Convolutional Networks

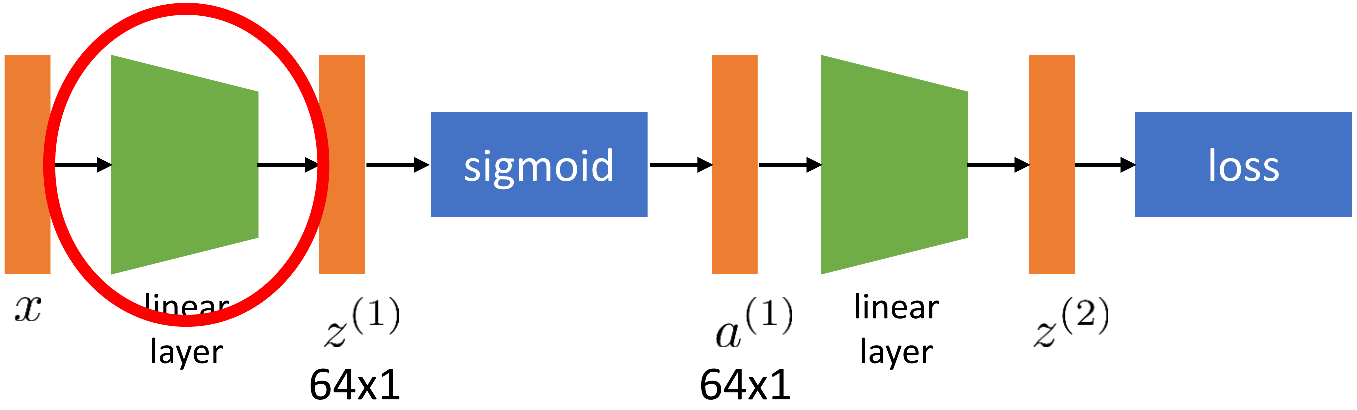
Designing, Visualizing and Understanding Deep Neural Networks

CS W182/282A

Instructor: Sergey Levine
UC Berkeley

Neural network with images

[object label]



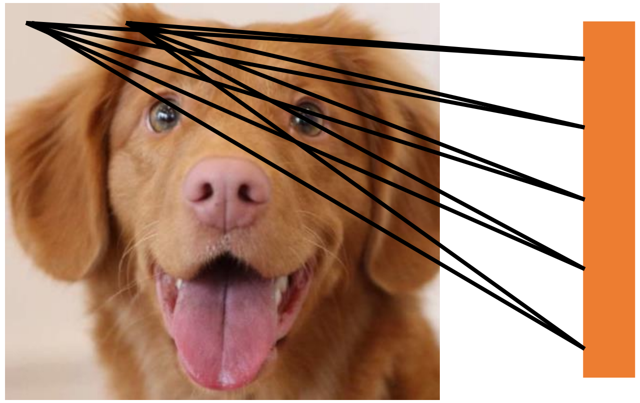


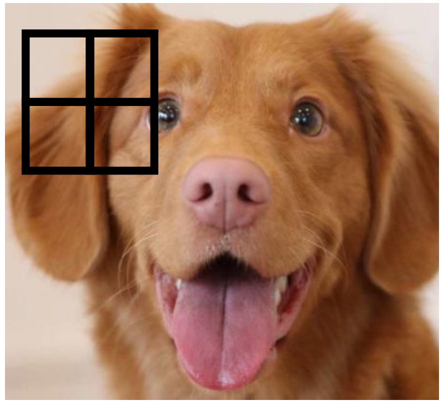
image is $128 \times 128 \times 3 = 49{,}152$

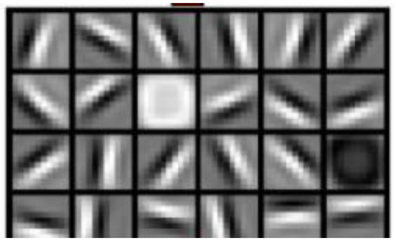
$z^{(1)}$ is 64-dim

$64 \times 49,152 \approx 3,000,000$

We need a better way!

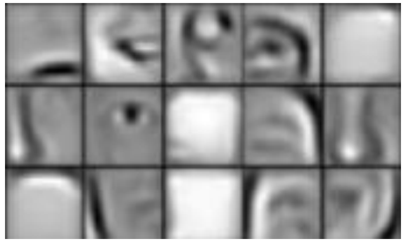
An idea...





Layer 1:

edge detectors?



Layer 2:

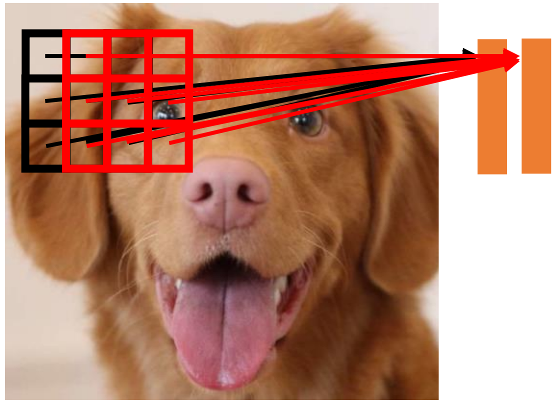
ears? noses?

Observation: many useful image features are **local**

to tell if a particular patch of image contains a feature, enough to look at the local patch

An idea...

Observation: many useful image features are **local**

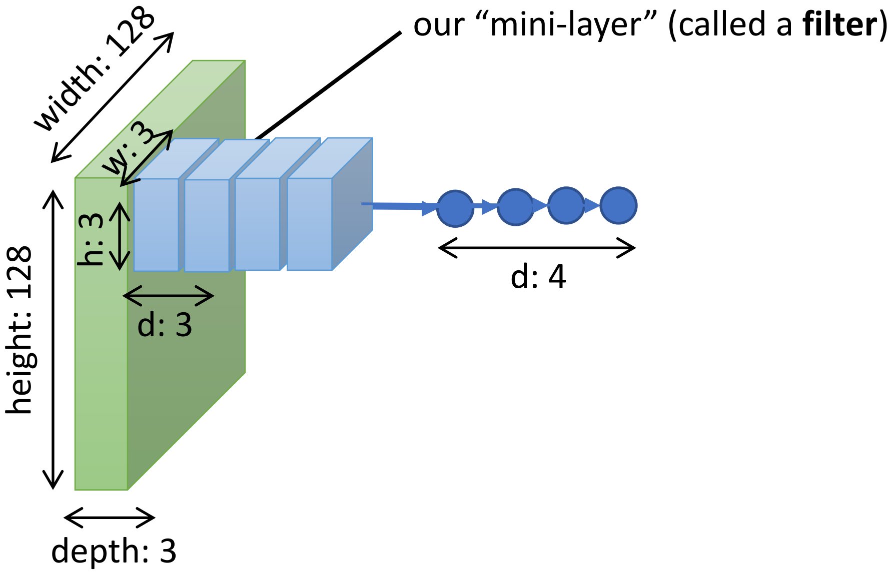


$$\text{patch is } 3 \times 3 \times 3 = 27$$

$$z^{(1)} \text{ is 64-dim}$$

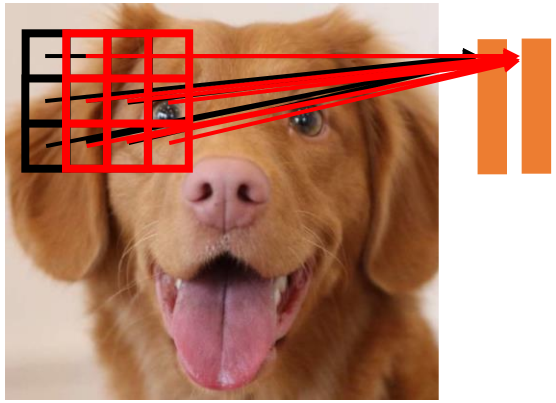
$$64 \times 27 = 1728$$

We get a **different** output at each image location!



An idea...

Observation: many useful image features are **local**

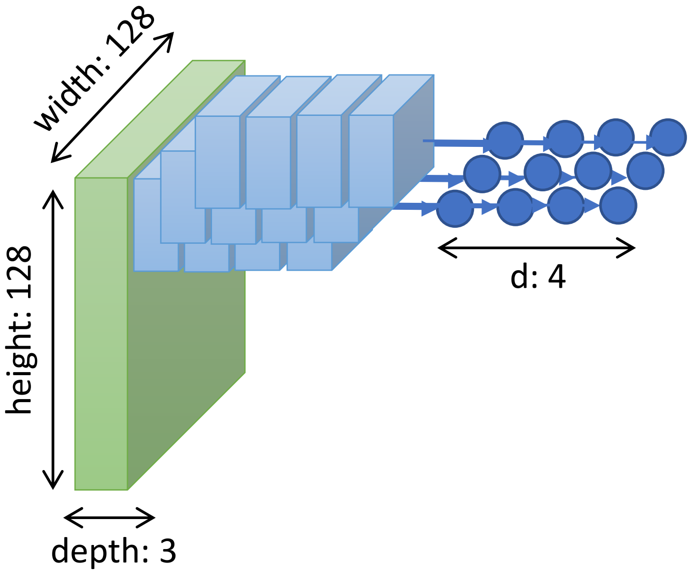


$$\text{patch is } 3 \times 3 \times 3 = 27$$

$$z^{(1)} \text{ is 64-dim}$$

$$64 \times 27 = 1728$$

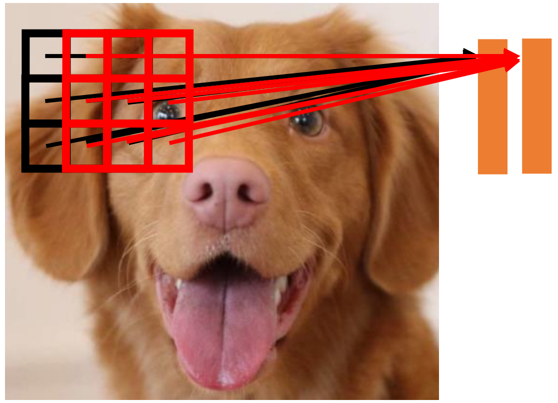
We get a **different** output at each image location!



don't forget to apply non-linearity!

An idea...

