# CNN Architectures

*Deep Learning*

Brad Quinton, Scott Chin

# CNN Architectures

- Ok so now we know about
  - Convolution Layers
  - Pooling Layers
- How do we put them together to make a good CNN architecture?
- How do we choose the various hyperparameter values?
- It’s useful to look at mainstream architectures
- Furthermore, a good place to start with a CNN solution, is to try a mainstream architecture

# Learning Objectives

- Look at successful CNN architectures to see how people tend to combine conv, pool, fully-connected layers to build CNNs and other innovations
- Learn some back-of-the-envelope metrics for comparing two architectures (# of FLOPs, # of parameters)
- Learn about Convolutional Layers that use 1x1 filters
- Get a little bit of a history lesson in Computer Vision Deep Learning
- Learn about Spatially-Separable and Depthwise-Separable Convolutions

# Overview

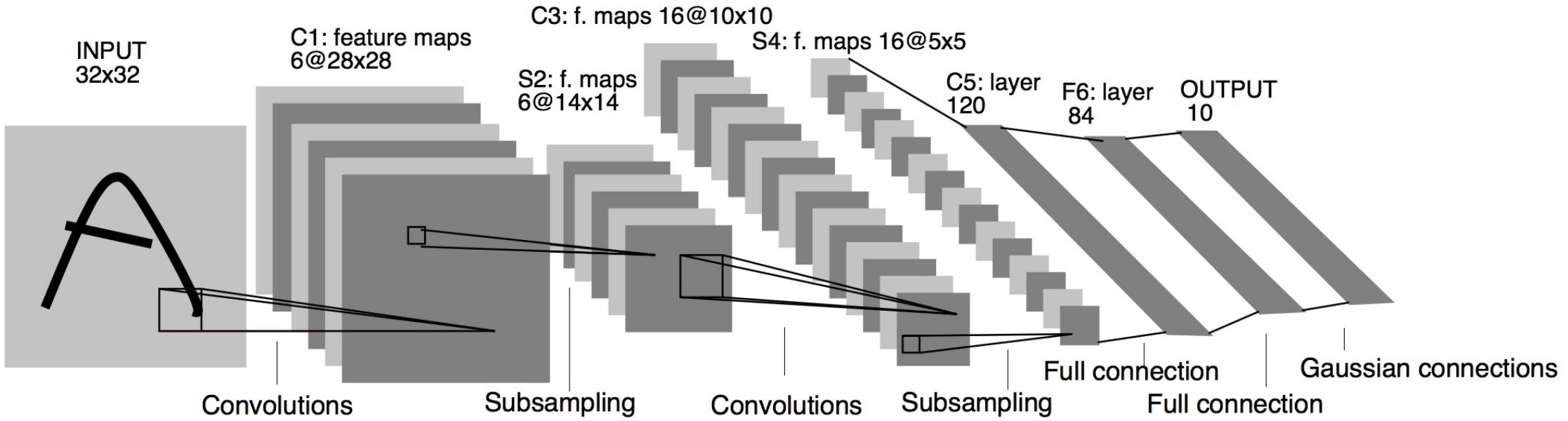
Famous Architectures (#Citations 2019, #Citations 2021)

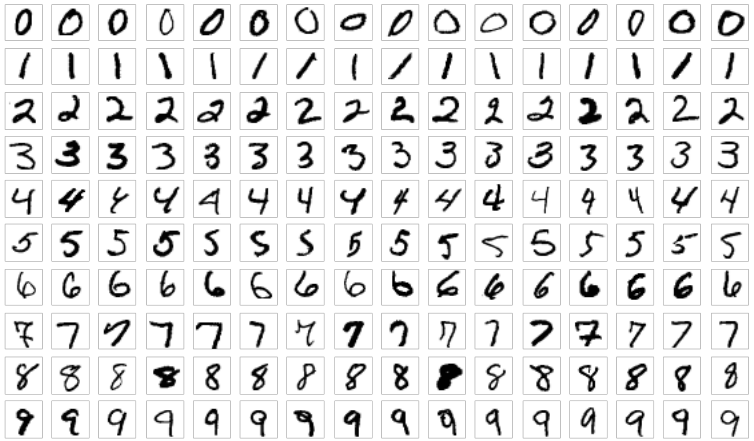
- LeNet (1998)– 24100 – 40764
- AlexNet (2012) – 55700 – 89497
- VGGNet (2014) – 33107 – 65972
- Inception (2014) – 18991 – 34786
- ResNets (2015) – 38314 - 93002
- EfficientNet (2019)

# LeNet (1998)

"Gradient-based learning applied to document recognition", LeCun, Bottou, Bengio, Haffner, 1998, http://vision.stanford.edu/cs598_spring07/papers/Lecun98.pdf

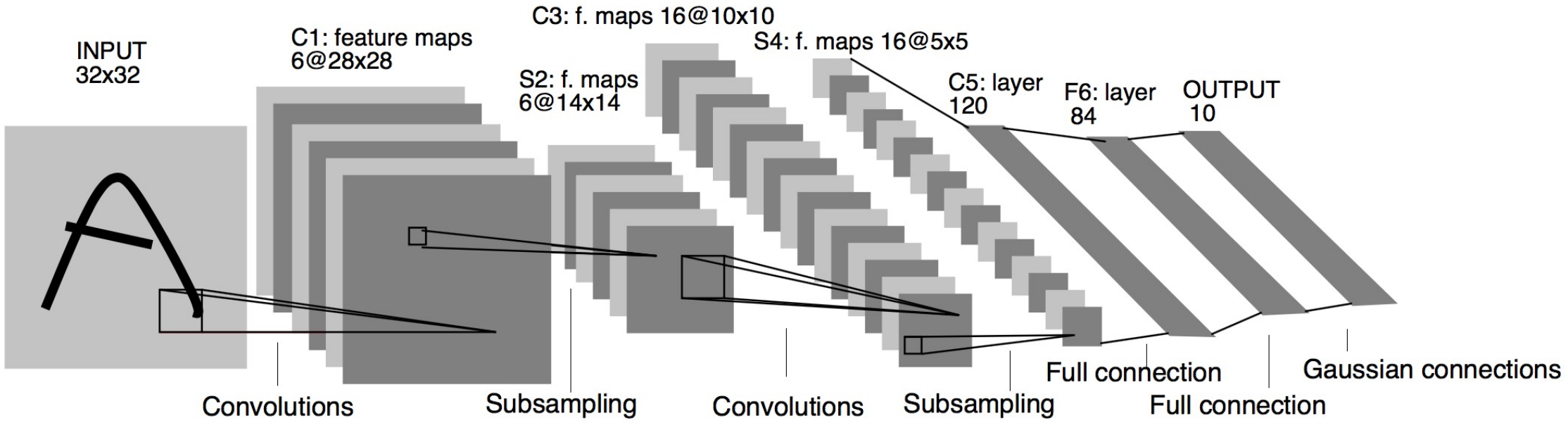
# LeNet Architecture



MNIST
Data Set

# LeNet Architecture




- A very regular and classic architecture where you stack convolutions and pooling a number of times, followed by some fully-connected layers

# Computational Resource Analysis

# Computational Resource Analysis

Need a way to compare two architectures in terms of

- Runtime Considerations
  - Once trained, how long to make a prediction (i.e. forward prop)?
  - How long will it take to train?
    - Backward prop computations
    - Number of parameters
- Memory Considerations
  - How much memory (e.g. GPU memory) will I need to
    - train this model (forward+backward prop)
    - make a prediction

# Number of floating point operations (FLOPs) for a Convolution Layer

- Convolution is a bunch of multiply-accumulate (MAC) operations. One MAC can be done in a single flop

# Number of floating point operations (FLOPs) for a Convolution Layer

- Convolution is a bunch of multiply-accumulate (MAC) operations. One MAC can be done in a single flop
- Given weights $(K, f, f, c_{in})$ and output of shape of $(h_{out}, w_{out}, c_{out})$
  - $(h_{out}, w_{out}, K)$ activations to compute
  - Each activation is a dot product between two $(f, f, c_{in})$ tensors
    → $f * f * c_{in}$ MACs
  - Total flops: $h_{out} * w_{out} * K * f * f * c_{in}$
    - Number of outputs * Number of flops to compute each output

# Number of floating point operations (FLOPs) for a Convolution Layer

- Convolution is a bunch of multiply-accumulate (MAC) operations. One MAC can be done in a single flop
- Given weights $(K, f, f, c_{in})$ and output of shape of $(h_{out}, w_{out}, c_{out})$
  - $(h_{out}, w_{out}, K)$ activations to compute
  - Each activation is a dot product between two $(f, f, c_{in})$ tensors
    → $f * f * c_{in}$ MACs
  - Total flops: $h_{out} * w_{out} * K * f * f * c_{in}$
    - Number of outputs * Number of flops to compute each output
- Example:
  - $(K, f, f, c_{in}) = (64,3,3,3)$ and output is $(225,225,64)$
  - Need $225 * 225 * 64 * 3 * 3 * 3 = 87.5$million flops

# Number of floating point operations (FLOPs) for Pooling Layer

- Givin a single $(f, f)$ region in which to pool,
  - Max pool is the comparison of $f * f$ numbers → $f * f$ flops
  - Avg pool is the addition of $f * f$ numbers → $f * f$ flops

# Number of floating point operations (FLOPs) for Pooling Layer

- Givin a single $(f, f)$ region in which to pool,
  - Max pool is the comparison of $f * f$ numbers → $f * f$ flops
  - Avg pool is the addition of $f * f$ numbers → $f * f$ flops
- Given a pooling layer with output shape $(h_{out}, w_{out}, c_{out})$
  - $h_{out} * w_{out} * c_{out}$ pooling regions to compute
  - Each pooling region needs $f * f$ flops
  - Total flops: $h_{out} * w_{out} * c_{out} * f * f$
    - Number of outputs * Number of flops to compute each output

# Number of floating point operations (FLOPs) for Pooling Layer

- Givin a single $(f, f)$ region in which to pool,
  - Max pool is the comparison of $f * f$ numbers → $f * f$ flops
  - Avg pool is the addition of $f * f$ numbers → $f * f$ flops
- Given a pooling layer with output shape $(h_{out}, w_{out}, c_{out})$
  - $h_{out} * w_{out} * c_{out}$ pooling regions to compute
  - Each pooling region needs $f * f$ flops
  - Total flops: $h_{out} * w_{out} * c_{out} * f * f$
    - Number of outputs * Number of flops to compute each output
- Example:
  - Pooling layer with $(2,2)$ pool regions and layer output is $(114,114,64)$
  - Need $114 * 114 * 64 * 2 * 2 =$ 3.3 million flops

# Number of floating point operations (FLOPs) for Fully Connected Layer

- Layer has $n_h^{[l]}$ units and $n_h^{[l-1]}$ inputs

# Number of floating point operations (FLOPs) for Fully Connected Layer

- Layer has $n_h^{[l]}$ units and $n_h^{[l-1]}$ inputs
- Output of each unit is weighted sum of $n_h^{[l-1]}$ numbers
  → $n_h^{[l-1]}$ MACs
- Output of all units is $n_h^{[l]} * n_h^{[l-1]}$

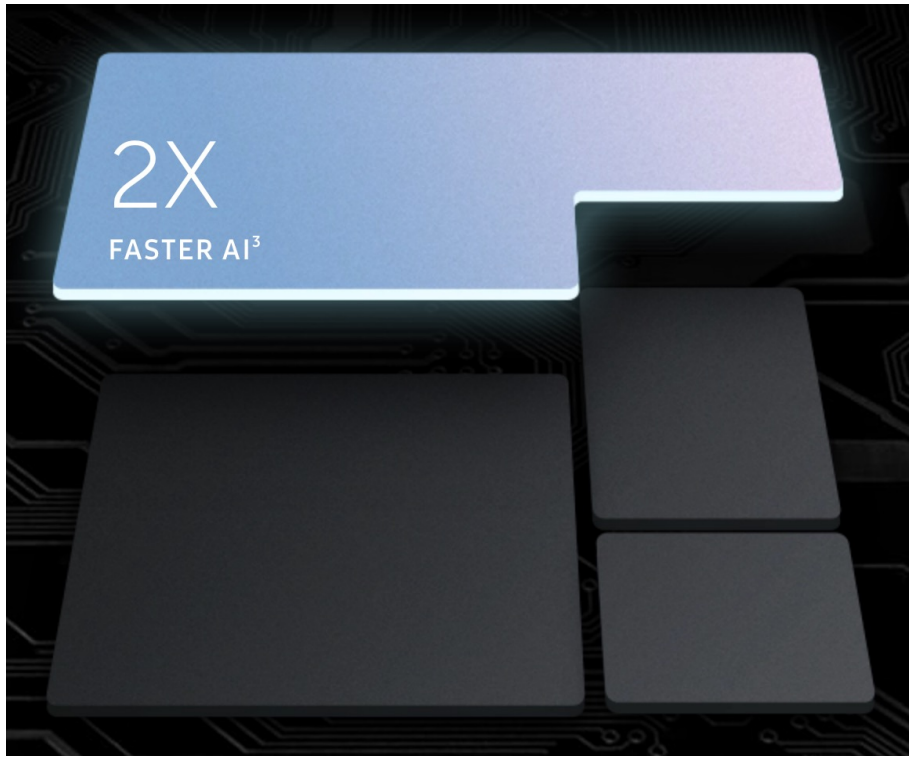
# Number of floating point operations (FLOPs) for Fully Connected Layer

- Layer has $n_h^{[l]}$ units and $n_h^{[l-1]}$ inputs
- Output of each unit is weighted sum of $n_h^{[l-1]}$ numbers
  → $n_h^{[l-1]}$ MACs
- Output of all units is $n_h^{[l]} * n_h^{[l-1]}$
- Example:
  - Layer with 4096 units and 4096 inputs
  - Need $4096 * 4096 = 16.8$ million flops

# Important! FLOPs vs Throughput, Latency, Power, Energy

- In reality, you really care how the “FLOPs” run on hardware
- But this depends on a lot of implementation details
  - Hardware Architecture
    - E.g. CPU/GPU/custom silicon
  - The way you write your code
    - Level of vectorization
    - Code-level optimizations
  - Compiler
    - How your code is mapped to the underlying hardware
- Your real metrics are things like throughput/latency/energy/power of your real application
- But for an academic analysis, #FLOPs is a reasonable proxy metric

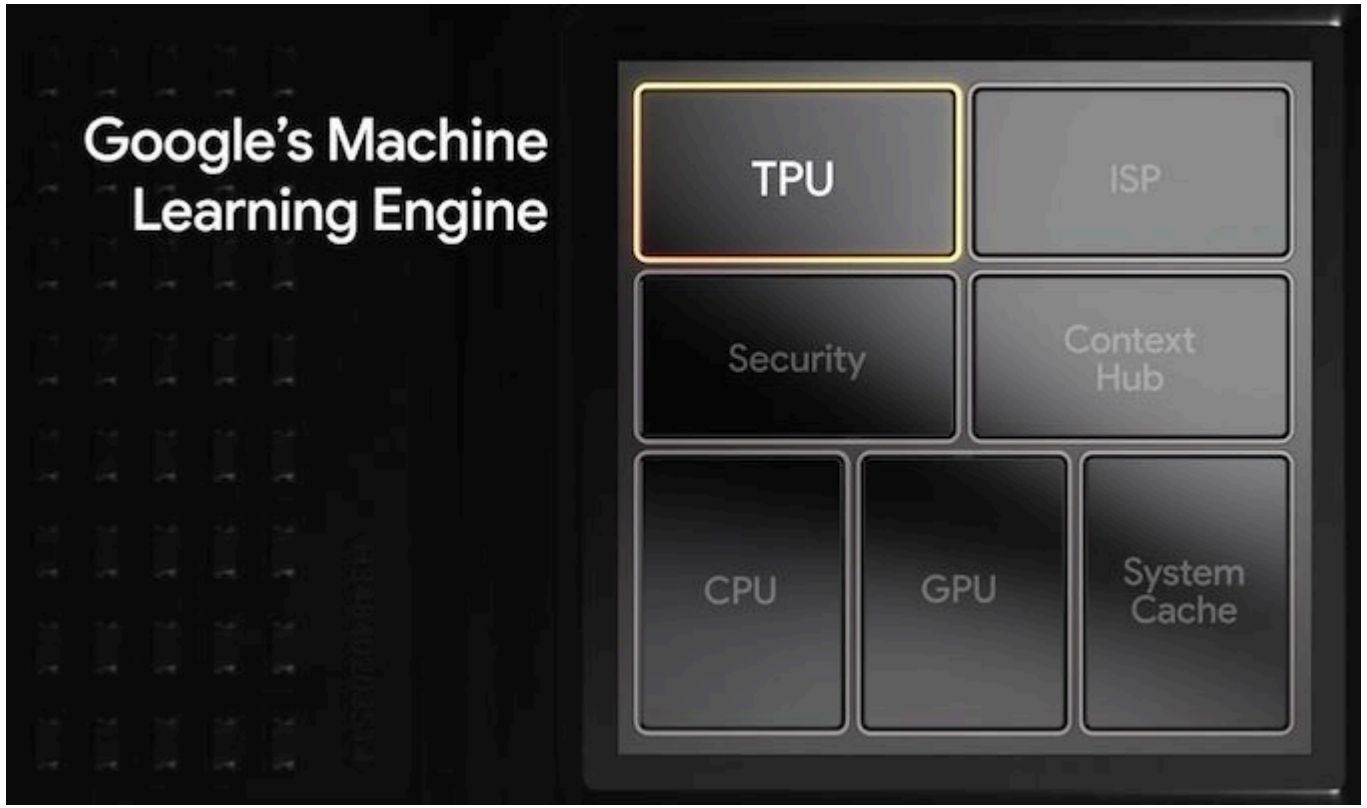
# Big Tech


- Apple A14 Processor dedicates almost 25% of the die to the Neural Engine!
- Qualcomm SnapDragon 888 dedicates almost 30% to the AI Engine


Brad Quinton, Scott Chin
https://www.samsung.com/ca/smartphones/galaxy-s21-ultra-5g/; https://www.androidauthority.com/apple-a14-bionic-chip-specs-explained-1159065/
Brad Quinton, Scott Chin

# Big Tech




https://www.anandtech.com/show/17032/tensor-soc-performance-efficiency

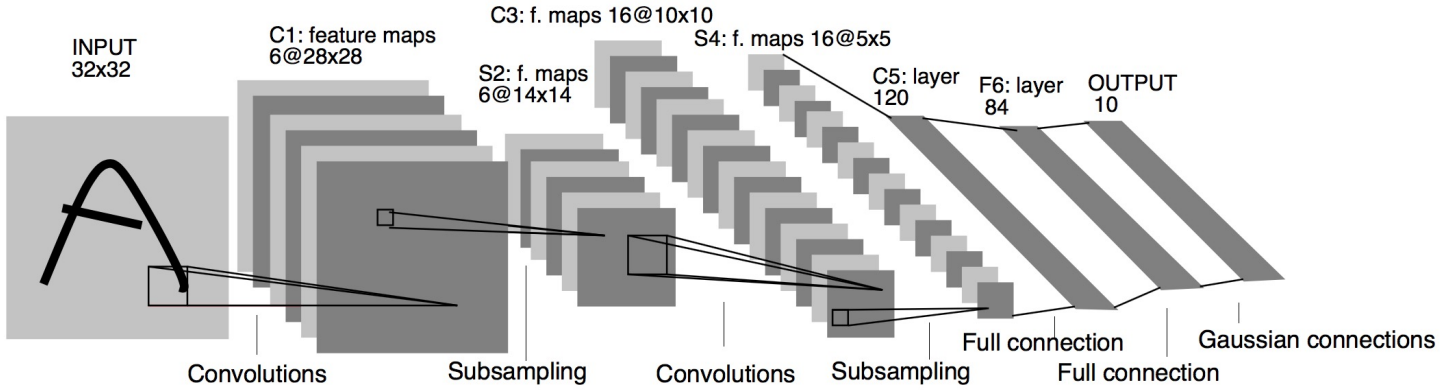
# Review: Number of Parameters per Layer

- Convolutional Layer:
  - ???
- MaxPooling Layer:
  - ???
- Fully Connected:
  - ???

