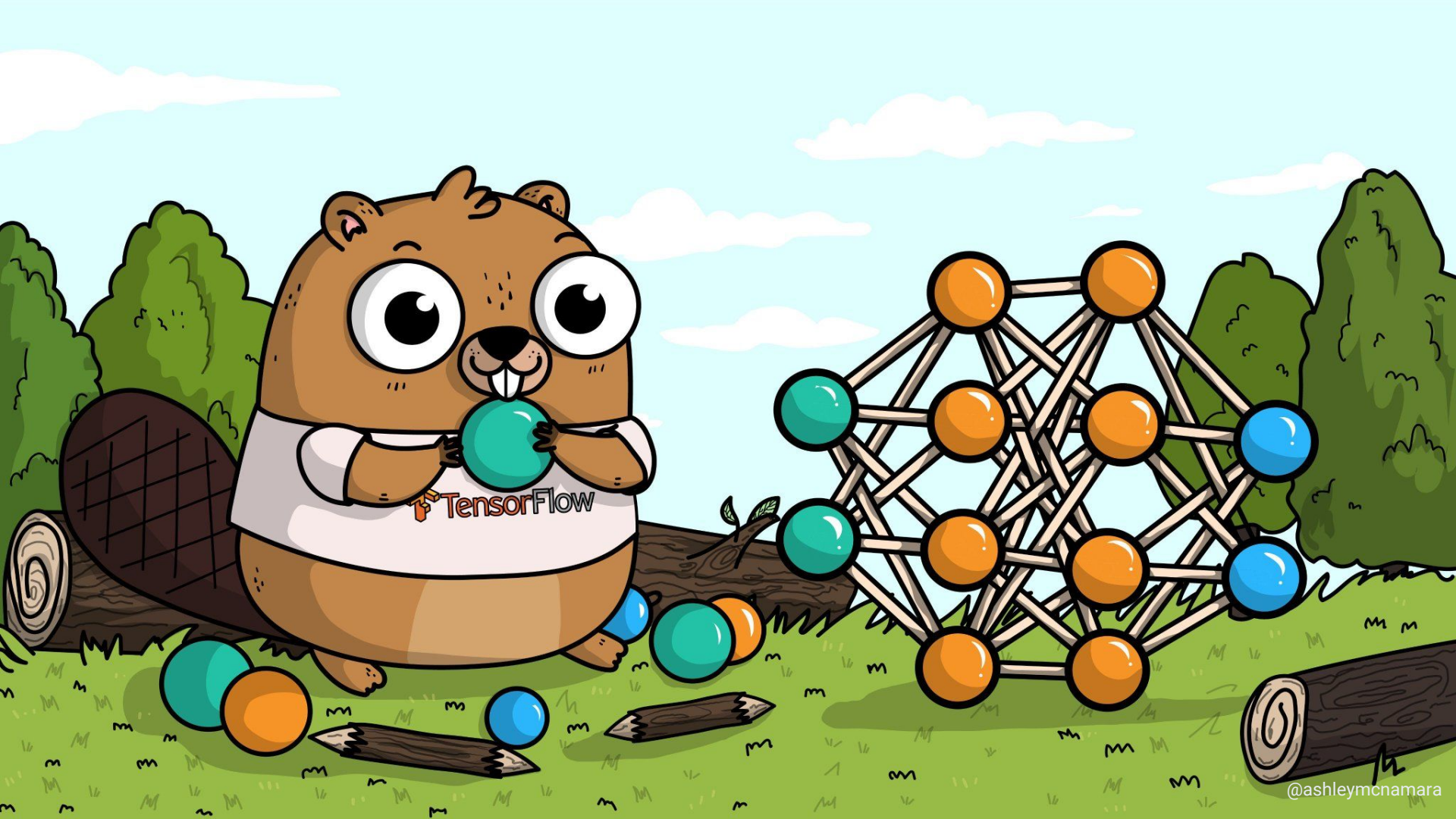

TensorFlow
@ashleymcnamara

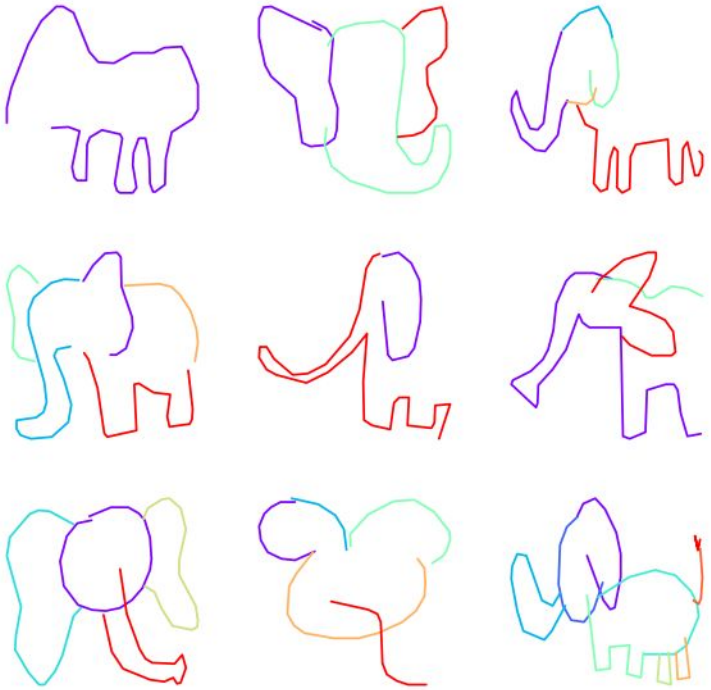
Applied Deep Learning

Lecture 3 • Feb 7th, 2019

Today

CNNs and transfer learning

- From scratch
- Reusing a pretrained model
- Fine-tuning
- Data augmentation
- Feature visualization



Administrative stuff

Will update syllabus tomorrow

- Just a minor change, swapping the order of these lectures

Likewise, will post assignment #2 tomorrow

- Hands on image classification + transfer learning

Code

- Backprop code upload to GitHub

CNNs

Convolution example

```
import scipy
from skimage import color, data
import matplotlib.pyplot as plt
img = data.astronaut()
img = color.rgb2gray(img)
plt.axis('off')
plt.imshow(img, cmap=plt.cm.gray)
```

BTW, I'm aware folk's from other departments may have a more sophisticated understanding of convolution. In ML - we basically mean sliding a filter across an image.

From hello-convolution.ipynb

Convolution example

Quick discussion: does anyone know who this is?

-1	-1	-1
-1	8	-1
-1	-1	-1

An edge detection filter. Intuition: dot product of the filter with a region of the image will be zero if all the pixels around the border have the same value as the center.

Convolution example

Eileen Collins

-1	-1	-1
-1	8	-1
-1	-1	-1

An edge detection filter. Intuition: dot product of the filter with a region of the image will be zero if all the pixels around the border have the same value as the center.

Launched the Chandra X-ray Observatory, in 1999

Convolution example

```
# simple edge detector
kernel = np.array([[-1,-1,-1],
                   [-1,8,-1],
                   [-1,-1,-1]])
result = scipy.signal.convolve2d(img, kernel, 'same')
plt.axis('off')
plt.imshow(result, cmap=plt.cm.gray)
```

From hello-convolution.ipynb

Convolution example

Eileen Collins

-1	-1	-1
-1	8	-1
-1	-1	-1

An edge detection filter. Intuition: dot product of the filter with a region of the image will be zero if all the pixels around the border have the same value as the center.

(May be easier to see with seismic)

Eileen Collins

-1	-1	-1
-1	8	-1
-1	-1	-1

An edge detection filter. Intuition: dot product of the filter with a region of the image will be zero if all the pixels around the border have the same value as the center.

Example

Of course, in a CNN these filters are learned

2	0	1	1
0	1	0	0
0	0	1	0
0	3	0	0

An input image (no padding)

1	0	1
0	0	0
0	1	0

A filter (3x3)

Output image (after convolving with stride 1)

Example

Of course, in a CNN these filters are learned

2	0	1	1
0	1	0	0
0	0	1	0
0	3	0	0

An input image (no padding)

1	0	1
0	0	0
0	1	0

A filter (3x3)

3	

Output image (after convolving with stride 1)

`np.tensordot(img[:3, :3], kernel)` or *2*1 + 0*0 + 1*1 + 0*0 + 1*0 + 0*0 + 0*0 + 0*1 + 1*0*

Example

Of course, in a CNN these filters are learned

2	0	1	1
0	1	0	0
0	0	1	0
0	3	0	0

An input image (no padding)

1	0	1
0	0	0
0	1	0

A filter (3x3)

3	2

Output image (after convolving with stride 1)

Example

Of course, in a CNN these filters are learned

2	0	1	1
0	1	0	0
0	0	1	0
0	3	0	0

An input image (no padding)

1	0	1
0	0	0
0	1	0

A filter (3x3)

3	2
3	

Output image (after convolving with stride 1)

Example

Of course, in a CNN these filters are learned

2	0	1	1
0	1	0	0
0	0	1	0
0	3	0	0

An input image (no padding)

1	0	1
0	0	0
0	1	0

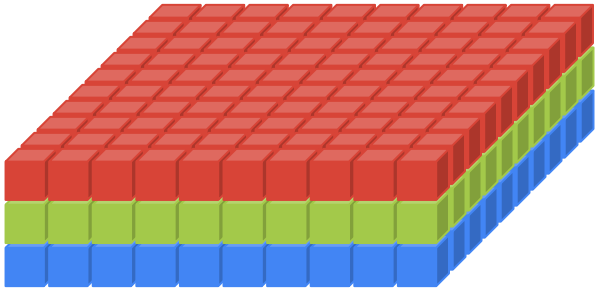
A filter (3x3)

3	2
3	1

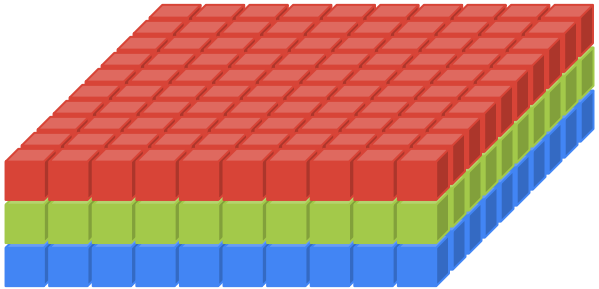
Output image (after convolving with stride 1)

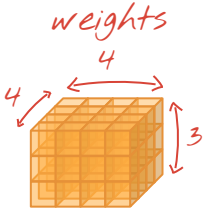
Now for 3d

- Several beautiful diagrams to follow courtesy of my colleague Martin Gorner (thank you!)

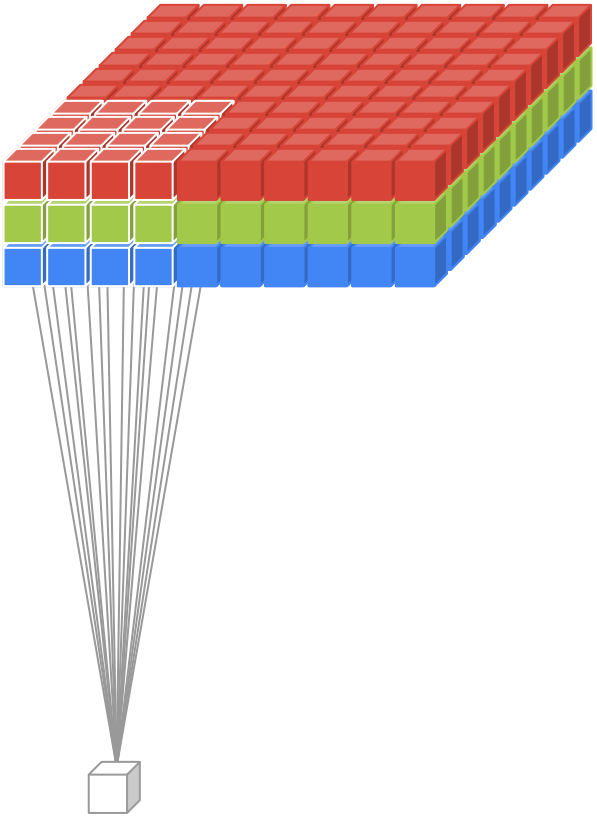


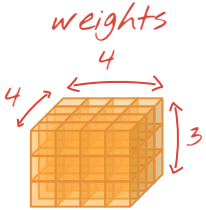
A RGB image as a 3d volume. Each color (or channel) is a layer.



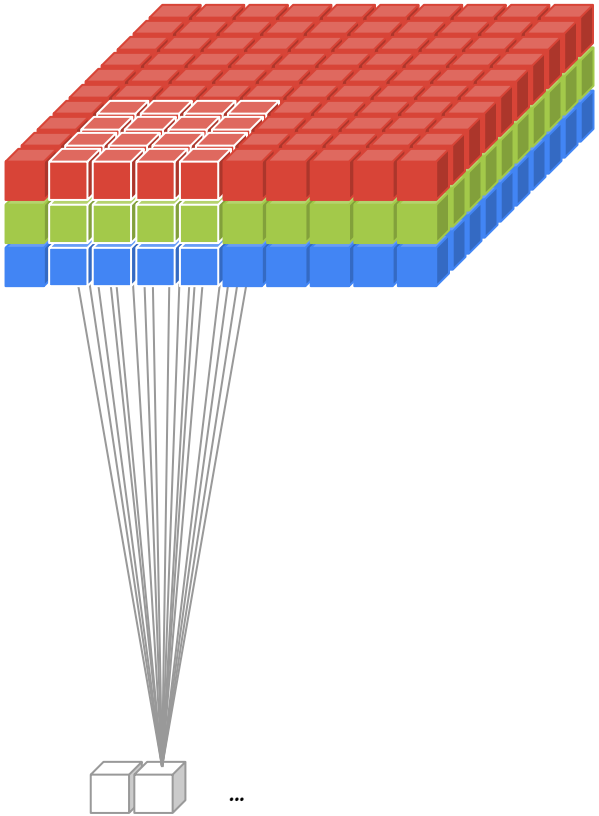


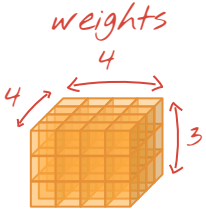
In 3d, our filters have width, height, and depth.



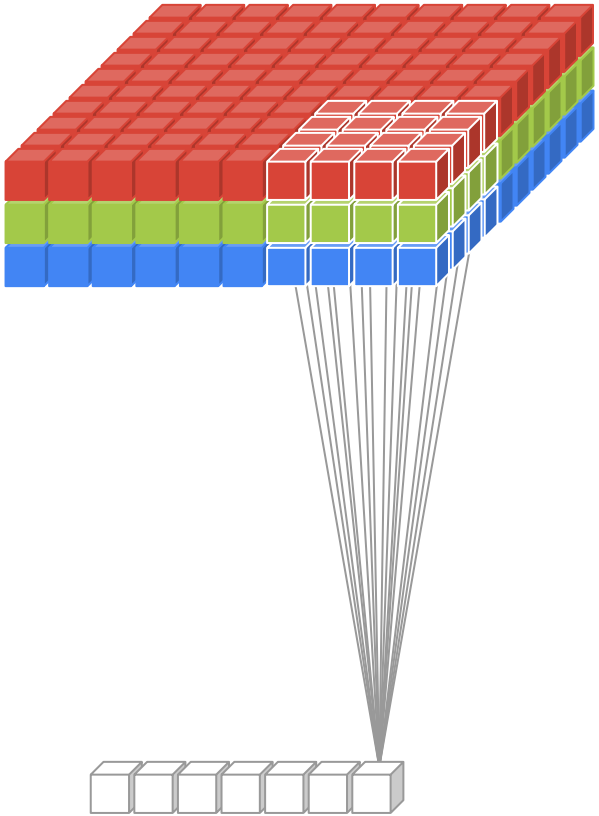


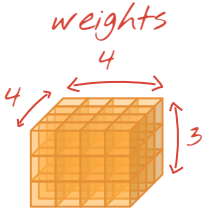
Applied in the same way as 2d (sum of weight * pixel value as they slide across the image).



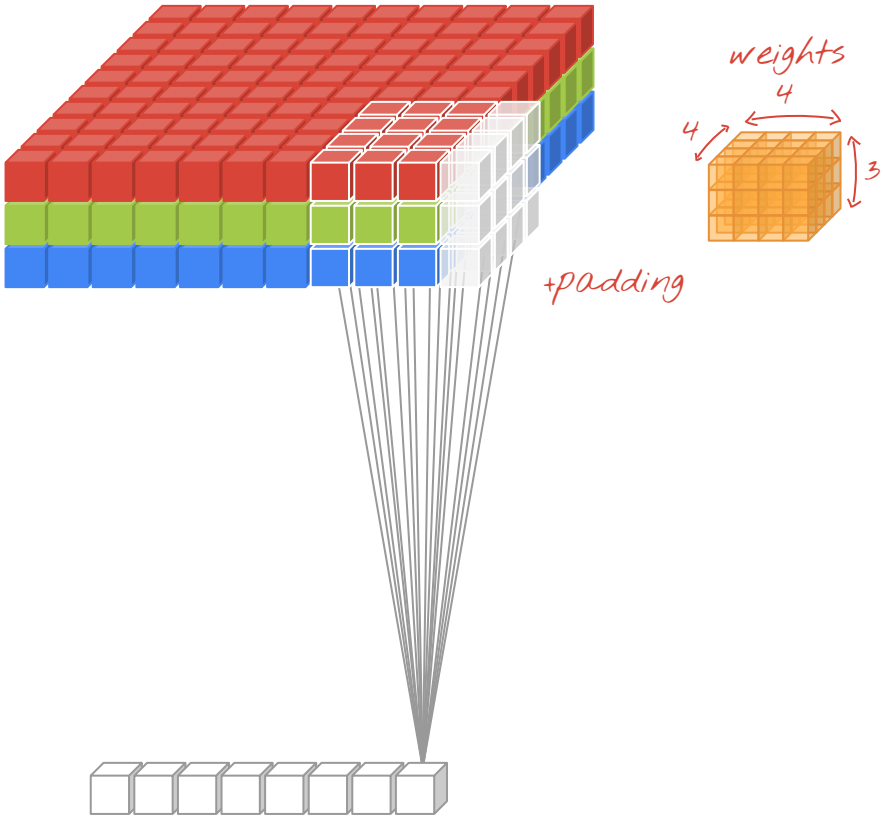


Applied in the same way as 2d (sum of weight * pixel value as they slide across the image).