Observation: many useful image features are **local**

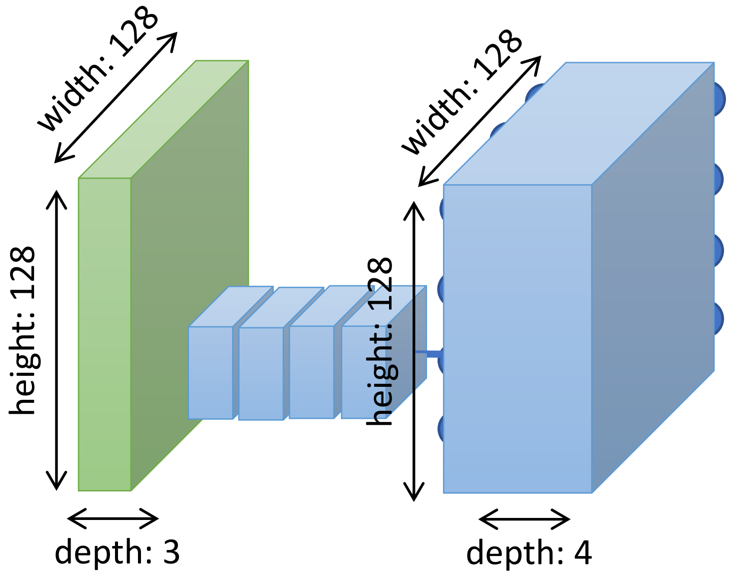


$$\text{patch is } 3 \times 3 \times 3 = 27$$

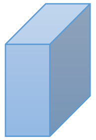
$$z^{(1)} \text{ is 64-dim}$$

$$64 \times 27 = 1728$$

We get a **different** output at each image location!

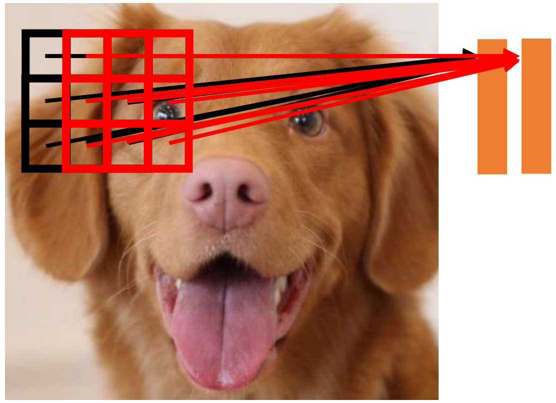


What do they look like?



An idea...

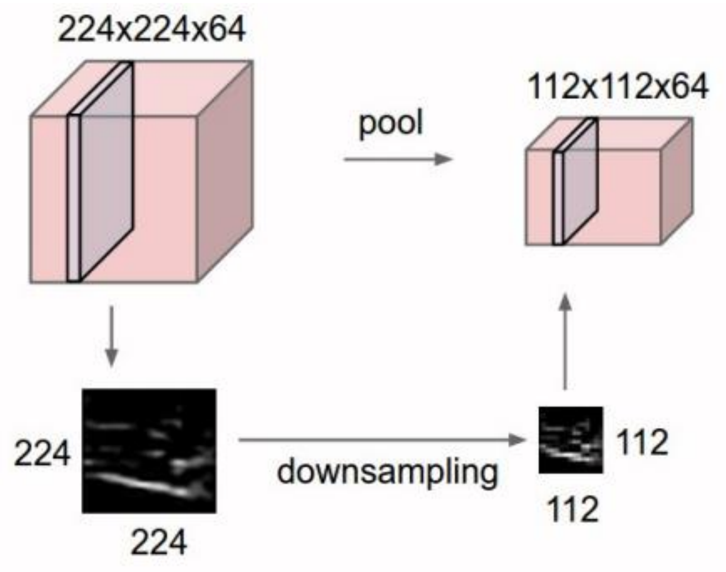
Observation: many useful image features are **local**



$$\text{patch is } 3 \times 3 \times 3 = 27$$

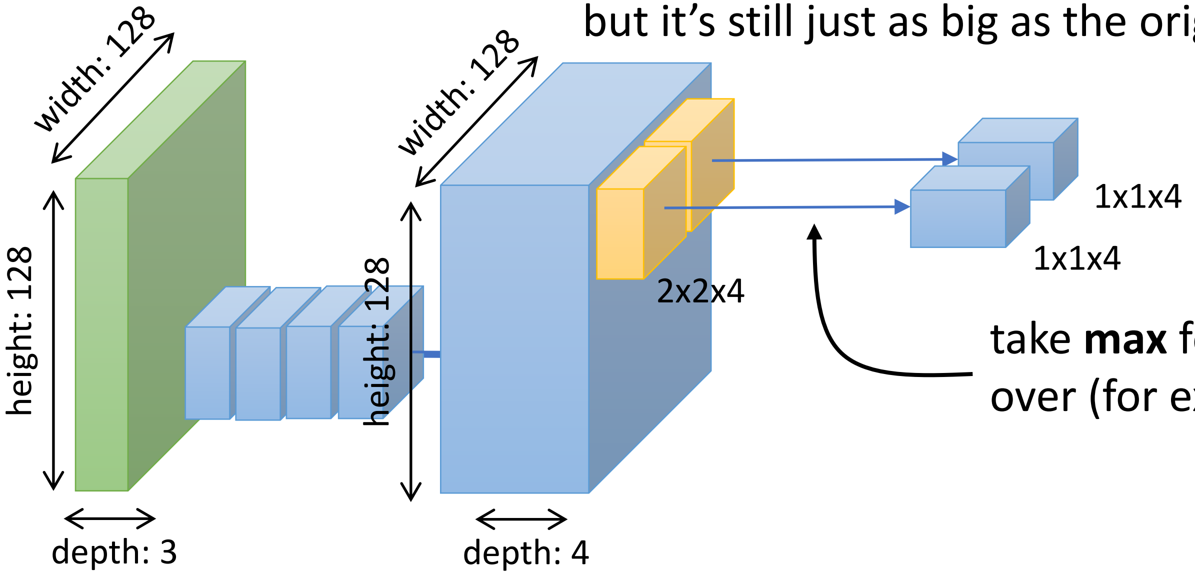
$$z^{(1)} \text{ is 64-dim}$$

$$64 \times 27 = 1728$$



We get a **different** output at each image location!

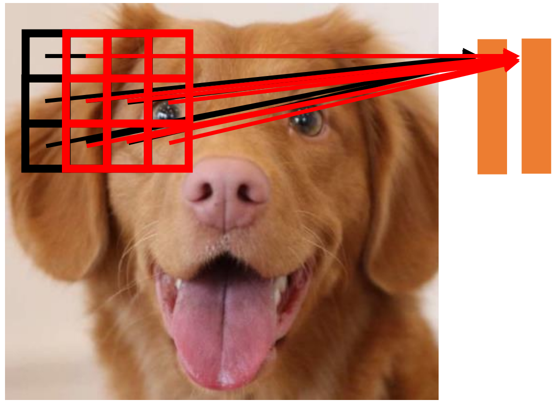
but it's still just as big as the original image!



take **max** for each channel
over (for ex) 2x2 region

An idea...

Observation: many useful image features are **local**

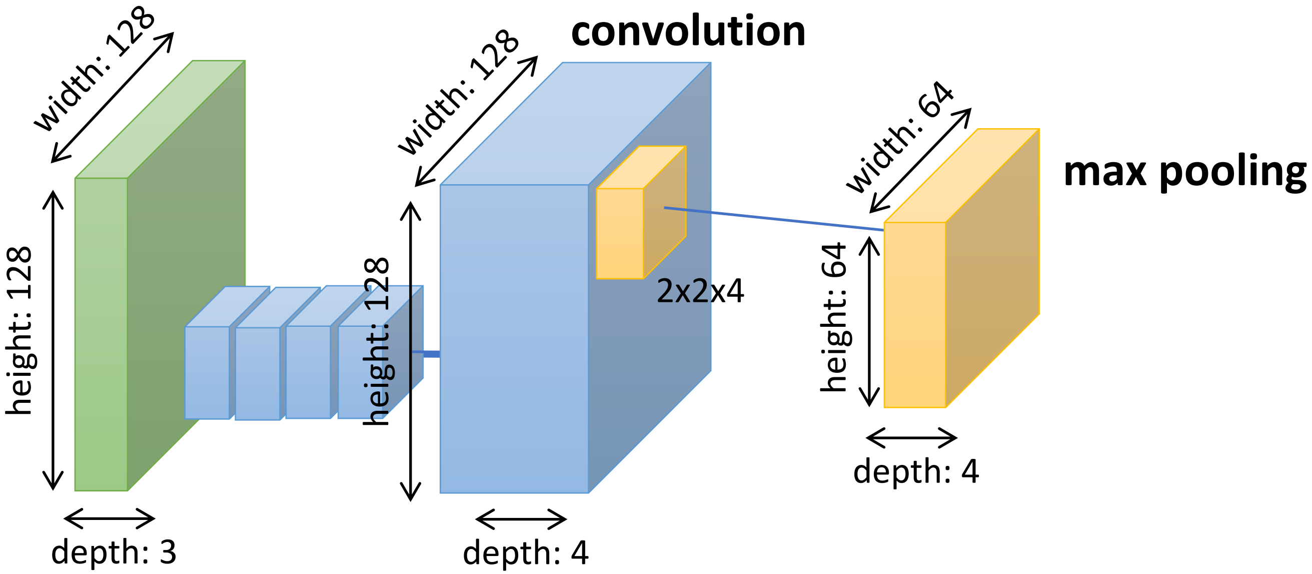


patch is $3 \times 3 \times 3 = 27$

$z^{(1)}$ is 64-dim

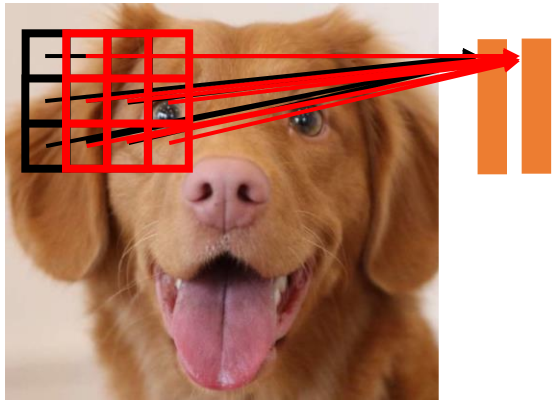
$64 \times 27 = 1728$

We get a **different** output at each image location!



An idea...