# Review: Number of Parameters per Layer

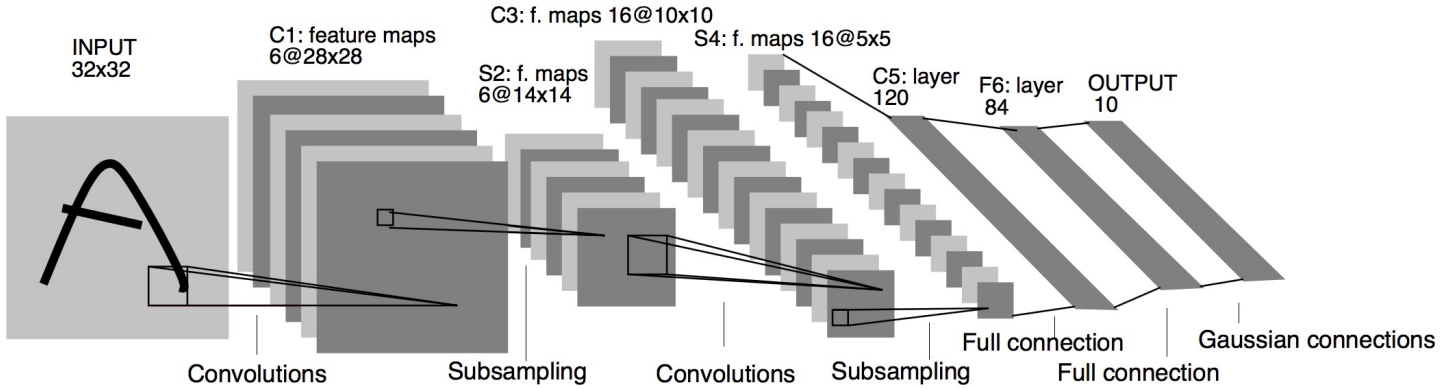
- Convolutional Layer:
  - $K(f * f * c_{in} + 1)$
- MaxPooling Layer:
  - 0
- Fully Connected:
  - $n_h^{[l]}\left(n_h^{[l-1]} + 1\right)$

# LeNet Analysis



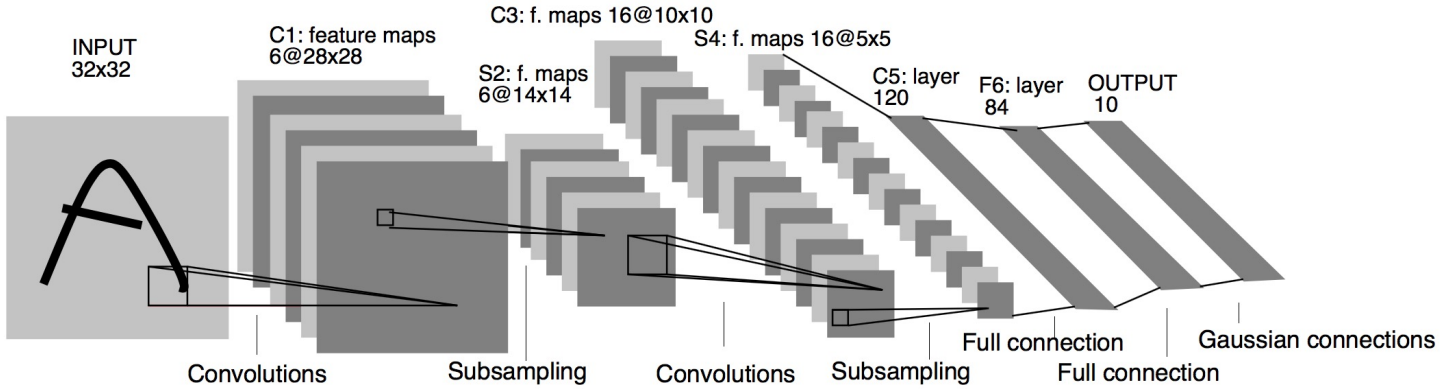

| Layer | HyperParams | Output Volume | # Parameters | flops |
|---|---|---|---|---|
| Input | | (32,32,1) | | |
| Conv | K=6, f=(5,5), s=(1,1) | | | |
| Avg. Pool | f=(2,2), s=(2,2) | | | |
| Conv | K=16, f=(5,5), s=(1,1) | | | |
| Avg. Pool | f=(2,2), s=(2,2) | | | |
| Flatten | | | | |
| FC | 120 units | | | |
| FC | 84 units | | | |
| FC (Output) | 10 units | | | |

# LeNet Analysis



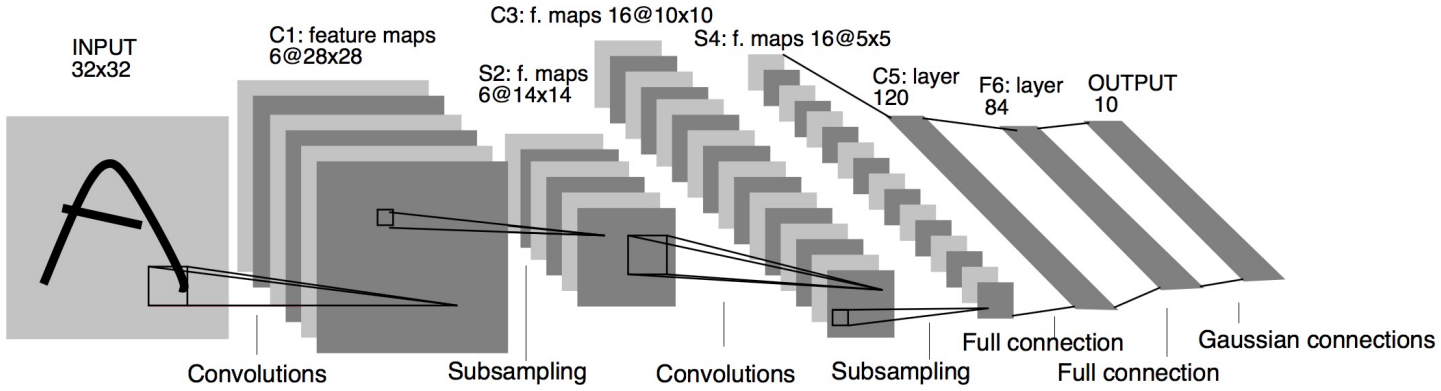

| Layer | HyperParams | Output Volume | # Parameters | flops |
|---|---|---|---|---|
| Input | | (32,32,1) | | |
| Conv | K=6, f=(5,5), s=(1,1) | ??? | | |
| Avg. Pool | f=(2,2), s=(2,2) | | | |
| Conv | K=16, f=(5,5), s=(1,1) | | | |
| Avg. Pool | f=(2,2), s=(2,2) | | | |
| Flatten | | | | |
| FC | 120 units | | | |
| FC | 84 units | | | |
| FC (Output) | 10 units | | | |

# LeNet Analysis



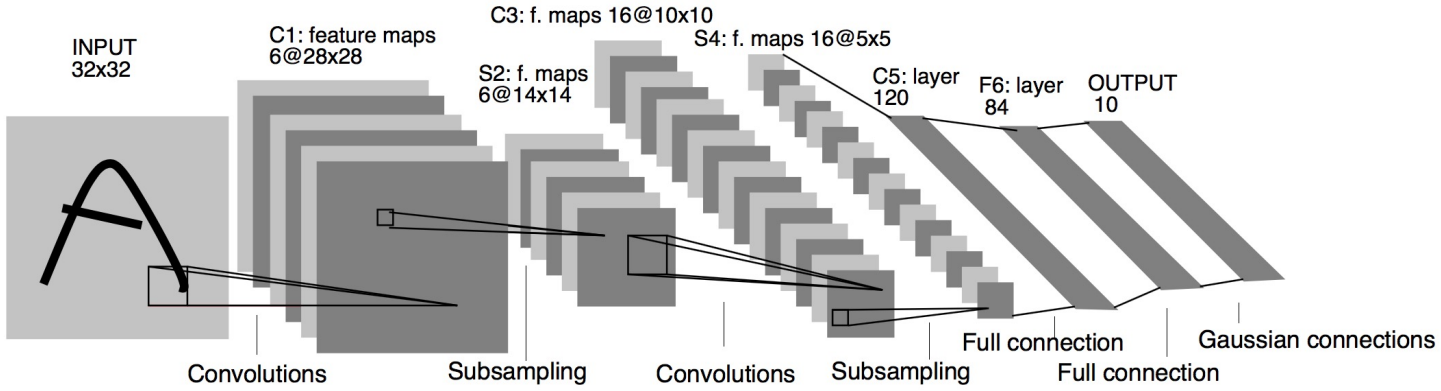

| Layer | HyperParams | Output Volume | # Parameters | flops |
|---|---|---|---|---|
| Input | | (32,32,1) | | |
| Conv | K=6, f=(5,5), s=(1,1) | (28,28,6) | | |
| Avg. Pool | f=(2,2), s=(2,2) | ??? | | |
| Conv | K=16, f=(5,5), s=(1,1) | | | |
| Avg. Pool | f=(2,2), s=(2,2) | | | |
| Flatten | | | | |
| FC | 120 units | | | |
| FC | 84 units | | | |
| FC (Output) | 10 units | | | |

# LeNet Analysis



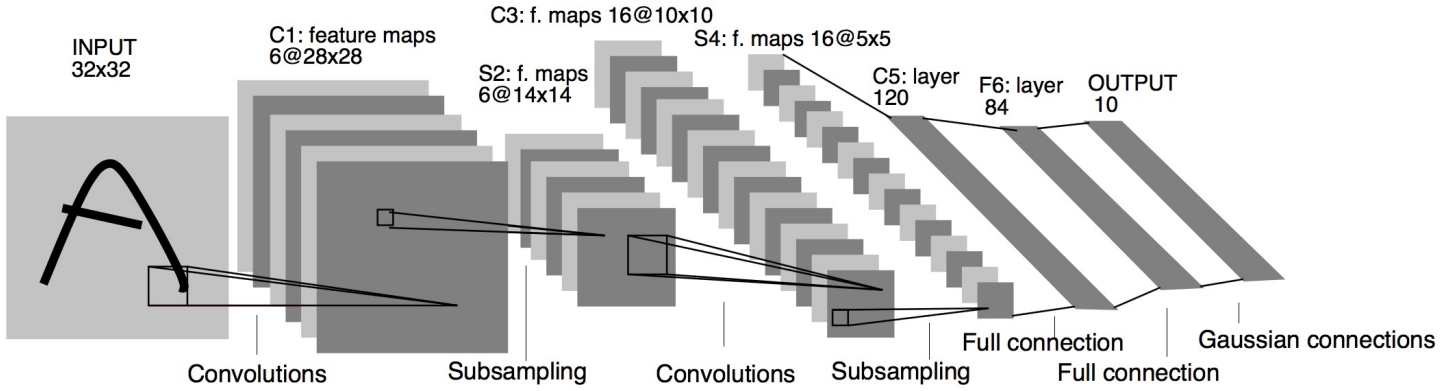

| Layer | HyperParams | Output Volume | # Parameters | flops |
|---|---|---|---|---|
| Input | | (32,32,1) | | |
| Conv | K=6, f=(5,5), s=(1,1) | (28,28,6) | | |
| Avg. Pool | f=(2,2), s=(2,2) | (14,14,6) | | |
| Conv | K=16, f=(5,5), s=(1,1) | (10,10,16) | | |
| Avg. Pool | f=(2,2), s=(2,2) | (5,5,16) | | |
| Flatten | | ??? | | |
| FC | 120 units | | | |
| FC | 84 units | | | |
| FC (Output) | 10 units | | | |

# LeNet Analysis



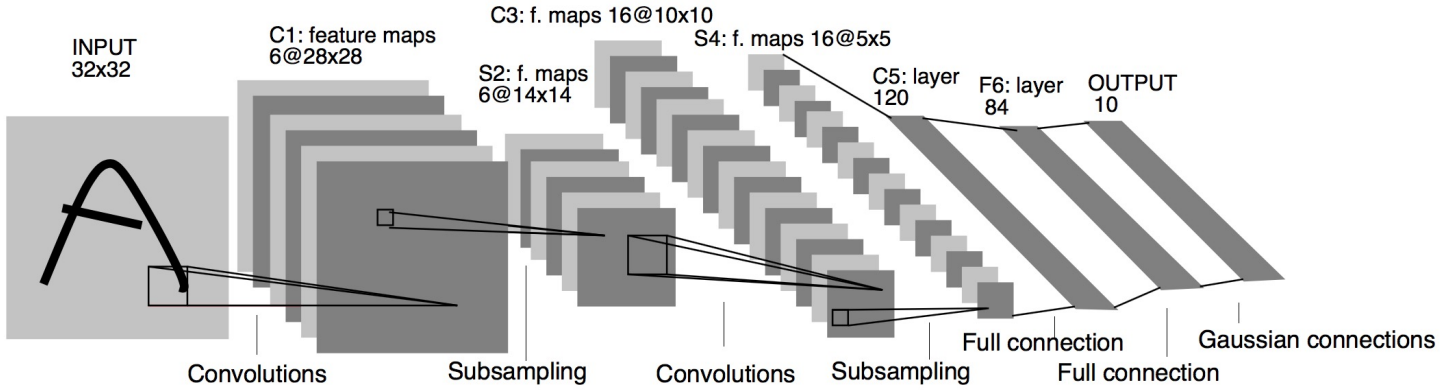

| Layer | HyperParams | Output Volume | # Parameters | flops |
|---|---|---|---|---|
| Input | | (32,32,1) | | |
| Conv | K=6, f=(5,5), s=(1,1) | (28,28,6) | | |
| Avg. Pool | f=(2,2), s=(2,2) | (14,14,6) | | |
| Conv | K=16, f=(5,5), s=(1,1) | (10,10,16) | | |
| Avg. Pool | f=(2,2), s=(2,2) | (5,5,16) | | |
| Flatten | | (400,) | | |
| FC | 120 units | (120,) | | |
| FC | 84 units | (84,) | | |
| FC (Output) | 10 units | (10,) | | |

# LeNet Analysis



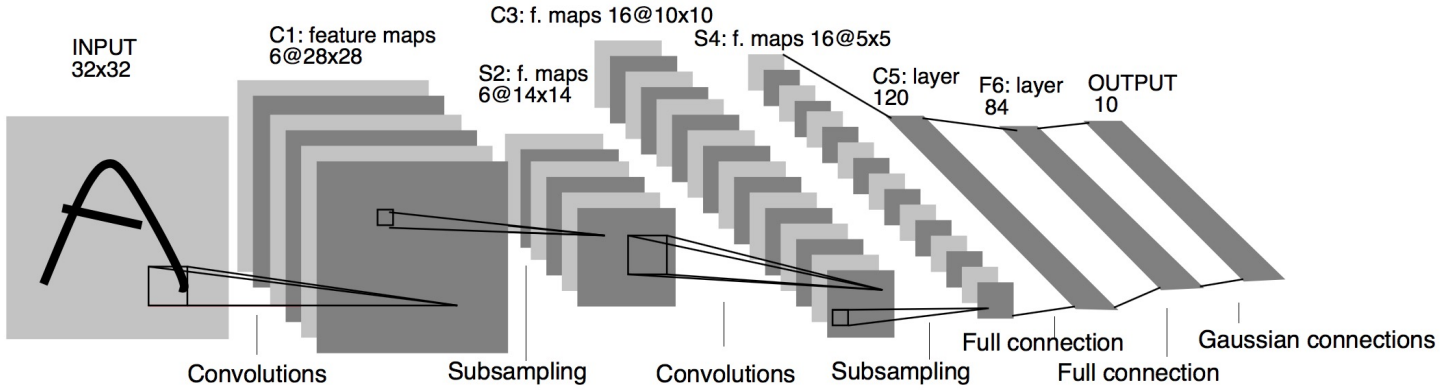

| Layer | HyperParams | Output Volume | # Parameters | flops |
|---|---|---|---|---|
| Input | | (32,32,1) | | |
| Conv | K=6, f=(5,5), s=(1,1) | (28,28,6) | ??? | |
| Avg. Pool | f=(2,2), s=(2,2) | (14,14,6) | | |
| Conv | K=16, f=(5,5), s=(1,1) | (10,10,16) | | |
| Avg. Pool | f=(2,2), s=(2,2) | (5,5,16) | | |
| Flatten | | (400,) | | |
| FC | 120 units | (120,) | | |
| FC | 84 units | (84,) | | |
| FC (Output) | 10 units | (10,) | | |

# LeNet Analysis



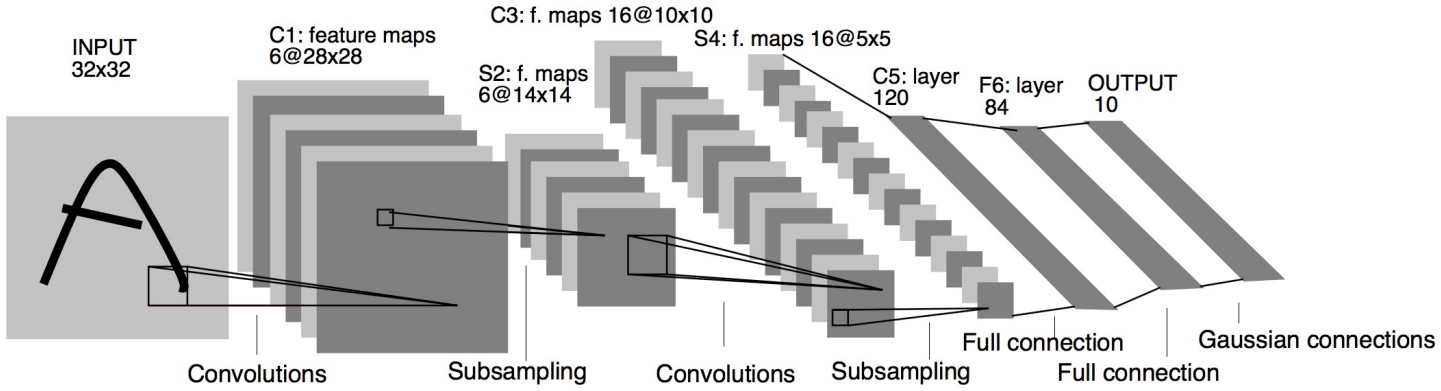

| Layer | HyperParams | Output Volume | # Parameters | flops |
|---|---|---|---|---|
| Input | | (32,32,1) | | |
| Conv | K=6, f=(5,5), s=(1,1) | (28,28,6) | 6*(5*5*1+1)=156 | |
| Avg. Pool | f=(2,2), s=(2,2) | (14,14,6) | ??? | |
| Conv | K=16, f=(5,5), s=(1,1) | (10,10,16) | | |
| Avg. Pool | f=(2,2), s=(2,2) | (5,5,16) | | |
| Flatten | | (400,) | | |
| FC | 120 units | (120,) | | |
| FC | 84 units | (84,) | | |
| FC (Output) | 10 units | (10,) | | |

