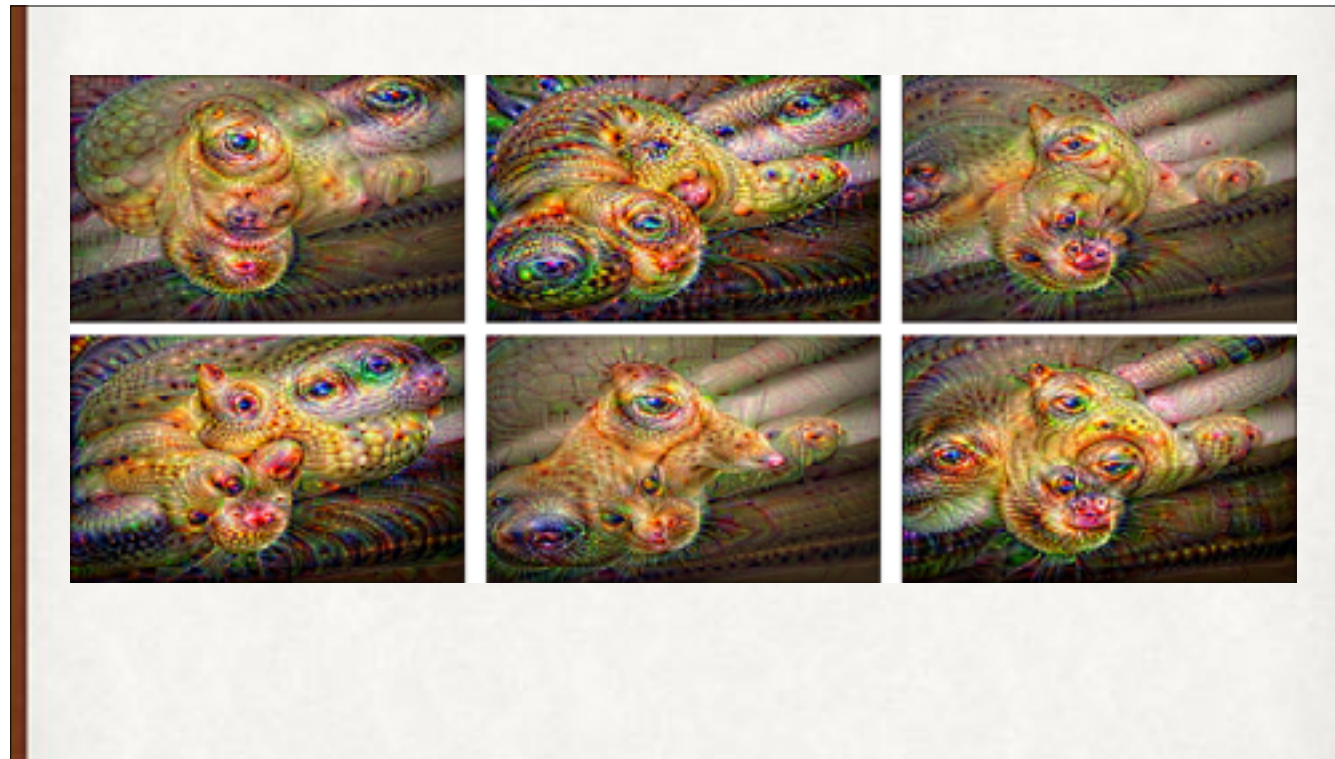
Obviously (hah!) this is what we get back. Originally called inceptionism, it's usually called deep dreaming. The filters are responding to little variations in the input, and the loop we've built causes those variations to get amplified and enhanced, causing the filters to respond more strongly, causing 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.



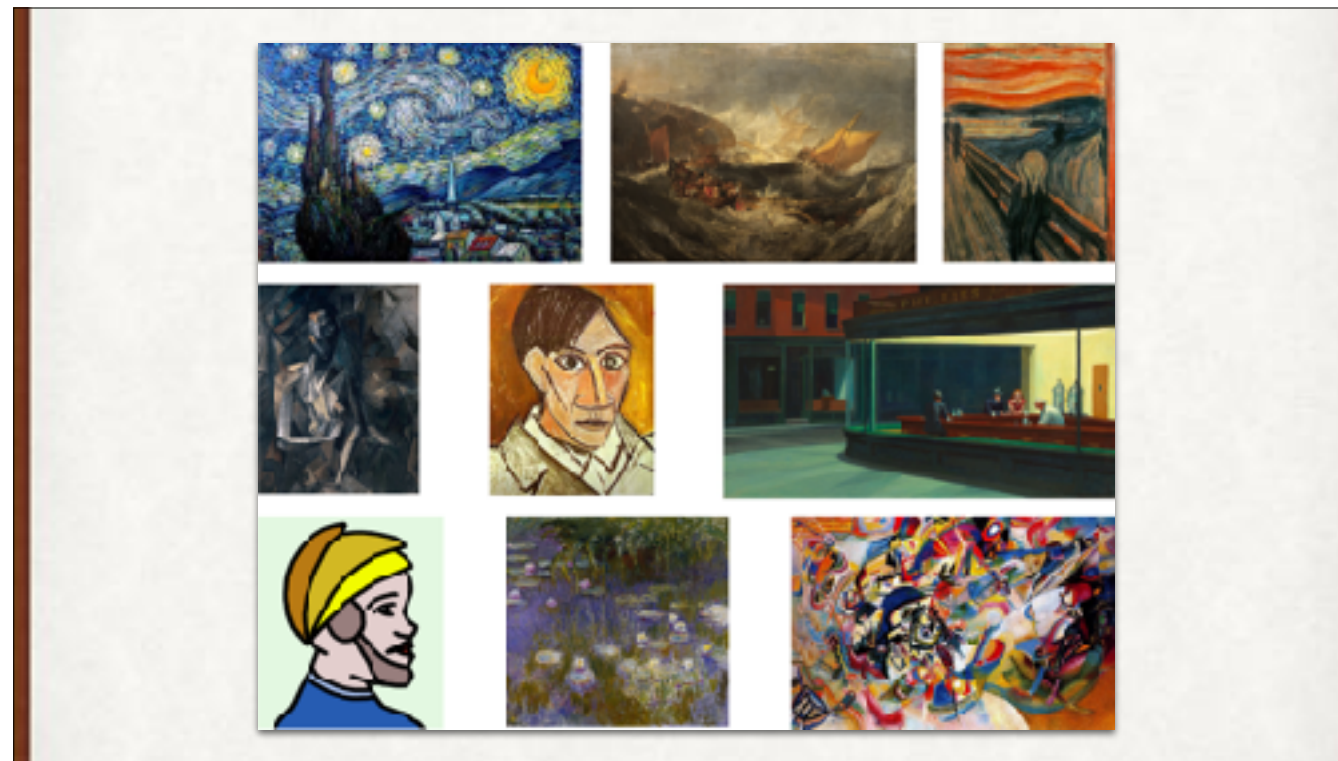
Deep dreaming on a picture of a cat.