Observation: many useful image features are **local**

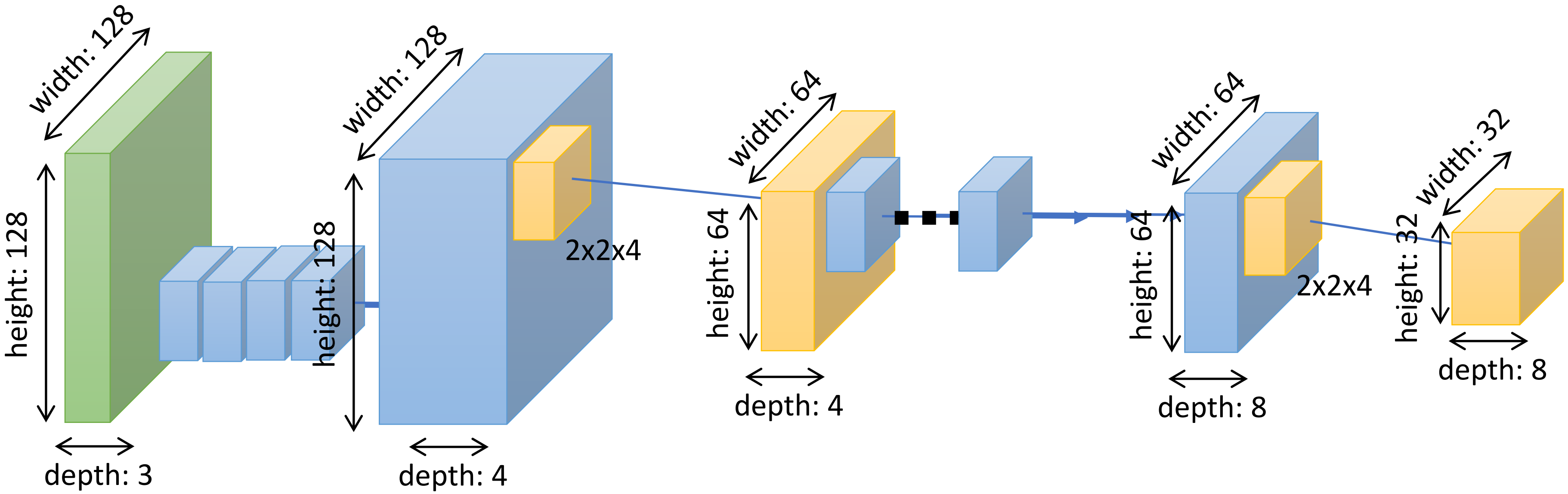


$$\text{patch is } 3 \times 3 \times 3 = 27$$

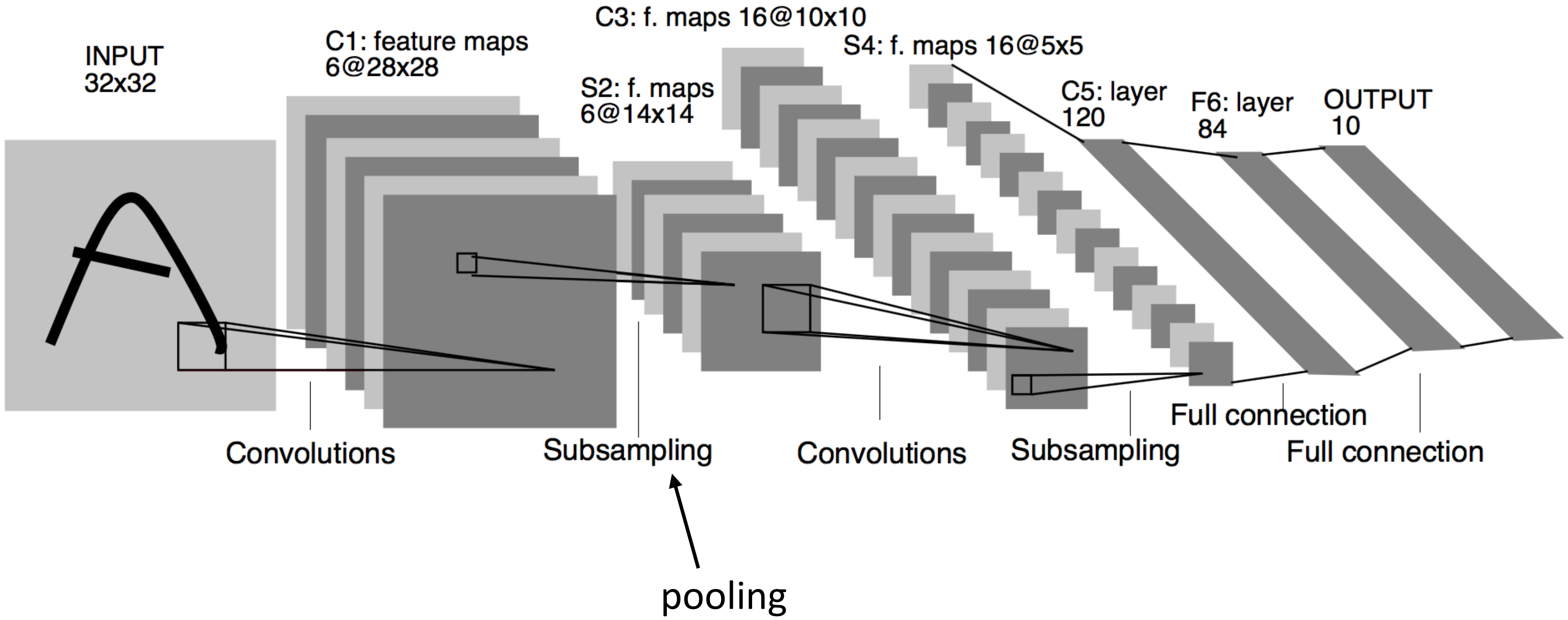
$$z^{(1)} \text{ is 64-dim}$$

$$64 \times 27 = 1728$$

We get a **different** output at each image location!



What does a real conv net look like?

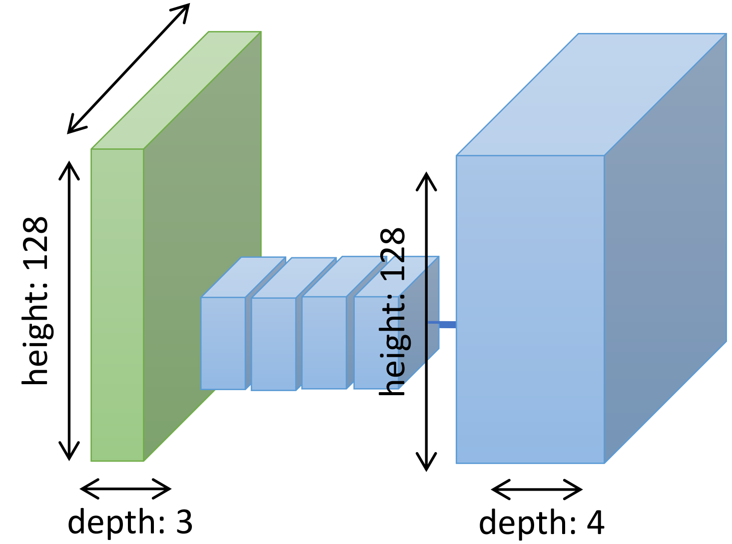


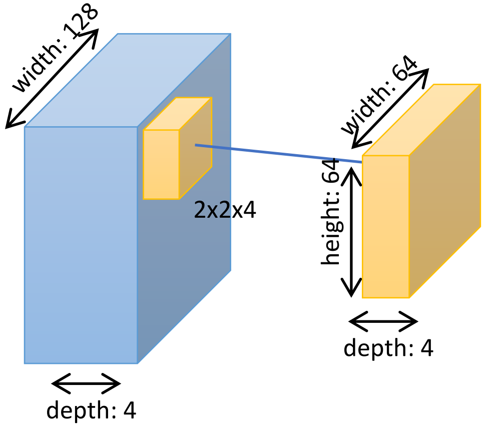
“LeNet” network for handwritten digit recognition

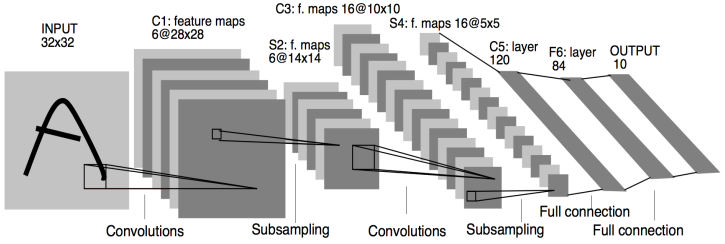
Implementing convolutional layers

Summary

- **Convolutional layer**
  - A way to avoid needing millions of parameters with images
  - Each layer is “local”
  - Each layer produces an “image” with (roughly) the same width & height, and number of channels = number of filters
- **Pooling**
  - If we ever want to get down to a single output, we must reduce resolution as we go
  - Max pooling: downsample the “image” at each layer, taking the max in each region
  - This makes it robust to small translation changes
- **Finishing it up**
  - At the end, we get something small enough that we can “flatten” it (turn it into a vector), and feed into a standard fully connected layer







ND arrays/tensors

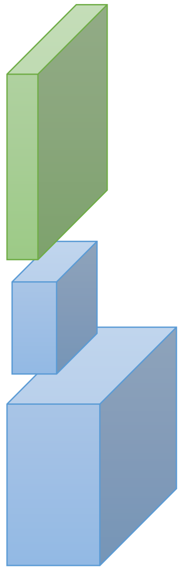
all these operations will involve N-dimensional arrays

often used synonymously with *tensor*

input image: $\text{HEIGHT} \times \text{WIDTH} \times \text{CHANNELS}$

filter: $\text{FLT.HEIGHT} \times \text{FLT.WIDTH} \times \text{OUTPUT CHAN} \times \text{INPUT CHAN}$

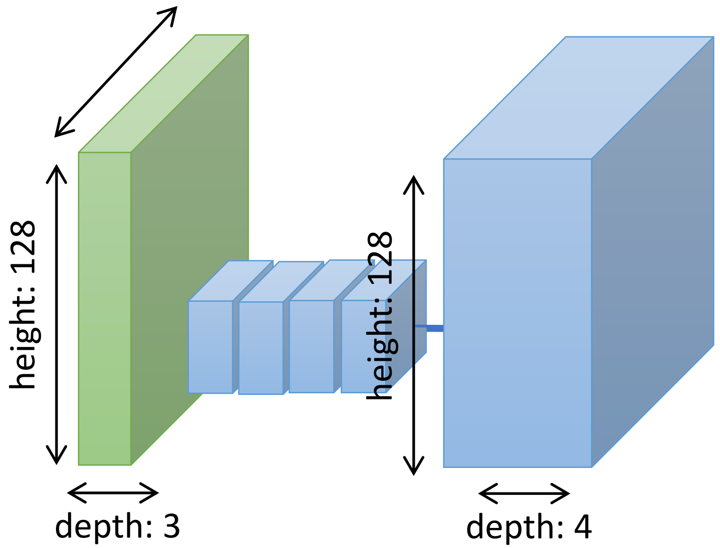
activations: $\text{HEIGHT} \times \text{WIDTH} \times \text{LAYER.CHANNELS}$