# LeNet Analysis



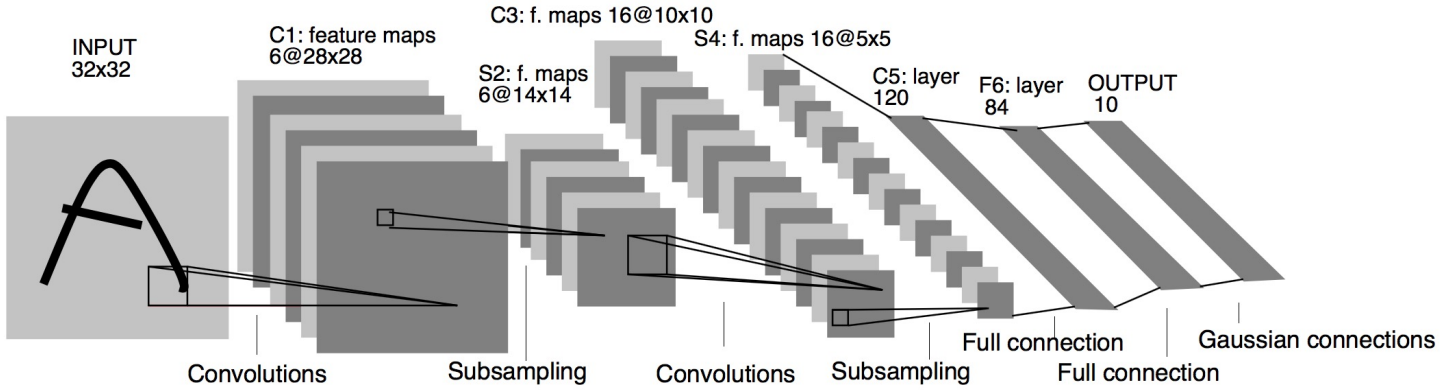

| Layer | HyperParams | Output Volume | # Parameters | flops |
|---|---|---|---|---|
| Input | | (32,32,1) | | |
| Conv | K=6, f=(5,5), s=(1,1) | (28,28,6) | 6*(5*5*1+1)=156 | |
| Avg. Pool | f=(2,2), s=(2,2) | (14,14,6) | | |
| Conv | K=16, f=(5,5), s=(1,1) | (10,10,16) | ??? | |
| Avg. Pool | f=(2,2), s=(2,2) | (5,5,16) | | |
| Flatten | | (400,) | | |
| FC | 120 units | (120,) | | |
| FC | 84 units | (84,) | | |
| FC (Output) | 10 units | (10,) | | |

# LeNet Analysis



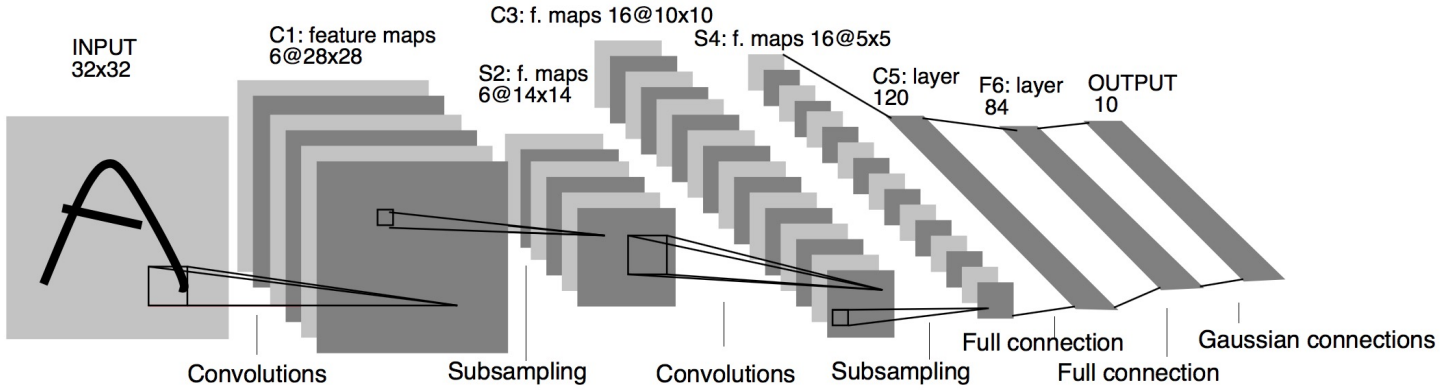

| Layer | HyperParams | Output Volume | # Parameters | flops |
|---|---|---|---|---|
| Input | | (32,32,1) | | |
| Conv | K=6, f=(5,5), s=(1,1) | (28,28,6) | 6*(5*5*1+1)=156 | |
| Avg. Pool | f=(2,2), s=(2,2) | (14,14,6) | | |
| Conv | K=16, f=(5,5), s=(1,1) | (10,10,16) | 16*(5*5*6+1)=2416 | |
| Avg. Pool | f=(2,2), s=(2,2) | (5,5,16) | | |
| Flatten | | (400,) | ??? | |
| FC | 120 units | (120,) | | |
| FC | 84 units | (84,) | | |
| FC (Output) | 10 units | (10,) | | |

# LeNet Analysis



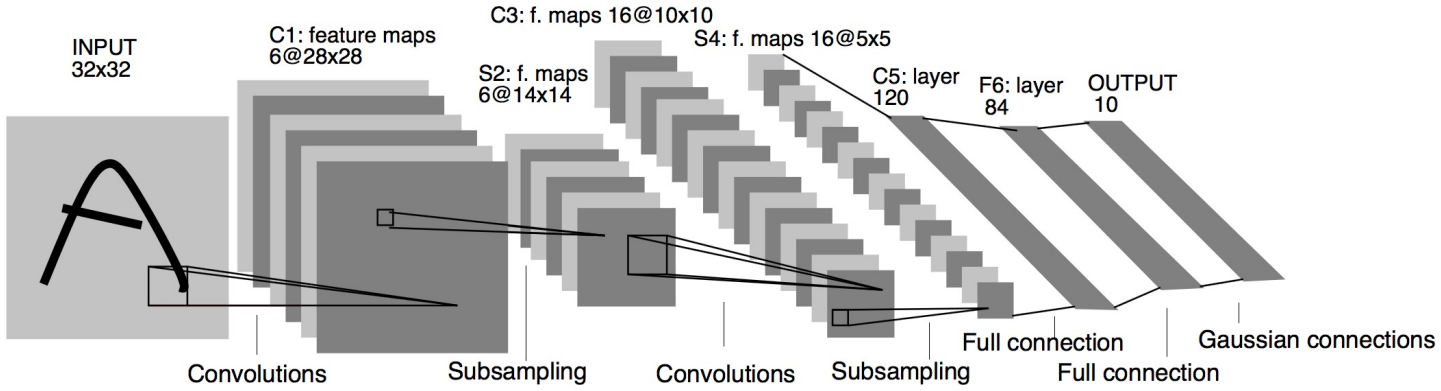

| Layer | HyperParams | Output Volume | # Parameters | flops |
|---|---|---|---|---|
| Input | | (32,32,1) | | |
| Conv | K=6, f=(5,5), s=(1,1) | (28,28,6) | 6*(5*5*1+1)=156 | |
| Avg. Pool | f=(2,2), s=(2,2) | (14,14,6) | | |
| Conv | K=16, f=(5,5), s=(1,1) | (10,10,16) | 16*(5*5*6+1)=2416 | |
| Avg. Pool | f=(2,2), s=(2,2) | (5,5,16) | | |
| Flatten | | (400,) | | |
| FC | 120 units | (120,) | ??? | |
| FC | 84 units | (84,) | | |
| FC (Output) | 10 units | (10,) | | |

# LeNet Analysis



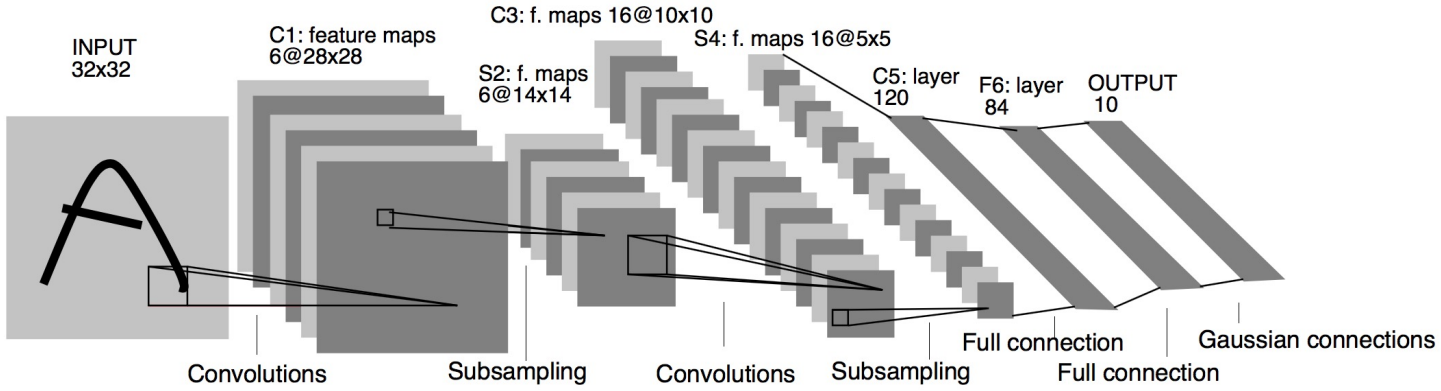

| Layer | HyperParams | Output Volume | # Parameters | flops |
|---|---|---|---|---|
| Input | | (32,32,1) | | |
| Conv | K=6, f=(5,5), s=(1,1) | (28,28,6) | 6*(5*5*1+1)=156 | |
| Avg. Pool | f=(2,2), s=(2,2) | (14,14,6) | | |
| Conv | K=16, f=(5,5), s=(1,1) | (10,10,16) | 16*(5*5*6+1)=2416 | |
| Avg. Pool | f=(2,2), s=(2,2) | (5,5,16) | | |
| Flatten | | (400,) | | |
| FC | 120 units | (120,) | 120*(400+1) = 48,120 | |
| FC | 84 units | (84,) | ??? | |
| FC (Output) | 10 units | (10,) | | |

# LeNet Analysis



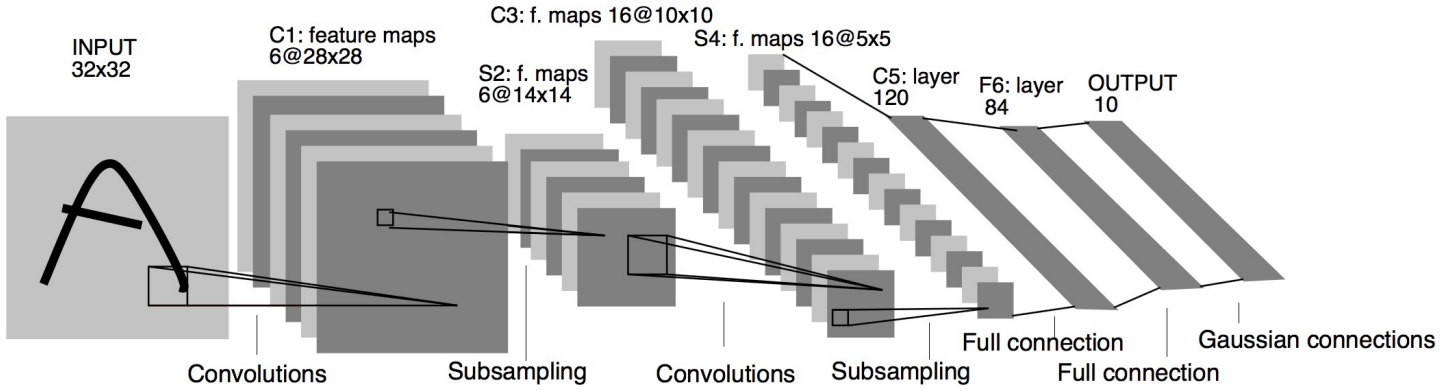

| Layer | HyperParams | Output Volume | # Parameters | flops |
|---|---|---|---|---|
| Input | | (32,32,1) | | |
| Conv | K=6, f=(5,5), s=(1,1) | (28,28,6) | 6*(5*5*1+1)=156 | |
| Avg. Pool | f=(2,2), s=(2,2) | (14,14,6) | | |
| Conv | K=16, f=(5,5), s=(1,1) | (10,10,16) | 16*(5*5*6+1)=2416 | |
| Avg. Pool | f=(2,2), s=(2,2) | (5,5,16) | | |
| Flatten | | (400,) | | |
| FC | 120 units | (120,) | 120*(400+1) = 48,120 | |
| FC | 84 units | (84,) | 84*(120+1)=10,164 | |
| FC (Output) | 10 units | (10,) | ??? | |

# LeNet Analysis



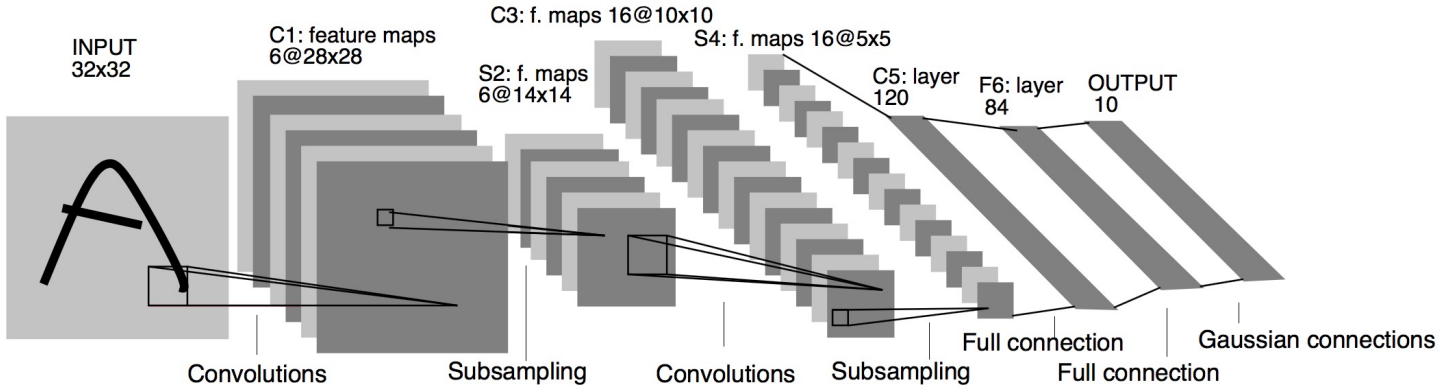

| Layer | HyperParams | Output Volume | # Parameters | flops |
|---|---|---|---|---|
| Input | | (32,32,1) | | |
| Conv | K=6, f=(5,5), s=(1,1) | (28,28,6) | 6*(5*5*1+1)=156 | |
| Avg. Pool | f=(2,2), s=(2,2) | (14,14,6) | | |
| Conv | K=16, f=(5,5), s=(1,1) | (10,10,16) | 16*(5*5*6+1)=2416 | |
| Avg. Pool | f=(2,2), s=(2,2) | (5,5,16) | | |
| Flatten | | (400,) | | |
| FC | 120 units | (120,) | 120*(400+1) = 48,120 | |
| FC | 84 units | (84,) | 84*(120+1)=10,164 | |
| FC (Output) | 10 units | (10,) | 10*(84+1)=850 | |

# LeNet Analysis




| Layer | HyperParams | Output Volume | # Parameters | flops |
|---|---|---|---|---|
| Input | | (32,32,1) | | |
| Conv | K=6, f=(5,5), s=(1,1) | (28,28,6) | 6*(5*5*1+1)=156 | ??? |
| Avg. Pool | f=(2,2), s=(2,2) | (14,14,6) | | |
| Conv | K=16, f=(5,5), s=(1,1) | (10,10,16) | 16*(5*5*6+1)=2416 | |
| Avg. Pool | f=(2,2), s=(2,2) | (5,5,16) | | |
| Flatten | | (400,) | | |
| FC | 120 units | (120,) | 120*(400+1) = 48,120 | |
| FC | 84 units | (84,) | 84*(120+1)=10,164 | |
| FC (Output) | 10 units | (10,) | 10*(84+1)=850 | |

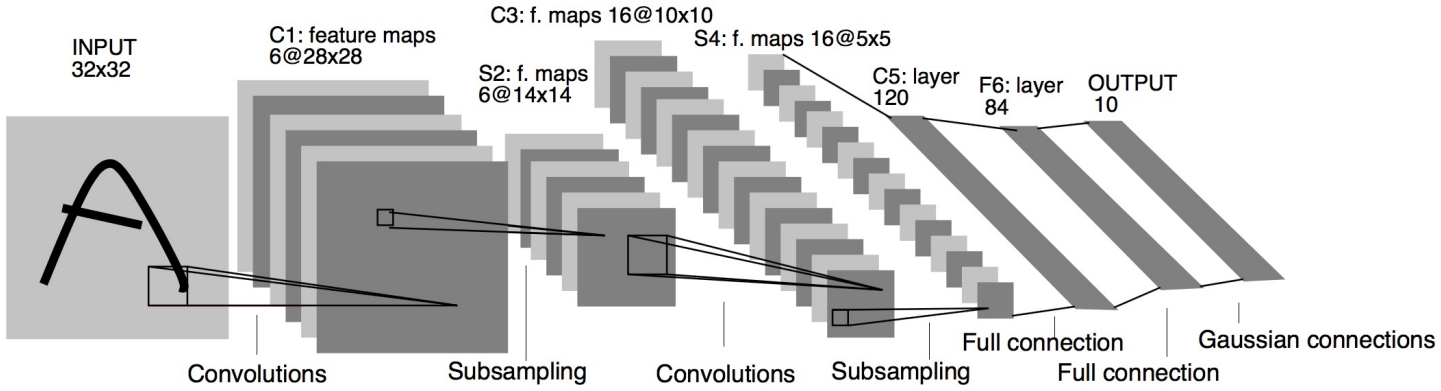
# LeNet Analysis




| Layer | HyperParams | Output Volume | # Parameters | flops |
|---|---|---|---|---|
| Input | | (32,32,1) | | |
| Conv | K=6, f=(5,5), s=(1,1) | (28,28,6) | 6*(5*5*1+1)=156 | 117k |
| Avg. Pool | f=(2,2), s=(2,2) | (14,14,6) | | |
| Conv | K=16, f=(5,5), s=(1,1) | (10,10,16) | 16*(5*5*6+1)=2416 | |
| Avg. Pool | f=(2,2), s=(2,2) | (5,5,16) | | |
| Flatten | | (400,) | | |
| FC | 120 units | (120,) | 120*(400+1) = 48,120 | |
| FC | 84 units | (84,) | 84*(120+1)=10,164 | |
| FC (Output) | 10 units | (10,) | 10*(84+1)=850 | |