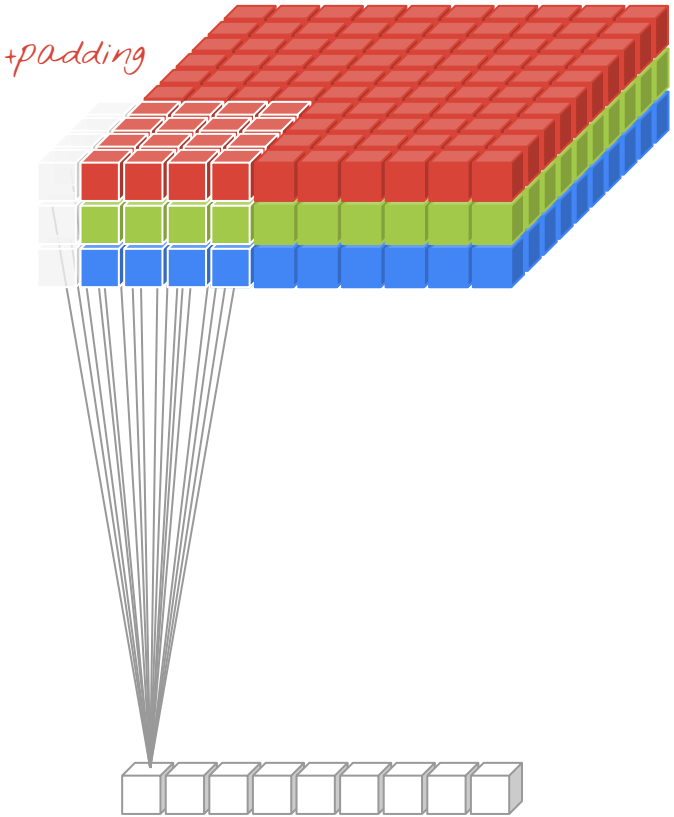


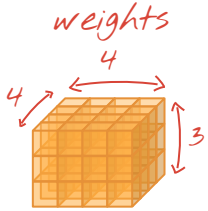


Notice the width of the output volume is smaller.

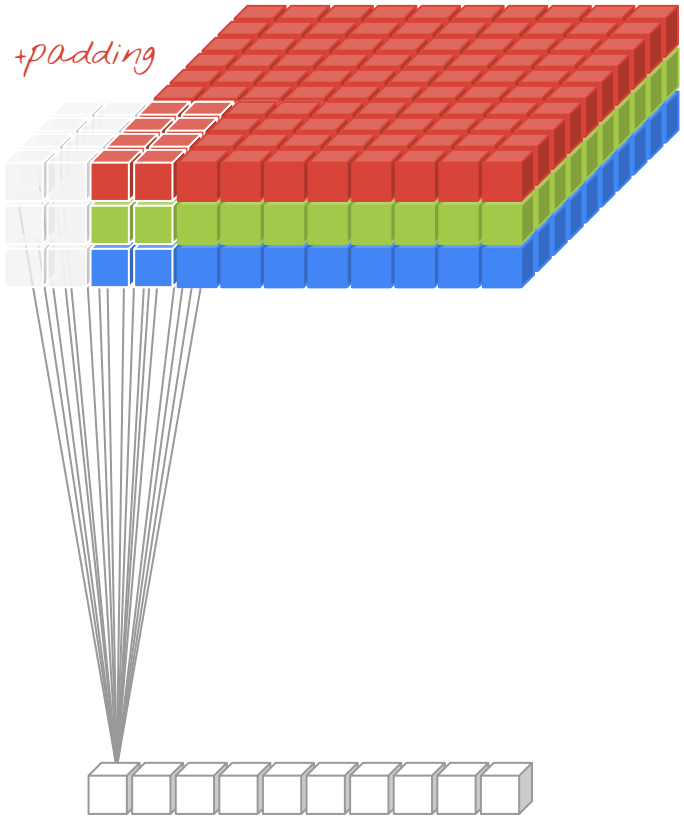


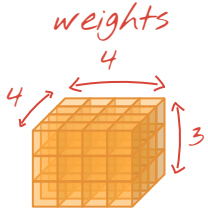
We can applying padding (zeros along the borders) to the original image to maintain the same output dimensions.



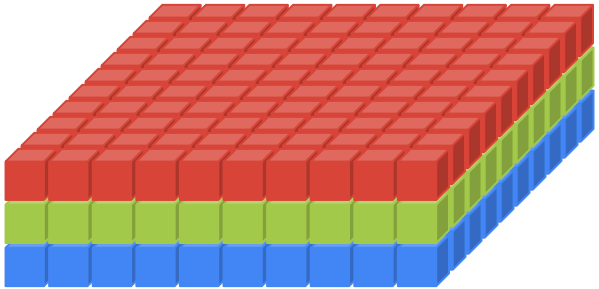


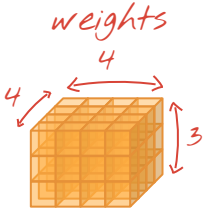
Adding more padding so the dimensions match (of course do this before beginning the convolution, most libraries offer a helper fn, more on that shortly).



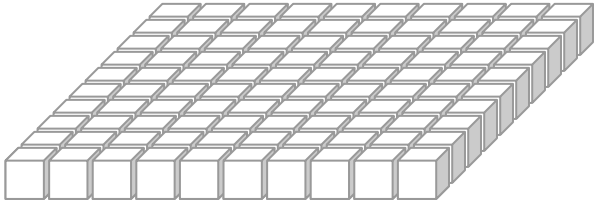


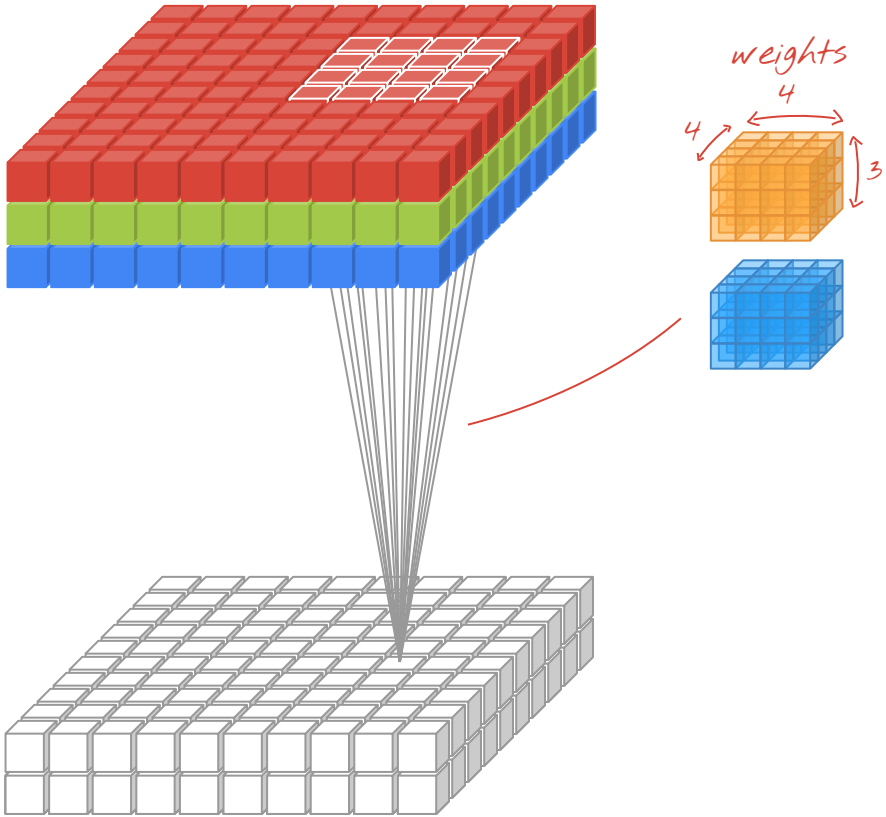
Adding more padding so the dimensions match (of course do this before beginning the convolution, most libraries offer a helper fn, more on that shortly).



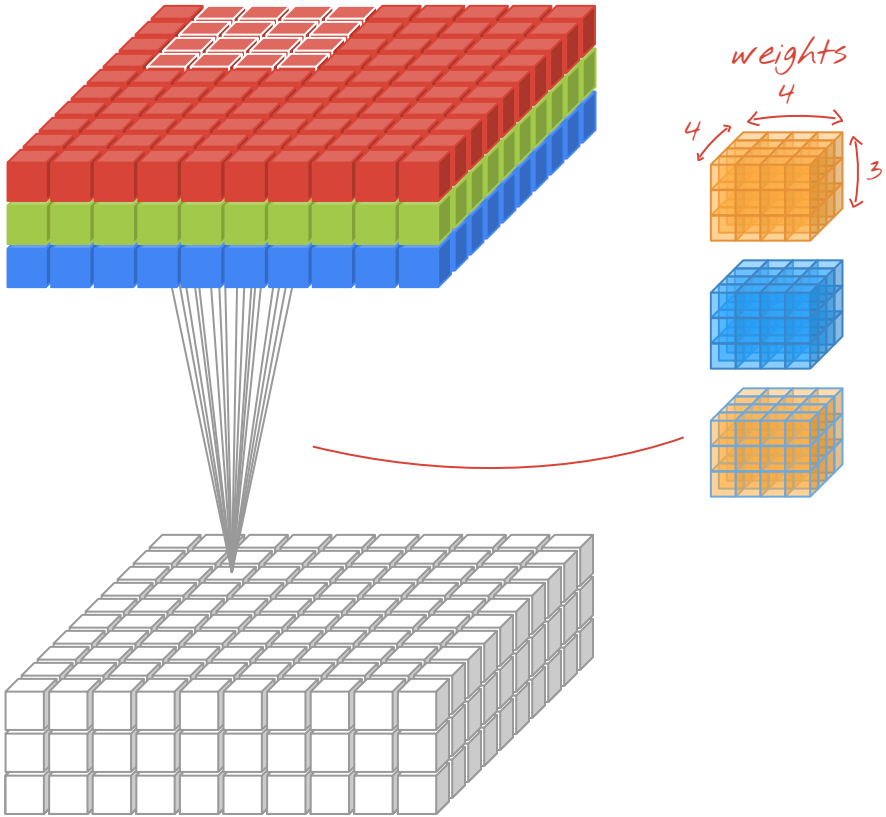


Applying the convolution over the rest of the input image.

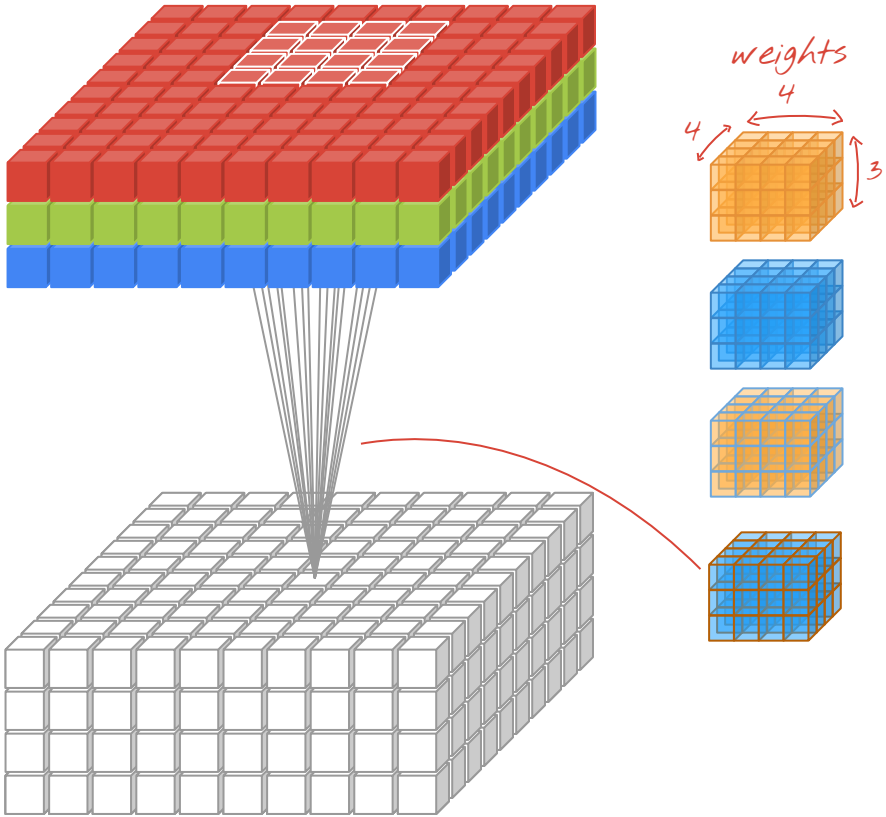




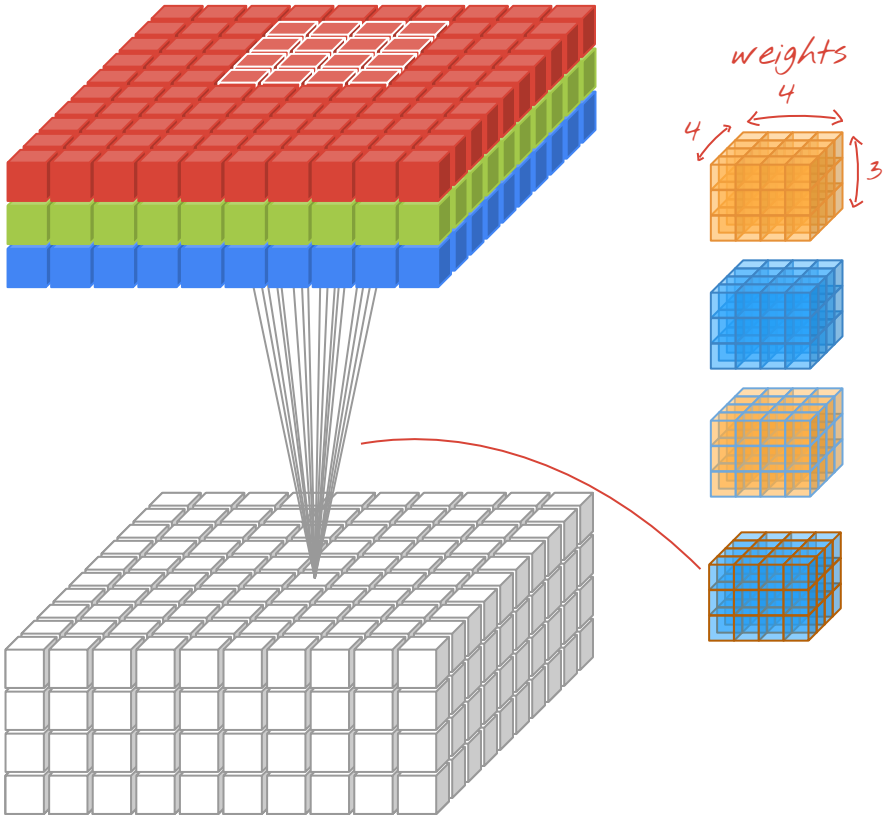
More filters, more output channels.



More filters, more output channels.



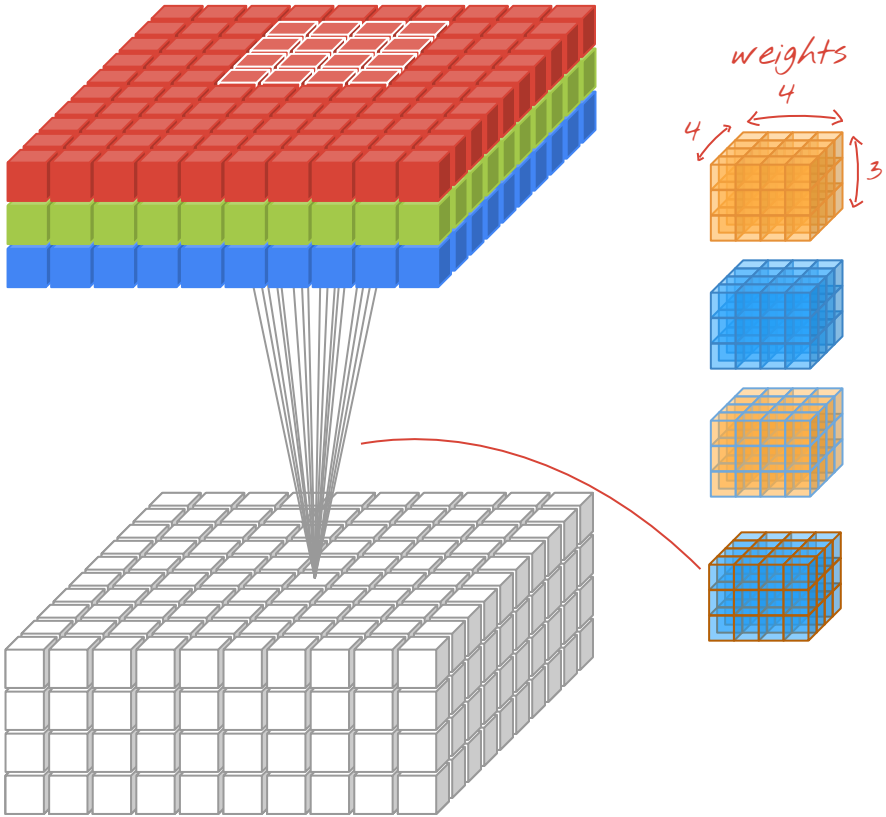
More filters, more output channels.



More filters, more output channels.