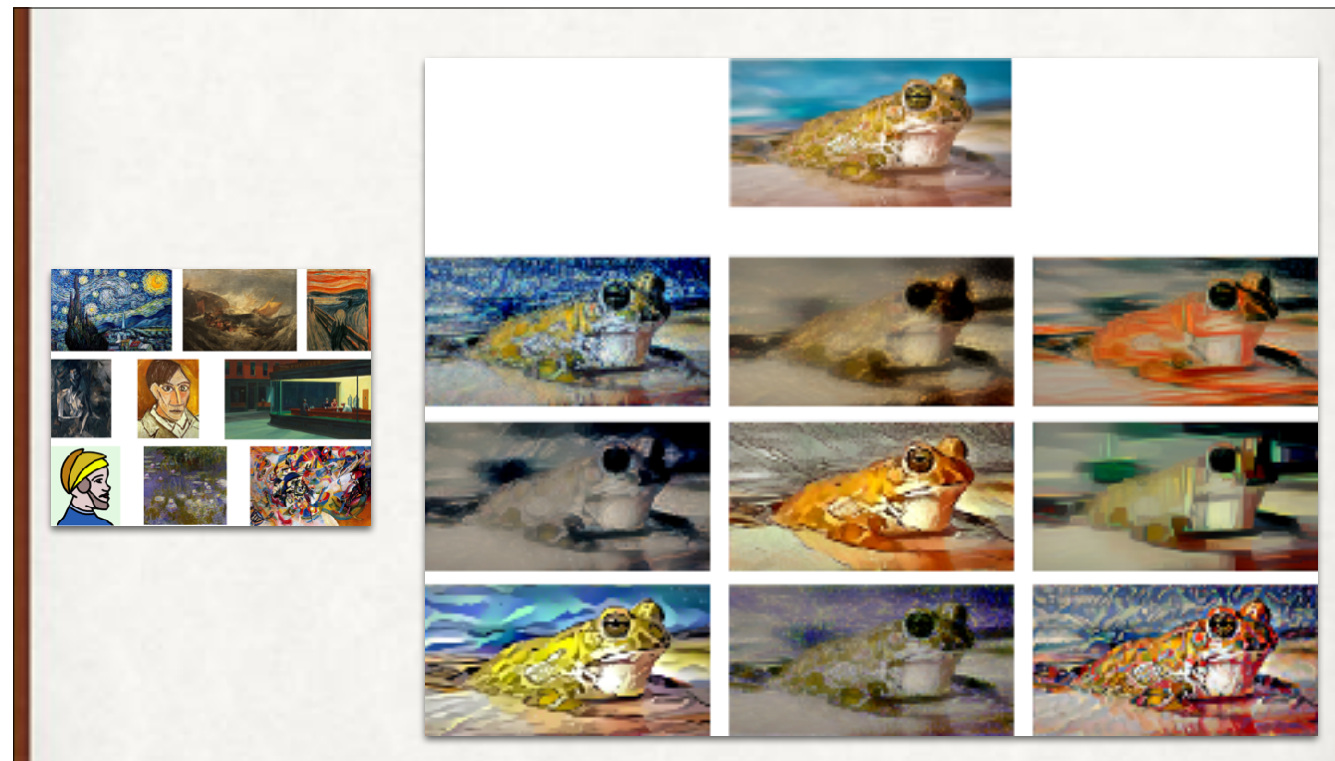
Using some other layers and weights to make other cat dreams. Different networks will give us different dreams, because their filters look for different kinds of image features. VGG16 is big on eyes, so we see lots of them.

Creative Application: Style Transfer

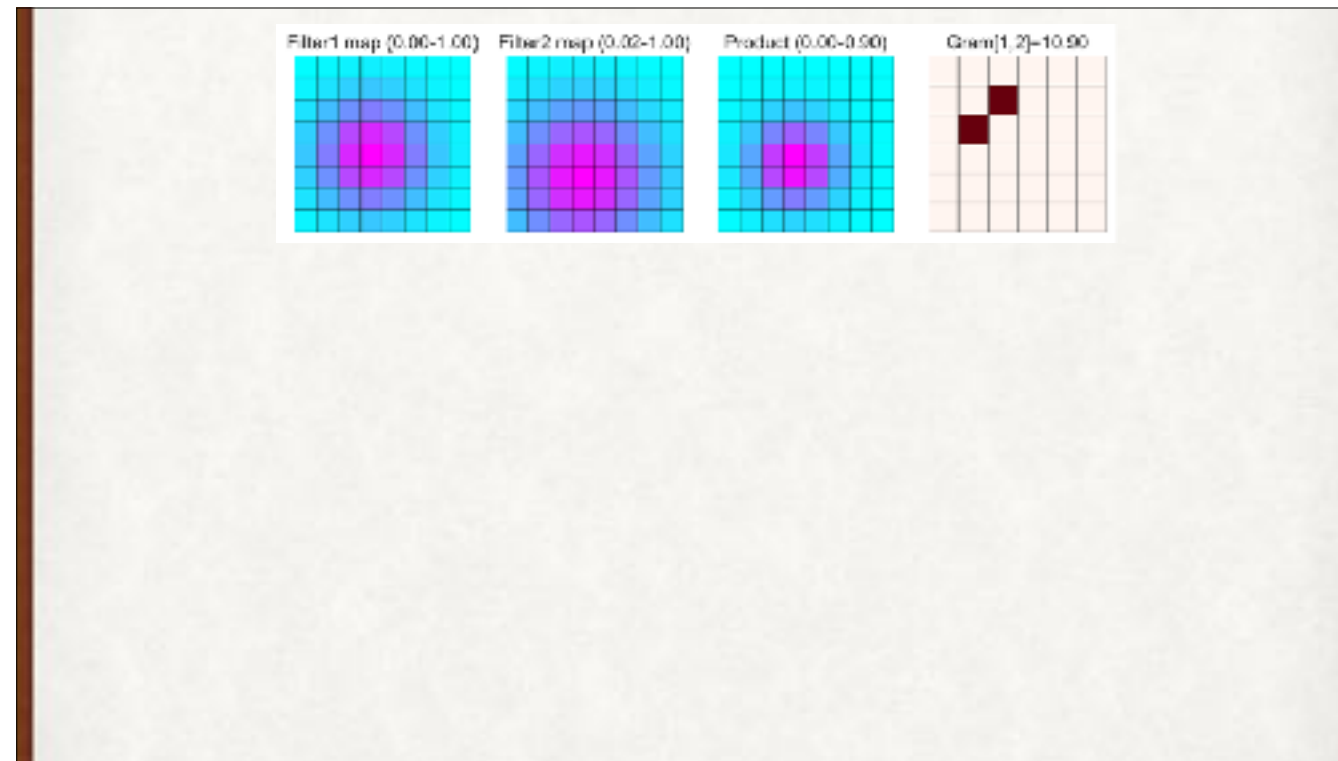
Style transfer means changing an image to look like one created by someone with a recognizable style (we use the word “style” in its popular sense, and don’t try to nail it down more specifically).