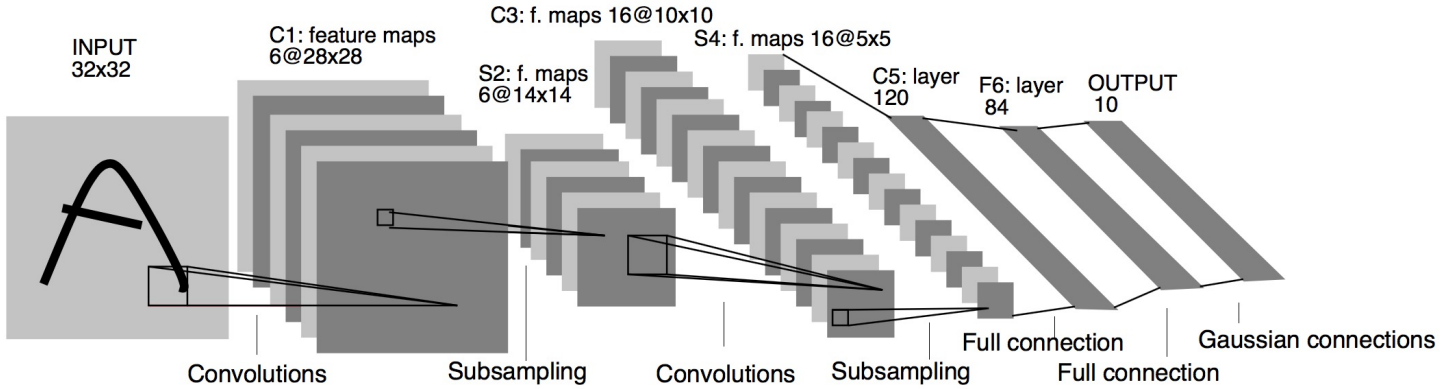
Flops: $h_{out} * w_{out} * K * f * f * c_{in}$ =28*28*6*5*5*1 = 117,600 flops

# LeNet Analysis




| Layer | HyperParams | Output Volume | # Parameters | flops |
|---|---|---|---|---|
| Input | | (32,32,1) | | |
| Conv | K=6, f=(5,5), s=(1,1) | (28,28,6) | 6*(5*5*1+1)=156 | 117k |
| Avg. Pool | f=(2,2), s=(2,2) | (14,14,6) | | ??? |
| Conv | K=16, f=(5,5), s=(1,1) | (10,10,16) | 16*(5*5*6+1)=2416 | |
| Avg. Pool | f=(2,2), s=(2,2) | (5,5,16) | | |
| Flatten | | (400,) | | |
| FC | 120 units | (120,) | 120*(400+1) = 48,120 | |
| FC | 84 units | (84,) | 84*(120+1)=10,164 | |
| FC (Output) | 10 units | (10,) | 10*(84+1)=850 | |

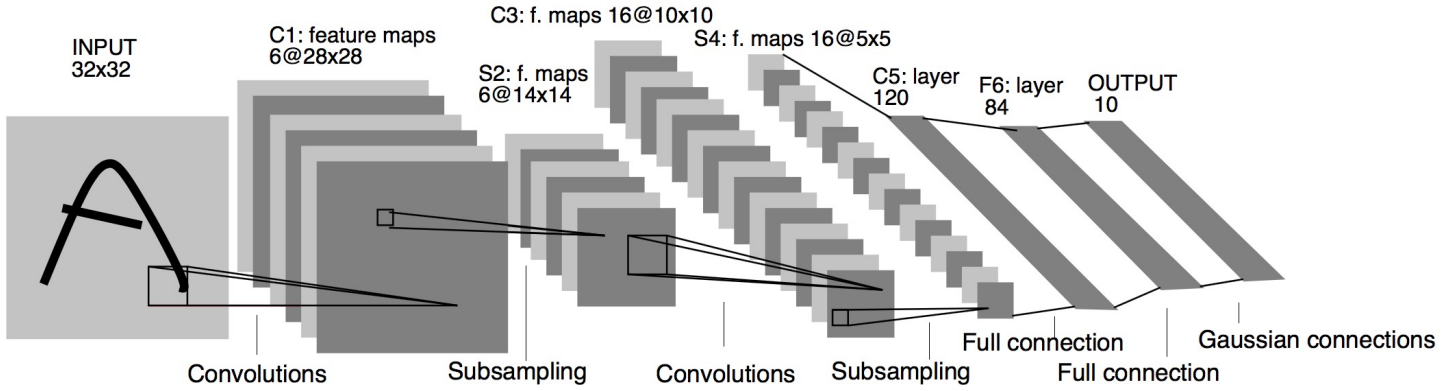
# LeNet Analysis




| Layer | HyperParams | Output Volume | # Parameters | flops |
|---|---|---|---|---|
| Input | | (32,32,1) | | |
| Conv | K=6, f=(5,5), s=(1,1) | (28,28,6) | 6*(5*5*1+1)=156 | 117k |
| Avg. Pool | f=(2,2), s=(2,2) | (14,14,6) | | 4074 |
| Conv | K=16, f=(5,5), s=(1,1) | (10,10,16) | 16*(5*5*6+1)=2416 | |
| Avg. Pool | f=(2,2), s=(2,2) | (5,5,16) | | |
| Flatten | | (400,) | | |
| FC | 120 units | (120,) | 120*(400+1) = 48,120 | |
| FC | 84 units | (84,) | 84*(120+1)=10,164 | |
| FC (Output) | 10 units | (10,) | 10*(84+1)=850 | |

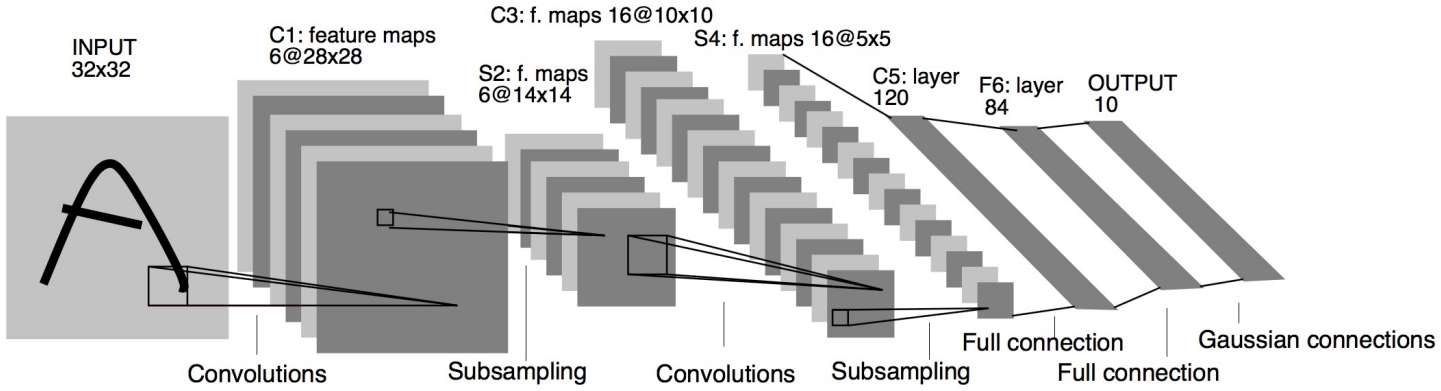
Flops: $h_{out} * w_{out} * c_{out} * f * f$ =14*14*6*2*2 = 4704 flops

# LeNet Analysis




| Layer | HyperParams | Output Volume | # Parameters | flops |
|---|---|---|---|---|
| Input | | (32,32,1) | | |
| Conv | K=6, f=(5,5), s=(1,1) | (28,28,6) | 6*(5*5*1+1)=156 | 117k |
| Avg. Pool | f=(2,2), s=(2,2) | (14,14,6) | | 4074 |
| Conv | K=16, f=(5,5), s=(1,1) | (10,10,16) | 16*(5*5*6+1)=2416 | 240k |
| Avg. Pool | f=(2,2), s=(2,2) | (5,5,16) | | 1.6k |
| Flatten | | (400,) | | 0 |
| FC | 120 units | (120,) | 120*(400+1) = 48,120 | ??? |
| FC | 84 units | (84,) | 84*(120+1)=10,164 | |
| FC (Output) | 10 units | (10,) | 10*(84+1)=850 | |

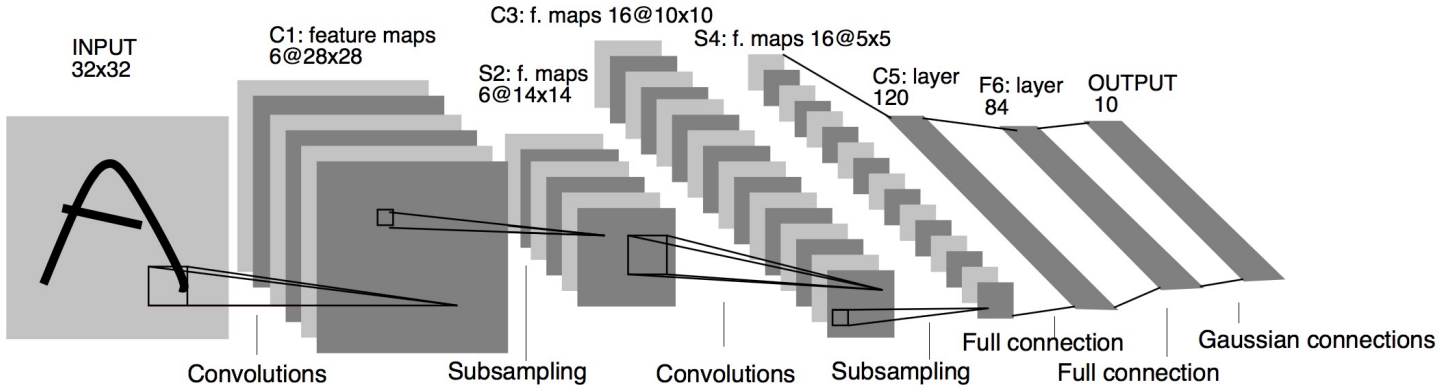
# LeNet Analysis




| Layer | HyperParams | Output Volume | # Parameters | flops |
|---|---|---|---|---|
| Input | | (32,32,1) | | |
| Conv | K=6, f=(5,5), s=(1,1) | (28,28,6) | 6*(5*5*1+1)=156 | 117k |
| Avg. Pool | f=(2,2), s=(2,2) | (14,14,6) | | 4074 |
| Conv | K=16, f=(5,5), s=(1,1) | (10,10,16) | 16*(5*5*6+1)=2416 | 240k |
| Avg. Pool | f=(2,2), s=(2,2) | (5,5,16) | | 1.6k |
| Flatten | | (400,) | | 0 |
| FC | 120 units | (120,) | 120*(400+1) = 48,120 | 48k |
| FC | 84 units | (84,) | 84*(120+1)=10,164 | |
| FC (Output) | 10 units | (10,) | 10*(84+1)=850 | |

Flops: $n_h^{[l]} * n_h^{[l-1]}$ =120*(400) = 48,000 flops

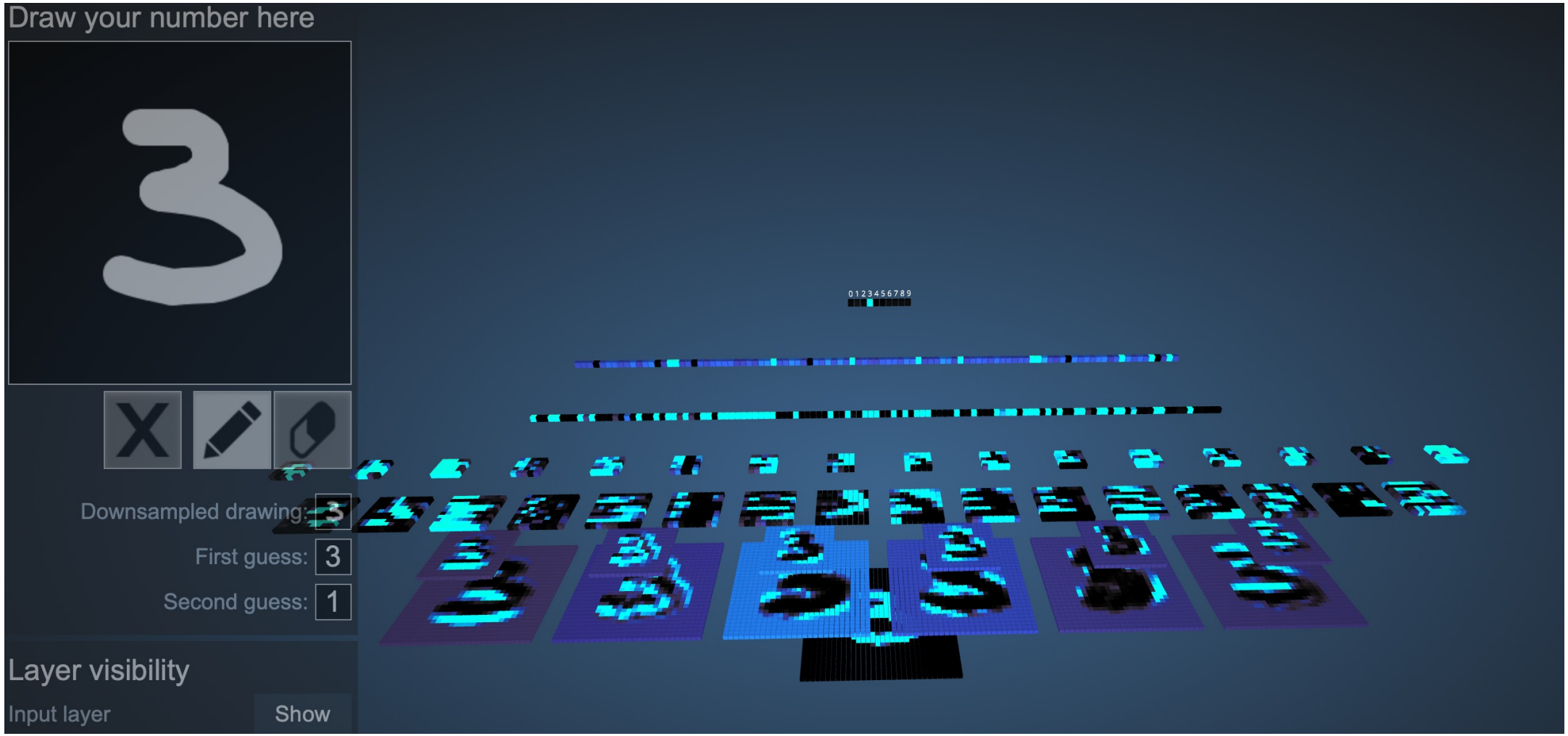
# LeNet Analysis




| Layer | HyperParams | Output Volume | # Parameters | flops |
|---|---|---|---|---|
| Input | | (32,32,1) | | |
| Conv | K=6, f=(5,5), s=(1,1) | (28,28,6) | 6*(5*5*1+1)=156 | 117k |
| Avg. Pool | f=(2,2), s=(2,2) | (14,14,6) | | 4074 |
| Conv | K=16, f=(5,5), s=(1,1) | (10,10,16) | 16*(5*5*6+1)=2416 | 240k |
| Avg. Pool | f=(2,2), s=(2,2) | (5,5,16) | | 1.6k |
| Flatten | | (400,) | | 0 |
| FC | 120 units | (120,) | 120*(400+1) = 48,120 | 48k |
| FC | 84 units | (84,) | 84*(120+1)=10,164 | 10k |
| FC (Output) | 10 units | (10,) | 10*(84+1)=850 | 840 |

~0.42MFlop for one forward pass, ~62k parameters

# Nice Visualization Demo




https://www.cs.ryerson.ca/~aharley/vis/conv/



Brad Quinton, Scott Chin

# ImageNet Large Scale Visual Recognition Competition (ILSVRC) 2010-2017

- Many mainstream CNN architectures got their claim to fame by doing well at the ILSVRC competition.
- The main driver in deep neural networks for computer vision and CNN architectures during much of that time

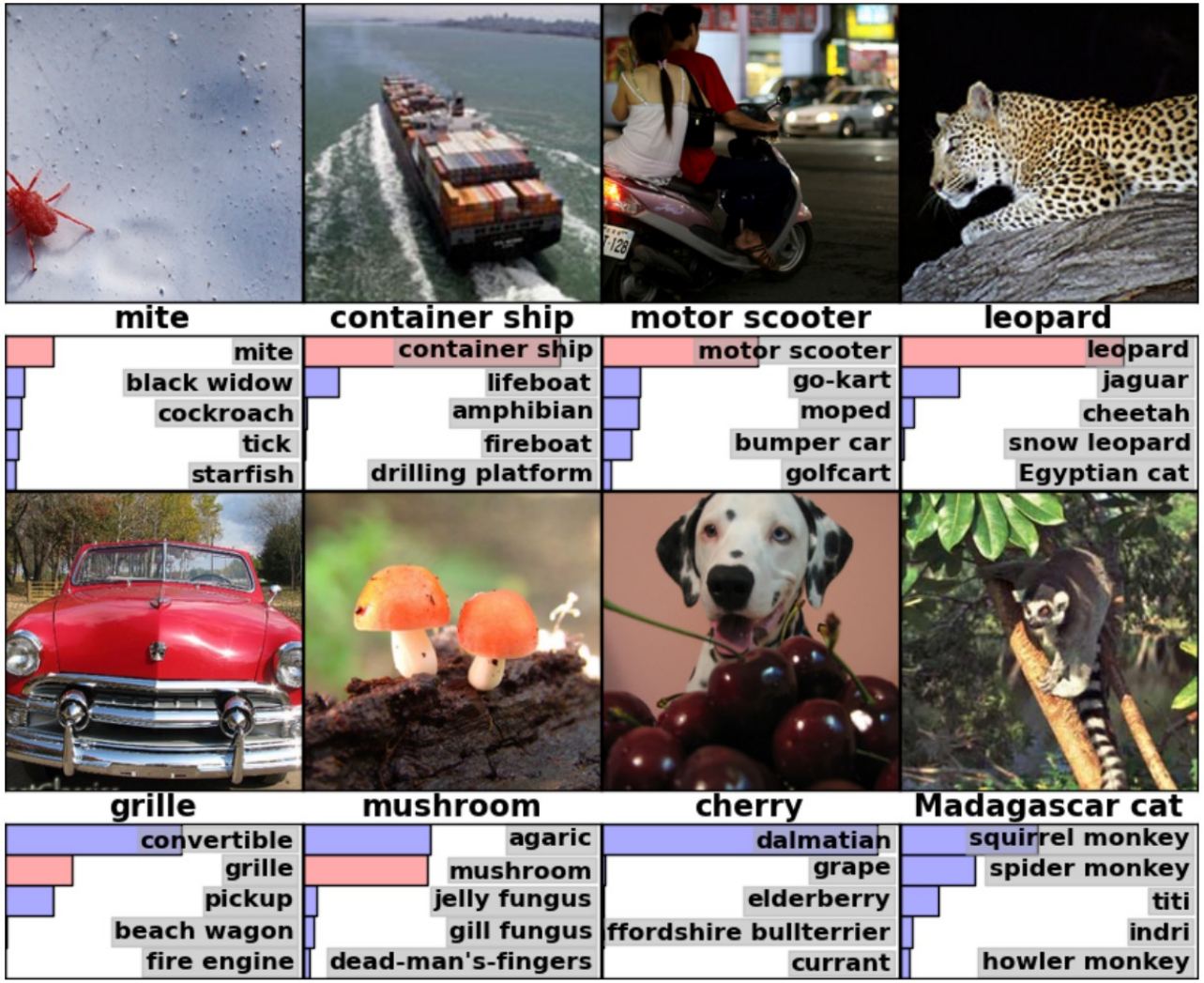
# Data Set

- ImageNet Dataset
  - 15 million+ labelled images in 22,000+ classes.