Notes:

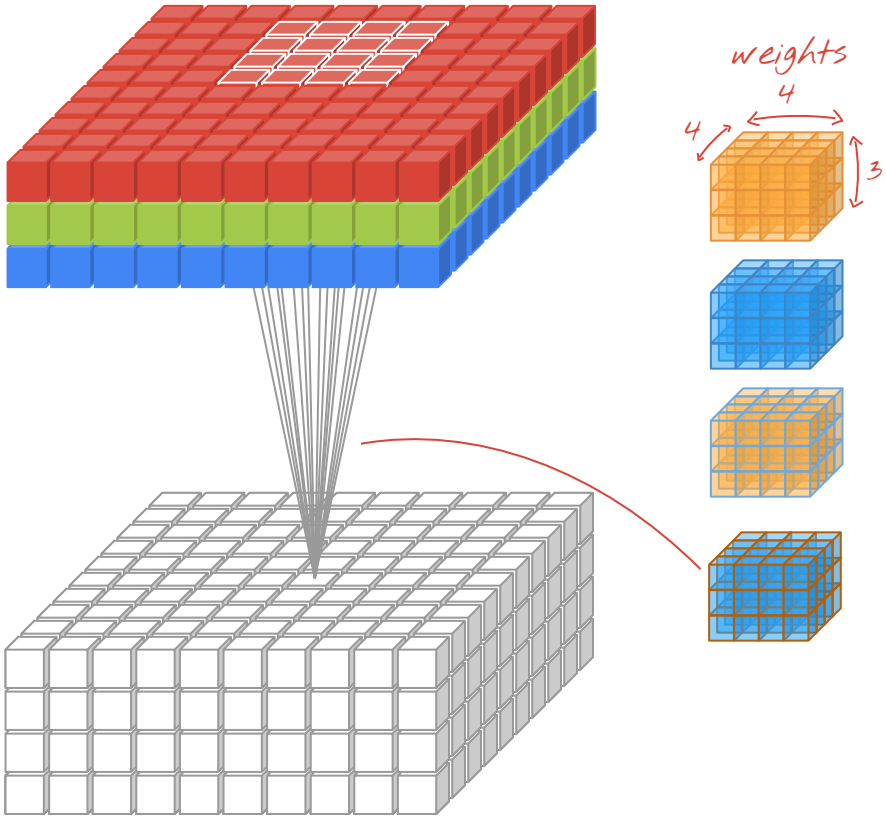
- Each filter learns one set of weights (it does not change between steps as it slides across the image)
- At this stage, we have four output channels. We can begin another layer of convolution using these as our input! The new bank of filters would each need depth 4.



More filters, more output channels.

Notes:

- Unlike the 2d filters from before, notice each filter connects to *every* input channel.
- This means they can compute sophisticated features. Initially, by looking at R,G,B - but later, by looking at combinations of learned features - like various edges, and later shapes / textures / and semantic features!

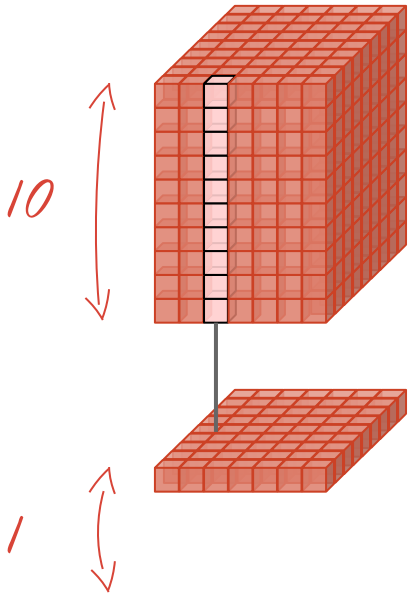


More filters, more output channels.

Notes:

- An output channel is also called a "feature map". It encodes the presence or absence (and degree of presence) of the feature it detects.
- CNNs are somewhat resistant to translation (an image shifted a bit will have a similar activation map, unless of course parts of it fall "off screen").

1d convolutions



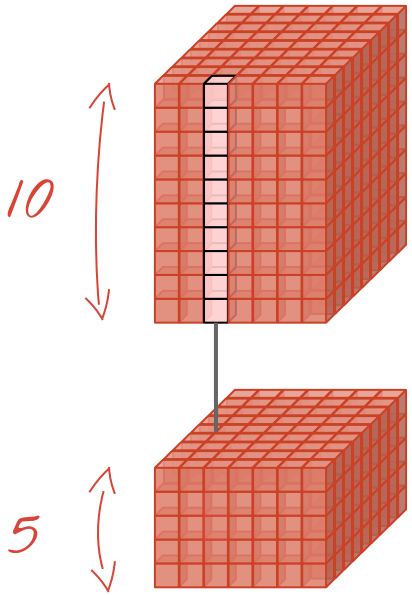
Weights:

1x1x10

Why 1x1 convolutions?

- **Efficiency**: reduces the depth (number of channels). Width and height are unchanged. To reduce the horizontal dimensions, you would use pooling (or increase the stride of the conv).
- The 1x1 conv computes a weighted sum of input channels (or features). This allows it to "**select**" certain combinations of features that are useful downstream.

1d convolutions



Weights:

1x1x10x5

Why 1x1 convolutions?

- **Efficiency**: reduces the depth (number of channels). Width and height are unchanged. To reduce the horizontal dimensions, you would use pooling (or increase the stride of the conv).
- The 1x1 conv computes a weighted sum of input channels (or features). This allows it to "**select**" certain combinations of features that are useful downstream.

Of course, you'll use a bank of these as well.

Max pooling

Destructive: reduces image size by 75%.
Only max values in each region pass through.
Usually applied spatially (to reduce x,y dimensions of an image), can also be applied depthwise (to reduce the number of channels).

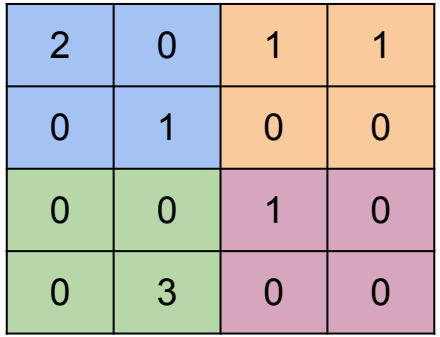
2	0	1	1
0	1	0	0
0	0	1	0
0	3	0	0

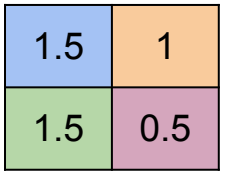
Max pooling with a 2x2 window and stride 2

2	1
3	1

Average pooling

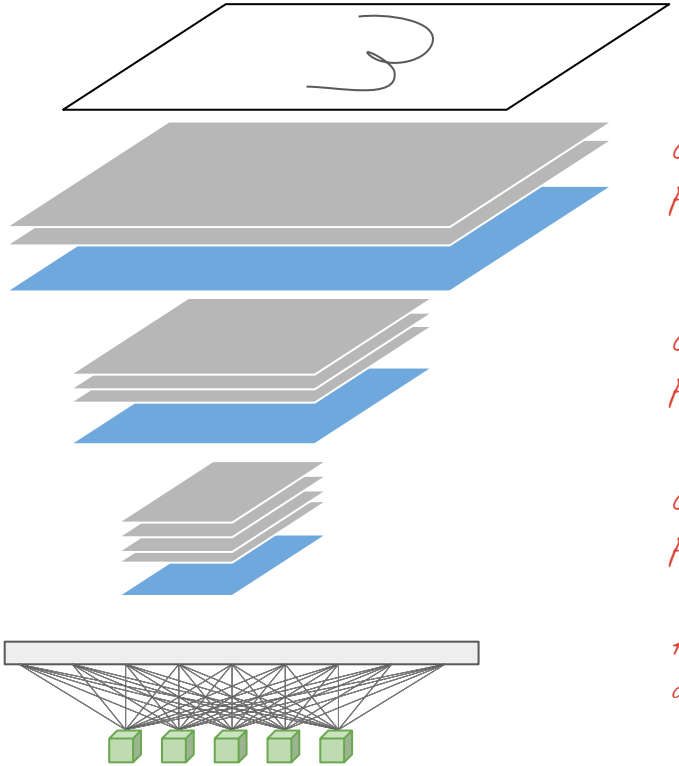
Same as previous slide, except average the windows.





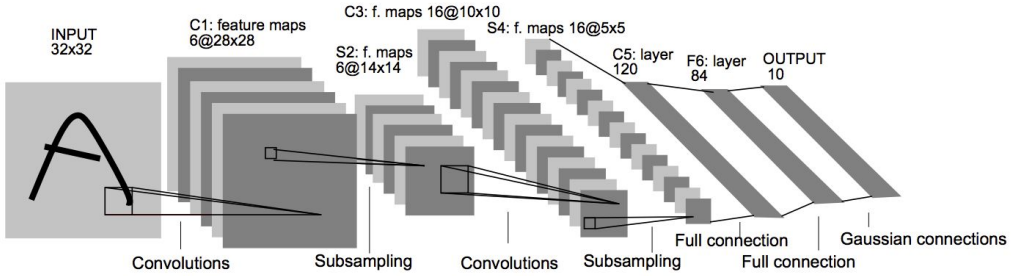
Common setup

- One or more stacks of conv / pool layers with relu activation, followed by a flatten then one or two dense layers.
- As we move through the network, feature maps become **smaller spatially**, and **increase in depth**.
- Features become increasingly abstract (but lose spatial information). E.g., "the image contained a eye, but not sure where it was").



LeNet-5 (1998)

This pattern (convolution, pool, convolution, pool, etc) became standard.



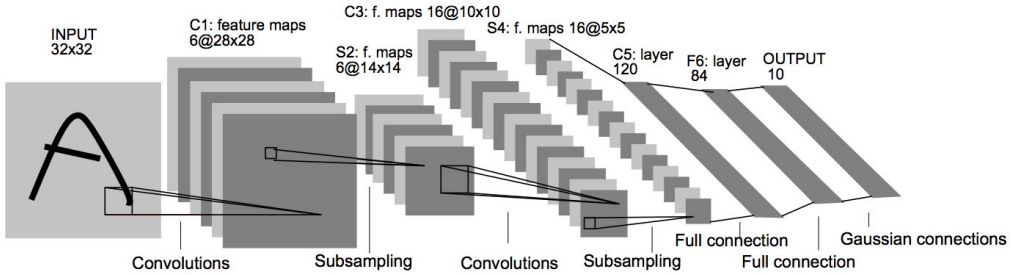
Introduced MNIST

Quick discussion: Why not just use Dense layers to recognize images?

http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf

LeNet-5 (1998)

This pattern (convolution, pool, convolution, pool, etc) became standard.



Introduced MNIST

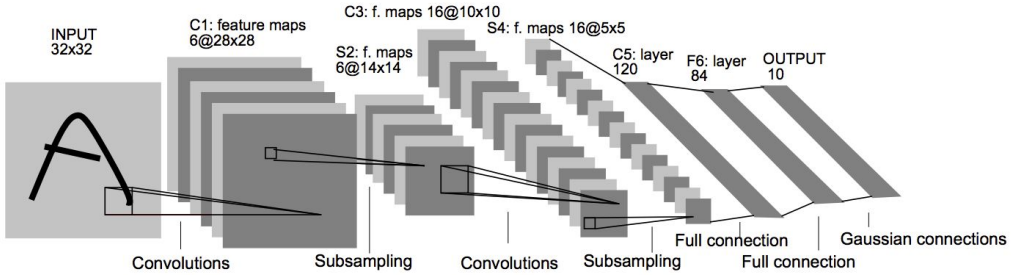
Quick discussion: Why not just use Dense layers to recognize images?

1. **Efficiency**: Assume input image is 150x150x3 = 67500 pixels. Say 1K neurons in first layer = 67.5M parameters (just in layer 1!)

http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf

LeNet-5 (1998)

This pattern (convolution, pool, convolution, pool, etc) became standard.



Introduced MNIST

Quick discussion: Why not just use Dense layers to recognize images?

1. **Efficiency**: Assume input image is 150x150x3 = 67500 pixels. Say 1K neurons in first layer = 67.5M parameters (just in layer 1!)

2. **Features must be detected separately at all locations**: Say detecting an edge is important. A Dense layer must learn to detect that edge at all positions (separately).

http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf

Example

A typical CNN in Keras

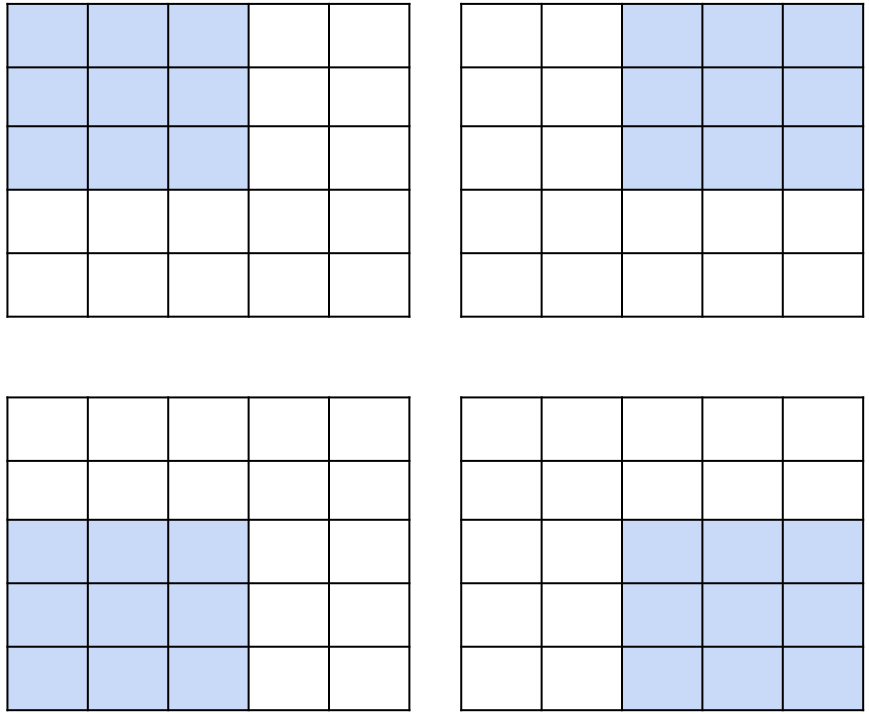
```
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=(28, 28, 1),
                 padding='same'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu',padding='same'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dense(10, activation='softmax'))
```

rows, cols, color channels

A good place to start

Stride

- Stride (aka step size as the filter slides across the image).
- Using numbers >1 will downsize the image.

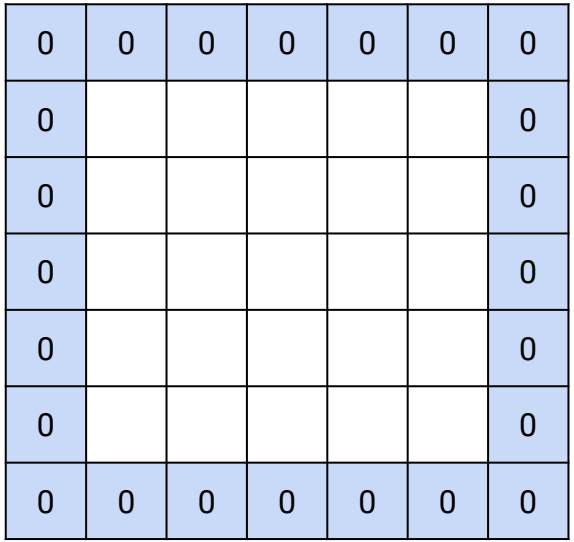


A 3x3 filter sliding over a 5x5 input with stride 2. Here, this results in a 2x2 output.

Padding

- **Valid**: no padding (only valid locations will be used in the conv)

- **Same**: (library adds zero padding so the output has the same width and height as the input)



A 3x3 filter sliding over a 5x5 input with "same" padding produces a 5x5 output.

keras.io/layers/convolutional/#conv2d

Example

A typical CNN in Keras

```
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=(28, 28, 1),
                 padding='same'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu',padding='same'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dense(10, activation='softmax'))
```

rows, cols, color channels

In transfer learning, referred to as the "base".

Referred to as the "top".

A good place to start

model.summary()

```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 28, 28, 32)        320
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 14, 14, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 14, 14, 64)        18496
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 7, 7, 64)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 3136)              0
_________________________________________________________________
dense_1 (Dense)              (None, 128)               401536
_________________________________________________________________
dense_2 (Dense)              (None, 10)                1290
=================================================================
Total params: 421,642
```

None refers to the batch size (determined at runtime).

model.summary()

```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 28, 28, 32)        320
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 14, 14, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 14, 14, 64)        18496
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 7, 7, 64)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 3136)              0
_________________________________________________________________
dense_1 (Dense)              (None, 128)               401536
_________________________________________________________________
dense_2 (Dense)              (None, 10)                1290
=================================================================
Total params: 421,642
```

Notice this Dense layer accounts for the majority of the weights

How many weights are in a dense layer?

flatten_1 (Flatten)	(None, 3136)	0
dense_1 (Dense)	(None, 128)	401536
dense_2 (Dense)	(None, 10)	1290

Fully connected, so...

input dimension * output dimension + one bias per output neuron

3136 * 128 + 128 = 401536

How many weights are in a dense layer?

flatten_1 (Flatten)	(None, 3136)	0
dense_1 (Dense)	(None, 128)	401536
dense_2 (Dense)	(None, 10)	1290

Fully connected, so...

input dimension * output dimension + one bias per output neuron

128 * 10 + 10 = 1290

How many weights are in a pooling layer?

conv2d_2 (Conv2D)	(None, 14, 14, 64)	18496
max_pooling2d_2 (MaxPooling2	(None, 7, 7, 64)	0
flatten_1 (Flatten)	(None, 3136)	0

Pooling layers have no parameters.

How many weights are in a conv layer?

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 28, 28, 32)	320
max_pooling2d_1 (MaxPooling2	(None, 14, 14, 32)	0
conv2d_2 (Conv2D)	(None, 14, 14, 64)	18496

Recall:

- Input shape: 28x28x1
- Filter size: 3x3

3*3 (filter size) * 1 (input depth) * 32 (# of filters) + 32 (bias, 1 per filter) = 320

How many weights are in a conv layer?