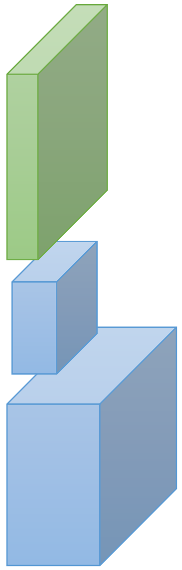
The “inner” (rightmost) dimensions work just like vectors/matrices

Matching “outer” dimensions (e.g., height/width) are treated as “broadcast” (i.e., elementwise operations)

Convolution operations performs a tiny matrix multiply at each position (like a tiny linear layer at each position)



Convolutional layer in equations



all these operationsn will involve N-dimensional arrays

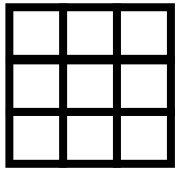
often used synonymously with *tensor*

input image: HEIGHT $\times$ WIDTH $\times$ CHANNELS

filter: FLT.HEIGHT $\times$ FLT.WIDTH $\times$ OUTPUT CHAN $\times$ INPUT CHAN

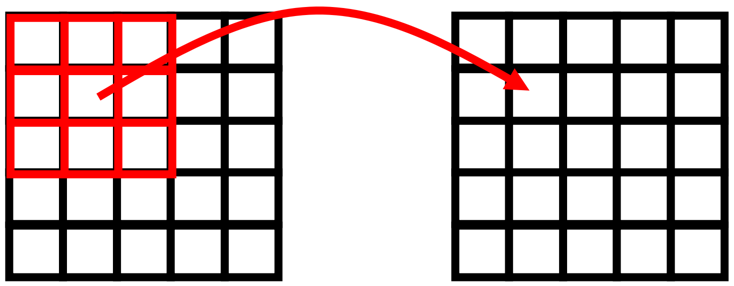
activations: HEIGHT $\times$ WIDTH $\times$ LAYER.CHANNELS

$$a^{(1)} \to z^{(2)} \qquad W^{(2)} : H_F \times W_F \times C_{\text{out}} \times C_{\text{in}}$$



$$H_{\text{in}} \times W_{\text{in}} \times C_{\text{in}} \qquad H_{\text{out}} \times W_{\text{out}} \times C_{\text{out}}$$

equal or **almost** equal (more on this later)



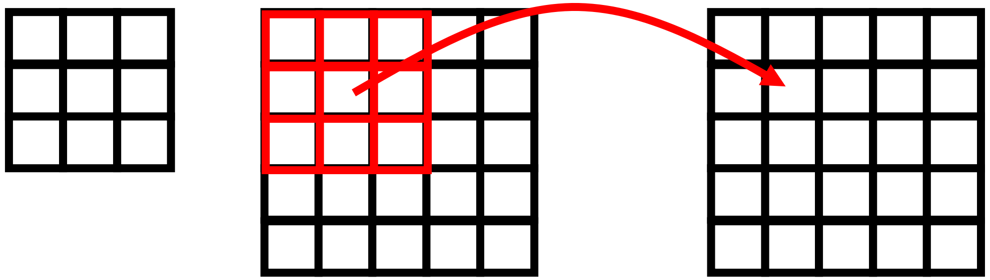
Convolutional layer in equations

$$a^{(1)} \rightarrow z^{(2)}$$

$$H_{\text{in}} \times W_{\text{in}} \times C_{\text{in}} \qquad H_{\text{out}} \times W_{\text{out}} \times C_{\text{out}}$$

equal or **almost** equal (more on this later)

$$W^{(2)} : H_F \times W_F \times C_{\text{out}} \times C_{\text{in}}$$



$$z^{(2)}[i,j,k] = \sum_{l=0}^{H_F-1} \sum_{m=0}^{H_W-1} \sum_{n=0}^{C_{\text{in}}-1} W^{(2)}[l,m,k,n]\, a^{(1)}[i+l-(H_F-1)/2, j+m-(H_W-1)/2, n]$$

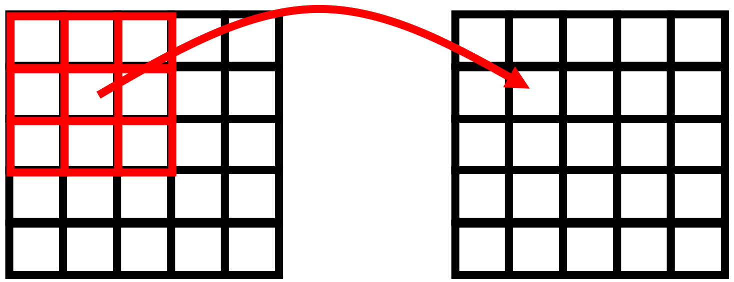
$$z^{(2)}[i,j] = \sum_{l=0}^{H_F-1} \sum_{m=0}^{H_W-1} W^{(2)}[l,m]\, a^{(1)}[i+l-(H_F-1)/2, j+m-(H_W-1)/2]$$

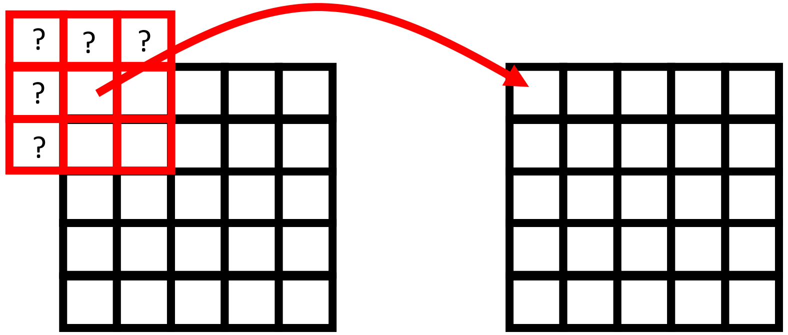
$$a^{(2)}[i,j,k] = \sigma(z^{(2)}[i,j,k])$$

Activation function applied per element, just like before

Simple principle, but a bit complicated to write

Padding and edges

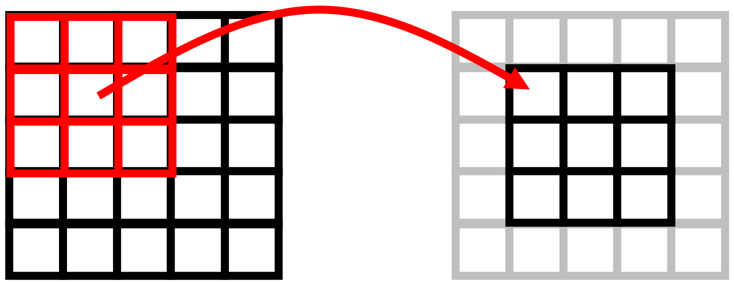




Problem: our activations shrink with every layer

Some people don't like this

Option 1: cut off the edges



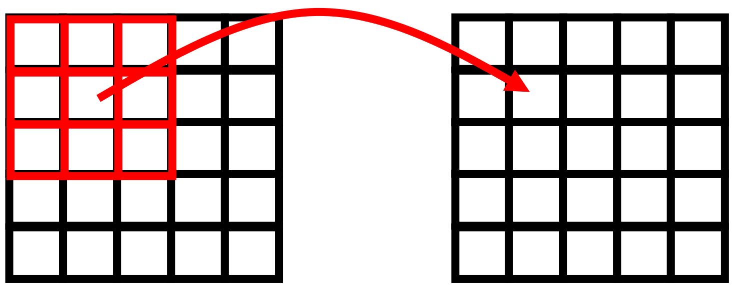
Pop quiz:
input is 32x32x3
filter is 5x5x6
what is the output in this case?

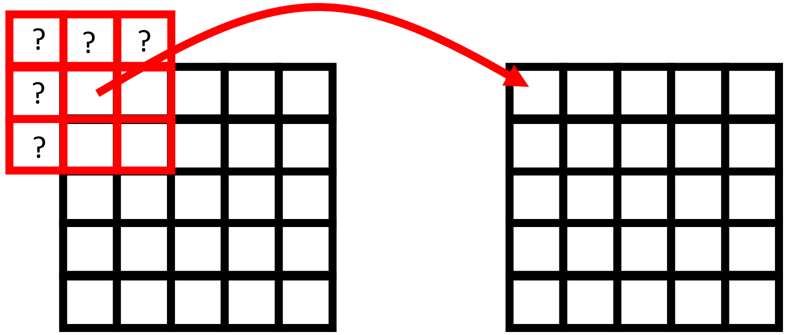
$$\text{“radius” is } (H_F - 1)/2 \text{ on each side} = 2$$

$$H_{\text{out}} = H_{\text{in}} - ((H_F - 1)/2) \times 2 = 28$$

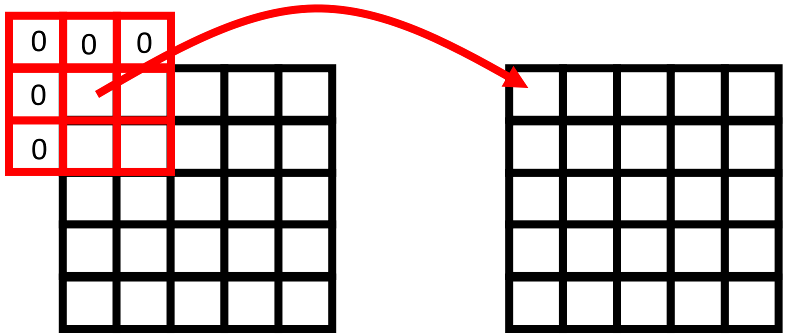
$$28 \times 28 \times 6$$

Padding and edges





Option 2: zero pad



Detail: remember to subtract the image mean first

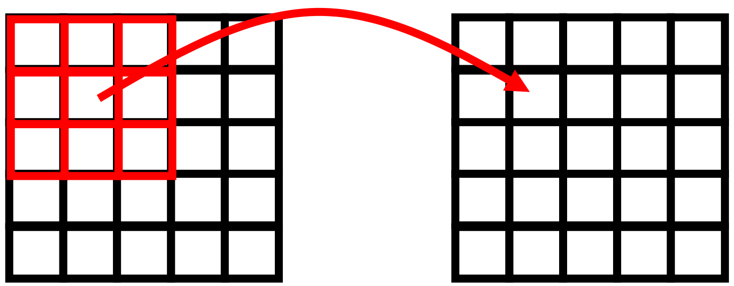
(fancier contrast normalization often used in practice)

Advantage: simple, size is preserved

Disadvantage: weird effect at boundary

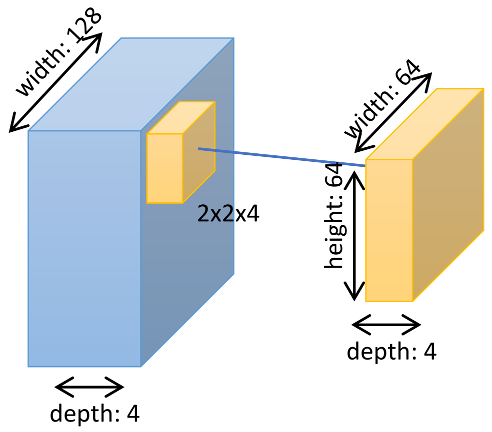
(this is usually not a problem, hence why this method is so popular)