- ILSVRC competition used a subset:
  1.2mil/150k/50k training/validation/test split

# Top-1 Error and Top-5 Error

- top-5 error rate is the fraction of test images for which the correct label is not among the five labels considered most probable by the model.
- top-1 error rate is the fraction of test images for which the correct label is not the prediction of the model.



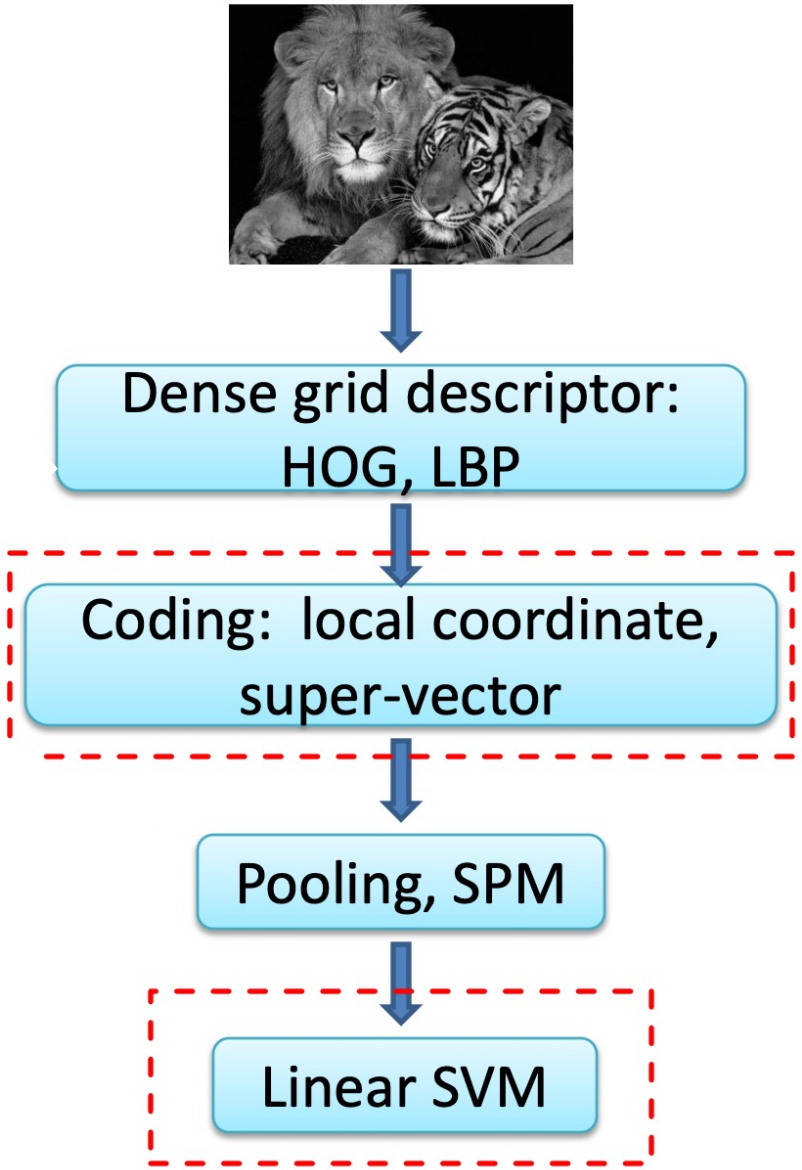

"ImageNet Classification with Deep Convolutional Neural Networks", Krizhevsky, Sutskever, Hinton, 2012, https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

# ILSVRC in 2010 and 2011

- Did NOT use Deep Learning techniques
- Hand Engineered Features
- “Classical” Machine Learning methods
- Final linear classifier (Support Vector Machine)

# 2010 ImageNet Winner

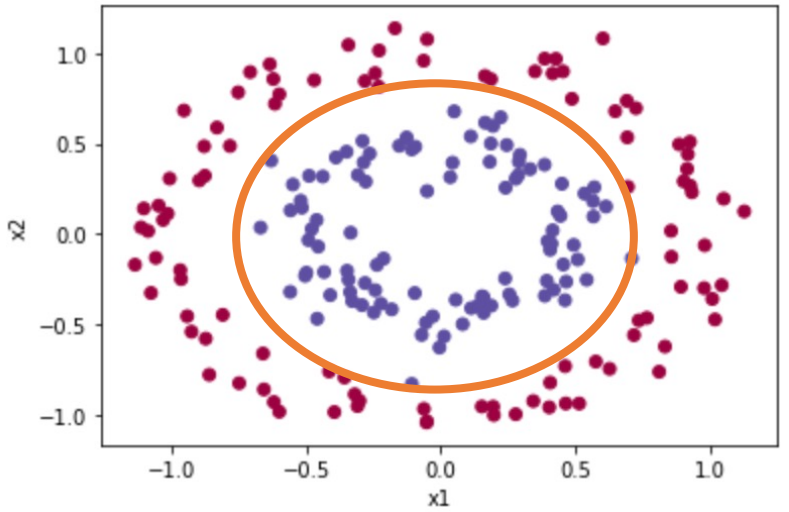
- Encode image into a long vector (100-200k elements)
- Encoding scheme is engineered
- Vectors go into a Support Vector Machine (SVM)
- SVMs
  - A machine learning algorithms
  - A linear classifier



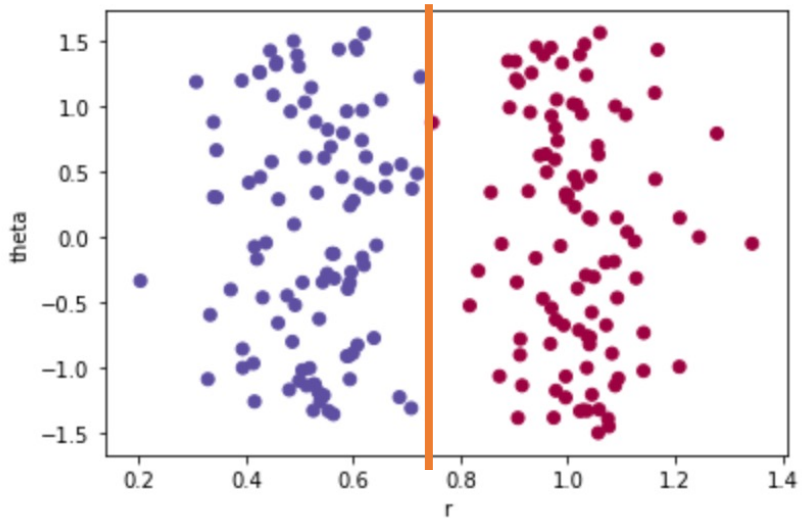

”imageNet classification: fast descriptor coding and large-scale SVM training”, Lin et al, 2010, http://www.image-net.org/challenges/LSVRC/2010/ILSVRC2010_NEC-UIUC.pdf

# From Lecture 8: Feature Engineering

- With Logistic regression (or any linear classifier), you can manually transform your features to encode non-linearity.
- This is called Feature Engineering and requires analysis and human effort
  - i.e. you engineer new synthetic features from the real ones
- Data is now linearly separable




$$r = \sqrt{x_1^2 + x_2^2} \qquad \theta = arctan\left(\frac{x_1}{x_2}\right)$$




Neural networks learn the best transform from the data without human assistance

# 2011 ImageNet Winner

## High-dimensional image signatures: Fisher Vectors (FV)

Perronnin, Sánchez and Mensink, "Improving the Fisher kernel for large-scale image classification", ECCV'10.

+

## Compression: Product Quantization (PQ)

Sánchez and Perronnin, "High-dimensional signature compression for large-scale image classification", CVPR'11.

+

## Simple machine learning: one-vs-all linear SVMs

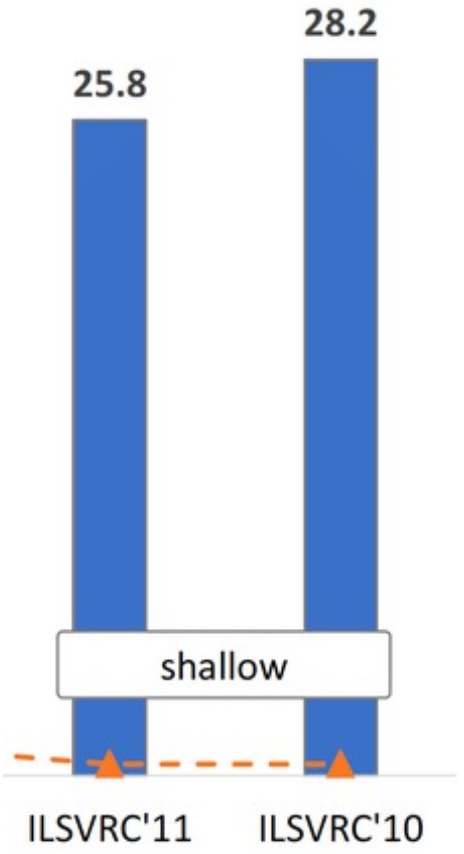
Linear classifiers learned in primal using Stochastic Gradient Descent (SGD)

"Compressed Fisher Vectors for LSVR", Perronnin, Sanchez, 2011, http://www.image-net.org/challenges/LSVRC/2011/ilsvrc11.pdf



Brad Quinton, Scott Chin

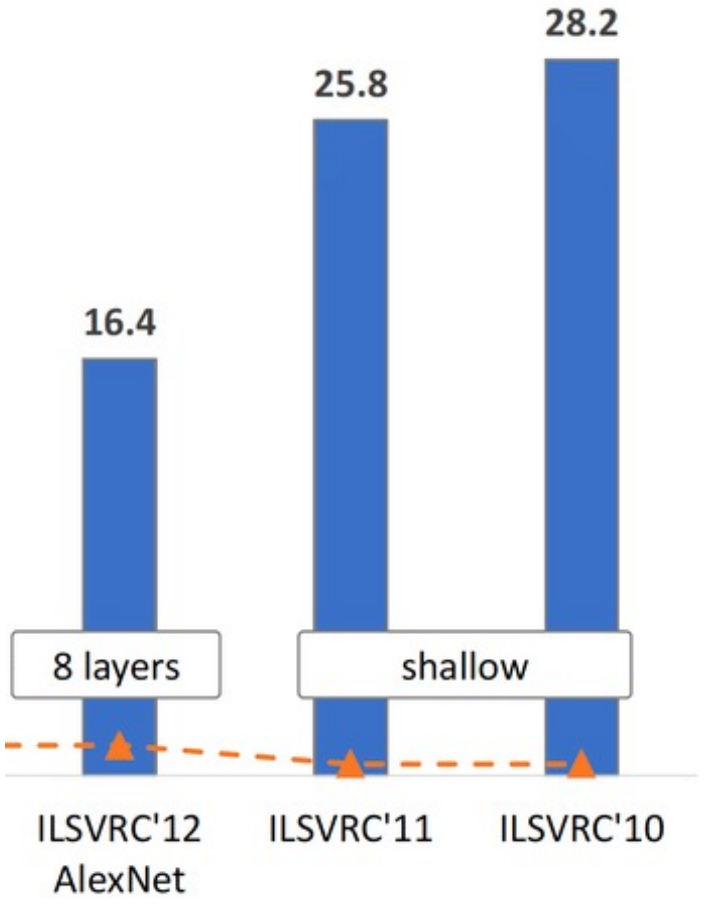
# Pre-Deep Learning at ImageNet






Brad Quinton, Scott Chin

# Deep Learning at ImageNet





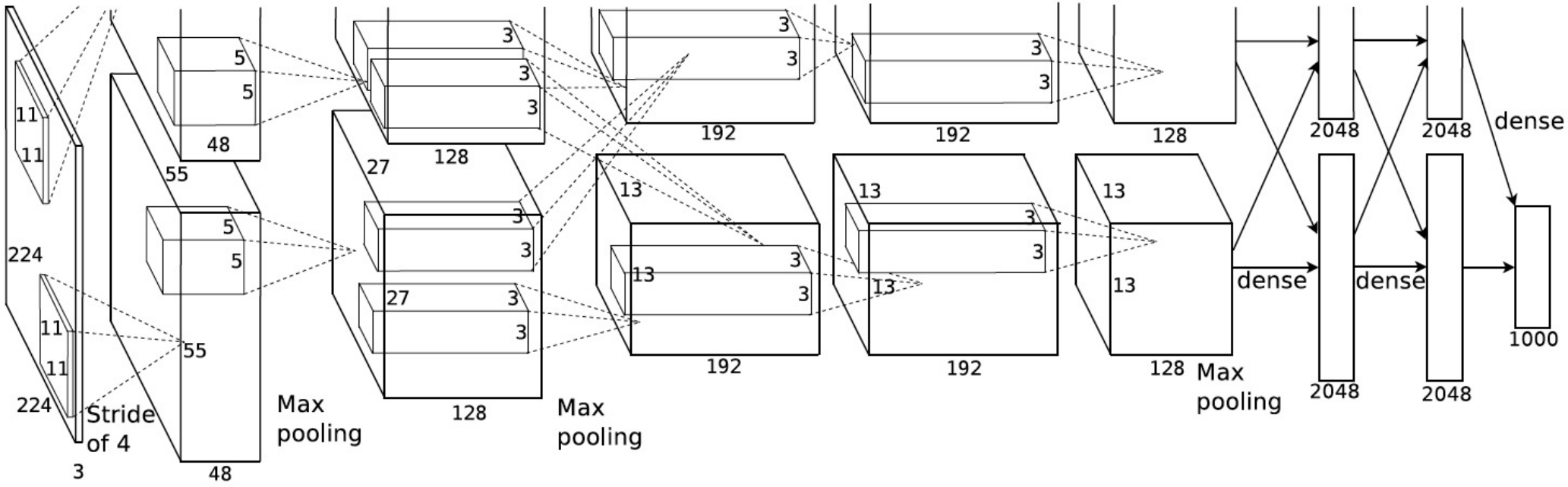

Brad Quinton, Scott Chin

# AlexNet (2012)

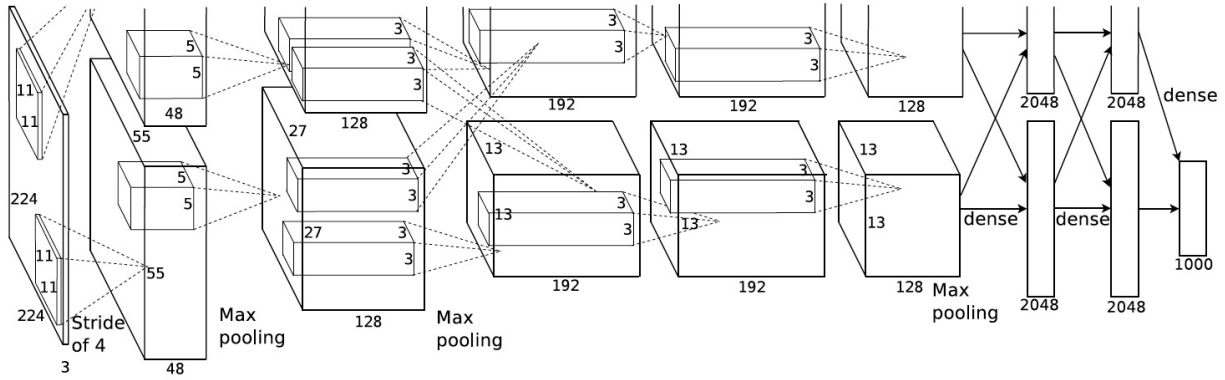
"ImageNet Classification with Deep Convolutional Neural Networks", Krizhevsky, Sutskever, Hinton, 2012,
https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

# AlexNet Architecture

- Popularized CNNs for computer vision
- 16% Top-5 error vs 26% for runner up

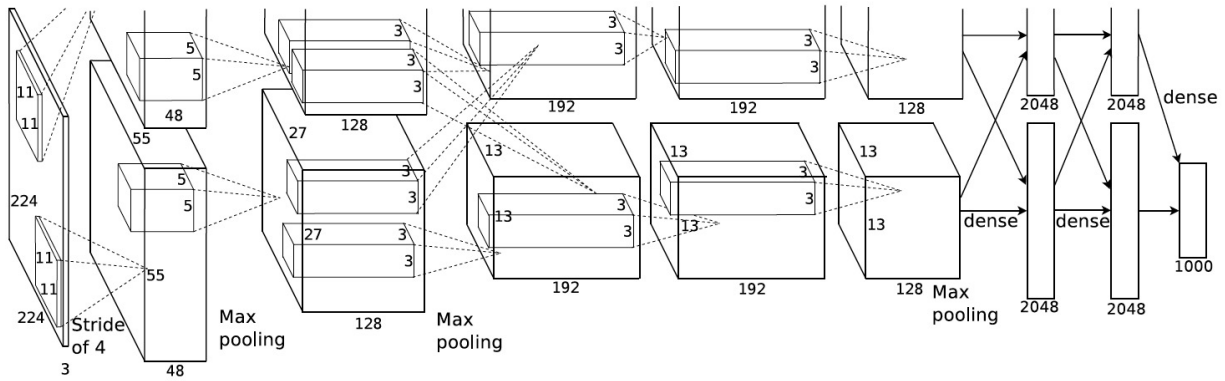


| Layer | HyperParams | Output | # Params | Mflops |
|---|---|---|---|---|
| Input | | (227,227,3) | | |
| Conv1 | K=96,f=(11,11,),s=(4,4) | ??? | | |
| Max Pool | f=(3,3),s=(2,2) | | | |
| Conv2 | K=256,f=(5,5),s=(1,1),p=same | | | |
| Max Pool | f=(3,3),s=(2,2) | | | |
| Conv3 | K=384,f=(3,3),s=(1,1),p=same | | | |
| Conv4 | K=384,f=(3,3),s=(1,1),p=same | | | |
| Conv5 | K=256,f=(3,3),s=(1,1),p=same | | | |
| Max Pool | f=(3,3),s=(2,2) | | | |




Brad Quinton, Scott Chin

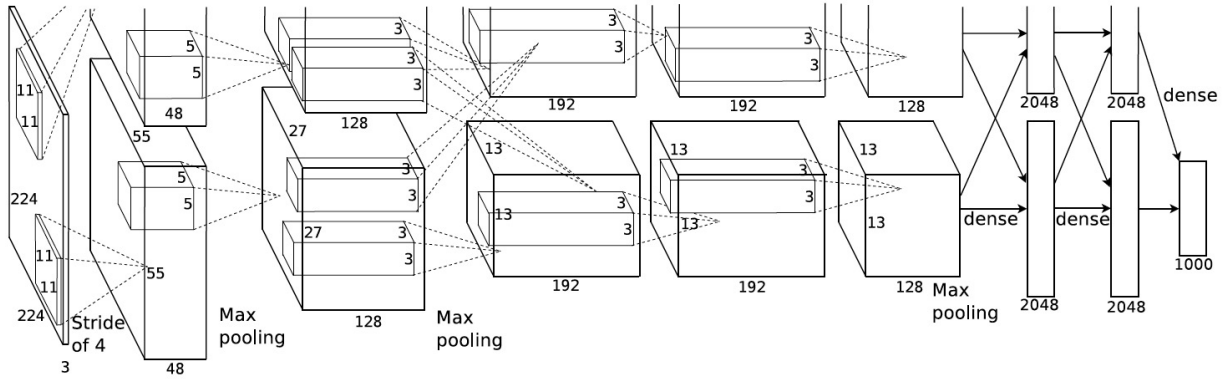
| Layer | HyperParams | Output | # Params | Mflops |
|---|---|---|---|---|
| Input | | (227,227,3) | | |
| Conv1 | K=96,f=(11,11,),s=(4,4) | (55,55,96) | ??? | |
| Max Pool | f=(3,3),s=(2,2) | | | |
| Conv2 | K=256,f=(5,5),s=(1,1),p=same | | | |
| Max Pool | f=(3,3),s=(2,2) | | | |
| Conv3 | K=384,f=(3,3),s=(1,1),p=same | | | |
| Conv4 | K=384,f=(3,3),s=(1,1),p=same | | | |
| Conv5 | K=256,f=(3,3),s=(1,1),p=same | | | |
| Max Pool | f=(3,3),s=(2,2) | | | |