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 28, 28, 32)	320
max_pooling2d_1 (MaxPooling2	(None, 14, 14, 32)	0
conv2d_2 (Conv2D)	(None, 14, 14, 64)	18496

Recall:

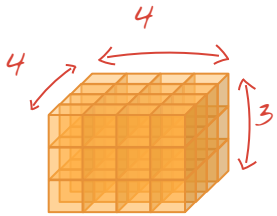
- Filter size: 3x3

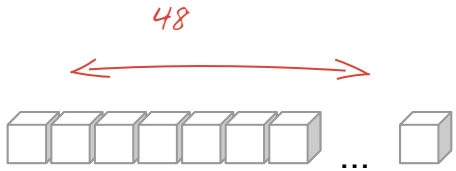
3*3 (filter size) * 32 (input depth) * 64 (# of filters) + 64 (bias, 1 per filter) = 18496

Flatten

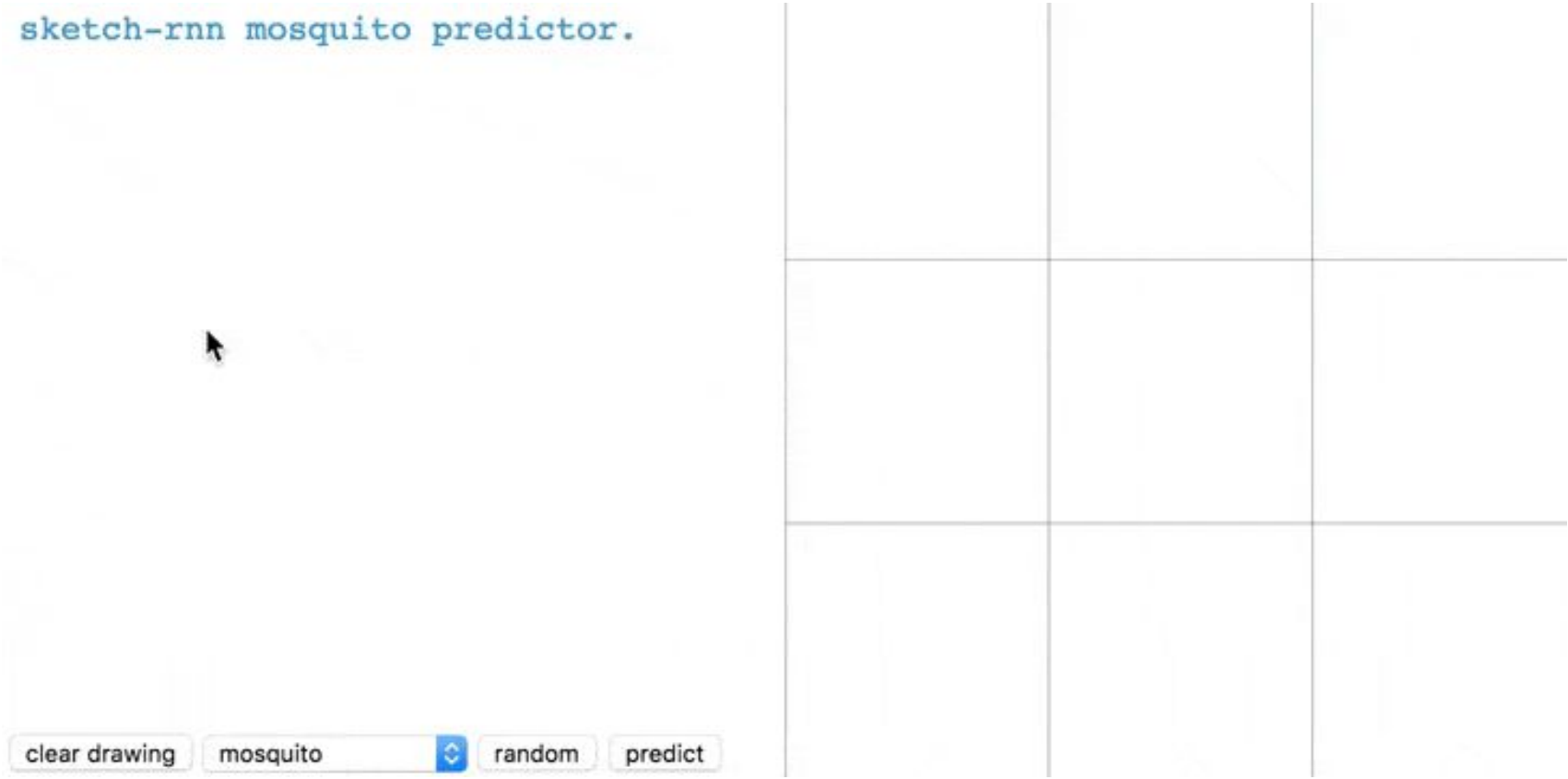
max_pooling2d_2 (MaxPooling2	(None, 7, 7, 64)	0
flatten_1 (Flatten)	(None, 3136)	0
dense_1 (Dense)	(None, 128)	401536

Just unstacks the volume above it into an array





QuickDraw: a fun dataset to play with


sketch-rnn mosquito predictor.
clear drawing
mosquito
random
predict

Sketch-RNN

Quick, Draw! The Data

Get the data Play the game

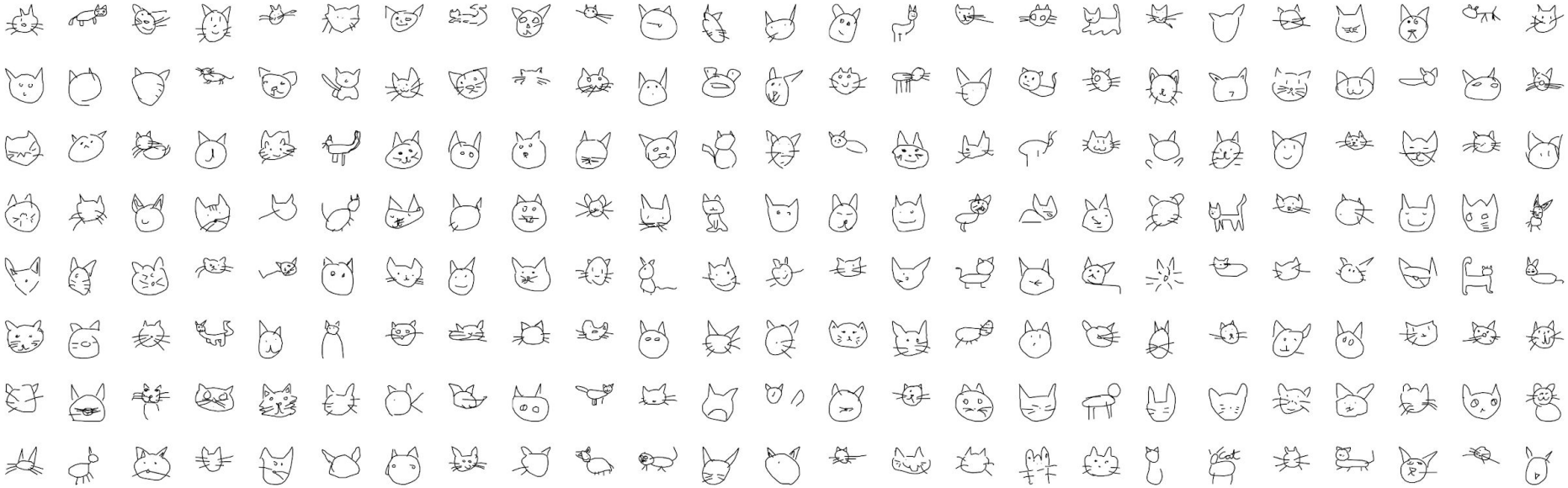
Now visualizing: cat

Randomize

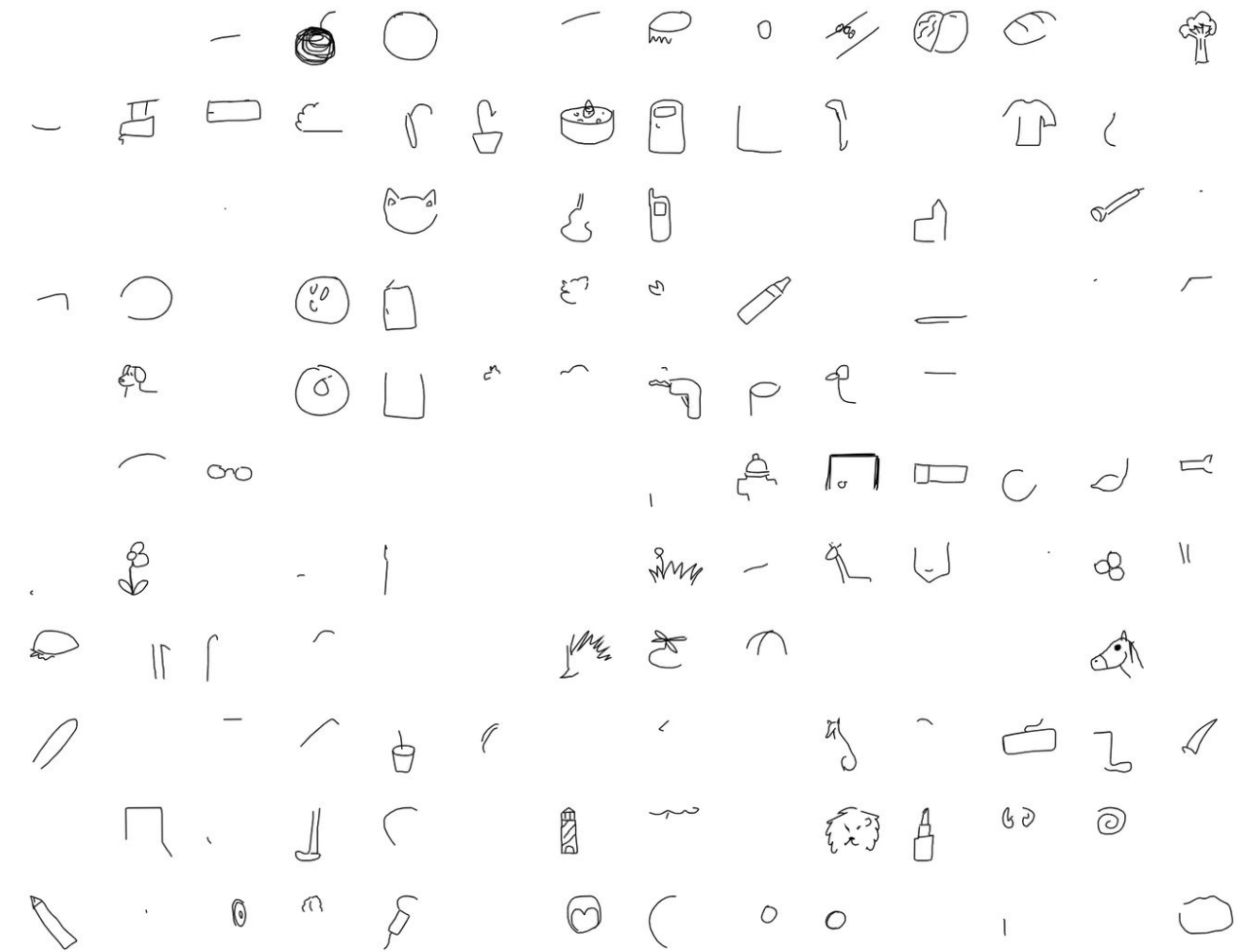
You are looking at 103,031 cat drawings made by real people... on the internet.

If you see something that shouldn't be here, simply select the drawing and click the flag icon.

It will help us make the collection better for everyone.

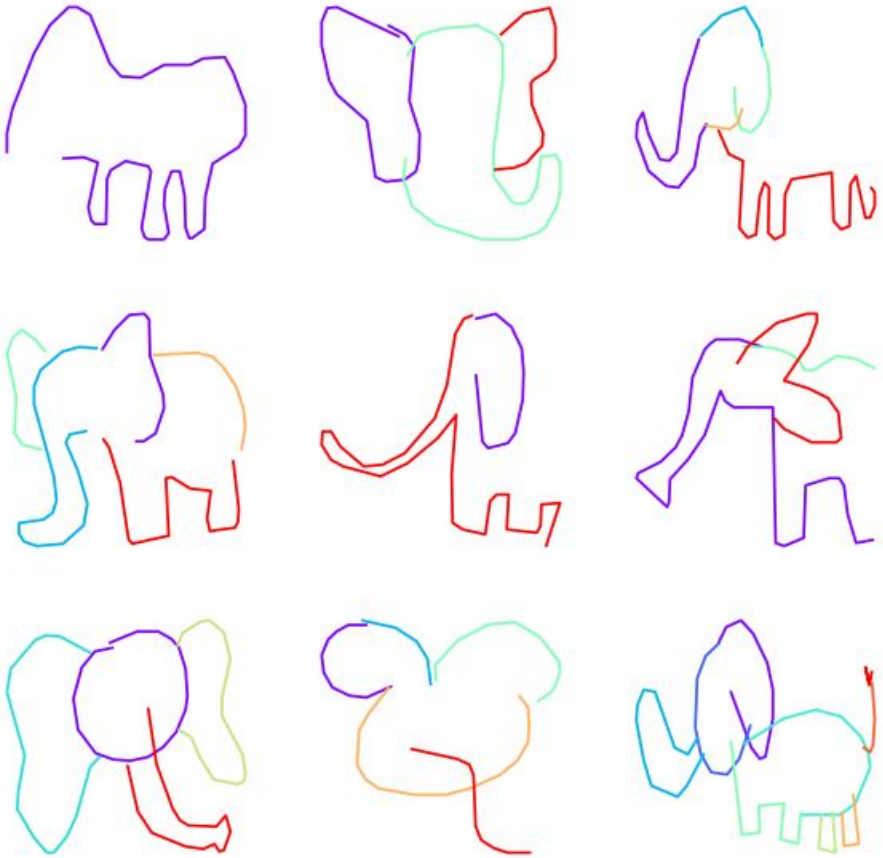


https://quickdraw.withgoogle.com/data/cat



Drawings are a sequence

- This dataset contains the path of every brush stroke.
- Can classify drawings either by treating them as images (using a **CNN** per usual), or by taking the sequence info into account (by using an **RNN**), or **both**!



- Each stroke is a unique color.
- Produced by the loader on CourseWorks.

A loader for this dataset

- Uploaded to GitHub
- You can use this to create variable size datasets (of N classes, with M instances per class).

A few well known models

Note

Many of these were trained on ImageNet for a week or more, with the fastest hardware available at the time. Several could now be trained in less than a day with a cluster of modern GPUs.

AlexNet

Key ideas

- Brought CNN's into mainstream.
- **ReLU** activation
- Won the 2012 ILSVRC by a large margin (17% top-5 error vs 26% for closest competitor)
- Similar idea to LeNet-5, though **much larger** / deeper (**60M vs 60K** parameters; 9 vs. 6 layers)
- Used data augmentation; Dropout

AlexNet

AlexNet

This result feels as interesting as the high-accuracy on ImageNet!

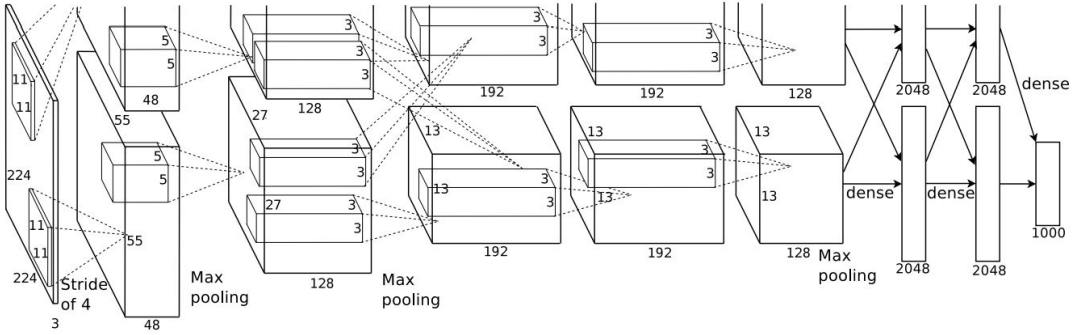


Figure 2: An illustration of the architecture of our CNN, explicitly showing the delineation of responsibilities between the two GPUs. One GPU runs the layer-parts at the top of the figure while the other runs the layer-parts at the bottom. The GPUs communicate only at certain layers. The network's input is 150,528-dimensional, and the number of neurons in the network's remaining layers is given by 253,440–186,624–64,896–64,896–43,264–4096–4096–1000.

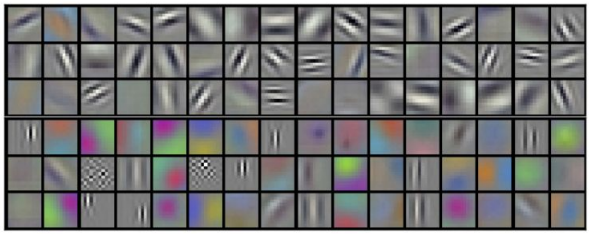
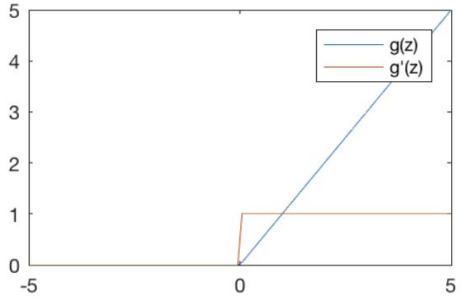


Figure 3: 96 convolutional kernels of size $11 \times 11 \times 3$ learned by the first convolutional layer on the $224 \times 224 \times 3$ input images. The top 48 kernels were learned on GPU 1 while the bottom 48 kernels were learned on GPU 2. See Section 6.1 for details.

AlexNet

ReLU activation (instead of tanh)

Rectified Linear Unit (ReLU)



$$g(z) = \max(0, z)$$

$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

Note: 6x increased training speed is **not** from 6x faster compute! The complexity of computing a conv is dominated by the many dot products, not by applying a pointwise activation function on the result.

Quick discussion: why does this help?