Strided convolutions



standard conv net structure at each layer:

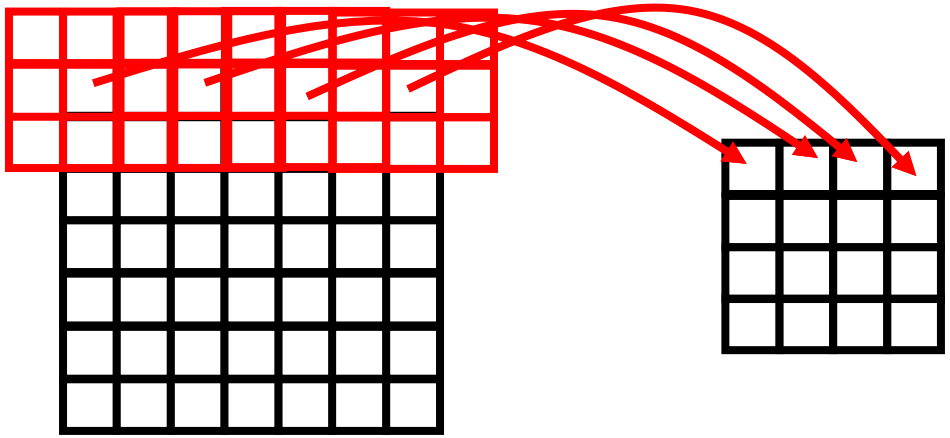
1. Apply conv, $H \times W \times C_{\text{in}} \rightarrow H \times W \times C_{\text{out}}$
2. Apply activation func σ, $H \times W \times C_{\text{out}} \rightarrow H \times W \times C_{\text{out}}$
3. Apply pooling (width N), $H \times W \times C_{\text{out}} \rightarrow H/N \times W/N \times C_{\text{out}}$



this can be very expensive computationally

$C_{\text{out}} \times C_{\text{in}}$ matrix multiply at each position in $H \times W$ image!

Idea: what if skip over some positions?



Amount of skipping is called the **stride**

Some people think that strided convolutions are just as good as conv + pooling

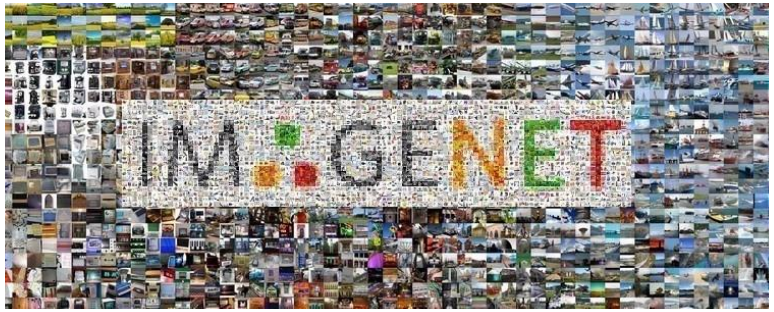
Examples of convolutional neural networks

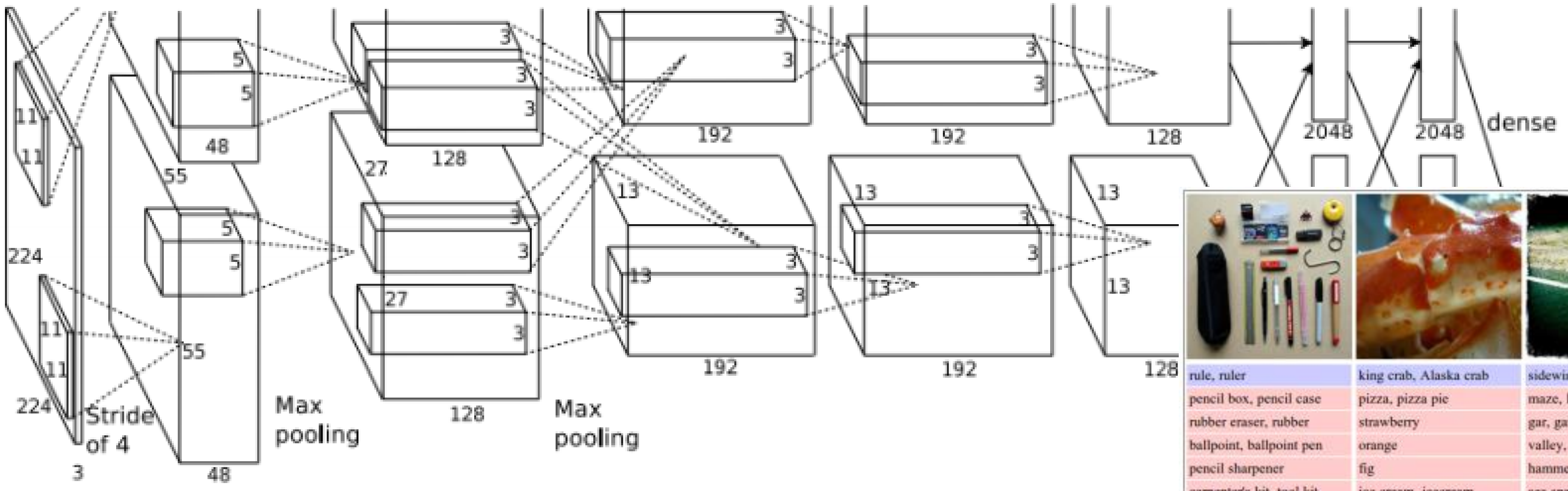
AlexNet

[Krizhevsky et al. 2012]

ILSVRC (ImageNet), 2009: 1.5 **million** images

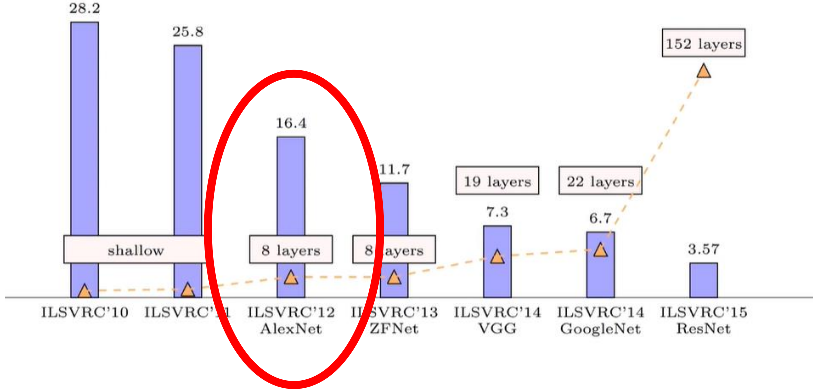
1000 categories





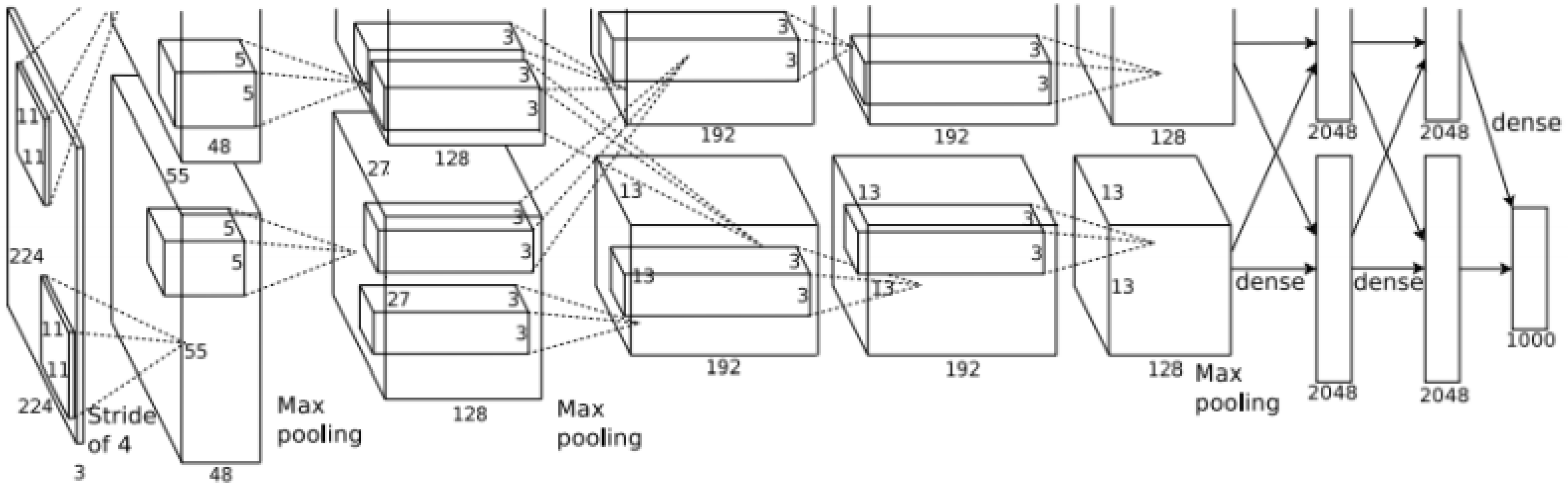
Why is this model important?