Brad Quinton, Scott Chin

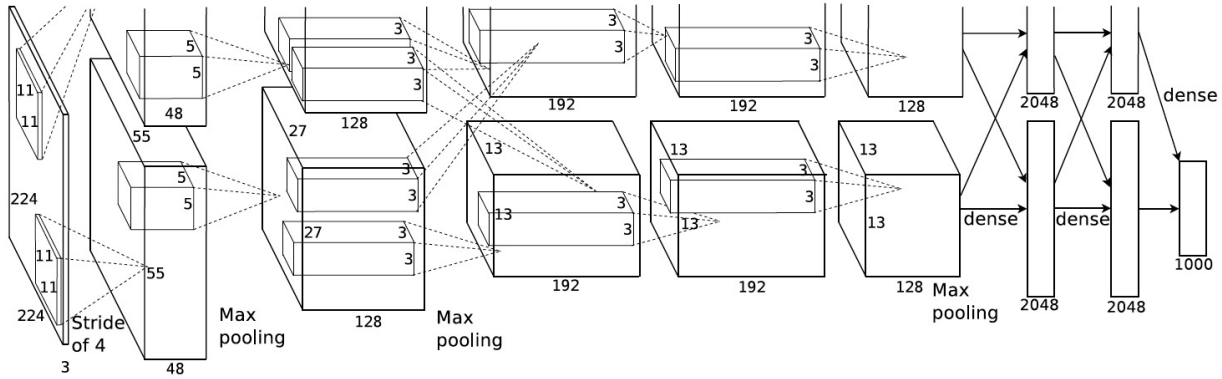
| Layer | HyperParams | Output | # Params | Mflops |
|---|---|---|---|---|
| Input | | (227,227,3) | | |
| Conv1 | K=96,f=(11,11,),s=(4,4) | (55,55,96) | 34,857 | |
| Max Pool | f=(3,3),s=(2,2) | | | |
| Conv2 | K=256,f=(5,5),s=(1,1),p=same | | | |
| Max Pool | f=(3,3),s=(2,2) | | | |
| Conv3 | K=384,f=(3,3),s=(1,1),p=same | | | |
| Conv4 | K=384,f=(3,3),s=(1,1),p=same | | | |
| Conv5 | K=256,f=(3,3),s=(1,1),p=same | | | |
| Max Pool | f=(3,3),s=(2,2) | | | |

$$\#\text{Params} = K(f * f * c_{in} + 1)$$




Brad Quinton, Scott Chin

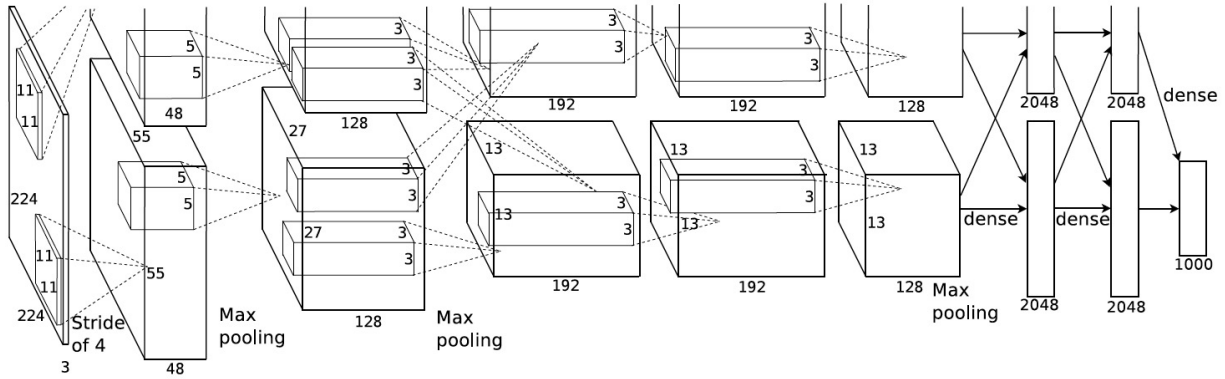
| Layer | HyperParams | Output | # Params | Mflops |
|---|---|---|---|---|
| Input | | (227,227,3) | | |
| Conv1 | K=96,f=(11,11,),s=(4,4) | (55,55,96) | 34,857 | ??? |
| Max Pool | f=(3,3),s=(2,2) | | | |
| Conv2 | K=256,f=(5,5),s=(1,1),p=same | | | |
| Max Pool | f=(3,3),s=(2,2) | | | |
| Conv3 | K=384,f=(3,3),s=(1,1),p=same | | | |
| Conv4 | K=384,f=(3,3),s=(1,1),p=same | | | |
| Conv5 | K=256,f=(3,3),s=(1,1),p=same | | | |
| Max Pool | f=(3,3),s=(2,2) | | | |




Brad Quinton, Scott Chin

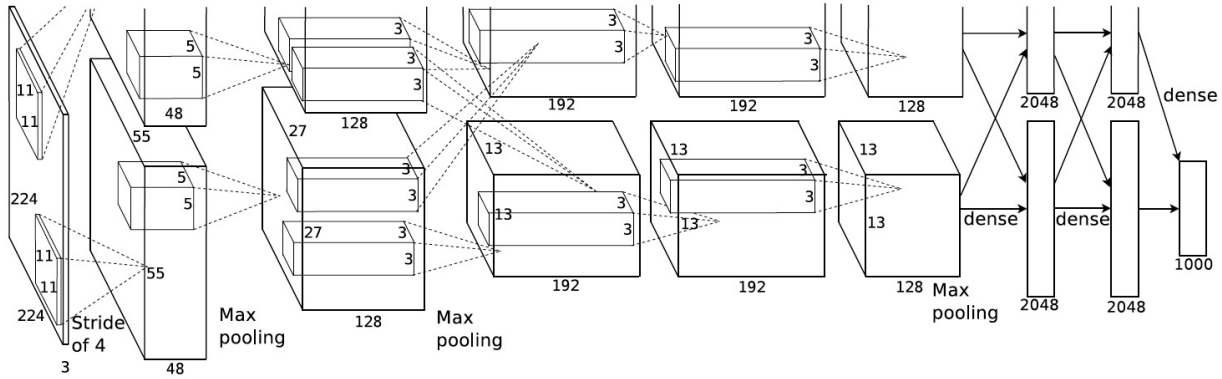
| Layer | HyperParams | Output | # Params | Mflops |
|---|---|---|---|---|
| Input | | (227,227,3) | | |
| Conv1 | K=96,f=(11,11,),s=(4,4) | (55,55,96) | 34,857 | 105 |
| Max Pool | f=(3,3),s=(2,2) | | | |
| Conv2 | K=256,f=(5,5),s=(1,1),p=same | | | |
| Max Pool | f=(3,3),s=(2,2) | | | |
| Conv3 | K=384,f=(3,3),s=(1,1),p=same | | | |
| Conv4 | K=384,f=(3,3),s=(1,1),p=same | | | |
| Conv5 | K=256,f=(3,3),s=(1,1),p=same | | | |
| Max Pool | f=(3,3),s=(2,2) | | | |

#MFlops = $h_{out} * w_{out} * c_{out} * f * f * c_{in}$



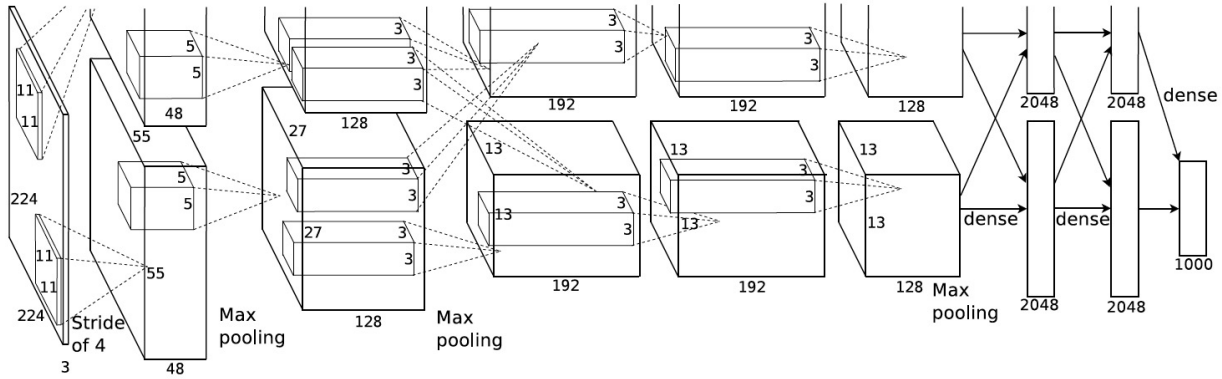

Brad Quinton, Scott Chin

| Layer | HyperParams | Output | # Params | Mflops |
|---|---|---|---|---|
| Input | | (227,227,3) | | |
| Conv1 | K=96,f=(11,11,),s=(4,4) | (55,55,96) | 34,857 | 105 |
| Max Pool | f=(3,3),s=(2,2) | ??? | | |
| Conv2 | K=256,f=(5,5),s=(1,1),p=same | | | |
| Max Pool | f=(3,3),s=(2,2) | | | |
| Conv3 | K=384,f=(3,3),s=(1,1),p=same | | | |
| Conv4 | K=384,f=(3,3),s=(1,1),p=same | | | |
| Conv5 | K=256,f=(3,3),s=(1,1),p=same | | | |
| Max Pool | f=(3,3),s=(2,2) | | | |




Brad Quinton, Scott Chin

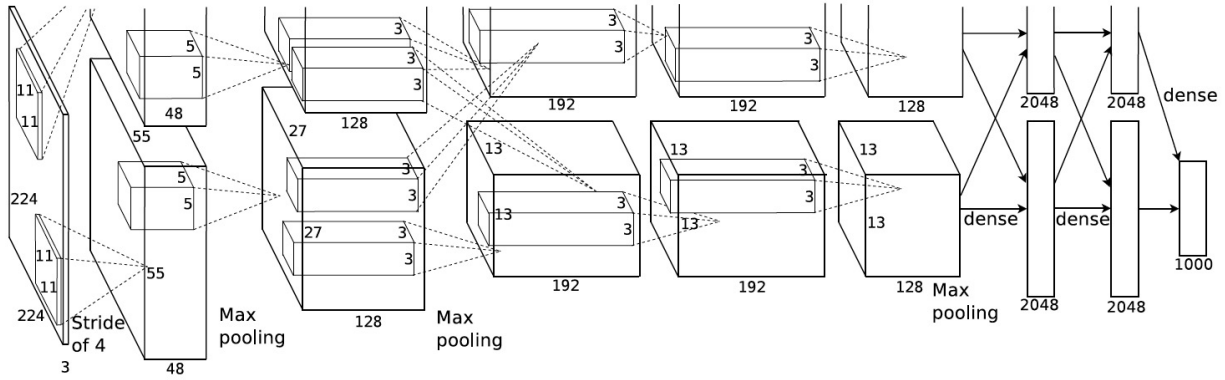
| Layer | HyperParams | Output | # Params | Mflops |
|---|---|---|---|---|
| Input | | (227,227,3) | | |
| Conv1 | K=96,f=(11,11,),s=(4,4) | (55,55,96) | 34,857 | 105 |
| Max Pool | f=(3,3),s=(2,2) | (27,27,96) | ??? | |
| Conv2 | K=256,f=(5,5),s=(1,1),p=same | | | |
| Max Pool | f=(3,3),s=(2,2) | | | |
| Conv3 | K=384,f=(3,3),s=(1,1),p=same | | | |
| Conv4 | K=384,f=(3,3),s=(1,1),p=same | | | |
| Conv5 | K=256,f=(3,3),s=(1,1),p=same | | | |
| Max Pool | f=(3,3),s=(2,2) | | | |




Brad Quinton, Scott Chin

| Layer | HyperParams | Output | # Params | Mflops |
|---|---|---|---|---|
| Input | | (227,227,3) | | |
| Conv1 | K=96,f=(11,11,),s=(4,4) | (55,55,96) | 34,857 | 105 |
| Max Pool | f=(3,3),s=(2,2) | (27,27,96) | 0 | ??? |
| Conv2 | K=256,f=(5,5),s=(1,1),p=same | | | |
| Max Pool | f=(3,3),s=(2,2) | | | |
| Conv3 | K=384,f=(3,3),s=(1,1),p=same | | | |
| Conv4 | K=384,f=(3,3),s=(1,1),p=same | | | |
| Conv5 | K=256,f=(3,3),s=(1,1),p=same | | | |
| Max Pool | f=(3,3),s=(2,2) | | | |

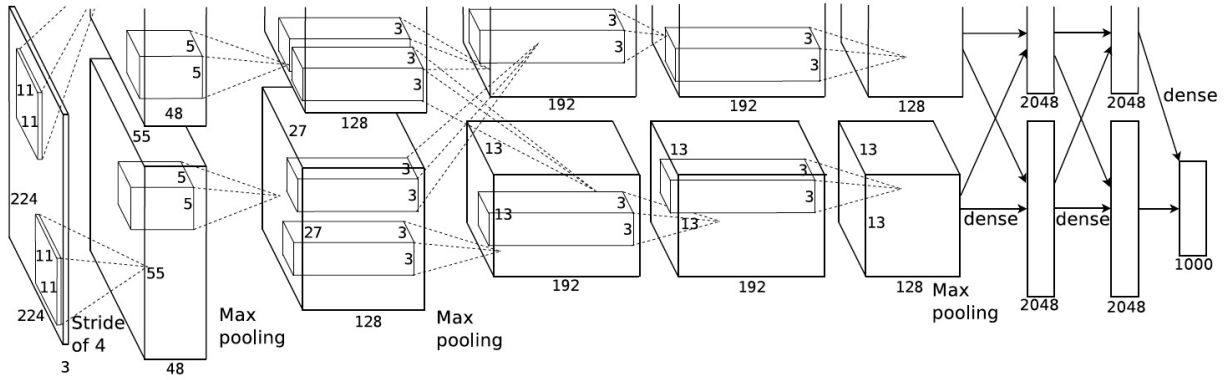
Remember! Pooling layers don't have learned parameters!




Brad Quinton, Scott Chin

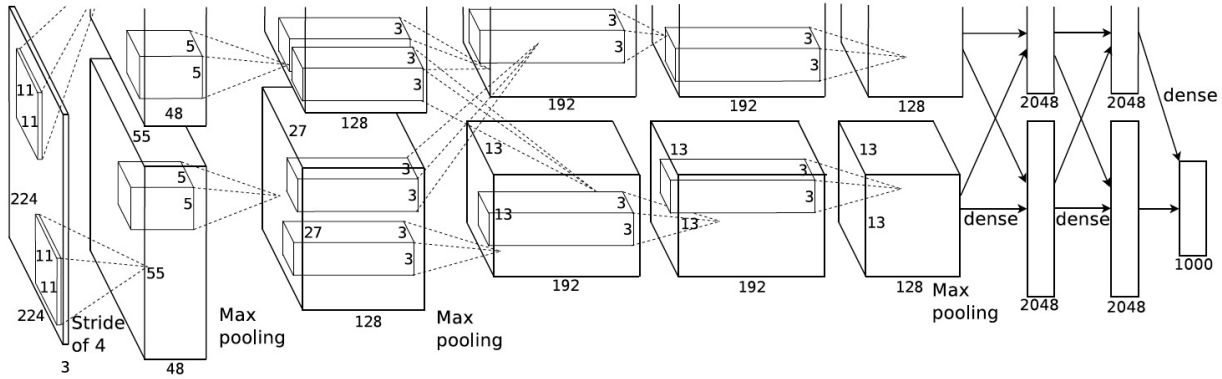
| Layer | HyperParams | Output | # Params | Mflops |
|---|---|---|---|---|
| Input | | (227,227,3) | | |
| Conv1 | K=96,f=(11,11,),s=(4,4) | (55,55,96) | 34,857 | 105 |
| Max Pool | f=(3,3),s=(2,2) | (27,27,96) | 0 | 0.6 |
| Conv2 | K=256,f=(5,5),s=(1,1),p=same | | | |
| Max Pool | f=(3,3),s=(2,2) | | | |
| Conv3 | K=384,f=(3,3),s=(1,1),p=same | | | |
| Conv4 | K=384,f=(3,3),s=(1,1),p=same | | | |
| Conv5 | K=256,f=(3,3),s=(1,1),p=same | | | |
| Max Pool | f=(3,3),s=(2,2) | | | |

#MFlops = $h_{out} * w_{out} * c_{out} * f * f$




Brad Quinton, Scott Chin

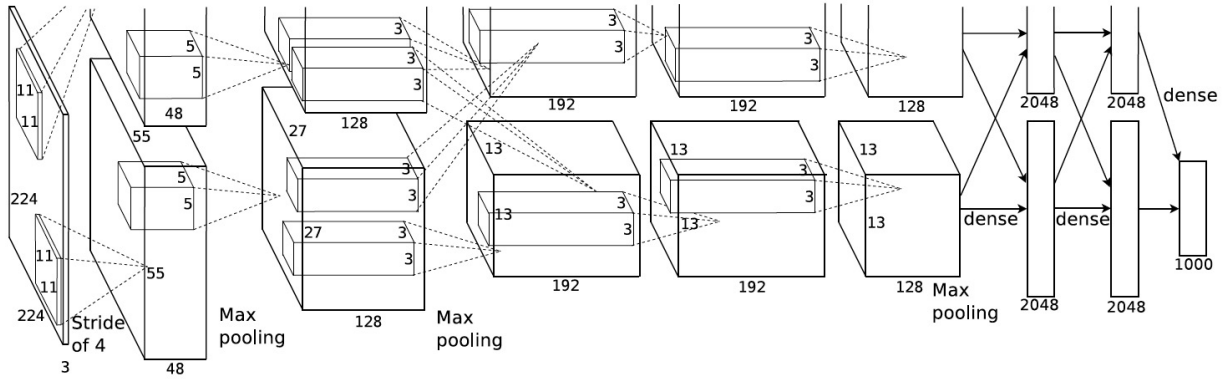
| Layer | HyperParams | Output | # Params | Mflops |
|---|---|---|---|---|
| Input | | (227,227,3) | | |
| Conv1 | K=96,f=(11,11,),s=(4,4) | (55,55,96) | 34,857 | 105 |
| Max Pool | f=(3,3),s=(2,2) | (27,27,96) | 0 | 0.6 |
| Conv2 | K=256,f=(5,5),s=(1,1),p=same | ??? | | |
| Max Pool | f=(3,3),s=(2,2) | | | |
| Conv3 | K=384,f=(3,3),s=(1,1),p=same | | | |
| Conv4 | K=384,f=(3,3),s=(1,1),p=same | | | |
| Conv5 | K=256,f=(3,3),s=(1,1),p=same | | | |
| Max Pool | f=(3,3),s=(2,2) | | | |