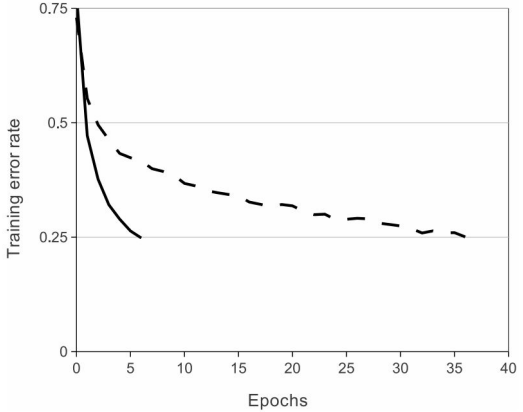


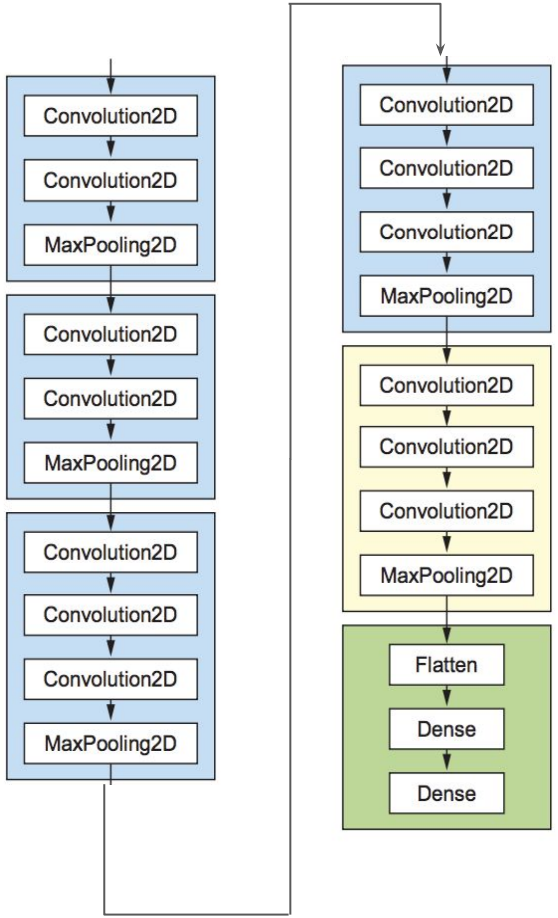
Figure 1: A four-layer convolutional neural network with ReLUs **(solid line)** reaches a 25% training error rate on CIFAR-10 six times faster than an equivalent network with tanh neurons **(dashed line)**. The learning rates for each network were chosen independently to make training as fast as possible. No regularization of any kind was employed. The magnitude of the effect demonstrated here varies with network architecture, but networks with ReLUs consistently learn several times faster than equivalents with saturating neurons.

AlexNet

VGG (2014)

Key ideas

- 2nd place in 2014
- Simple, clean architecture (but inefficient, with a huge number
- Became common to use in other experiments later on (e.g., sty



VGG

VGG-19

```
from keras.applications.vgg19 import VGG19
model = VGG19(weights='imagenet', include_top=True)
model.summary()
```

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 224, 224, 3)	0
block1_conv1 (Conv2D)	(None, 224, 224, 64)	1792
block1_conv2 (Conv2D)	(None, 224, 224, 64)	36928
block1_pool (MaxPooling2D)	(None, 112, 112, 64)	0
block2_conv1 (Conv2D)	(None, 112, 112, 128)	73856
block2_conv2 (Conv2D)	(None, 112, 112, 128)	147584
block2_pool (MaxPooling2D)	(None, 56, 56, 128)	0
block3_conv1 (Conv2D)	(None, 56, 56, 256)	295168
block3_conv2 (Conv2D)	(None, 56, 56, 256)	590080
block3_conv3 (Conv2D)	(None, 56, 56, 256)	590080
block3_conv4 (Conv2D)	(None, 56, 56, 256)	590080
block3_pool (MaxPooling2D)	(None, 28, 28, 256)	0
block4_conv1 (Conv2D)	(None, 28, 28, 512)	1180160
block4_conv2 (Conv2D)	(None, 28, 28, 512)	2359808
block4_conv3 (Conv2D)	(None, 28, 28, 512)	2359808
block4_conv4 (Conv2D)	(None, 28, 28, 512)	2359808
block4_pool (MaxPooling2D)	(None, 14, 14, 512)	0
block5_conv1 (Conv2D)	(None, 14, 14, 512)	2359808
block5_conv2 (Conv2D)	(None, 14, 14, 512)	2359808
block5_conv3 (Conv2D)	(None, 14, 14, 512)	2359808
block5_conv4 (Conv2D)	(None, 14, 14, 512)	2359808
block5_pool (MaxPooling2D)	(None, 7, 7, 512)	0
flatten (Flatten)	(None, 25088)	0
fc1 (Dense)	(None, 4096)	102764544
fc2 (Dense)	(None, 4096)	16781312
predictions (Dense)	(None, 1000)	4097000

Total params: 143,667,240
Trainable params: 143,667,240
Non-trainable params: 0

Notice ~80% of the parameters are in the last couple Dense connected layers...

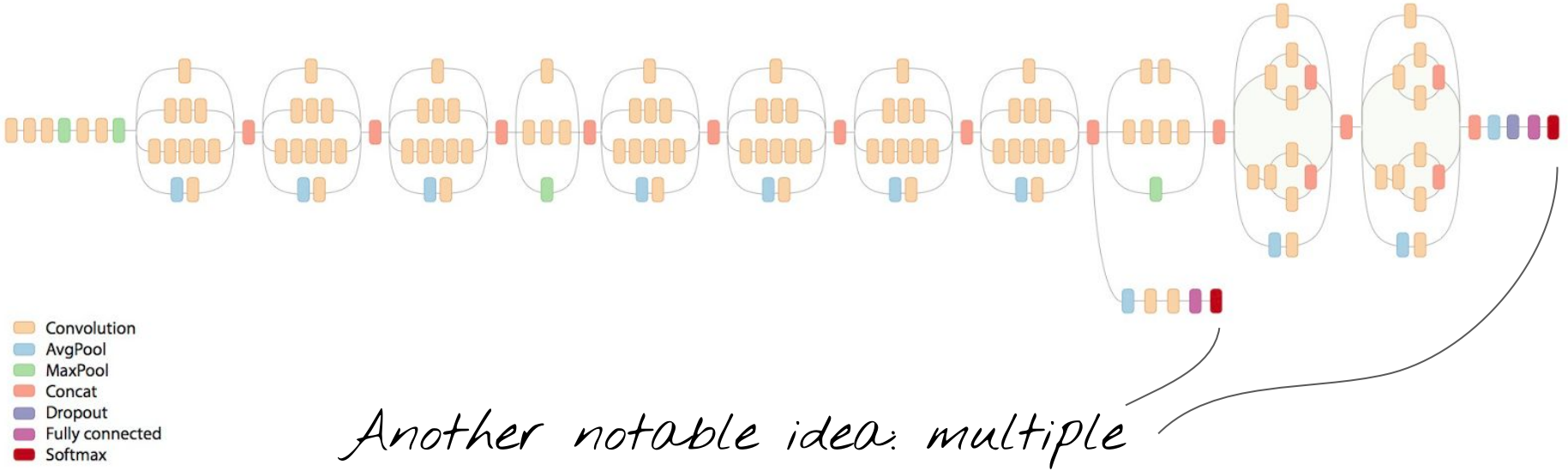
Inception (2015)

Key ideas

- Emphasized efficiency: **10x** fewer parameters than AlexNet (by eliminating the two large dense layers at the bottom).

- Inception module: Got the community thinking about designs beyond a simple stack of conv/pool layers.

GoogLeNet



Another notable idea: multiple outputs / loss functions in the same model.

https://ai.googleblog.com/2016/08/improving-inception-and-image.html

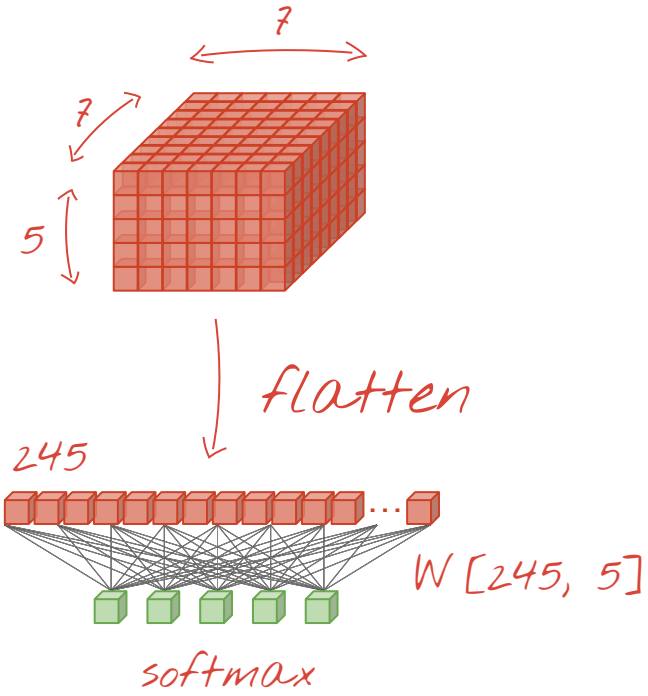
```
from keras.applications.inception_v3 import InceptionV3
model = InceptionV3(weights='imagenet', include_top=True)
model.summary()
```

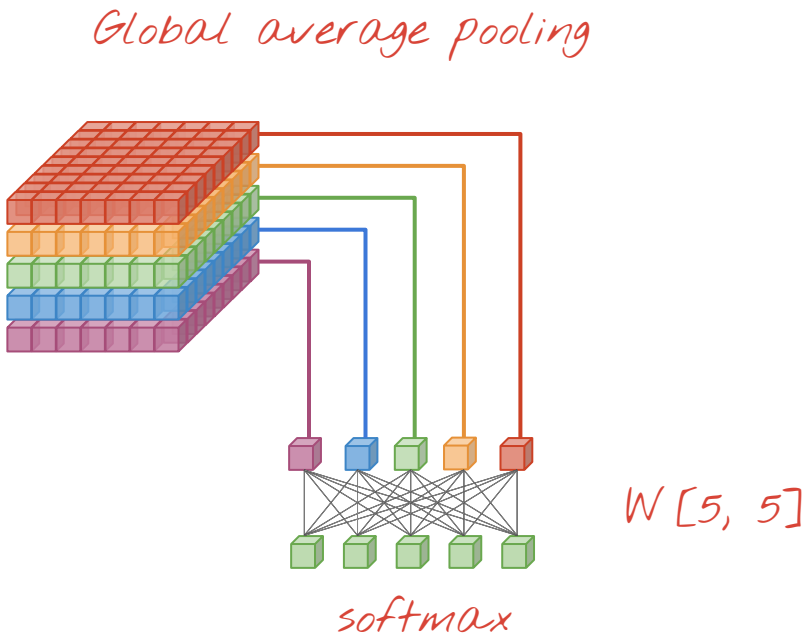
22 layers, then GlobalAveragePooling. What's this?

```
__________________________________________________________________________________________________
mixed10 (Concatenate)           (None, None, None, 2 0           activation_86[0][0]
                                                                 mixed9_1[0][0]
                                                                 concatenate_2[0][0]
                                                                 activation_94[0][0]
__________________________________________________________________________________________________
avg_pool (GlobalAveragePooling2 (None, 2048)         0           mixed10[0][0]
__________________________________________________________________________________________________
predictions (Dense)             (None, 1000)         2049000     avg_pool[0][0]
==================================================================================================
Total params: 23,851,784
Trainable params: 23,817,352
Non-trainable params: 34,432
__________________________________________________________________________________________________
```

Global average pooling

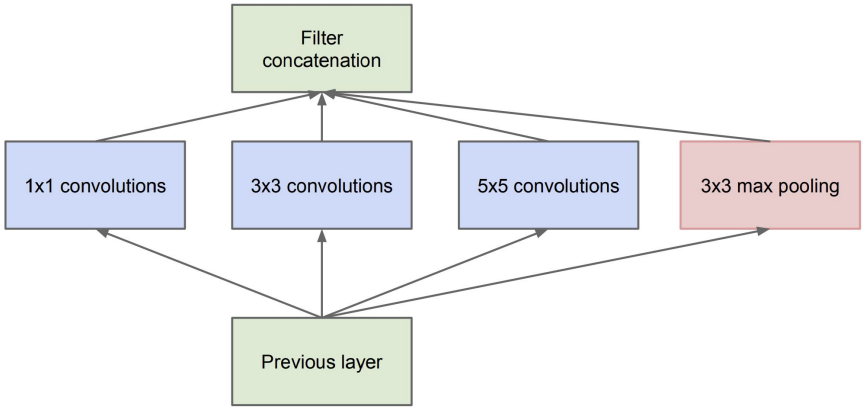
Although the last dense layers in CNNs contain most of the weights, they were found to be less helpful than thought in the Inception paper. Global average pooling prior to the dense layers greatly reduces the number of weights, without changing accuracy much.

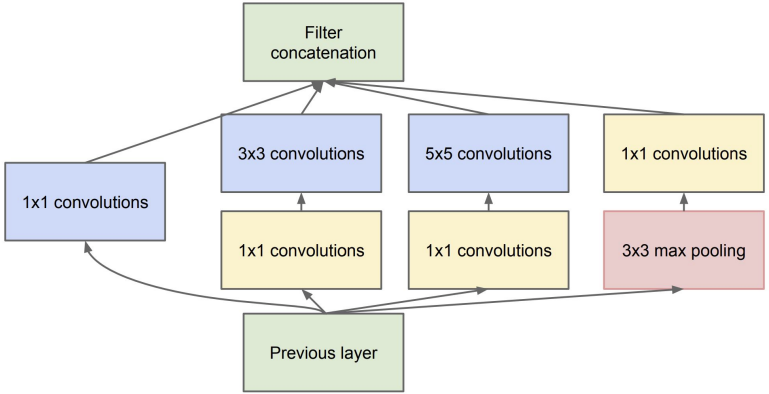




Inception module

Basic idea: instead of choosing which size of filter to use, run a few in parallel and let the network sort out which are useful. Results merged by stacking depthwise.





GoogLeNet

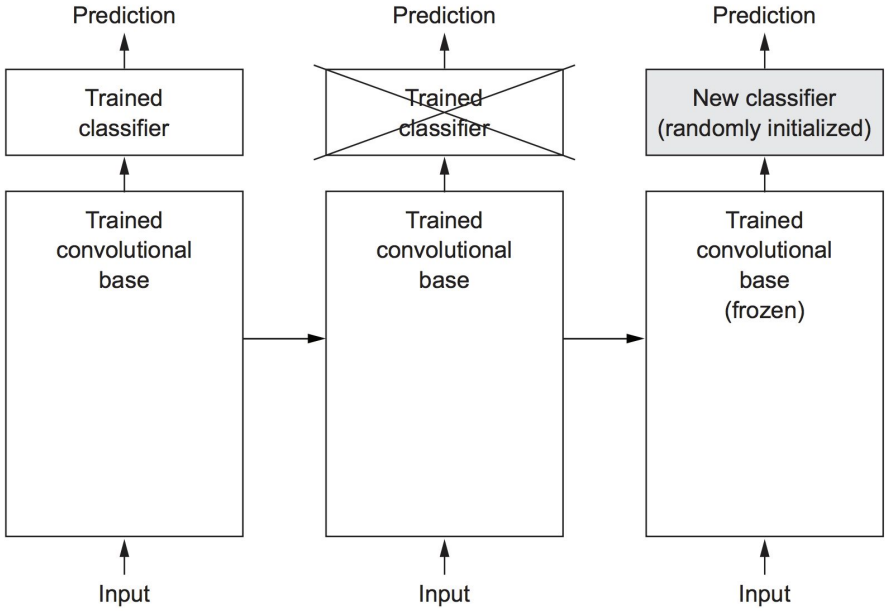
Left image: first idea (problem: output size too large).
Right image: using 1d convs to reduce size.

Transfer learning

Idea: knowledge (weights) learned on one task may be useful on another.

- *The base of a CNN learns a feature hierarchy (edges -> shapes -> textures -> .... -> semantic features (eye detectors, ear detectors, etc).*
- *Earlier features may generalize to other tasks (especially if trained on a large amount of data, say, ImageNet).*

I'd be curious to hear if you have experience with this in another domain!

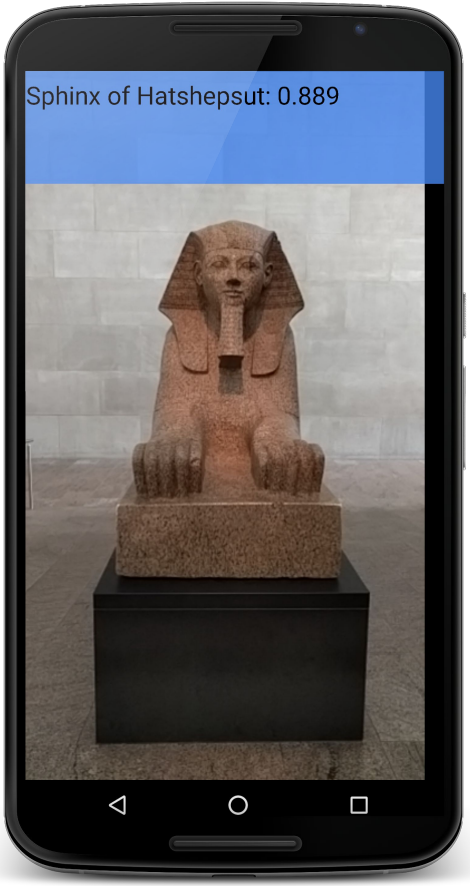


I have stolen this diagram from the book.