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



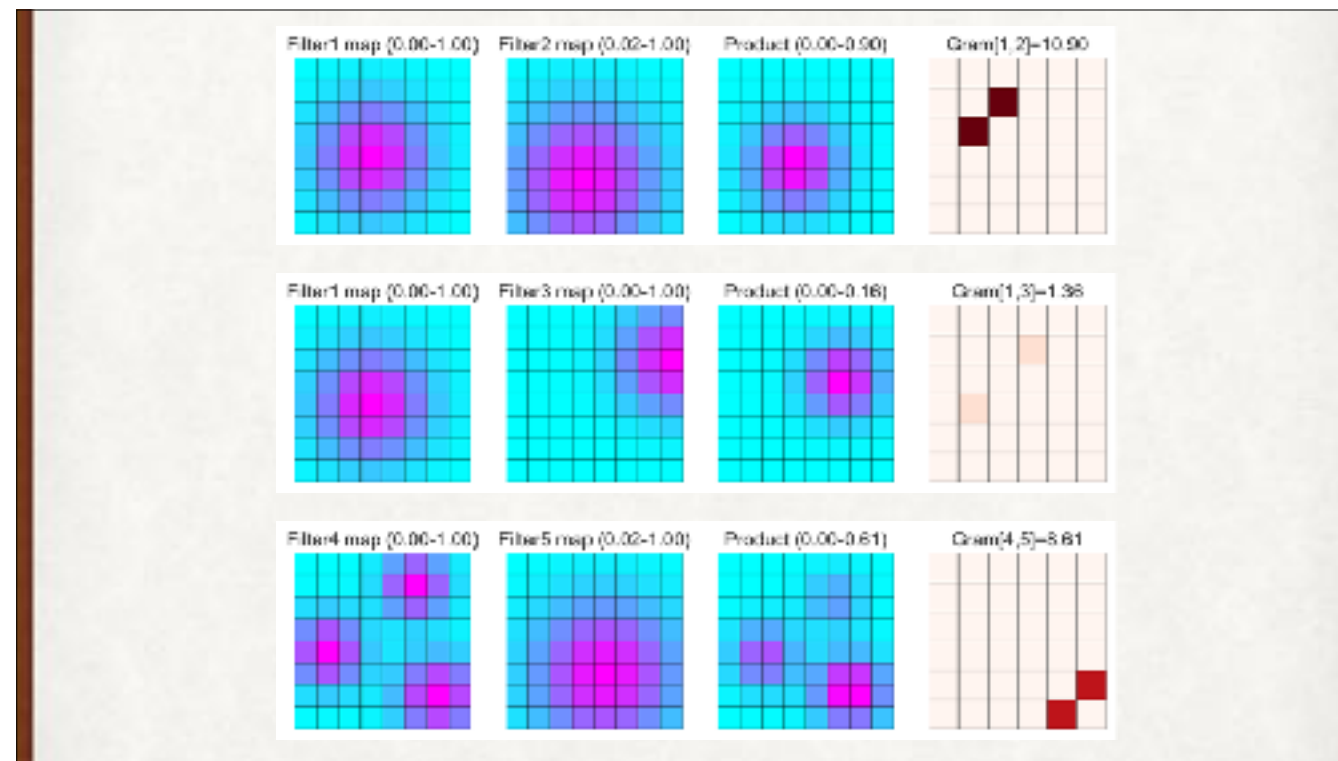
Wow. How do we do this?



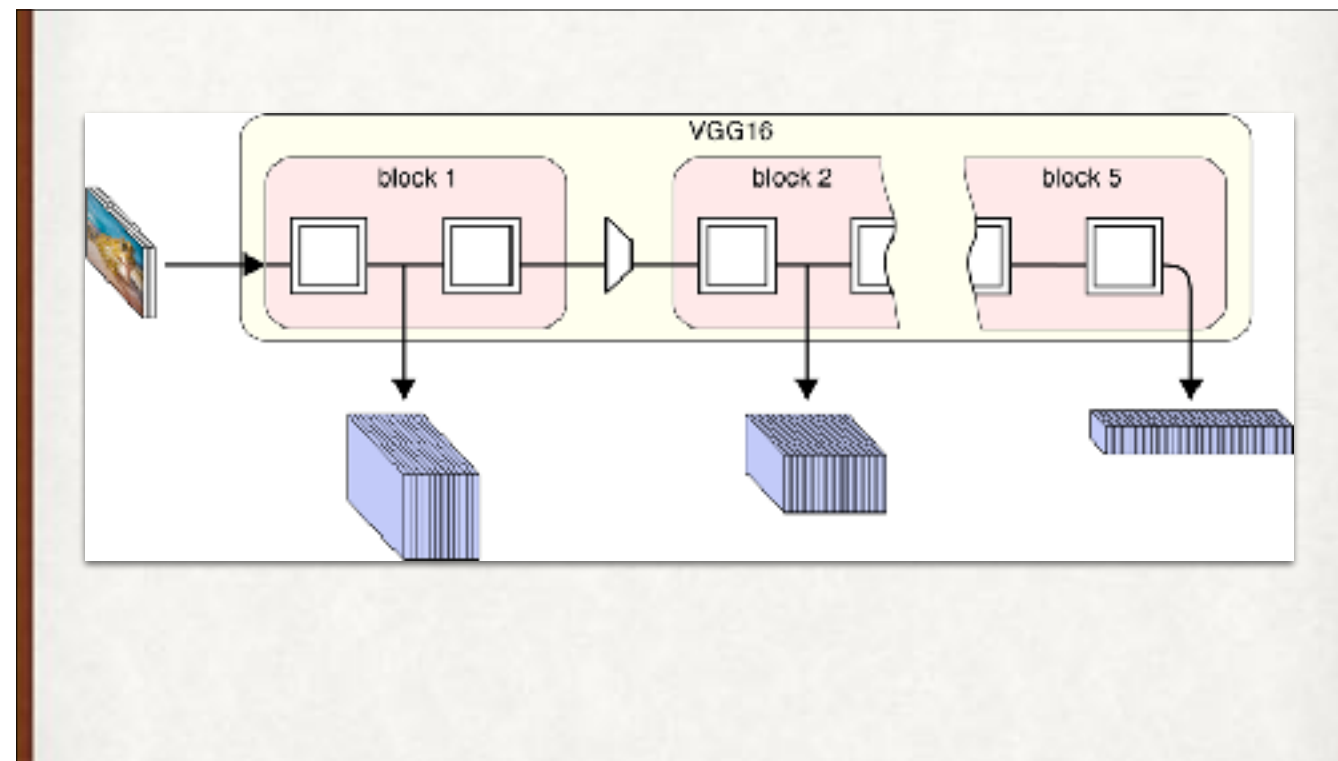
The Gram matrix measures how much two filters fire simultaneously on the same input. When different filters both respond to the same location in their input, the Gram value will be high. So we're finding where both filters responded significantly. These two filters are both responding strongly to many of the same pixels, so their Gram value is high (a dark dot in the matrix).