Brad Quinton, Scott Chin

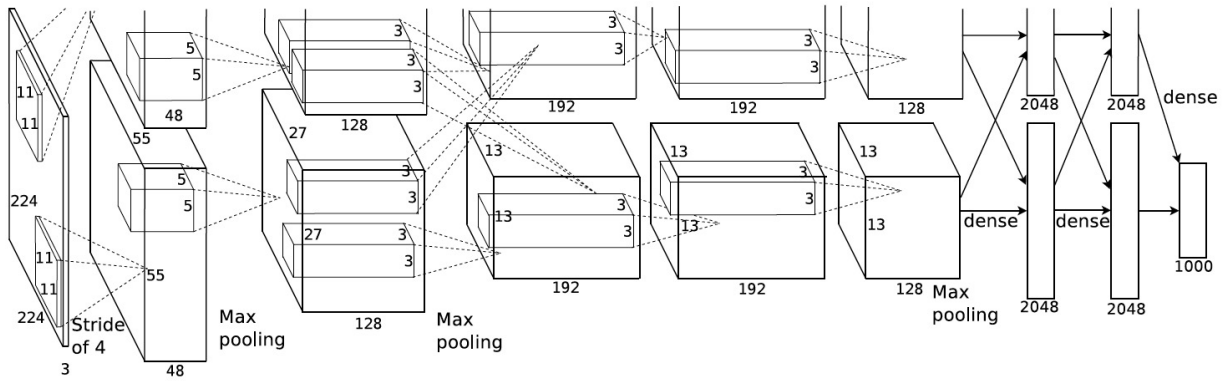
| Layer | HyperParams | Output | # Params | Mflops |
|---|---|---|---|---|
| Input | | (227,227,3) | | |
| Conv1 | K=96,f=(11,11,),s=(4,4) | (55,55,96) | 34,857 | 105 |
| Max Pool | f=(3,3),s=(2,2) | (27,27,96) | 0 | 0.6 |
| Conv2 | K=256,f=(5,5),s=(1,1),p=same | (27,27,256) | | |
| Max Pool | f=(3,3),s=(2,2) | | | |
| Conv3 | K=384,f=(3,3),s=(1,1),p=sa | | | |
| Conv4 | K=384,f=(3,3),s=(1,1),p=same | | | |
| Conv5 | K=256,f=(3,3),s=(1,1),p=same | | | |
| Max Pool | f=(3,3),s=(2,2) | | | |

"same" padding preserves spatial dimensions




Brad Quinton, Scott Chin

| Layer | HyperParams | Output | # Params | Mflops |
|---|---|---|---|---|
| Input | | (227,227,3) | | |
| Conv1 | K=96,f=(11,11,),s=(4,4) | (55,55,96) | 34,857 | 105 |
| Max Pool | f=(3,3),s=(2,2) | (27,27,96) | 0 | 0.6 |
| Conv2 | K=256,f=(5,5),s=(1,1),p=same | (27,27,256) | 614,656 | 448 |
| Max Pool | f=(3,3),s=(2,2) | (13,13,256) | 0 | 0.4 |
| Conv3 | K=384,f=(3,3),s=(1,1),p=same | (13,13,384) | 885,120 | 150 |
| Conv4 | K=384,f=(3,3),s=(1,1),p=same | (13,13,384) | 1,327,488 | 224 |
| Conv5 | K=256,f=(3,3),s=(1,1),p=same | (13,13,256) | 884,992 | 150 |
| Max Pool | f=(3,3),s=(2,2) | (6,6,256) | 0 | 0.08 |
| | | | 0 | |
| | | | | |
| | | | | |
| | | | | |




Brad Quinton, Scott Chin

| Layer | HyperParams | Output | # Params | Mflops |
|---|---|---|---|---|
| Input | | (227,227,3) | | |
| Conv1 | K=96,f=(11,11,),s=(4,4) | (55,55,96) | 34,857 | 105 |
| Max Pool | f=(3,3),s=(2,2) | (27,27,96) | 0 | 0.6 |
| Conv2 | K=256,f=(5,5),s=(1,1),p=same | (27,27,256) | 614,656 | 448 |
| Max Pool | f=(3,3),s=(2,2) | (13,13,256) | 0 | 0.4 |
| Conv3 | K=384,f=(3,3),s=(1,1),p=same | (13,13,384) | 885,120 | 150 |
| Conv4 | K=384,f=(3,3),s=(1,1),p=same | (13,13,384) | 1,327,488 | 224 |
| Conv5 | K=256,f=(3,3),s=(1,1),p=same | (13,13,256) | 884,992 | 150 |
| Max Pool | f=(3,3),s=(2,2) | (6,6,256) | 0 | 0.08 |
| Flatten | | ??? | 0 | |
| FC | n=4096 | | | |
| FC | n=4096 | | | |
| FC(softmax) | n=1000 | | | |

| Layer | HyperParams | Output | # Params | Mflops |
|---|---|---|---|---|
| Input | | (227,227,3) | | |
| Conv1 | K=96,f=(11,11,),s=(4,4) | (55,55,96) | 34,857 | 105 |
| Max Pool | f=(3,3),s=(2,2) | (27,27,96) | 0 | 0.6 |
| Conv2 | K=256,f=(5,5),s=(1,1),p=same | (27,27,256) | 614,656 | 448 |
| Max Pool | f=(3,3),s=(2,2) | (13,13,256) | 0 | 0.4 |
| Conv3 | K=384,f=(3,3),s=(1,1),p=same | (13,13,384) | 885,120 | 150 |
| Conv4 | K=384,f=(3,3),s=(1,1),p=same | (13,13,384) | 1,327,488 | 224 |
| Conv5 | K=256,f=(3,3),s=(1,1),p=same | (13,13,256) | 884,992 | 150 |
| Max Pool | f=(3,3),s=(2,2) | (6,6,256) | 0 | 0.08 |
| Flatten | | (9216,) | 0 | |
| FC | n=4096 | (4096,) | ??? | |
| FC | n=4096 | (4096,) | | |
| FC(softmax) | n=1000 | (1000,) | | |

| Layer | HyperParams | Output | # Params | Mflops |
|---|---|---|---|---|
| Input | | (227,227,3) | | |
| Conv1 | K=96,f=(11,11,),s=(4,4) | (55,55,96) | 34,857 | 105 |
| Max Pool | f=(3,3),s=(2,2) | (27,27,96) | 0 | 0.6 |
| Conv2 | K=256,f=(5,5),s=(1,1),p=same | (27,27,256) | 614,656 | 448 |
| Max Pool | f=(3,3),s=(2,2) | (13,13,256) | 0 | 0.4 |
| Conv3 | K=384,f=(3,3),s=(1,1),p=same | (13,13,384) | 885,120 | 150 |
| Conv4 | K=384,f=(3,3),s=(1,1),p=same | (13,13,384) | 1,327,488 | 224 |
| Conv5 | K=256,f=(3,3),s=(1,1),p=same | (13,13,256) | 884,992 | 150 |
| Max Pool | f=(3,3),s=(2,2) | (6,6,256 | | |
| Flatten | | (9216,) | | |
| FC | n=4096 | (4096,) | 37,752,832 | ??? |
| FC | n=4096 | (4096,) | | |
| FC(softmax) | n=1000 | (1000,) | | |

$$\#\text{Params} = n_h^{[l]}\left(n_h^{[l-1]} + 1\right)$$

| Layer | HyperParams | Output | # Params | Mflops |
|---|---|---|---|---|
| Input | | (227,227,3) | | |
| Conv1 | K=96,f=(11,11,),s=(4,4) | (55,55,96) | 34,857 | 105 |
| Max Pool | f=(3,3),s=(2,2) | (27,27,96) | 0 | 0.6 |
| Conv2 | K=256,f=(5,5),s=(1,1),p=same | (27,27,256) | 614,656 | 448 |
| Max Pool | f=(3,3),s=(2,2) | (13,13,256) | 0 | 0.4 |
| Conv3 | K=384,f=(3,3),s=(1,1),p=same | (13,13,384) | 885,120 | 150 |
| Conv4 | K=384,f=(3,3),s=(1,1),p=same | (13,13,384) | 1,327,488 | 224 |
| Conv5 | K=256,f=(3,3),s=(1,1),p=same | (13,13,256) | 884,992 | 150 |
| Max Pool | f=(3,3),s=(2,2) | (6,6,256) | | |
| Flatten | | (9216,) | 0 | |
| FC | n=4096 | (4096,) | 37,752,832 | 38 |
| FC | n=4096 | (4096,) | | |
| FC(softmax) | n=1000 | (1000,) | | |

$$\#\text{MFLOPs} = n_h^{[l-1]} * n_h^{[l]}$$

| Layer | HyperParams | Output | # Params | Mflops |
|---|---|---|---|---|
| Input | | (227,227,3) | | |
| Conv1 | K=96,f=(11,11,),s=(4,4) | (55,55,96) | 34,857 | 105 |
| Max Pool | f=(3,3),s=(2,2) | (27,27,96) | 0 | 0.6 |
| Conv2 | K=256,f=(5,5),s=(1,1),p=same | (27,27,256) | 614,656 | 448 |
| Max Pool | f=(3,3),s=(2,2) | (13,13,256) | 0 | 0.4 |
| Conv3 | K=384,f=(3,3),s=(1,1),p=same | (13,13,384) | 885,120 | 150 |
| Conv4 | K=384,f=(3,3),s=(1,1),p=same | (13,13,384) | 1,327,488 | 224 |
| Conv5 | K=256,f=(3,3),s=(1,1),p=same | (13,13,256) | 884,992 | 150 |
| Max Pool | f=(3,3),s=(2,2) | (6,6,256) | 0 | 0.08 |
| Flatten | | (9216,) | 0 | |
| FC | n=4096 | (4096,) | 37,752,832 | 38 |
| FC | n=4096 | (4096,) | 16,781,312 | 17 |
| FC(softmax) | n=1000 | (1000,) | 4,097,000 | 4 |

| Layer | HyperParams | Output | # Params | Mflops |
|---|---|---|---|---|
| Input | | (227,227,3) | | |
| Conv1 | K=96,f=(11,11,),s=(4,4) | (55,55,96) | 34,857 | 105 |
| | | 27,27,96) | 0 | 0.6 |
| | | 27,27,256) | 614,656 | 448 |
| | | 13,13,256) | 0 | 0.4 |
| | | (13,13,384) | 885,120 | 150 |
| Conv4 | K=384,f=(3,3),s=(1,1),p=same | (13,13,384) | 1,327,488 | 224 |
| Conv5 | K=256,f=(3,3),s=(1,1),p=same | (13,13,256) | 884,992 | 150 |
| Max Pool | f=(3,3),s=(2,2) | (6,6,256) | 0 | 0.08 |
| Flatten | | (9216,) | 0 | |
| FC | n=4096 | (4096,) | 37,752,832 | 38 |
| FC | n=4096 | (4096,) | 16,781,312 | 17 |
| FC(softmax) | n=1000 | (1000,) | 4,097,000 | 4 |

AlexNet

- ~1000 MFlop for one forward pass
- ~62 million parameters

| Layer | HyperParams | Output | # Params | Mflops |
|---|---|---|---|---|
| Input | | (227,227,3) | | |
| Conv1 | K=96,f=(11,11,),s=(4,4) | (55,55,96) | 34,857 | 105 |
| | | ,27,27,96) | 0 | 0.6 |
| | | ,27,27,256) | 614,656 | 448 |
| | | 3,13,256) | 0 | 0.4 |
| Conv3 | K=384,f=(3,3),s=(1,1),p=same | (13,13,384) | 885,120 | 150 |
| | | 3,13,384) | 1,327,488 | 224 |
| | | 3,13,256) | 884,992 | 150 |
| | | ,6,256) | 0 | 0.08 |
| Flatten | | 9216,) | 0 | |
| FC | n=4096 | (4096,) | 37,752,832 | 38 |
| FC | n=4096 | (4096,) | 16,781,312 | 17 |
| FC(softmax) | n=1000 | (1000,) | 4,097,000 | 4 |

AlexNet

- ~1000 MFlop for one forward pass
- ~62 million parameters

Compared to Lenet5 (1998)

- ~0.42 MFlop for one forward pass
- ~0.062 million parameters

| Layer | HyperParams | Output | # Params | Mflops |
|---|---|---|---|---|
| Input | | (227,227,3) | | |
| Conv1 | K=96,f=(11,11,),s=(4,4) | (55,55,96) | 34,857 | 105 |
| | | 96) | 0 | 0.6 |
| | | 256) | 614,656 | 448 |
| | | 256) | 0 | 0.4 |
| | | 384) | 885,120 | 150 |
| Conv4 | K=384,f=(3,3),s=(1,1),p=same | (13,13,384) | 1,327,488 | 224 |
| Conv5 | K=256,f=(3,3),s=(1,1),p=same | (13,13,256) | 884,992 | 150 |
| Max Pool | f=(3,3),s=(2,2) | (6,6,256) | 0 | 0.08 |
| Flatten | | (9216,) | 0 | |
| FC | n=4096 | (4096,) | 37,752,832 | 38 |
| FC | n=4096 | (4096,) | 16,781,312 | 17 |
| FC(softmax) | n=1000 | (1000,) | 4,097,000 | 4 |

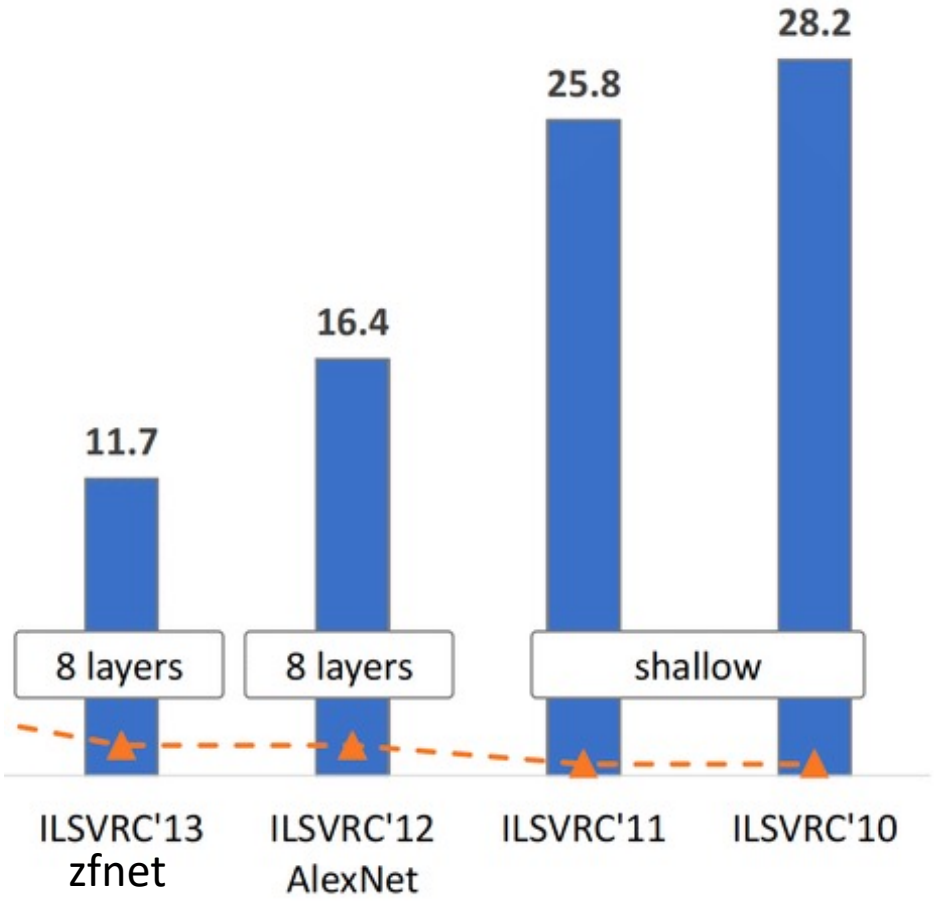
Some more observations:

- Most params in full—connected layers
- Most computation in Conv layers
- Pooling layers pretty much “free”

# Some other take-aways from AlexNet

- Popularized ReLUs for CNNs
  - Found that networks with ReLU consistently “learned” faster.
  - E.g. On CIFAR10, achieved 25% training error in 6x fewer training epochs
- Overlapping Pooling
  - Overlapping Pooling.  Found this helped them reduce top1 error by 0.4% and top5 error by 0.3%
  - Observed that overlapping pooling helped model generalize (reduce overfit)
- Used Local Response Normalization layers
- Architecture hyperparameters chosen by trial-and-error

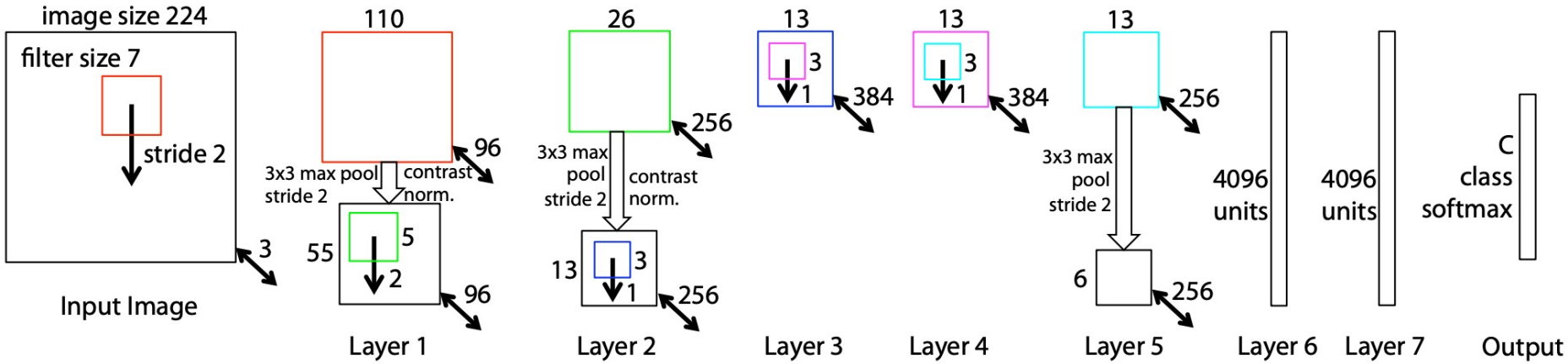
# Deep Learning at ImageNet



# ZFNet (2013)

“Visualizing and Understanding Convolutional Networks”, Zeiler, Fergus, 2013, https://arxiv.org/abs/1311.2901

# A Bigger AlexNet

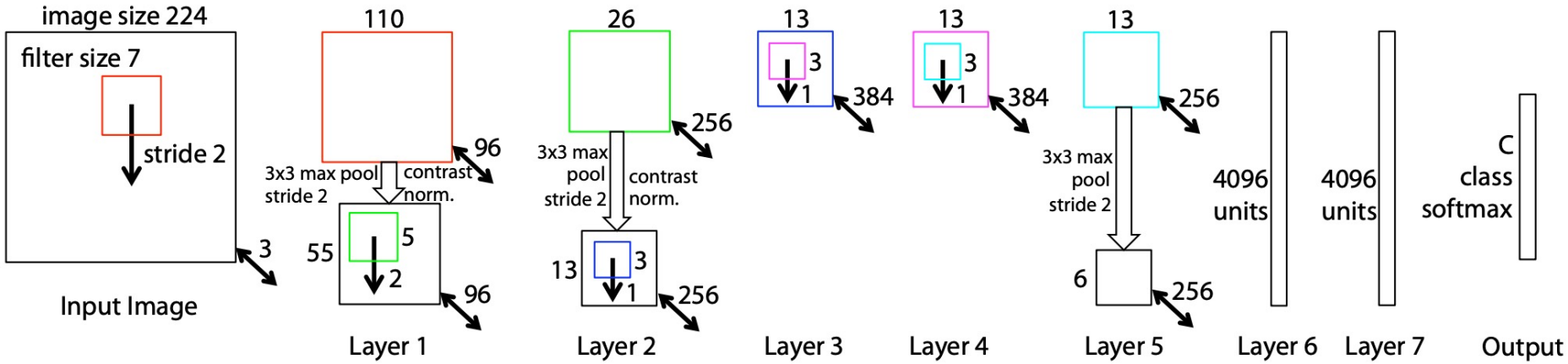



- Conv1 7x7 stride 2 instead of 11x11 stride 4
- Conv3 512 filters instead of 384
- Conv4 1024 instead of 384
- Conv5 512 instead of 384