Monet - Sunflowers: 0.992

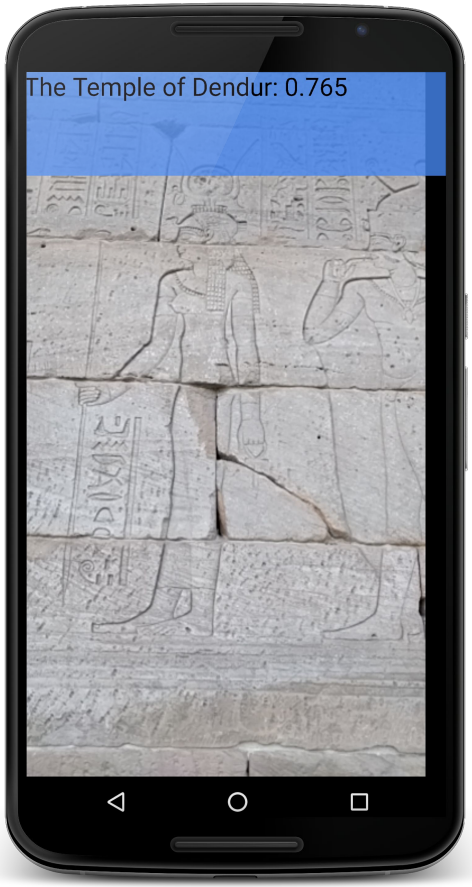
Train an accurate model with a small amount of data!

- Collected just **~50 photographs** (at various angles) of different paintings / sculptures in the Met (50 photos / each).
- Retrained Inception with weights from ImageNet. Deployed model on Android.
- Idea afterwards (not mine, it's a good one!): instead of taking a bunch of photos (labor intensive), instead - **take a video** - then slice into frames with FFmpeg. You could probably build a model in a few minutes this way!



Sphinx of Hatshepsut

Bernini - Autumn in the Guise of Priapus



The Temple of Dendur

Reuse an existing model

Deep Learning with Python covers these techniques well (certainly better than any tutorials we currently have on TensorFlow.org). Use that as a reference for assignment #3.

1. Retrain from scratch (same architecture, new weights).

   Pros: all layers well suited for your data; important if your data does *not* resemble ImageNet (say, in medical imaging). Cons: need a bunch of data (1M images or more).

2. Reuse the conv base (weights unchanged) as a feature extractor, train a new dense layer top.

   Pros: you only need a small amount of data! Works great if your images are similar to ImageNet. Cons: probably could be more accurate with a bit more work.

3. Also fine-tune the top couple conv layers.

   Idea: the top layers in the CNN are probably less relevant to your domain. After training your dense layer a bit, unfreeze these weights and continue training with a *low* learning rate (tricky, gradients on your untrained Dense layer are high!)

Models for image classification with weights trained on ImageNet

- Xception
- VGG16
- VGG19
- ResNet50
- InceptionV3
- InceptionResNetV2
- MobileNet
- DenseNet
- NASNet

You may need to run !pip install -U Keras to import these (there's a bug in an older version)

https://keras.io/applications/
github.com/tensorflow/models/tree/master/official/keras_application_models

ResNet

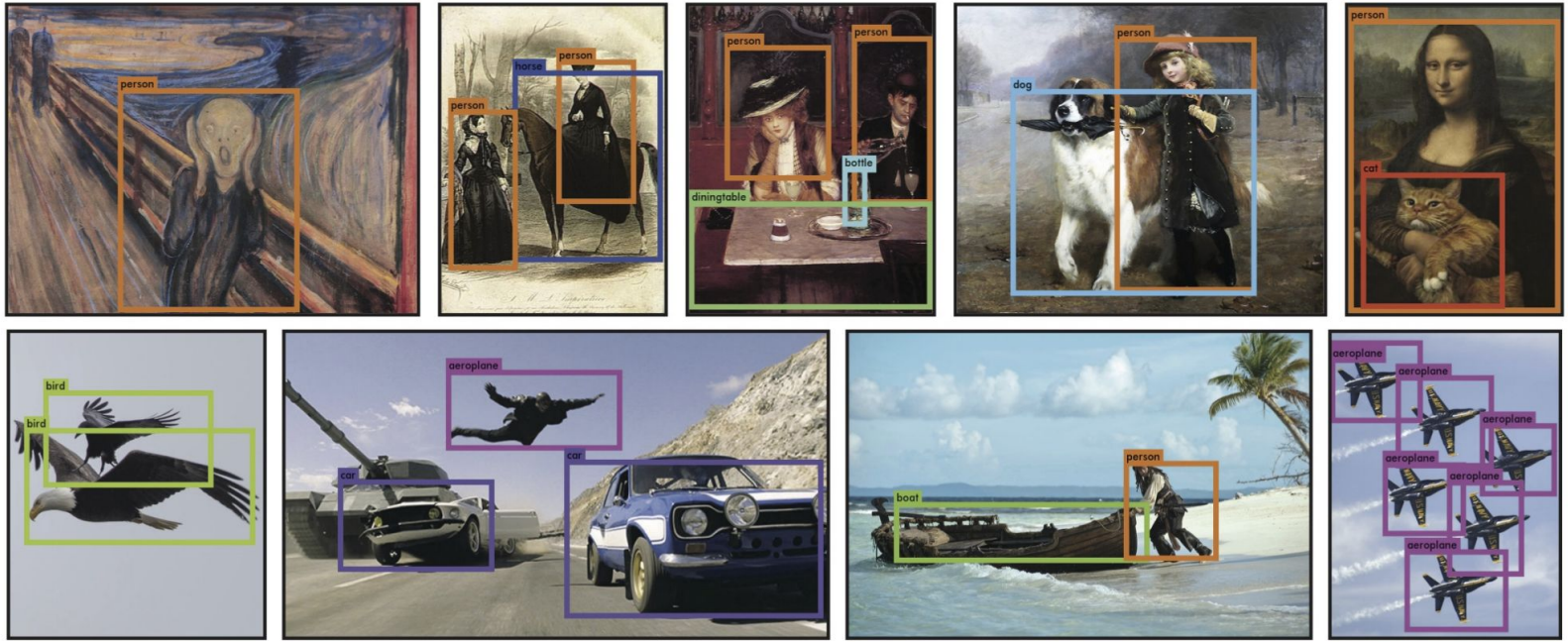
Basic idea:

- 152 layers deep (wow)
- Problems with deep networks? Vanishing gradient.
- Mitigated by adding residual connections allowing signal to propagate the signal.

ResNet, Microsoft - 2015

Beyond image classification

YOLO



You only look once, 2015

YOLO

Single forward pass object detection

- Resizes the input image to 448 × 448
- Runs a CNN (very much like the ones we've seen)
- Output gives bounding boxes and class labels

Previous object detection strategies were much slower (YOLO was ~25 ms / inference in 2015)

You only look once, 2015

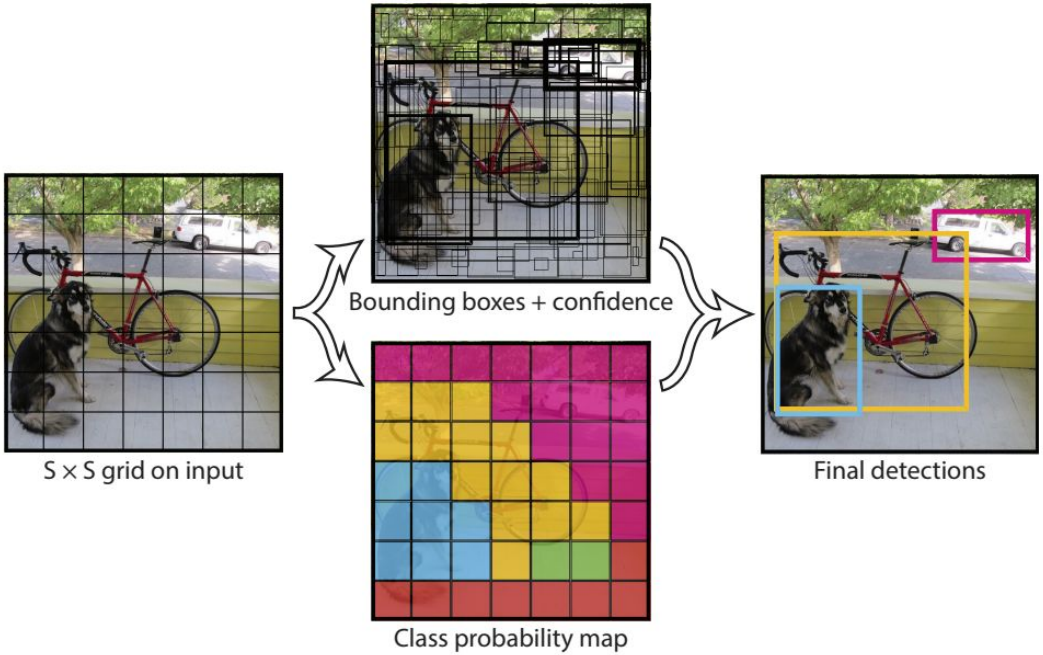


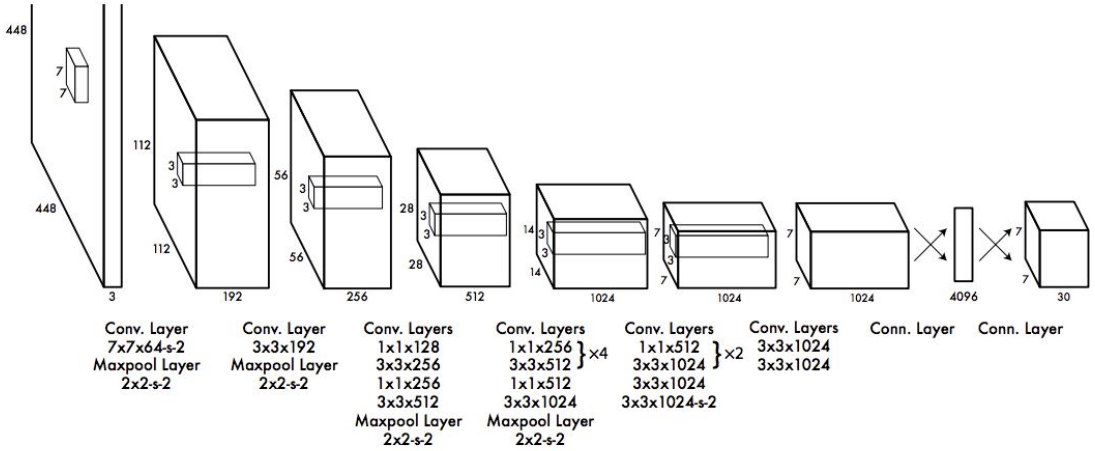
Image is divided into a S × S grid. For each grid cell, predict B bounding boxes, and C class probabilities.

Predictions: include x, y, w, h, confidence. (x,y) gives center of the bounding box. Confidence gives IOU estimate between predicted box and ground truth.

One prediction for each class, for each box. So for C=20, S=7, and B=2, output from network is S × S × (B ∗ 5 + C) = 7 x 7 x 30.

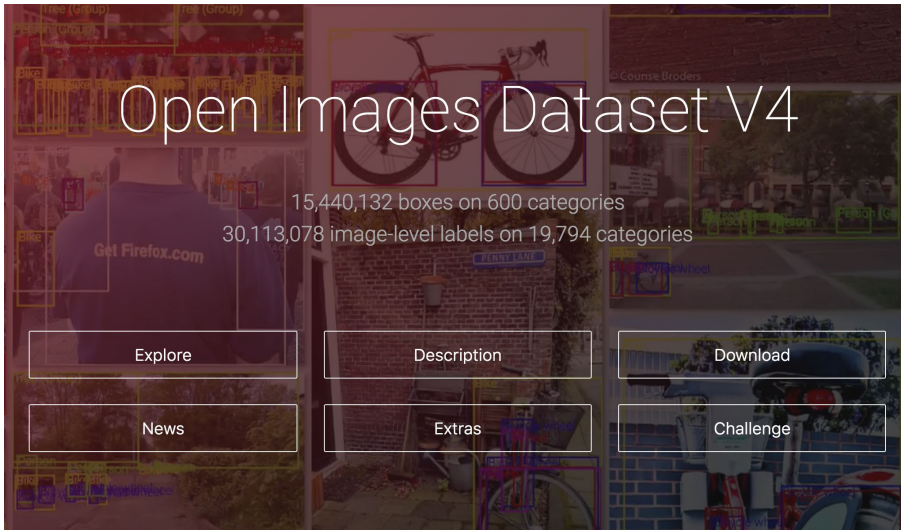
You only look once, 2015

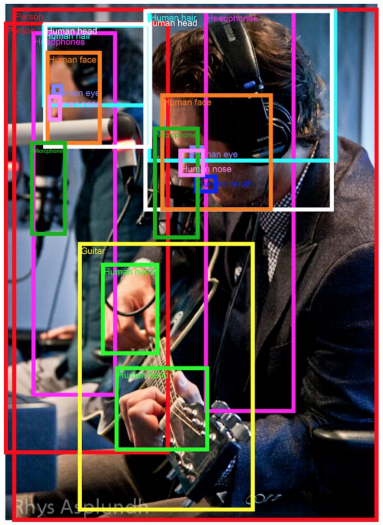
Design

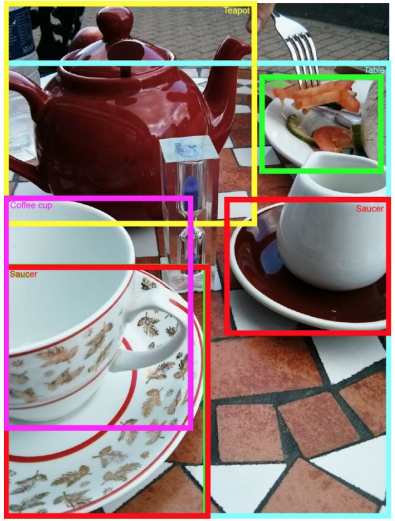


A stack of 24 conv layers of various sizes, with max pooling. Easiest way to think of the output is by having a dense layers at the end).

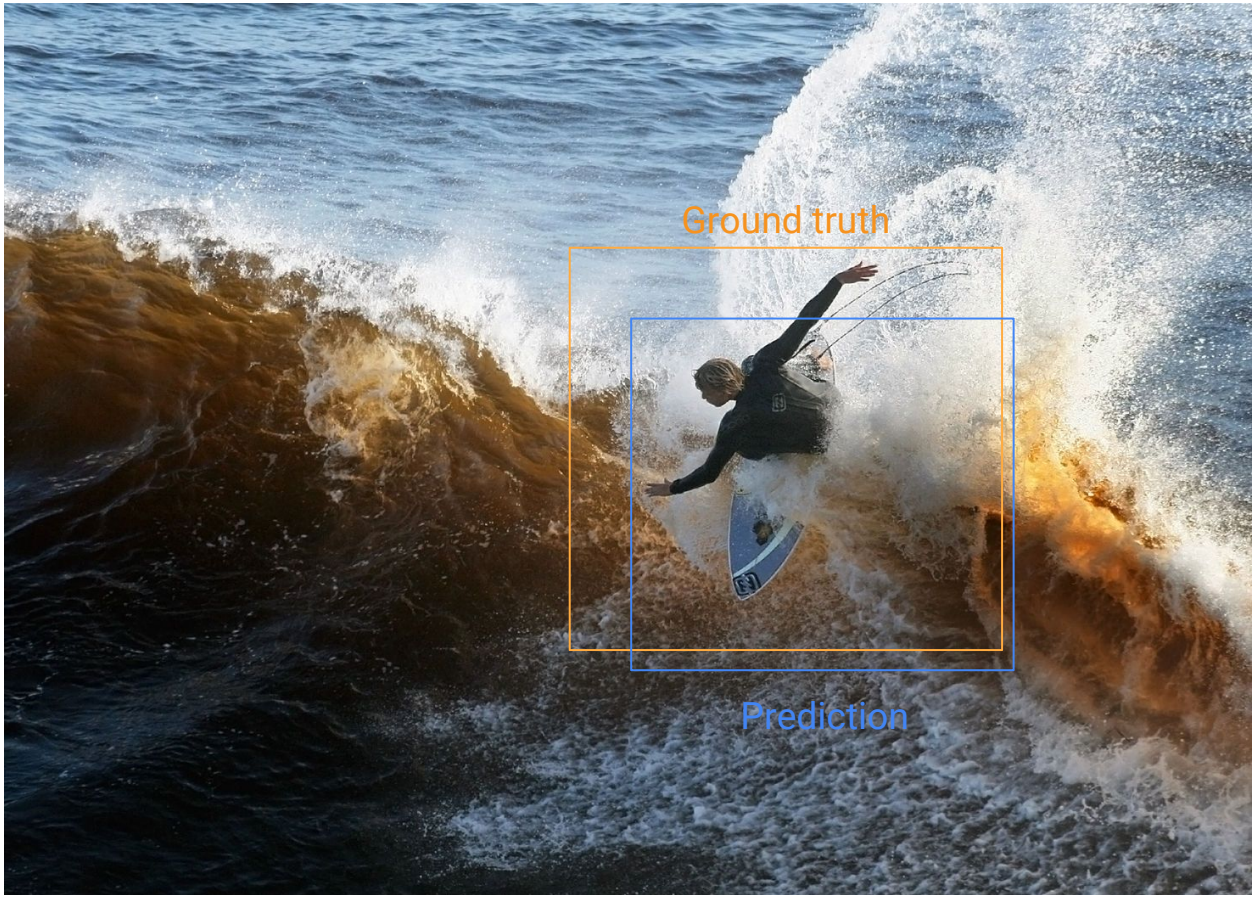
Recent dataset: 15M bounding boxes (600 categories); 30M image-level labels (20K categories)



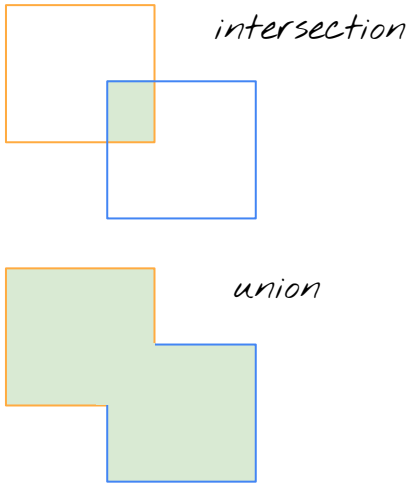




https://ai.googleblog.com/2018/04/announcing-open-images-v4-and-eccv-2018.html



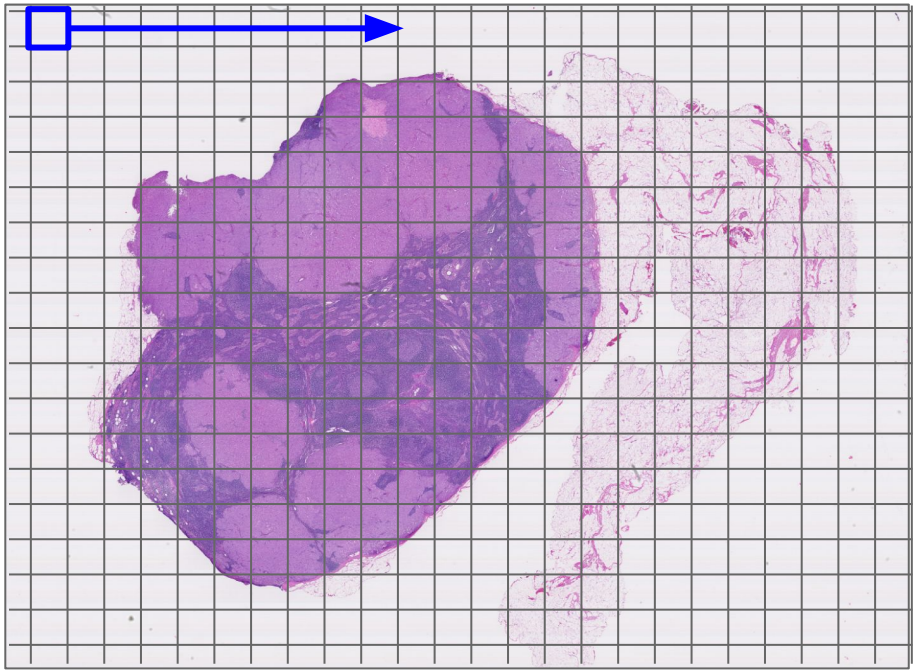
IOU (Intersection over union)



A surfer at the wave

Much more on this in lecture 8.