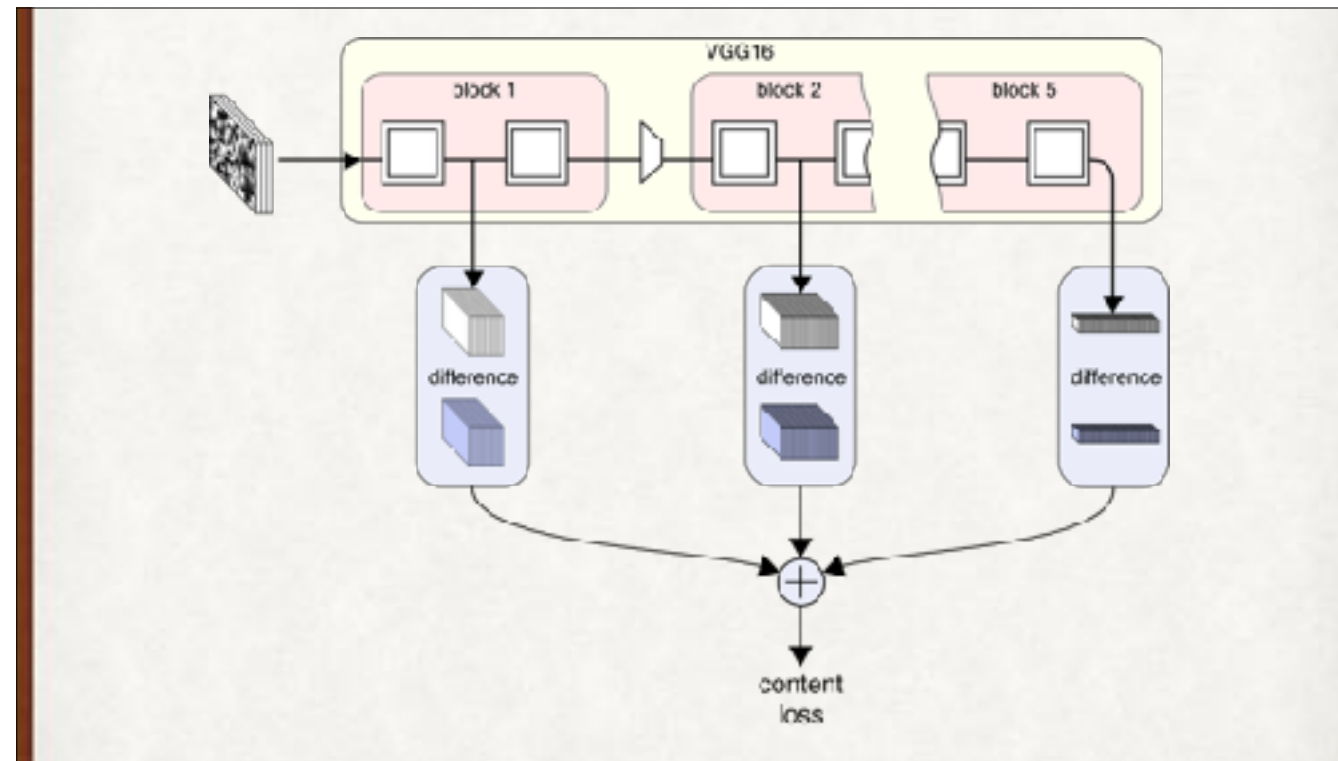
These two filters are responding to different location, so their Gram value is low (a light dot).















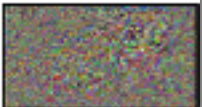
The two filters at the bottom have some overlap, so their Gram value is moderate. But what the heck does this have to do with style?

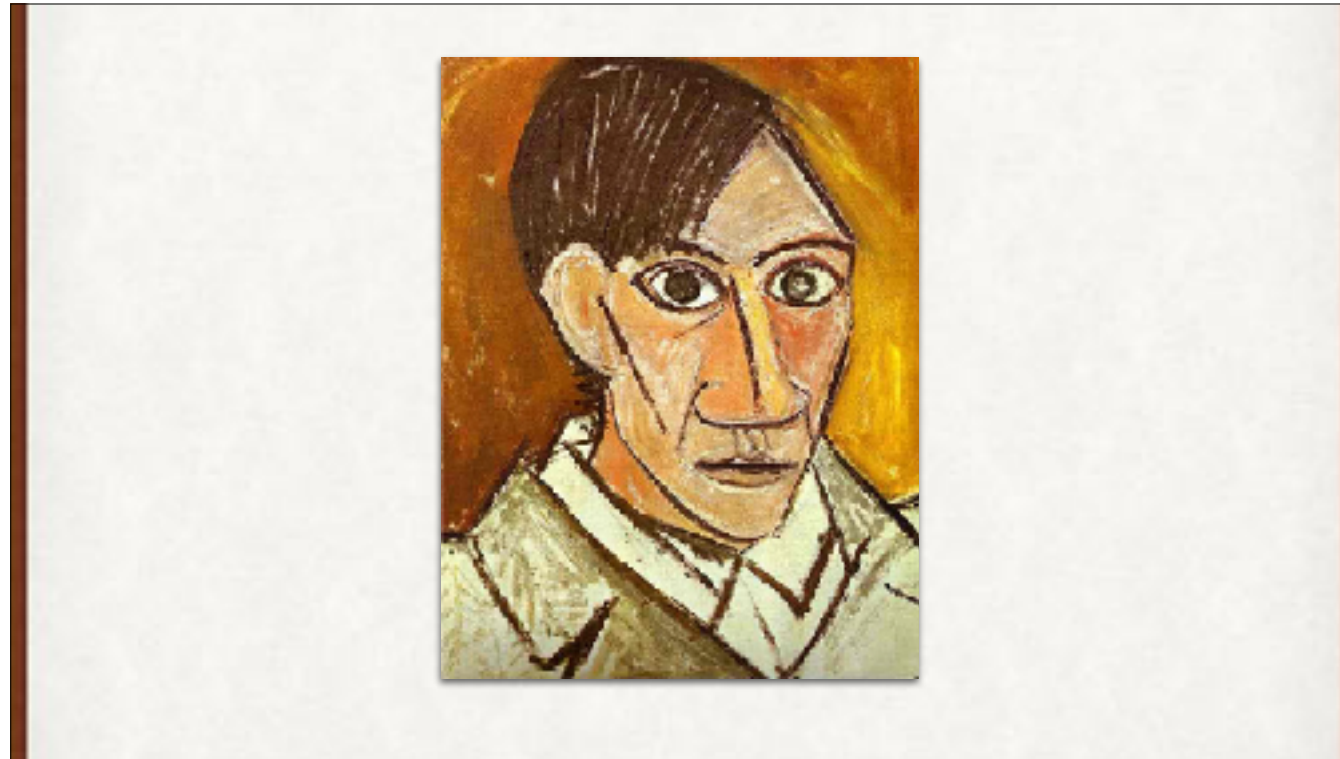


We'll come back to the Gram matrices. For now, let's feed the frog into VGG16, and save the outputs of each layer.