- “Classic” medium-depth convolutional network design (a bit like a modernized version of LeNet)
- Widely known for being the first neural network to attain state-of-the-art results on the ImageNet large-scale visual recognition challenge (ILSVRC)



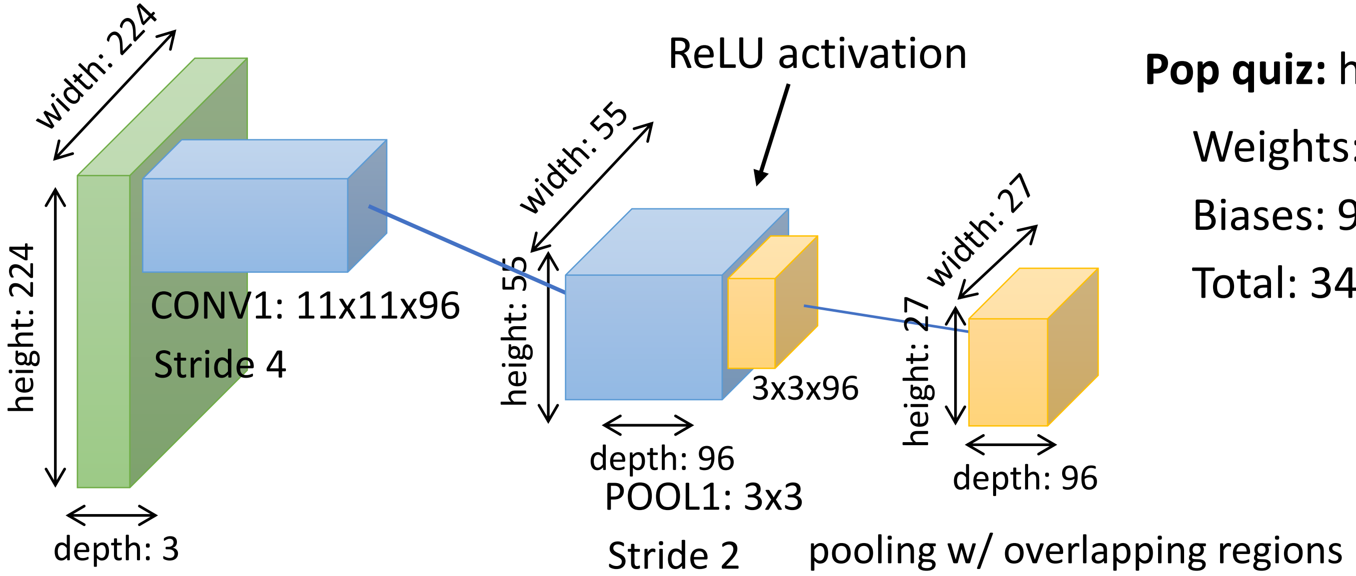
AlexNet

[Krizhevsky et al. 2012]



trained on two GPUs, hence why the diagram is "split"

... we don't worry about this sort of thing these days



Pop quiz: how many parameters in CONV1?

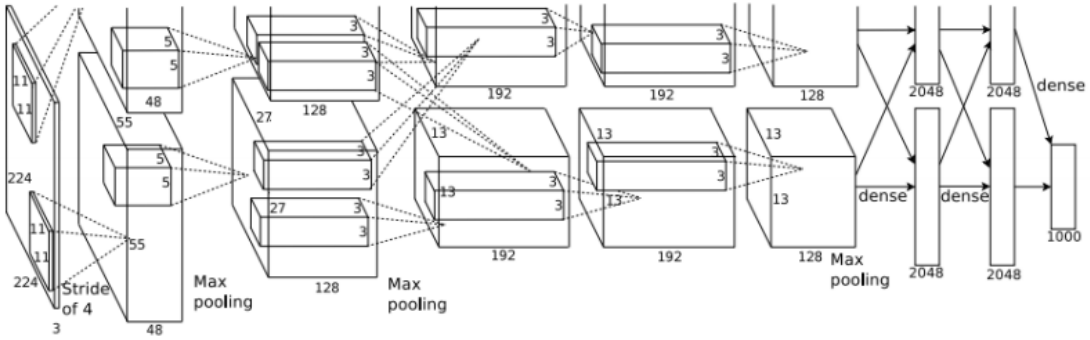
Weights: 11x11x3x96 = 34,848

Biases: 96

Total: 34,944

AlexNet

[Krizhevsky et al. 2012]



trained on two GPUs, hence why the diagram is “split”

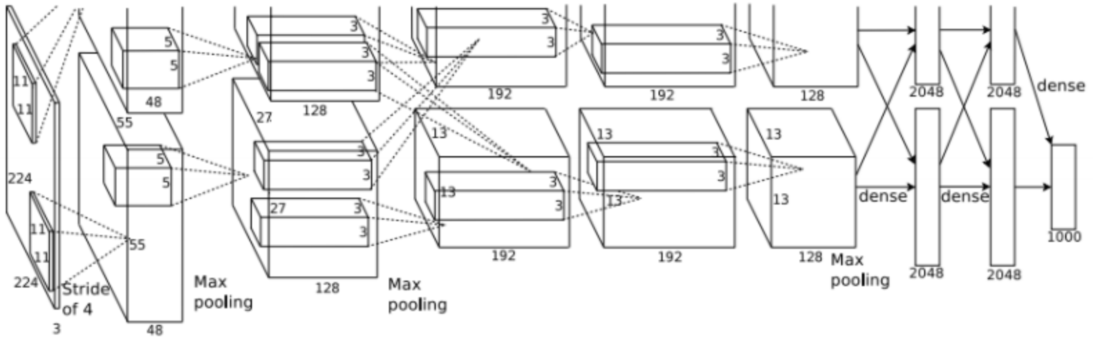
… we don’t worry about this sort of thing these days

CONV1: 11x11x96, Stride 4, maps 224x224x3 -> 55x55x96 [without zero padding]
POOL1: 3x3x96, Stride 2, maps 55x55x96 -> 27x27x96
NORM1: Local normalization layer [not widely used anymore, but we’ll talk about normalization later]
CONV2: 5x5x256, Stride 1, maps 27x27x96 -> 27x27x256 [**with** zero padding]
POOL2: 3x3x256, Stride 2, maps 27x27x256 -> 13x13x256
NORM2: Local normalization layer
CONV3: 3x3x384, Stride 1, maps 13x13x256 -> 13x13x384 [**with** zero padding]
CONV4: 3x3x384, Stride 1, maps 13x13x384 -> 13x13x384 [**with** zero padding]
CONV5: 3x3x256, Stride 1, maps 13x13x256 -> 13x13x256 [**with** zero padding]
POOL3: 3x3x256, Stride 2, maps 13x13x256 -> 6x6x256
FC6: 6x6x256 -> 9,216 -> 4,096 [matrix is 4,096 x 9,216]
FC7: 4,096 -> 4,096
FC8: 4,096 -> 1,000
SOFTMAX

AlexNet

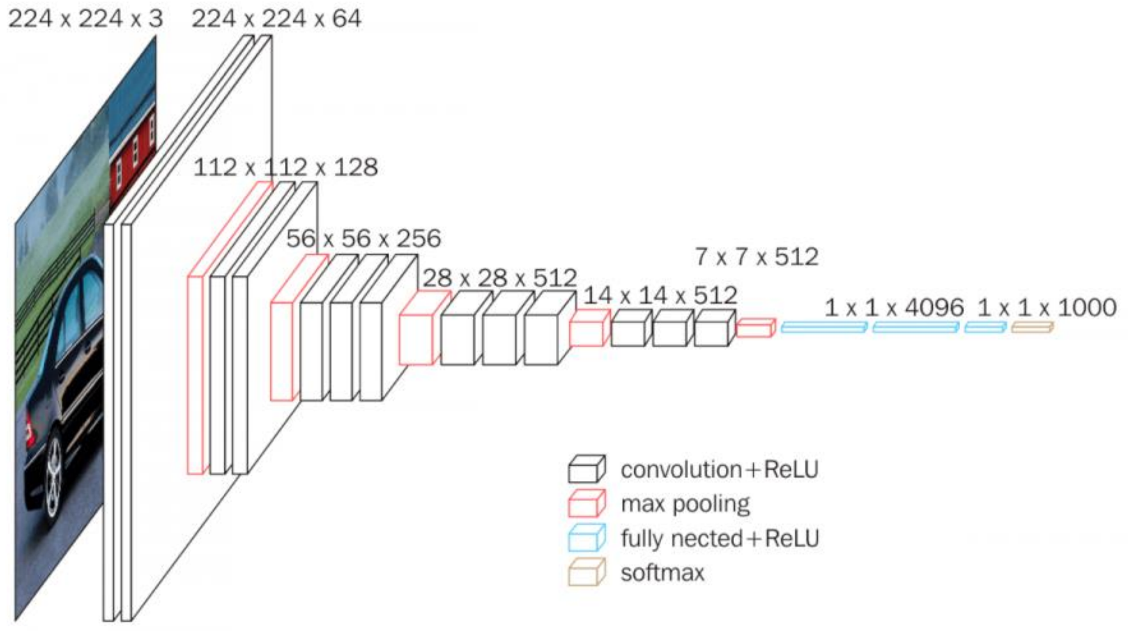
[Krizhevsky et al. 2012]

- Don't forget: ReLU nonlinearities after every CONV or FC layer (except the last one!)
- Trained with regularization (we'll learn about these later):
  - Data augmentation
  - Dropout
- Local normalization (not used much anymore, but there are other types of normalization we do use)



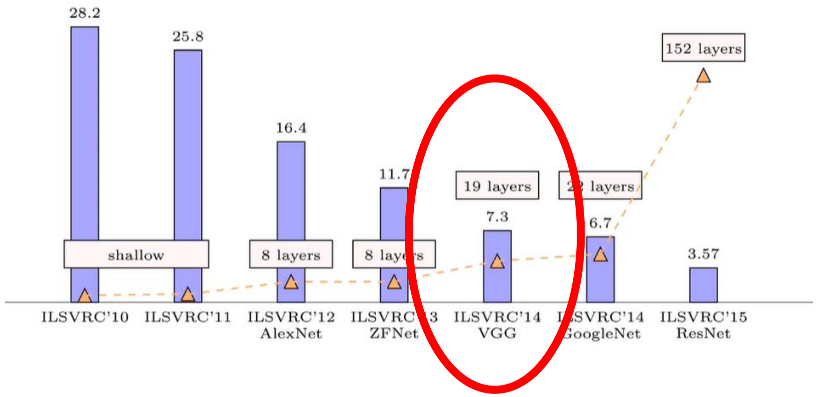
CONV1: 11x11x96, Stride 4, maps 224x224x3 -> 55x55x96 [without zero padding]
POOL1: 3x3x96, Stride 2, maps 55x55x96 -> 27x27x96
NORM1: Local normalization layer
CONV2: 5x5x256, Stride 1, maps 27x27x96 -> 27x27x256 [**with** zero padding]
POOL2: 3x3x256, Stride 2, maps 27x27x256 -> 13x13x256
NORM2: Local normalization layer
CONV3: 3x3x384, Stride 1, maps 13x13x256 -> 13x13x384 [**with** zero padding]
CONV4: 3x3x384, Stride 1, maps 13x13x384 -> 13x13x384 [**with** zero padding]
CONV5: 3x3x256, Stride 1, maps 13x13x256 -> 13x13x256 [**with** zero padding]
POOL3: 3x3x256, Stride 2, maps 13x13x256 -> 6x6x256
FC6: 6x6x256 -> 9,216 -> 4,096 [matrix is 4,096 x 9,216]
FC7: 4,096 -> 4,096
FC8: 4,096 -> 1,000
SOFTMAX