Other approaches (also powered by CNN)



Approach: Divide images into small patches; Train a model on to classify patches as malignant / benign. Slide window over new image, classifying each patch, produce heatmap.

Pros

- Works with images of any size, potentially more accurate. Simpler, easier to debug and explain.

Cons

- YOLO can "look" at the entire image all at once. Not obvious how to incorporate different views of the image..
- Super slow (must classify each grid, many forward passes).

Apply DL like the great data scientist you are, not like the image classification researcher you aren't.

- 99% of the time, you are better off retraining or fine tuning an existing well known architecture for your task, rather than designing a novel architecture from scratch (unless your goal is to do research in image classification *itself*).

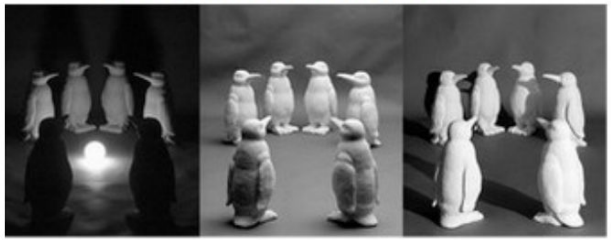
- This is true even for fine grained tasks, like medical imaging.

Data augmentation

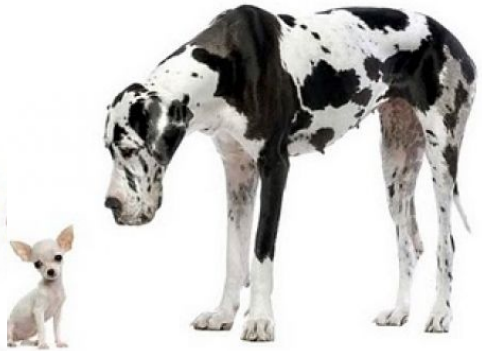
Many sources of variation. Ideally, want to capture these in our training data to help our model generalize - but probably won't have enough training data to do so.

Viewpoint variation

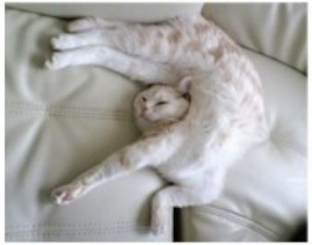
Illumination conditions



Scale variation



Deformation



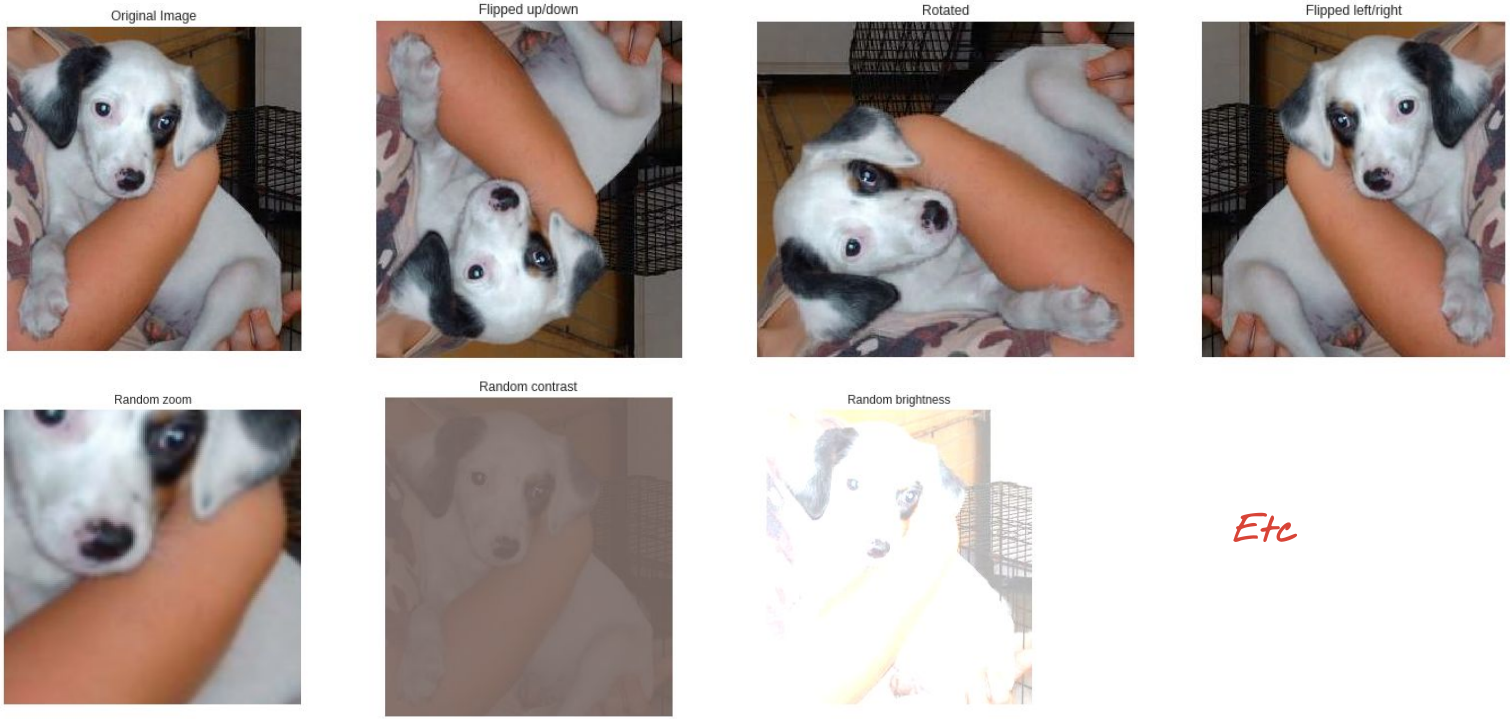
Occlusion

Background clutter

Intra-class variation

Parts of this slide borrowed from MIT (with permission).

Data augmentation



Etc

Less obvious transforms may be relevant for your domain

Data augmentation may be applied to the **training set only**. Never to validation or test.

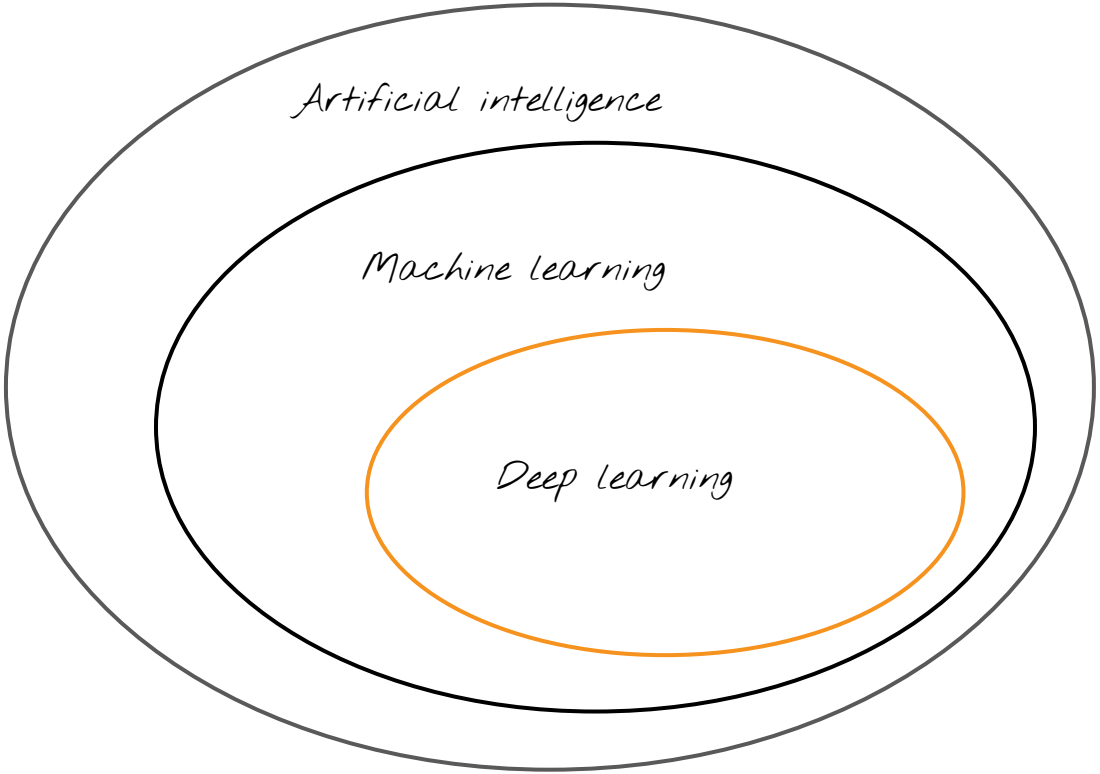
Computationally expensive (increasing the size of your training set by a factor of n_augmentations).

Most libraries provide data augmentation tools. I'll try to cover tf.data in lecture 6 to show how to do this at scale, though 99% of the time simpler options are more than adequate and your best bet.

https://keras.io/preprocessing/image/

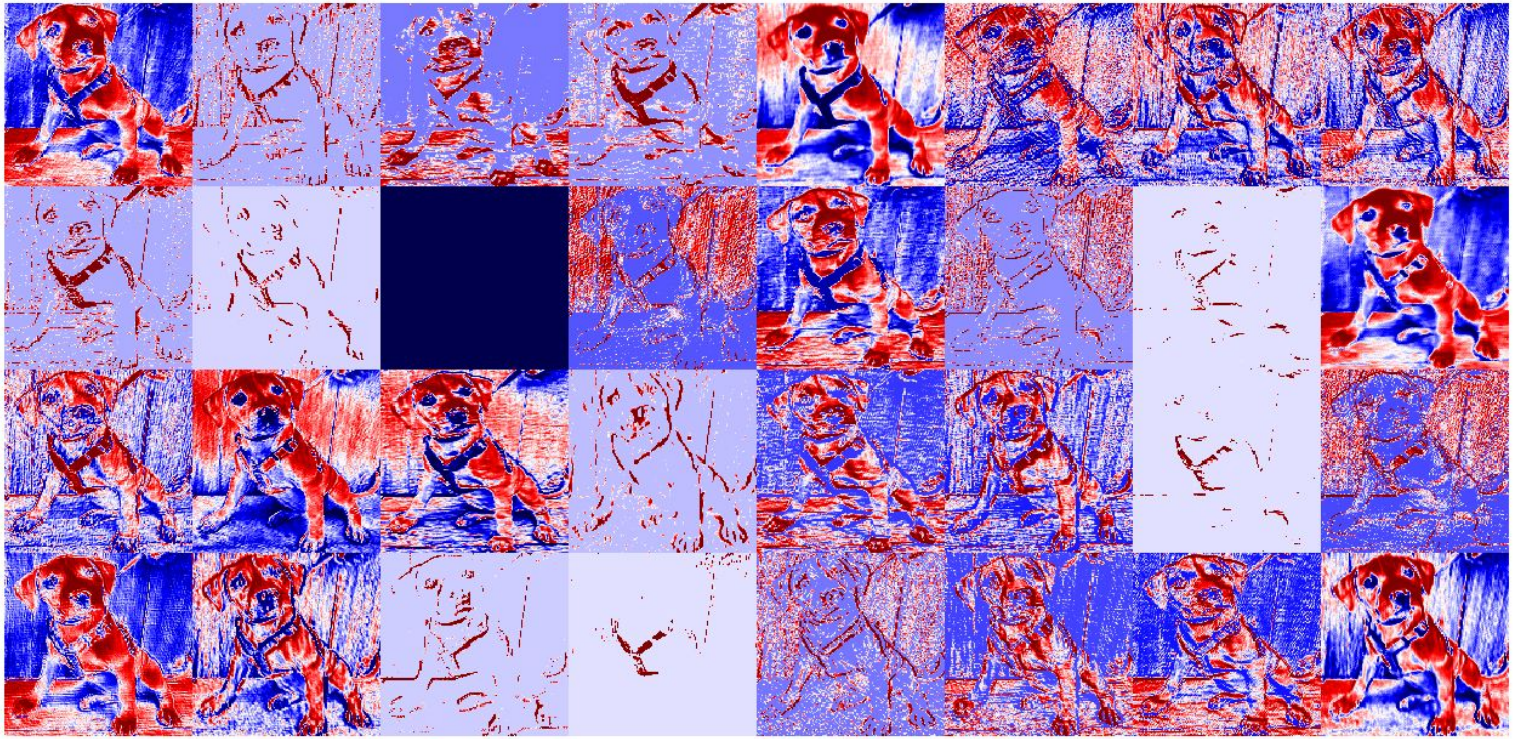
random_jpeg_quality

Feature visualization

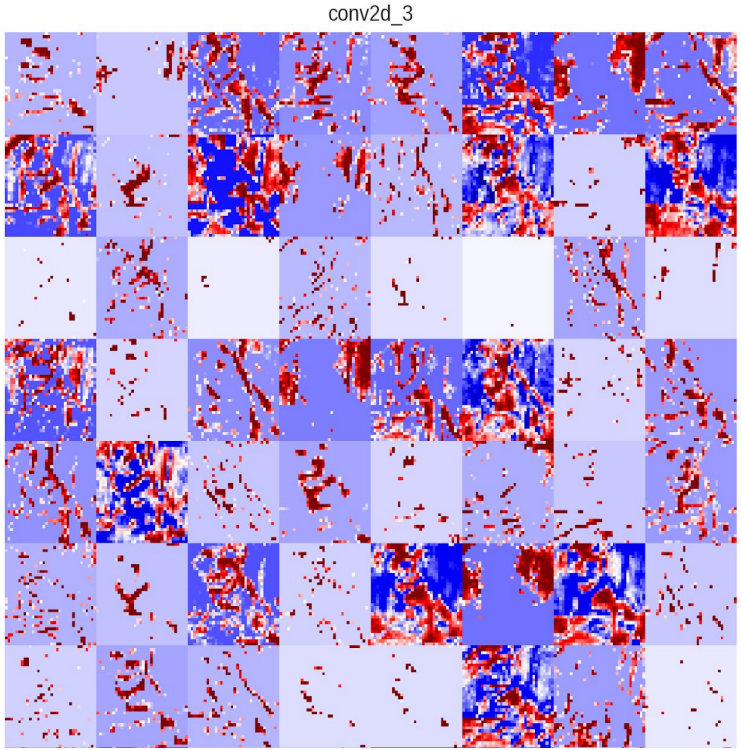


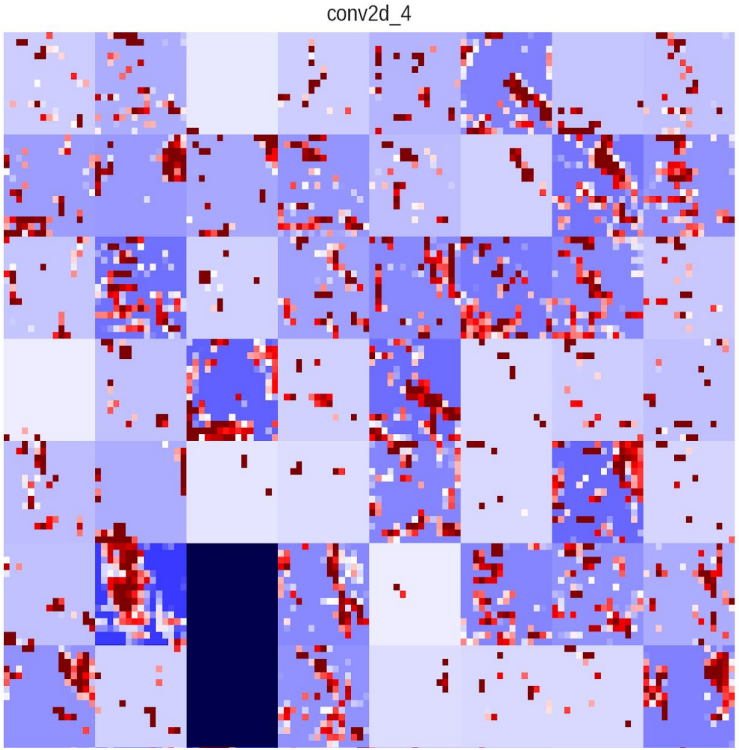
CNNs provide a couple avenues to see representation learning in action.