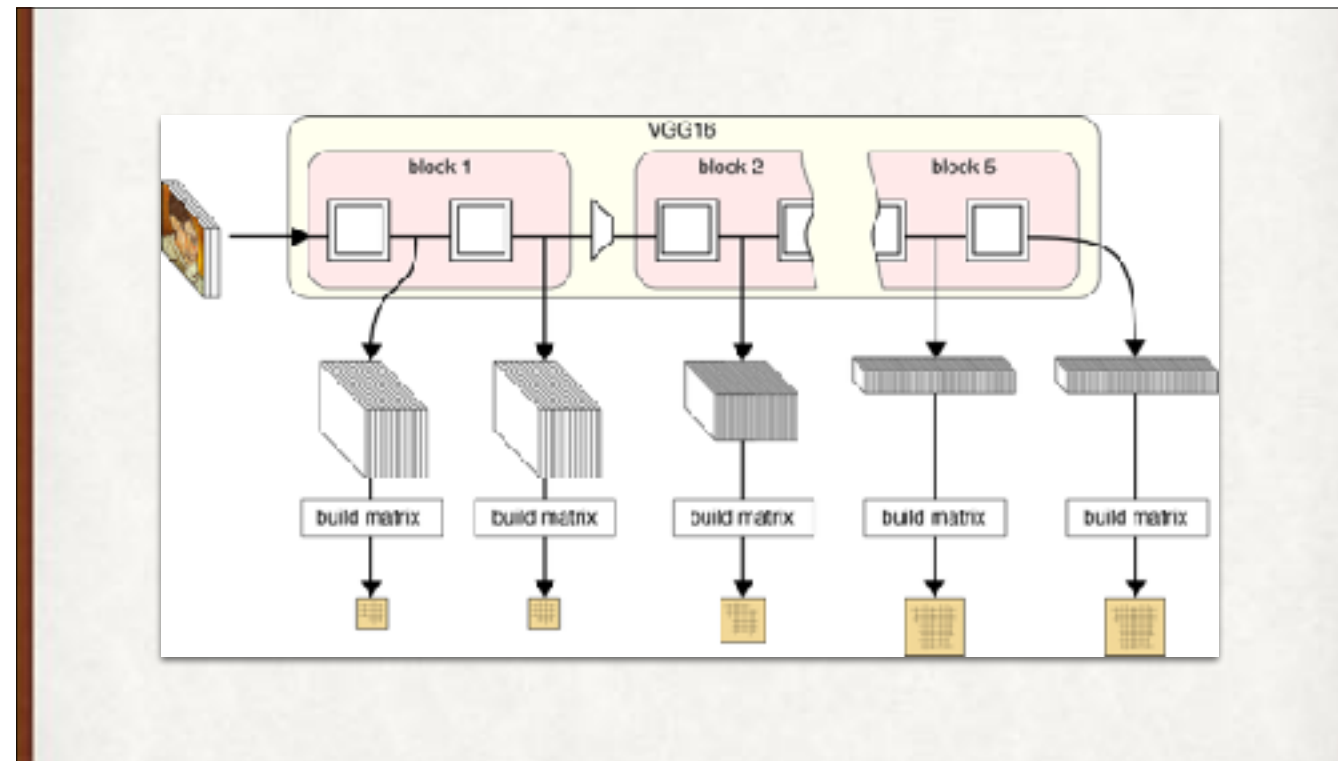


We'll run noise into VGG16, and compare the total output of each layer with what we saved for the frog. Adding those all together gives us the "content loss," or how different the noise is from the original picture of the frog.

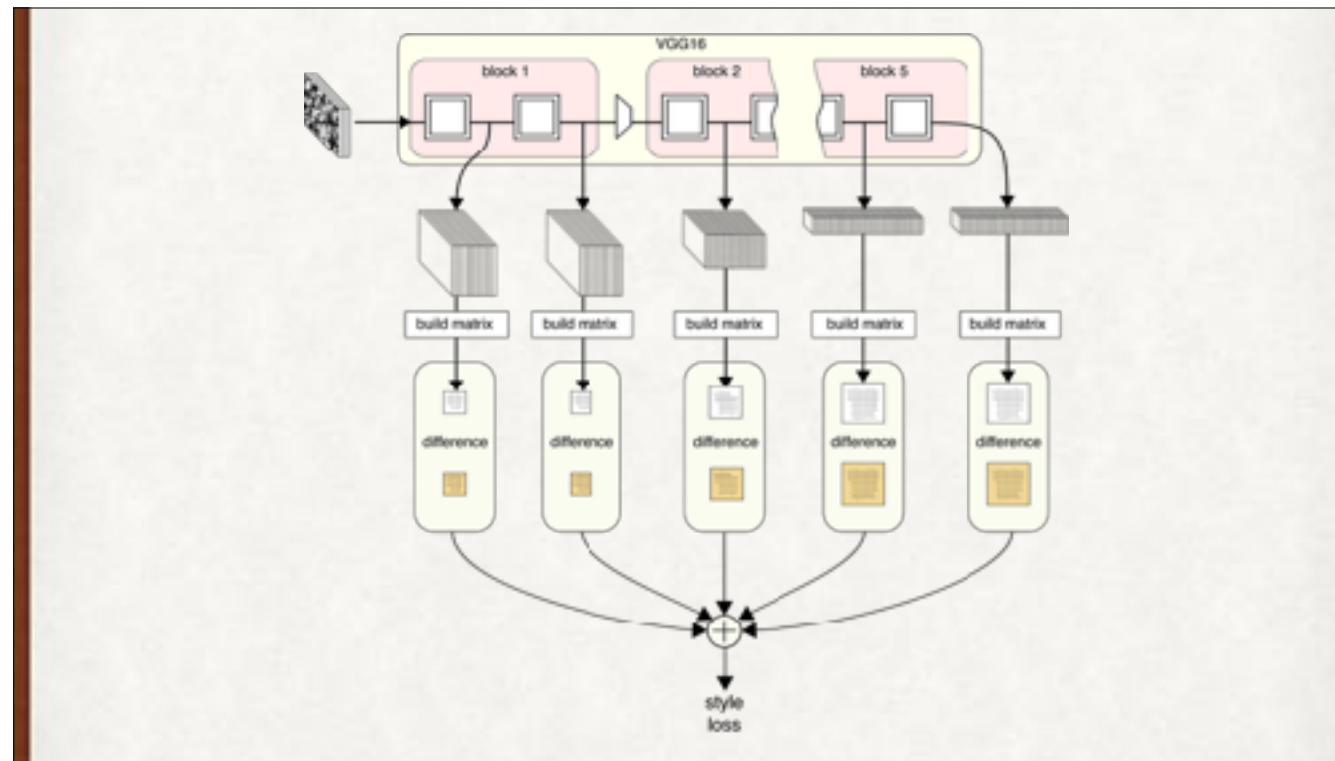
	conv 1	conv 2	conv 3
block 1			
block 2			
block 3			
block 4			
block 5			