VGG

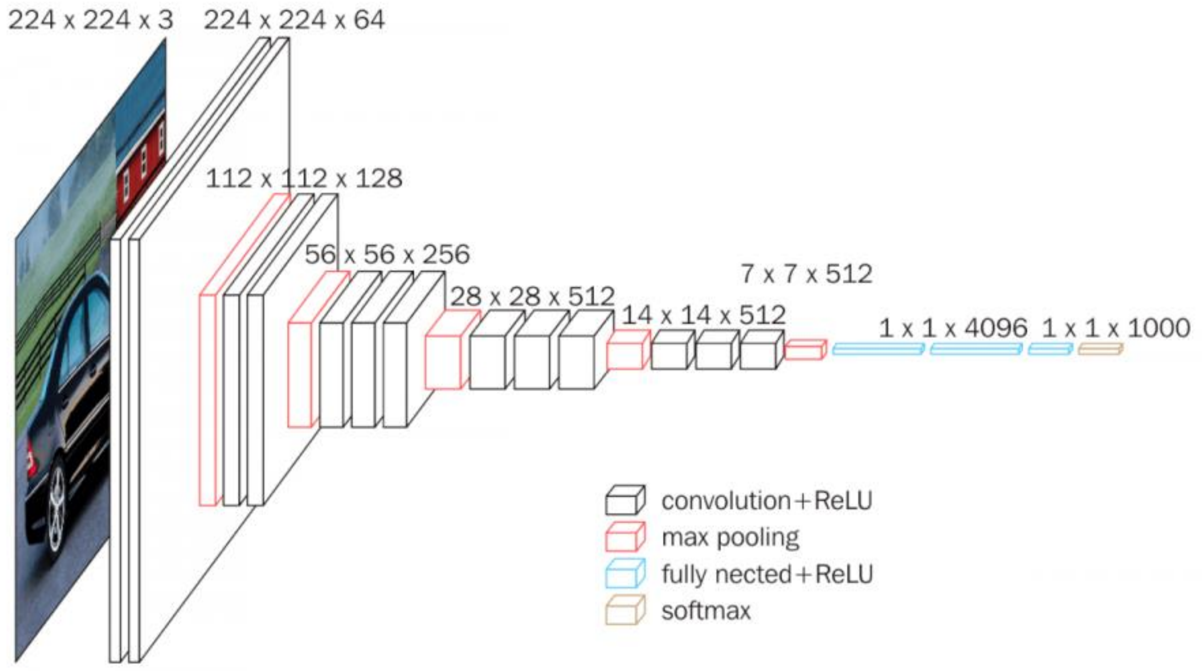


Why is this model important?

- ➢ Still often used today
- ➢ Big increase in **depth** over previous best model
- ➢ Start seeing "homogenous" stacks of multiple convolutions interspersed with resolution reduction



VGG



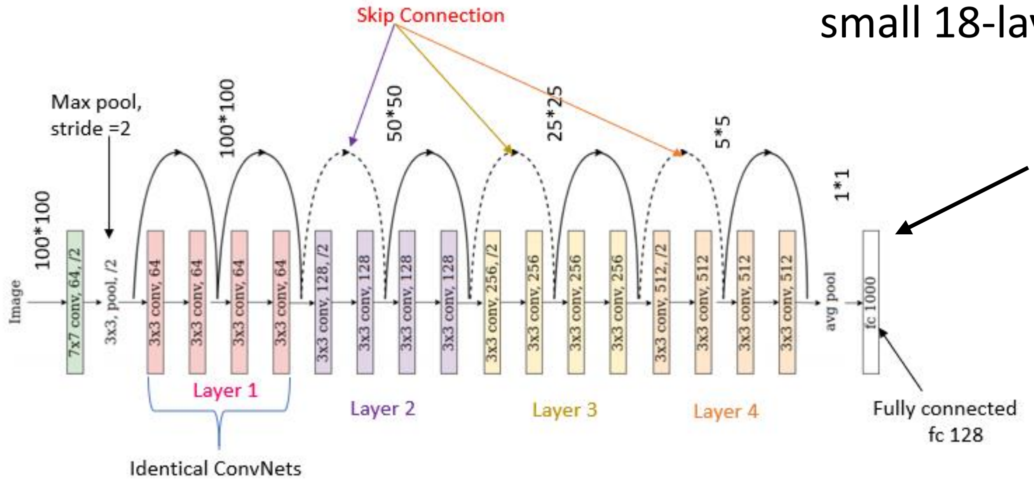
CONV: 3x3x64, maps 224x224x3 -> 224x224x64
CONV: 3x3x64, maps 224x224x64 -> 224x224x64
POOL: 2x2, maps 224x224x64 -> 112x112x64
CONV: 3x3x128, maps 112x112x64 -> 112x112x128
CONV: 3x3x128, maps 112x112x128 -> 112x112x128
POOL: 2x2, maps 112x112x128 -> 56x56x128
CONV: 3x3x256, maps 56x56x128 -> 56x56x256
CONV: 3x3x256, maps 56x56x256 -> 56x56x256
CONV: 3x3x256, maps 56x56x256 -> 56x56x256
POOL: 2x2, maps 56x56x256 -> 28x28x256
CONV: 3x3x512, maps 28x28x256 -> 28x28x512
CONV: 3x3x512, maps 28x28x512 -> 28x28x512
CONV: 3x3x512, maps 28x28x512 -> 28x28x512
POOL: 2x2, maps 28x28x512 -> 14x14x512
CONV: 3x3x512, maps 14x14x512 -> 14x14x512
CONV: 3x3x512, maps 14x14x512 -> 14x14x512
CONV: 3x3x512, maps 14x14x512 -> 14x14x512
POOL: 2x2, maps 14x14x512 -> 7x7x512
FC: 7x7x512 -> 25,088 -> 4,096 ← almost all parameters are here
FC: 4,096 -> 4,096
FC: 4,096 -> 1,000
SOFTMAX

- More layers = more processing, which is why we see repeated blocks
- Which parts use the most memory?
- Which parts have the most parameters?

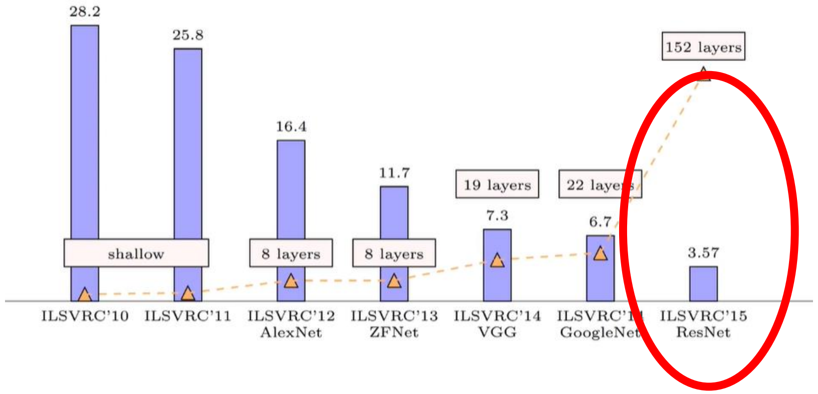
ResNet

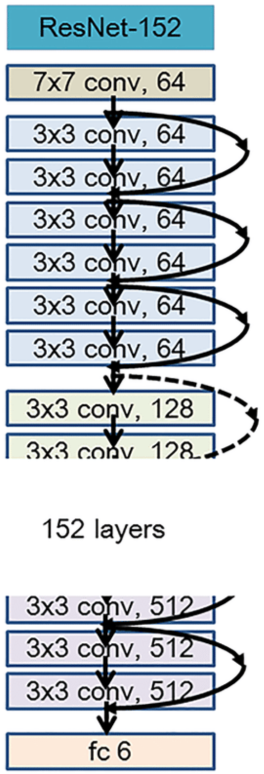
152 layers!

small 18-layer prototype (ResNet-18)



don’t bother with huge FC layer at the end, just average pool over all positions and have one linear layer