conv2d_1

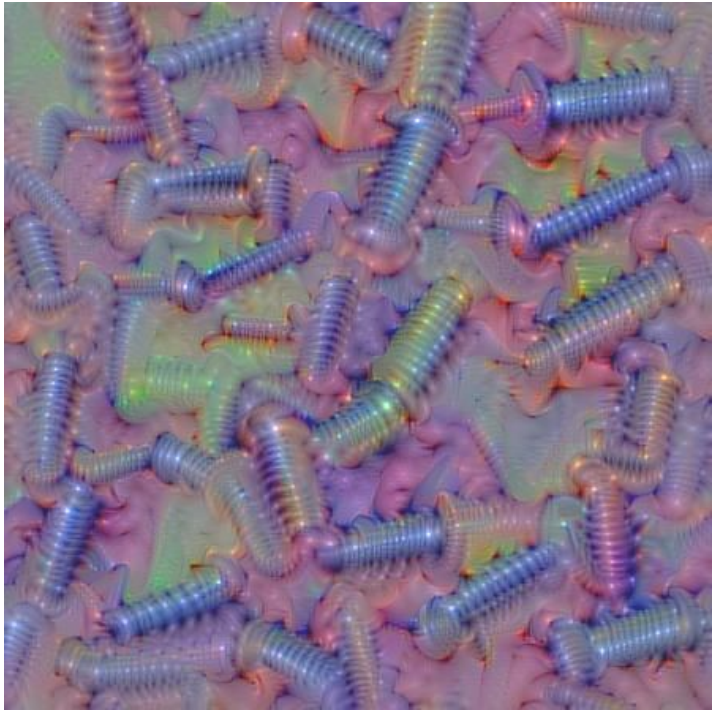


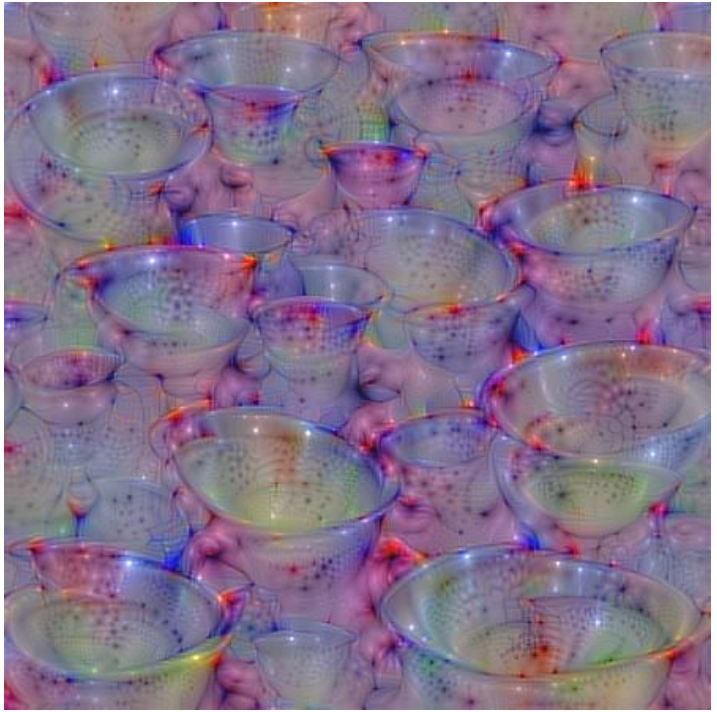
Idea: Forward an image through the network, show the output of each feature map.





Notice resolution is decreasing as we move up in layers. Representation tells us more about "what" was in the image, rather than "where".



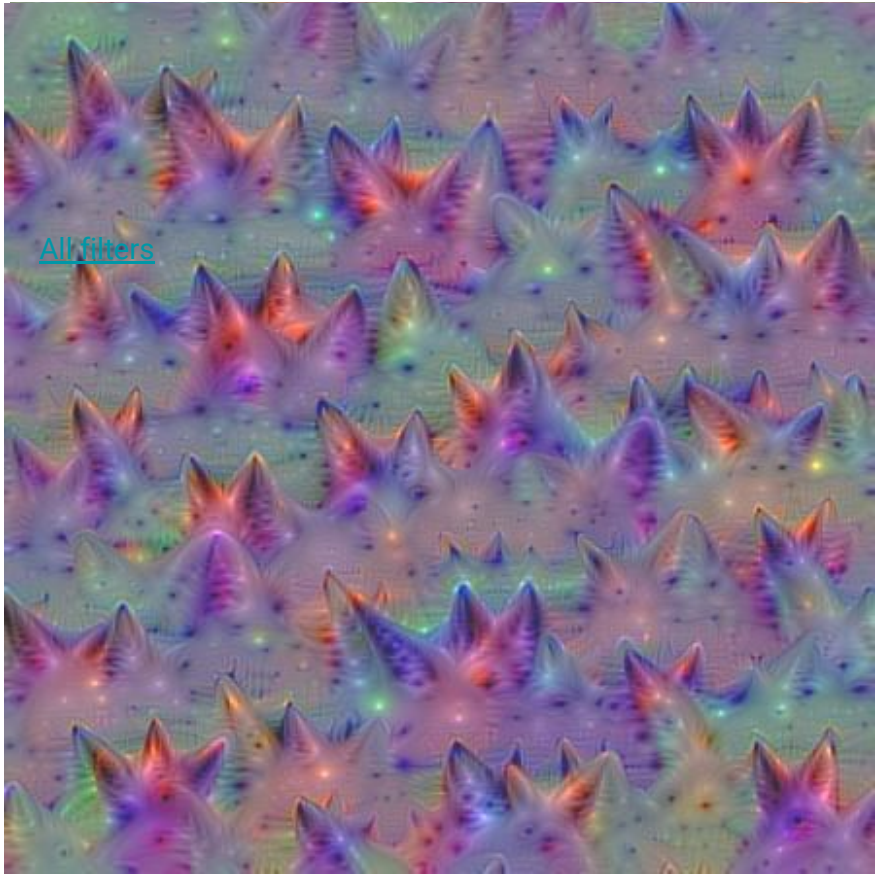


Idea: Create a loss function that's the mean activation of a feature map! Starting from a random noise image, calculate gradients of loss w.r.t pixels, optimize per usual.

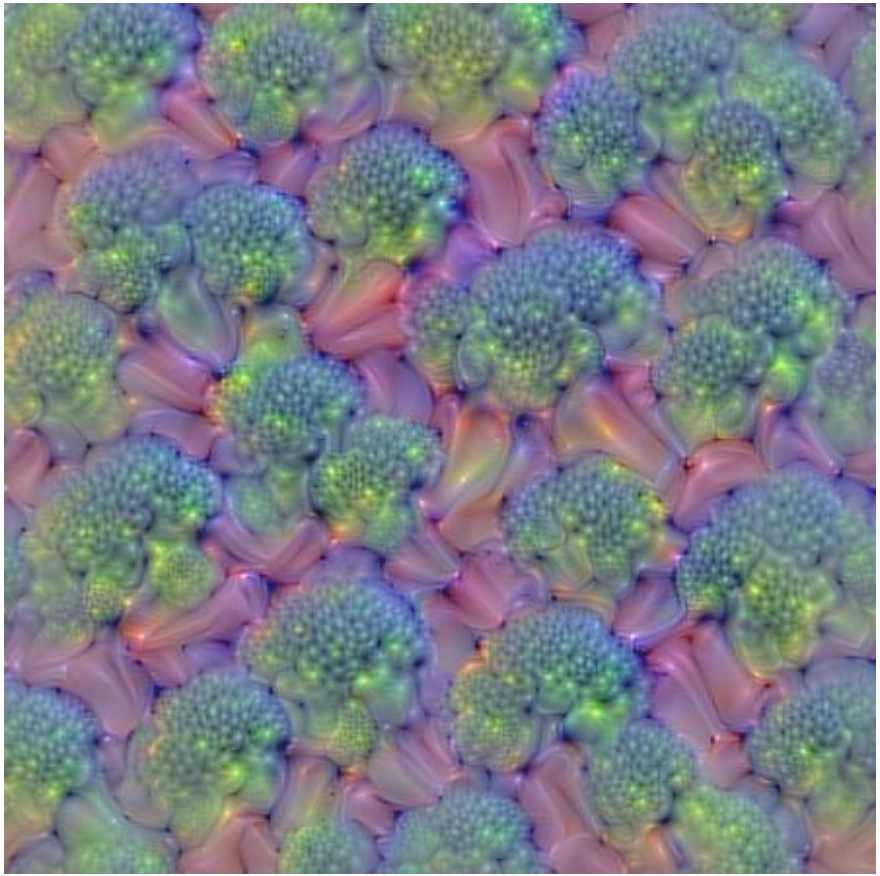
All filters

Screws / Corkscrews?

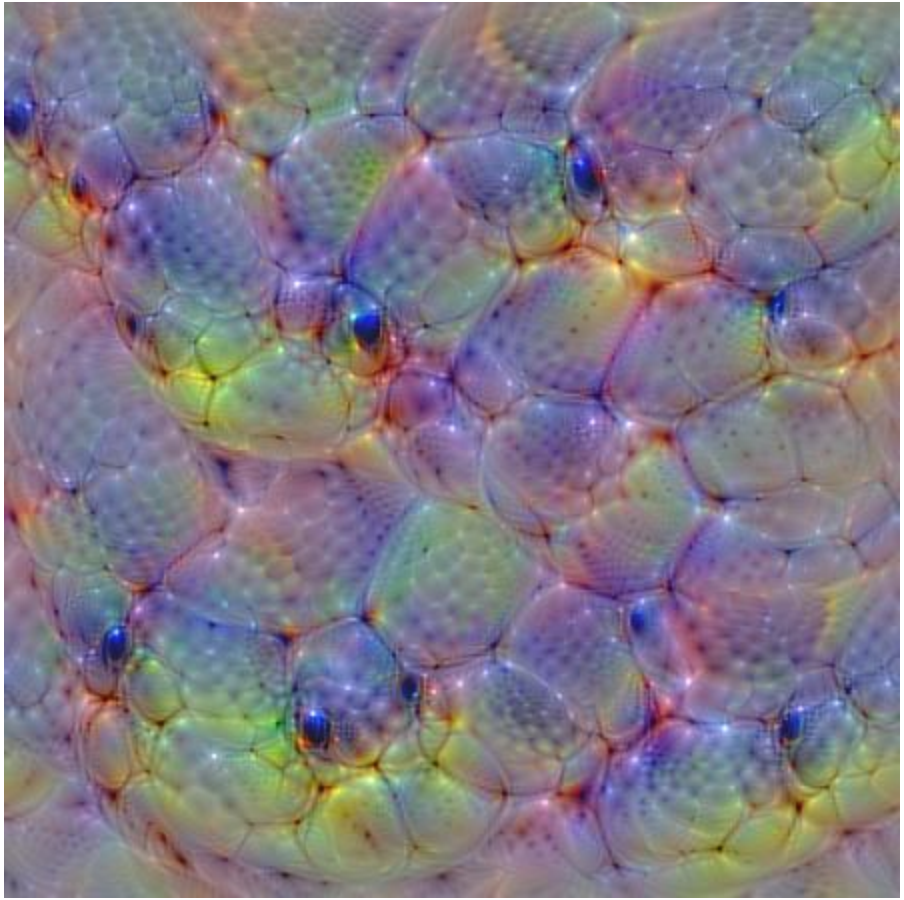
Wine glass?



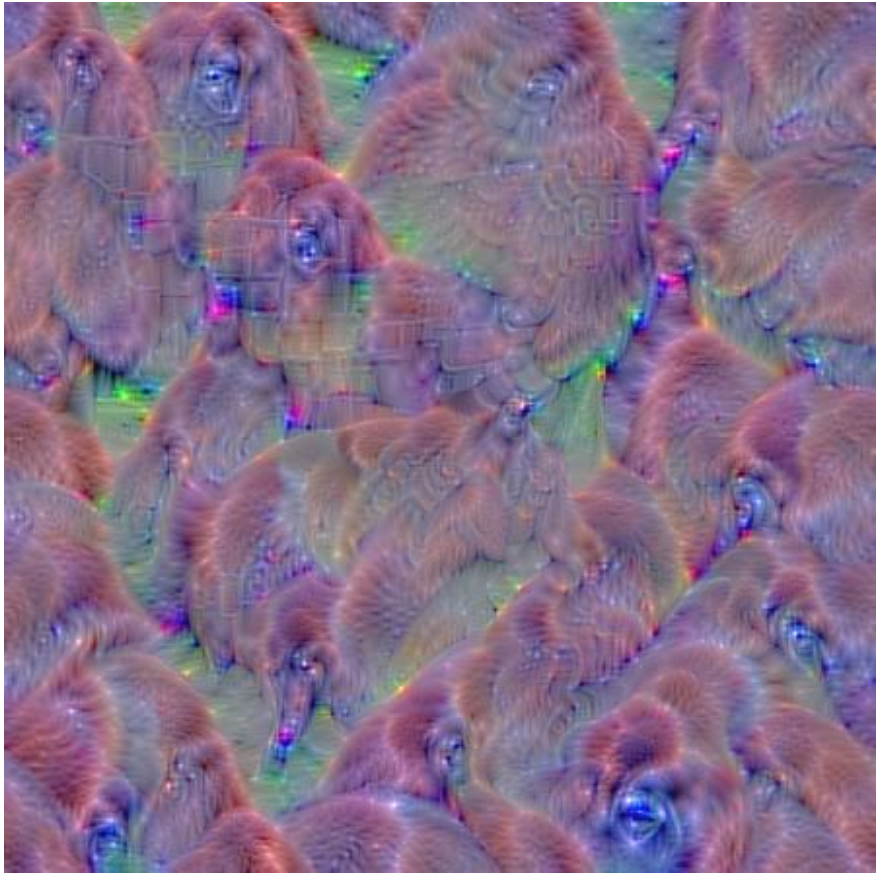
Ears?



Broccoli / Cauliflower?



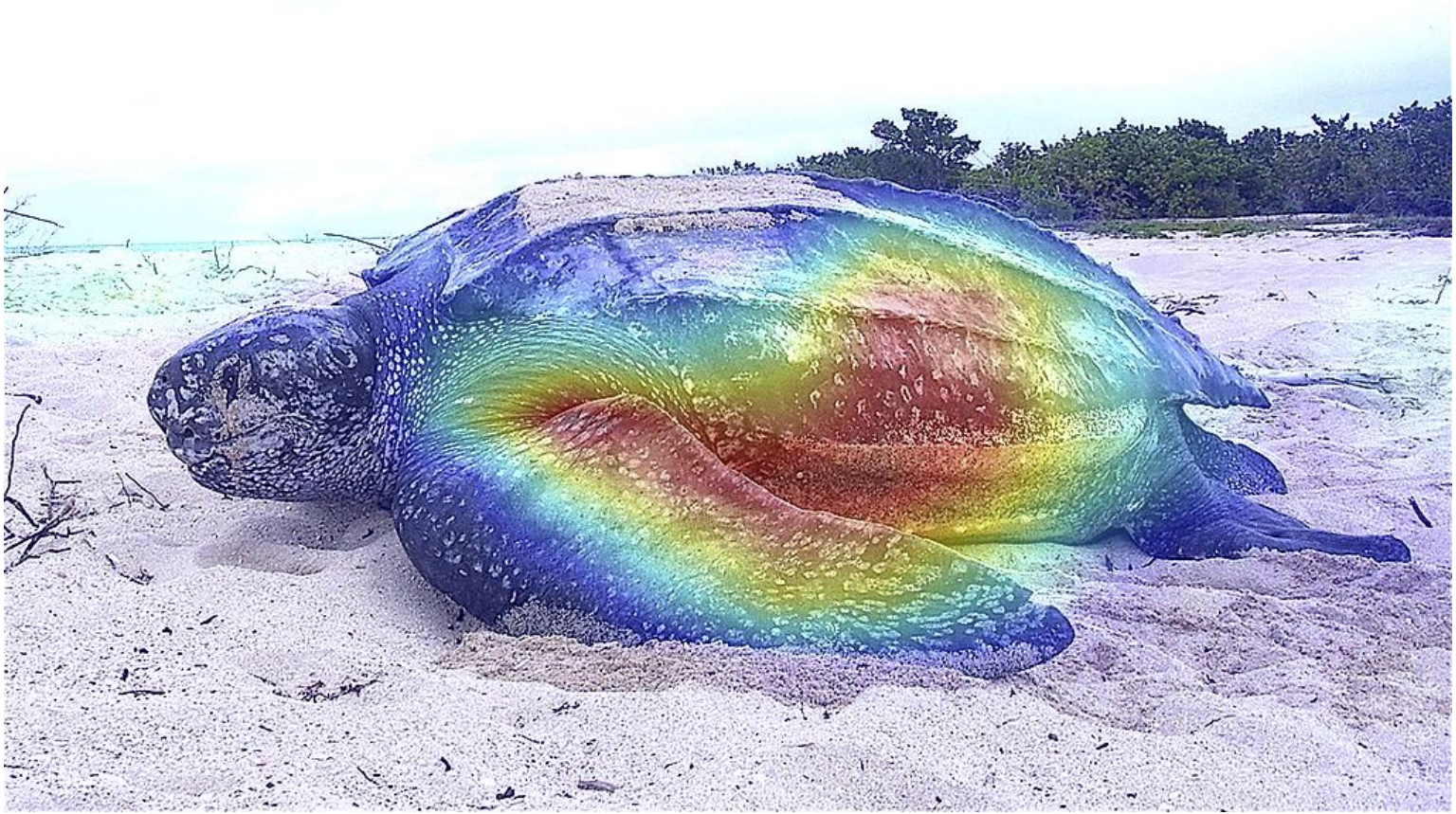
Snakes?



Gorillas / chimpanzees?

Idea: choose the last conv layer. Take all feature maps. Record weight from the dense connection to some class. Next, calculate gradient of each feature map with respect to the inputs. Weight those by "importance" from dense layer.

Leatherback turtle



Leatherback turtle

Loggerhead turtle

Next time

Code

- 3.1-backprop uploaded to GitHub (we'll cover next week)

Reading

- Deep Learning with Python: 4, 5.
- Deep Learning: 9

Assignments

- **Reminder: A1 due next week**
- A2 will be posted tomorrow (will extend the due date a bit)