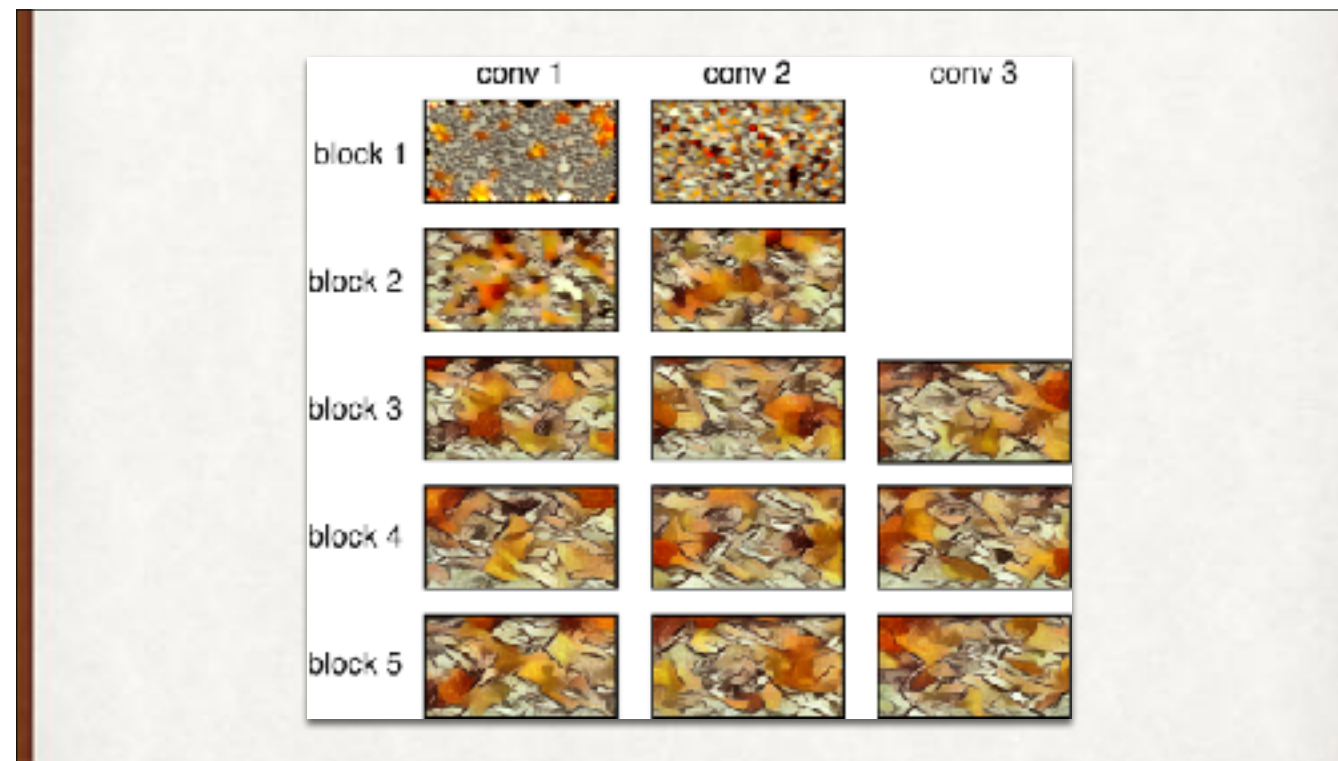
To do this, let's pick this very stylized 1907 self-portrait by Pablo Picasso. People sure looked weird back then.



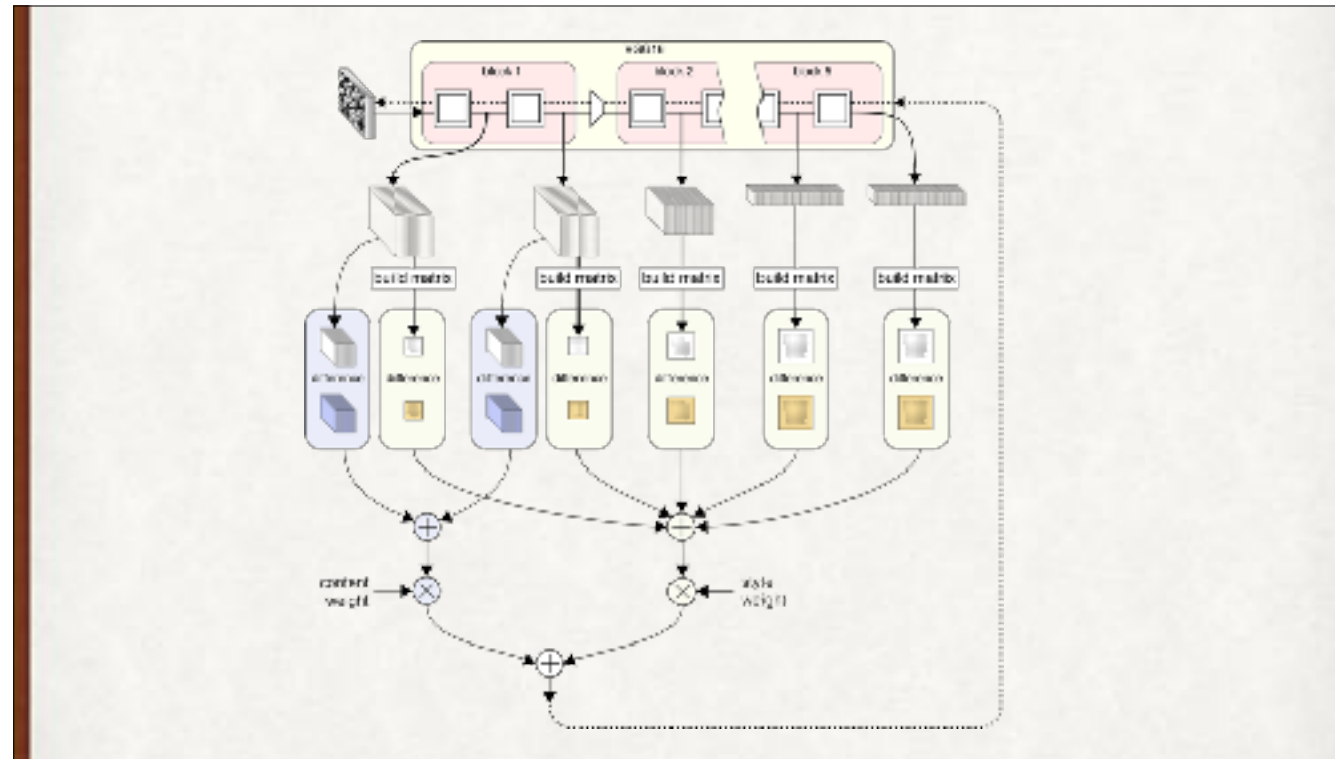
Run the painting through VGG16, get the output from each layer, and build the Gram matrix telling us where different filters activated in the same place. Save those matrices. Remember, these matrices tell us to what degree each pair of filters responded strongly in the same locations of their input.