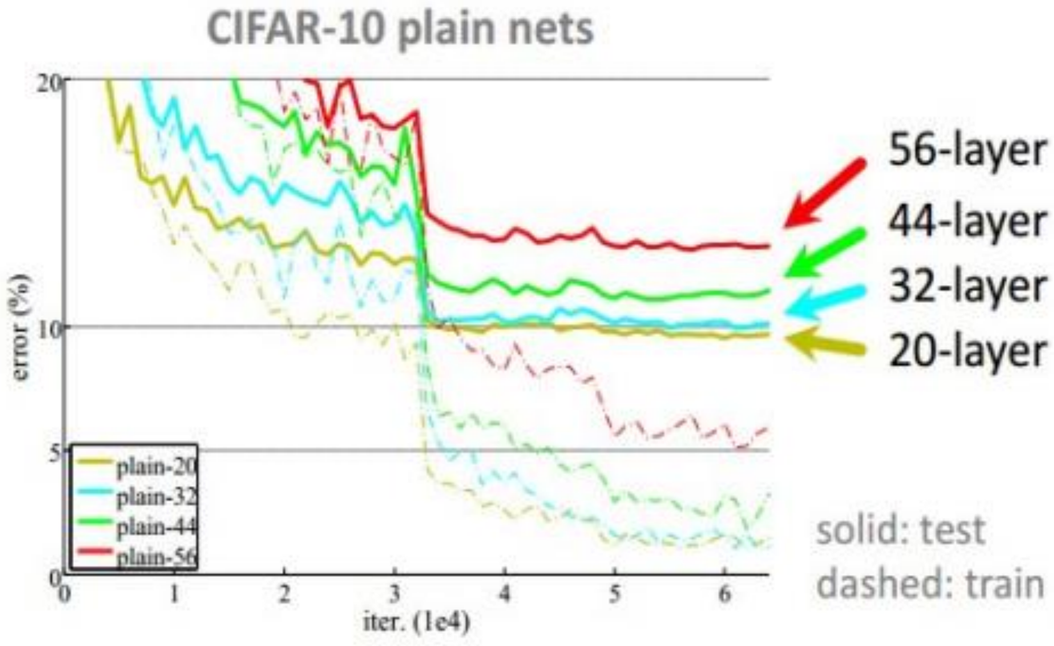


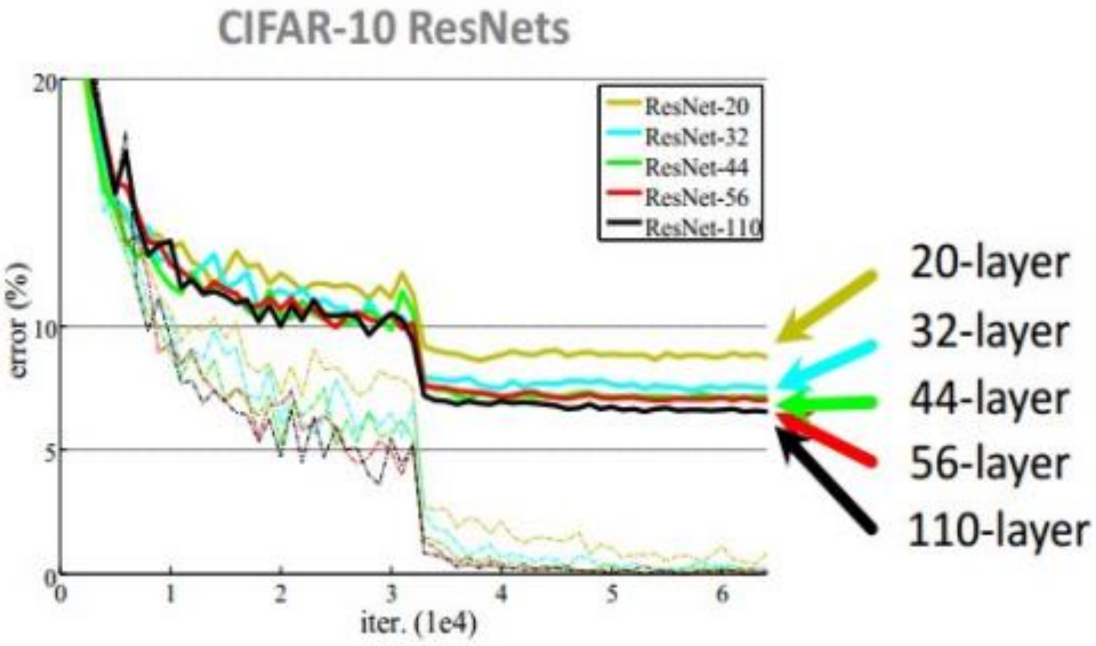


Images from He et al., and others

ResNet

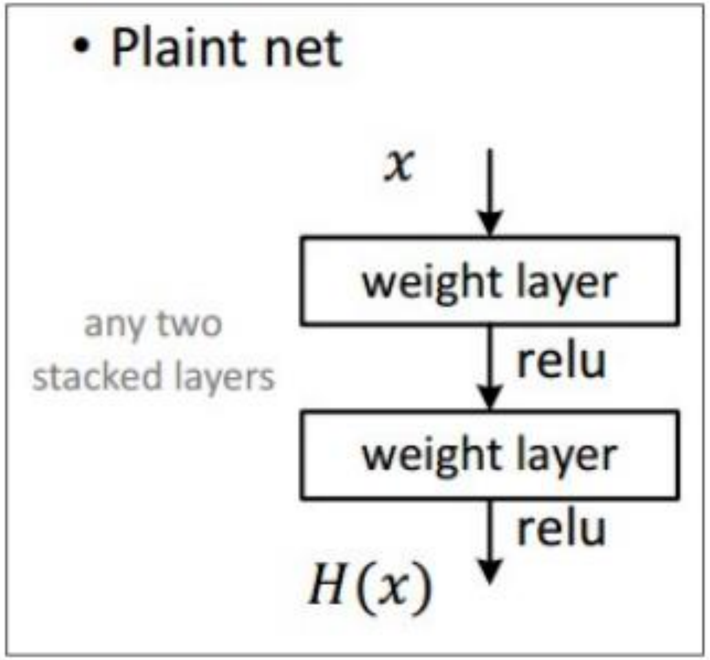
CIFAR-10 experiments

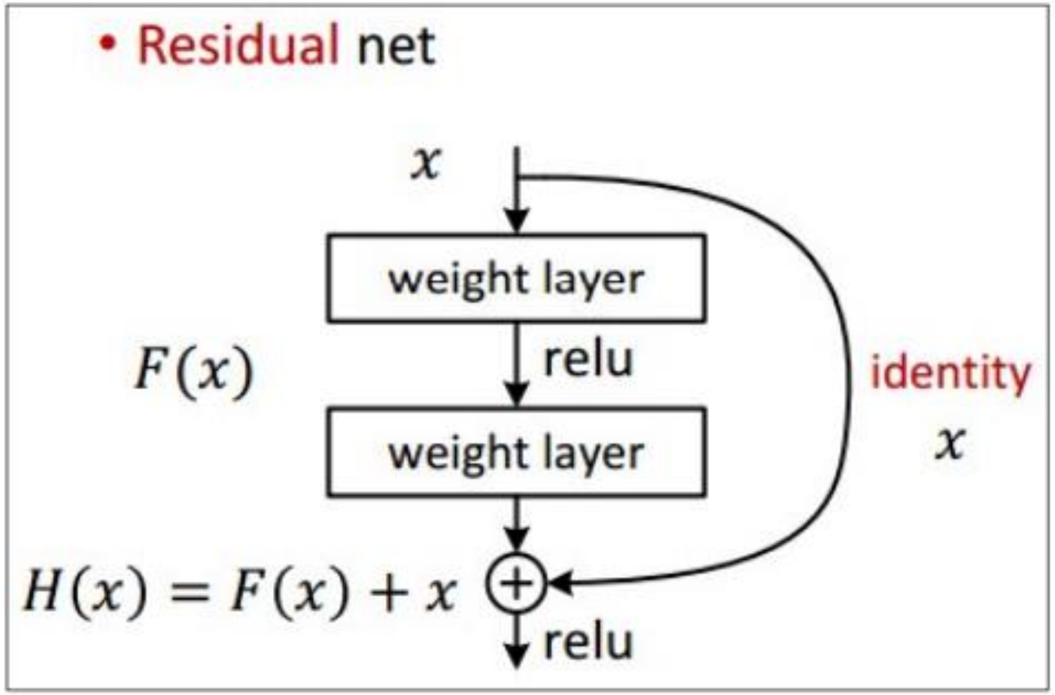




Images from He et al., and others

What’s the main idea?





Why is this a good idea?

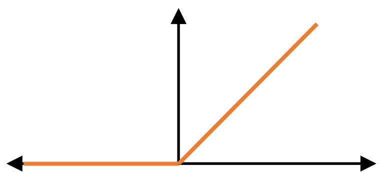
Images from He et al., and others

Why are deep networks hard to train?

$$\frac{d\mathcal{L}}{dW^{(1)}} = \frac{dz^{(1)}}{dW^{(1)}} \frac{da^{(1)}}{dz^{(1)}} \frac{dz^{(2)}}{da^{(1)}} \frac{d\mathcal{L}}{dz^{(2)}}$$

$$\frac{d\mathcal{L}}{dW^{(1)}} = J_1 J_2 J_3 \dots J_n \frac{d\mathcal{L}}{dz^{(n)}}$$

$$\text{ReLU:} \left(\frac{df}{dz}\right)_i = \mathrm{Ind}(z_i \geq 0)$$



If we multiply many many numbers together, what will we get?

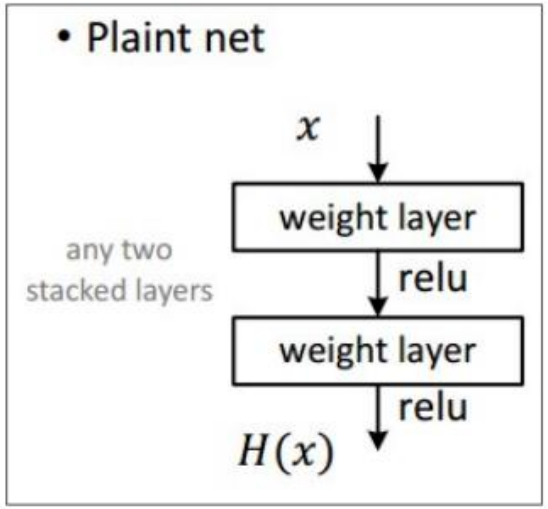
If most of the numbers are < 1, we get 0

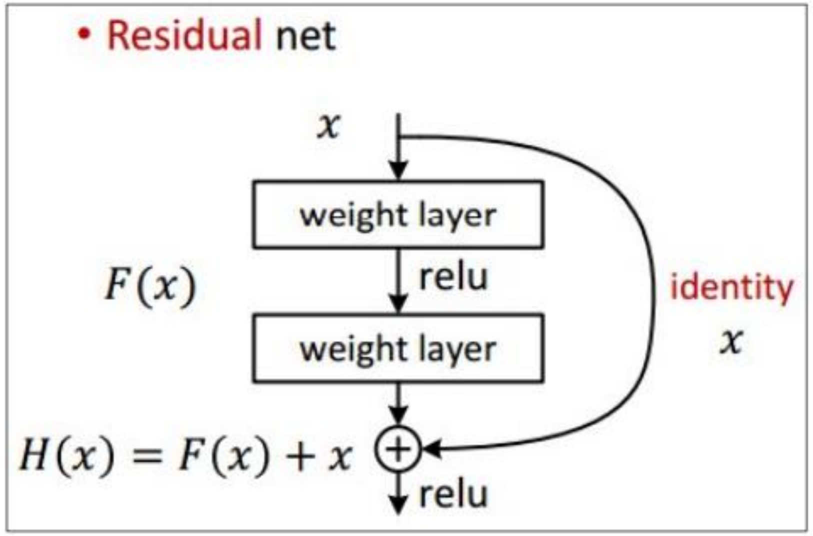
If most of the numbers are > 1, we get infinity

We only get a reasonable answer if the numbers are all close to 1!

For matrices, this means we want $J_i \approx \mathbf{I}$

So why is this a good idea?





$$\frac{dH}{dx}$$

could be **big** or **small**

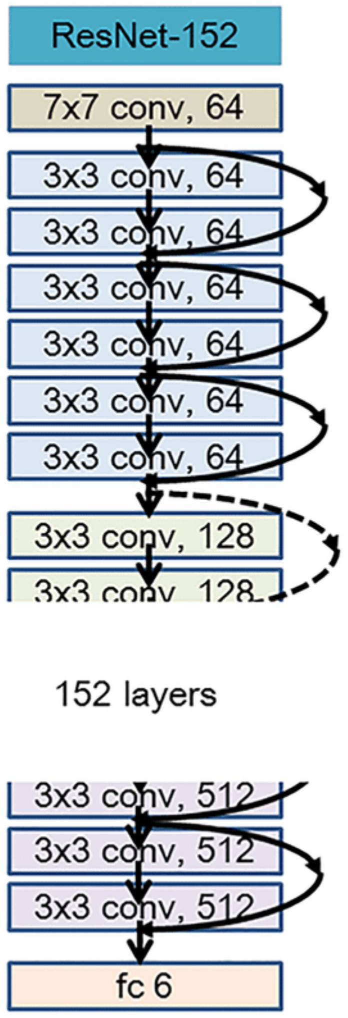
not close to $\mathbf{I}$

$$\frac{dH}{dx} = \frac{dF}{dx} + \mathbf{I}$$

If weights are not too big,
this will be small(ish)

Images from He et al., and others

ResNet



- "Generic" blocks with many layers, interspersed with a few pooling operations
- No giant FC layer at the end, just mean pool over all x/y positions and a small(ish) FC layer to go into the softmax
- Residual layers to provide for good gradient flow