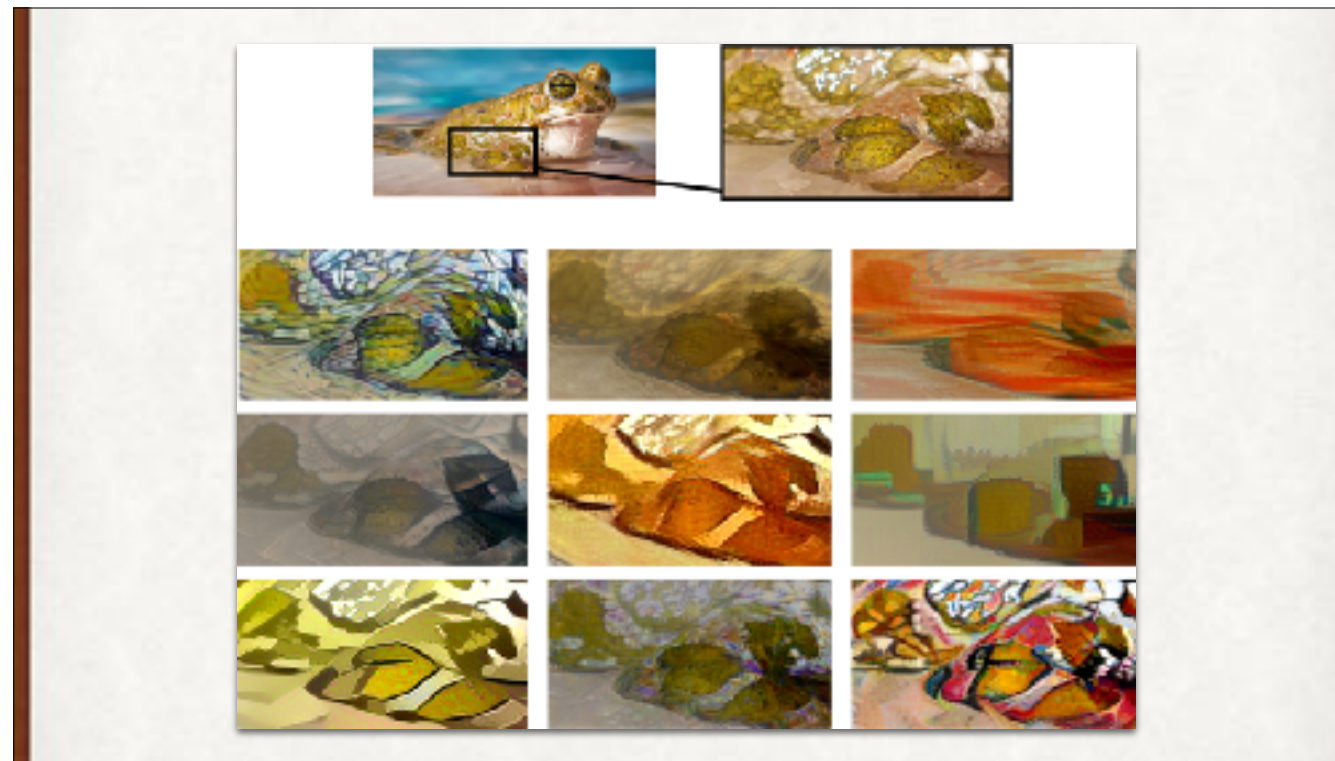
Now that we have the painting's Gram matrices saved, run noise through VGG16, build its Gram matrices, and find out how much they differ from the ones we saved. Add that all up to get the "Gram matrix loss." We'll see that this captures the "style" information we're looking for.



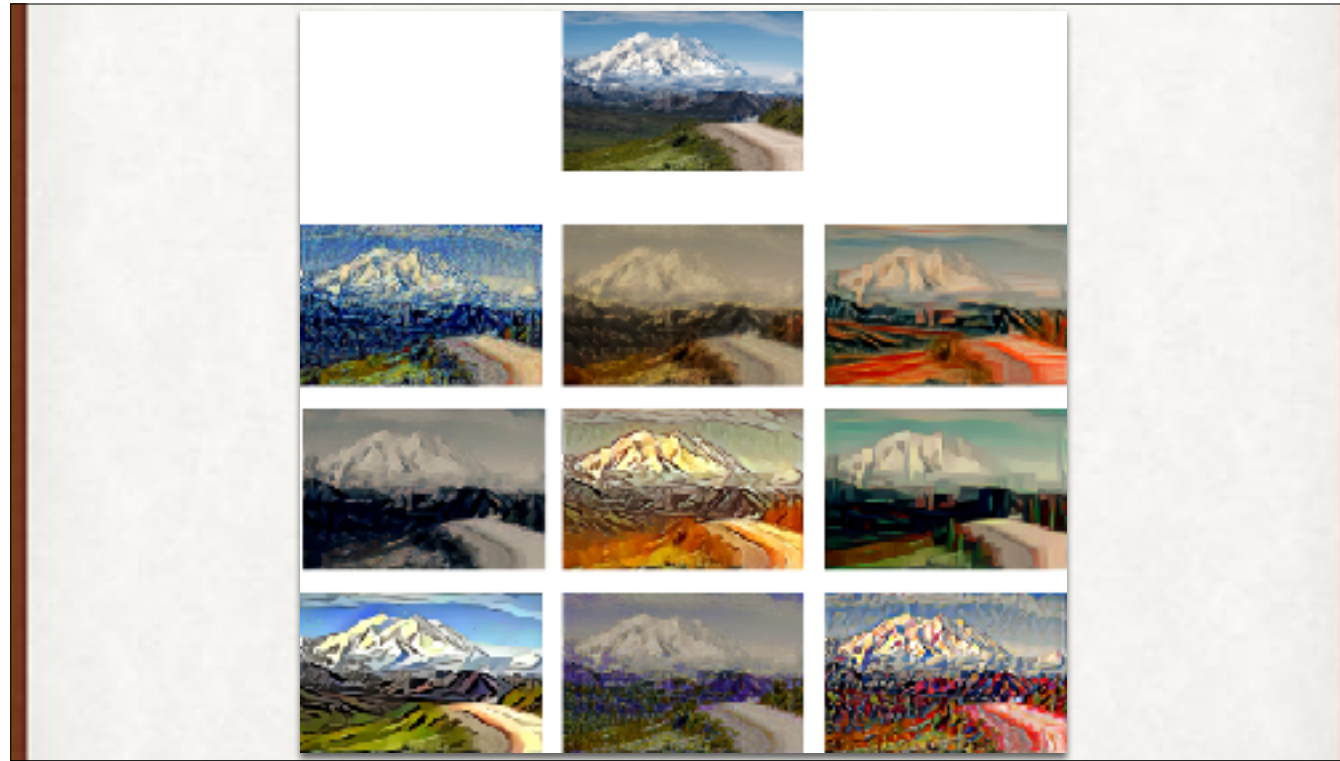
Let's instead use the sum of all the outputs from the start up to and including a given layer. Starting with noise, these outputs are looking a lot like the style used in the self portrait, even though there's no content information at all. They have the right colors, they're grouped into coherent shapes 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.



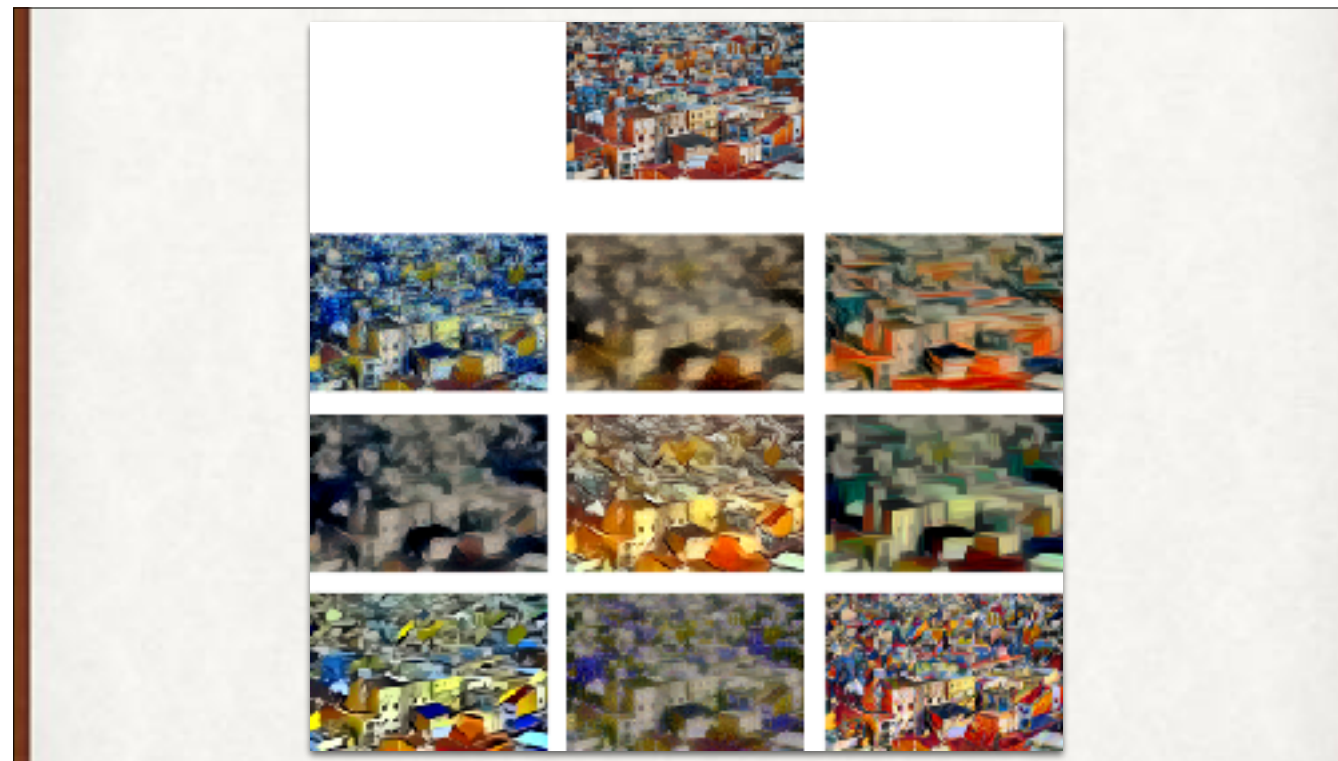
Here's the overall technique for style transfer. We start with a noisy image. We'll add the content and style losses together, after weighting them to give them the importances we desire. Here we're only measuring content loss from the first two layers. The summed loss is our error signal, which we'll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



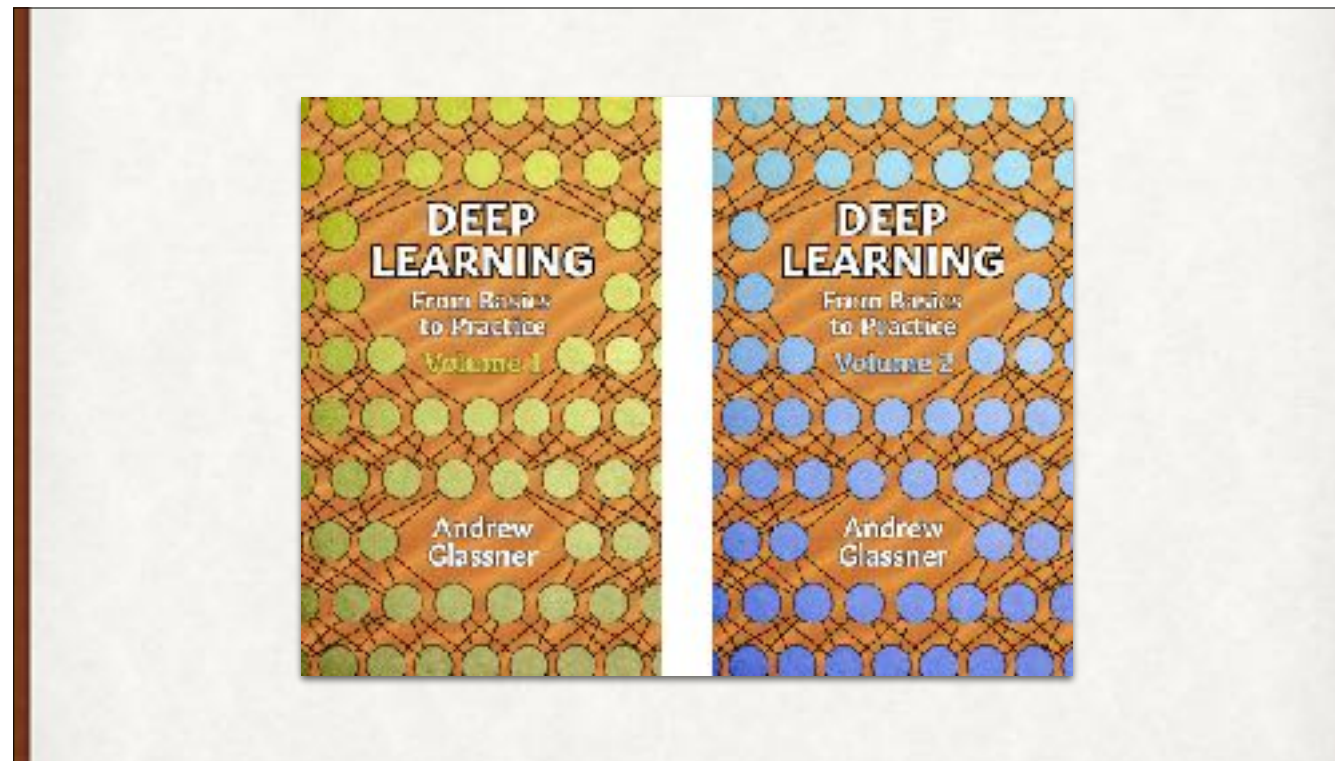
These are not just simple blends or color adjustments to the frog. They're really unique images. Here's a closeup of the near leg from each image.



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



And a photo of a town.



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**

 **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.