Note: This figure from their paper shows a smaller variant of ZFNet with fewer filters per layer

"Visualizing and Understanding Convolutional Networks", Zeiler, Fergus, 2013, https://arxiv.org/abs/1311.2901

# A Bigger AlexNet

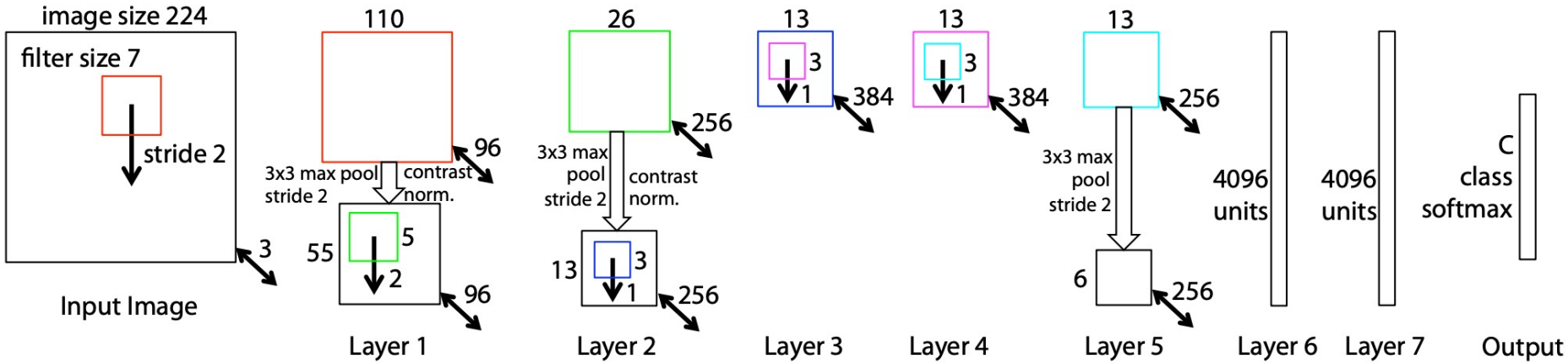



- Conv1 7x7 stride 2 instead of 11x11 stride 4 ← More resolution
- Conv3 512 filters instead of 384
- Conv4 1024 instead of 384
- Conv5 512 instead of 384

Note: This figure from their paper shows a smaller variant of ZFNet with fewer filters per layer

"Visualizing and Understanding Convolutional Networks", Zeiler, Fergus, 2013, https://arxiv.org/abs/1311.2901

# A Bigger AlexNet




- Conv1 7x7 stride 2 instead of 11x11 stride 4 ← More resolution
- Conv3 512 filters instead of 384
- Conv4 1024 instead of 384
- Conv5 512 instead of 384

} More Capacity for learning different features

Note: This figure from their paper shows a smaller variant of ZFNet with fewer filters per layer

"Visualizing and Understanding Convolutional Networks", Zeiler, Fergus, 2013, https://arxiv.org/abs/1311.2901

| Layer | HyperParams | Output | # Params | Mflops |
|---|---|---|---|---|
| Input | | (224,224,3) | | |
| Conv1 | K=96,f=(7,7),s=(2,2) | (110,110,96) | 14,208 | 170.8 |
| Max Pool | f=(3,3),s=(2,2) | (55,55,96) | 0 | 2.6 |
| Conv2 | K=256,f=(5,5),s=(1,1),p=same | (26,26,256) | 614,656 | 415.3 |
| Max Pool | f=(3,3),s=(2,2) | (13,13,256) | 0 | 0.09 |
| Conv3 | K=512,f=(3,3),s=(1,1),p=same | (13,13,512) | 1,180,160 | 199.4 |
| Conv4 | K=1024,f=(3,3),s=(1,1),p=same | (13,13,1024) | 4,719,616 | 797.4 |
| Conv5 | K=512,f=(3,3),s=(1,1),p=same | (13,13,512) | 4,719,616 | 797.4 |
| Max Pool | f=(3,3),s=(2,2) | (6,6,512) | 0 | 0.17 |
| Flatten | | (18432,) | 0 | |
| FC | n=4096 | (4096,) | 75,515,904 | 75.5 |
| FC | n=4096 | (4096,) | 16,781,312 | 16.8 |
| FC(softmax) | n=1000 | (1000,) | 4,097,000 | 4.1 |

| Layer | HyperParams | Output | # Params | Mflops |
|---|---|---|---|---|
| Input | | (224,224,3) | | |
| Conv1 | K=96,f=(7,7),s=(2,2) | (110,110,96) | 14,208 | 170.8 |
| | | 55,55,96) | 0 | 2.6 |
| | | 26,26,256) | 614,656 | 415.3 |
| | | 13,13,256) | 0 | 0.09 |
| Conv3 | K=512,f=(3,3),s=(1,1),p=same | (13,13,512) | 1,180,160 | 199.4 |
| | | 13,13,1024) | 4,719,616 | 797.4 |
| | | 13,13,512) | 4,719,616 | 797.4 |
| | | 6,6,512) | 0 | 0.17 |
| Flatten | | 18432,) | 0 | |
| FC | n=4096 | (4096,) | 75,515,904 | 75.5 |
| FC | n=4096 | (4096,) | 16,781,312 | 16.8 |
| FC(softmax) | n=1000 | (1000,) | 4,097,000 | 4.1 |

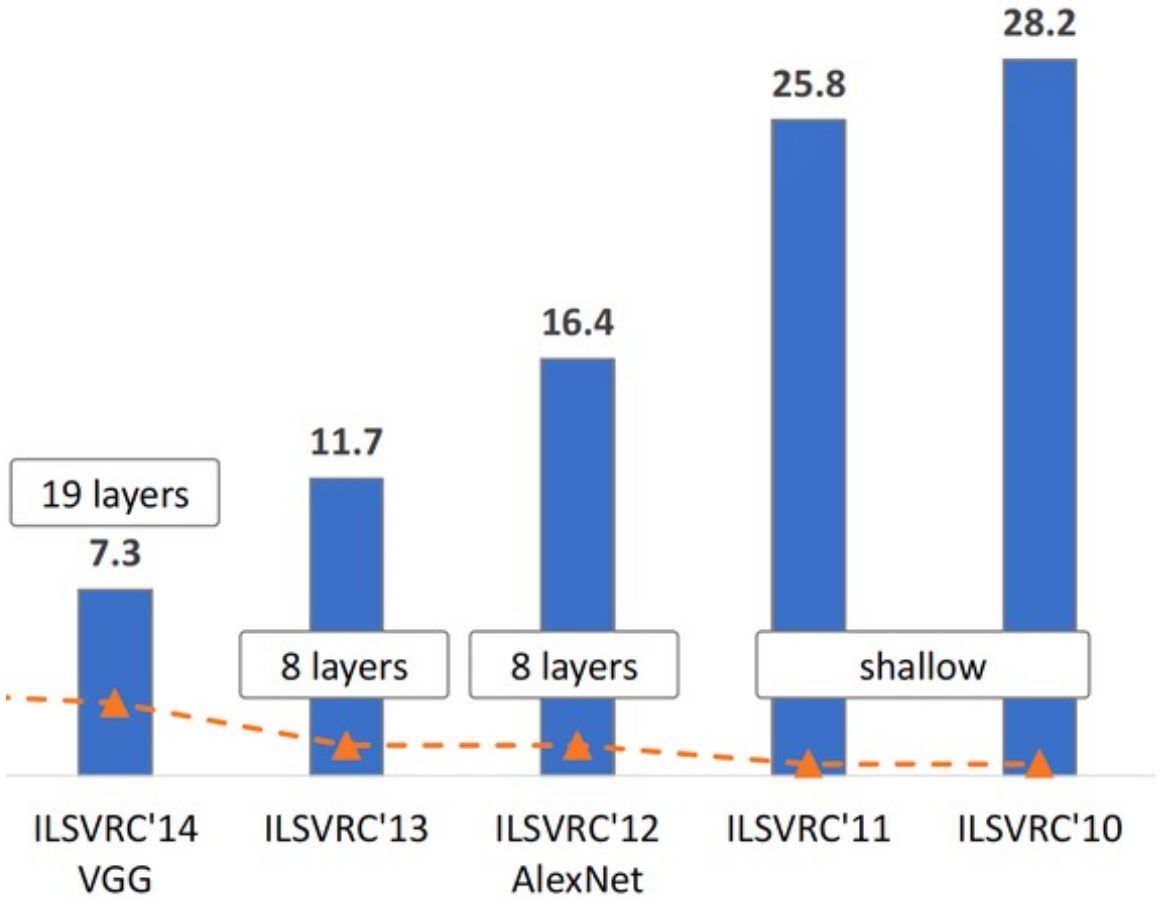
ZFNet
- ~2.47 GFlop for one forward pass
- ~108 million parameters

Compared to AlexNet
- ~1 GFlop for one forward pass
- ~62 million parameters

# A few takeways from ZFNet

- Bigger capacity is still better (for this problem at least)
- Still use trial-and-error for architecture design
- No consideration of computation efficiency (reduced top-5 error by 4.7% at the cost of 2.5x computation and 60% more parameters)

# Deep Learning at ImageNet



# VGGNet

“Very Deep Convolutional Networks for Large-Scale Image Recognition”, Simonyan, Zisserman, 2014, https://arxiv.org/abs/1409.1556

# Systematic Design Principles

1. All conv layers are 3x3 stride 1, same pad
2. All max pool layers are 2x2 stride 2
3. The conv layer following a pool layer will have enough filters to double the volume channel size

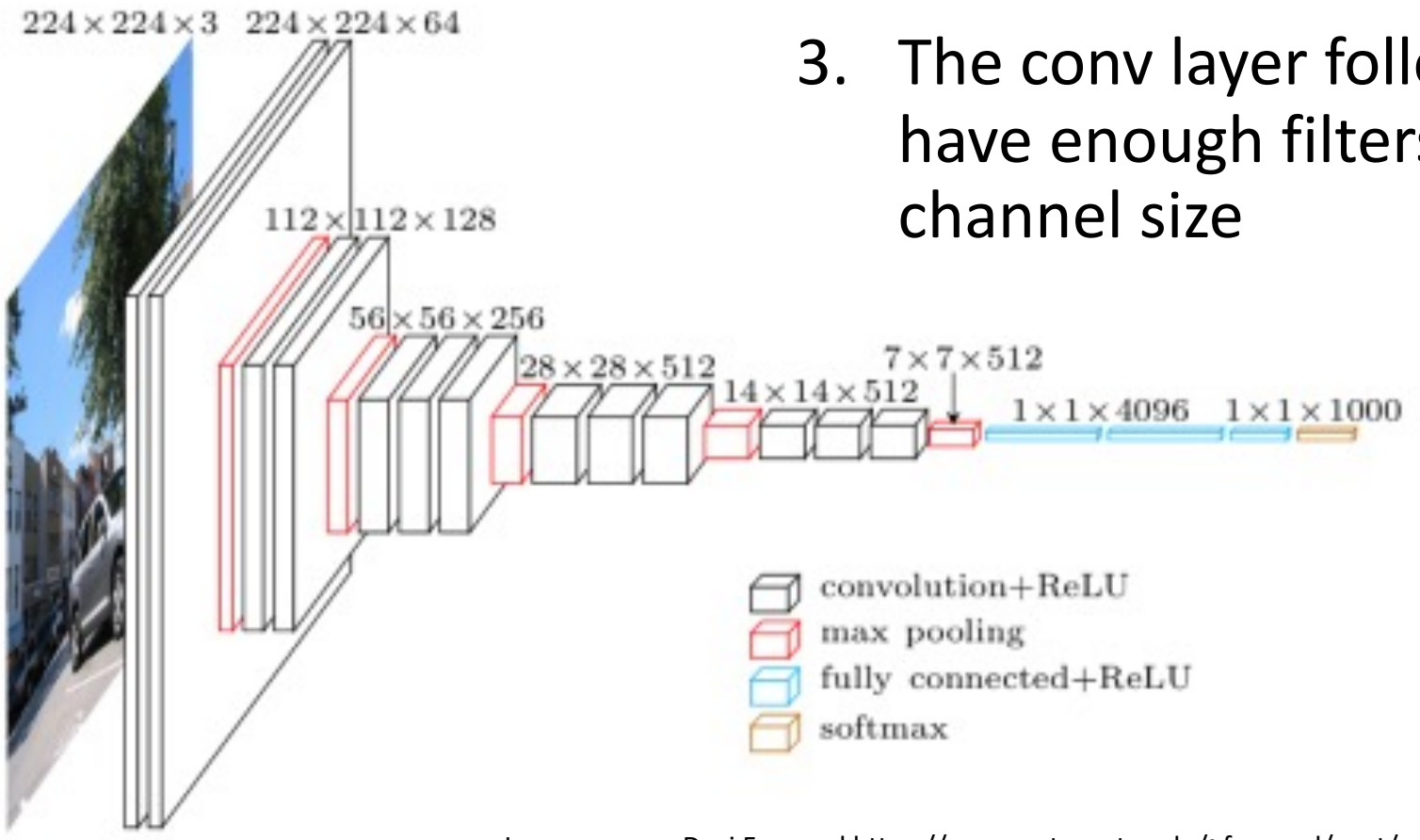



Image source: Davi Frossard https://www.cs.toronto.edu/~frossard/post/vgg16/

# VGGNet is a class of architectures

- Using design rules, a number of architectures were evaluated
- Each architecture has 5 **stages**
- A stage consists of 1-4 conv layers followed by max pool
- The ones that people talk about are VGG16 and VGG19, with 16 and weight 19 layers, respectively
- Around 138million parameters

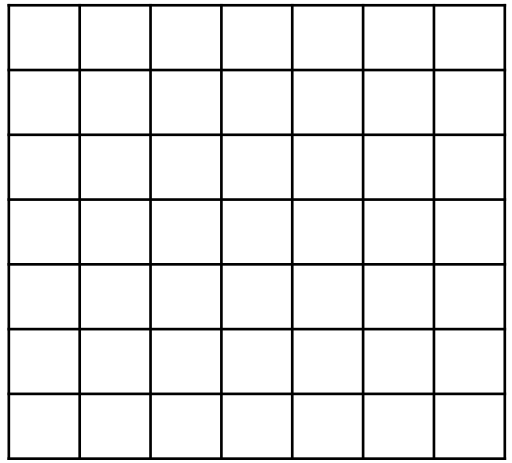
| ConvNet Configuration | | | | | |
|---|---|---|---|---|---|
| A | A-LRN | B | C | D | E |
| 11 weight layers | 11 weight layers | 13 weight layers | 16 weight layers | 16 weight layers | 19 weight layers |
| input (224 × 224 RGB image) | | | | | |
| conv3-64 | conv3-64<br>**LRN** | conv3-64<br>**conv3-64** | conv3-64<br>conv3-64 | conv3-64<br>conv3-64 | conv3-64<br>conv3-64 |
| maxpool | | | | | |
| conv3-128 | conv3-128 | conv3-128<br>**conv3-128** | conv3-128<br>conv3-128 | conv3-128<br>conv3-128 | conv3-128<br>conv3-128 |
| maxpool | | | | | |
| conv3-256<br>conv3-256 | conv3-256<br>conv3-256 | conv3-256<br>conv3-256 | conv3-256<br>conv3-256<br>**conv1-256** | conv3-256<br>conv3-256<br>**conv3-256** | conv3-256<br>conv3-256<br>conv3-256<br>**conv3-256** |
| maxpool | | | | | |
| conv3-512<br>conv3-512 | conv3-512<br>conv3-512 | conv3-512<br>conv3-512 | conv3-512<br>conv3-512<br>**conv1-512** | conv3-512<br>conv3-512<br>**conv3-512** | conv3-512<br>conv3-512<br>conv3-512<br>**conv3-512** |
| maxpool | | | | | |
| conv3-512<br>conv3-512 | conv3-512<br>conv3-512 | conv3-512<br>conv3-512 | conv3-512<br>conv3-512<br>**conv1-512** | conv3-512<br>conv3-512<br>**conv3-512** | conv3-512<br>conv3-512<br>conv3-512<br>**conv3-512** |
| maxpool | | | | | |
| FC-4096 | | | | | |
| FC-4096 | | | | | |
| FC-1000 | | | | | |
| soft-max | | | | | |

# Only 3x3, stride 1, same padding conv layers

- Why could this be a good choice?
- Let’s compare to using larger filters

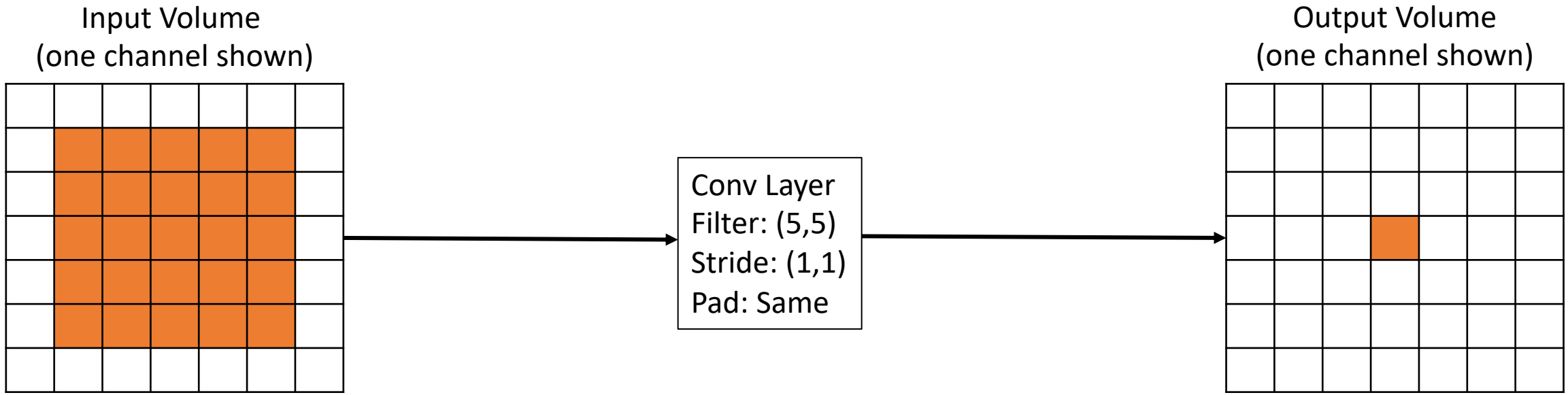
# Only 3x3, stride 1, same padding conv layers

Input Volume
(one channel shown)



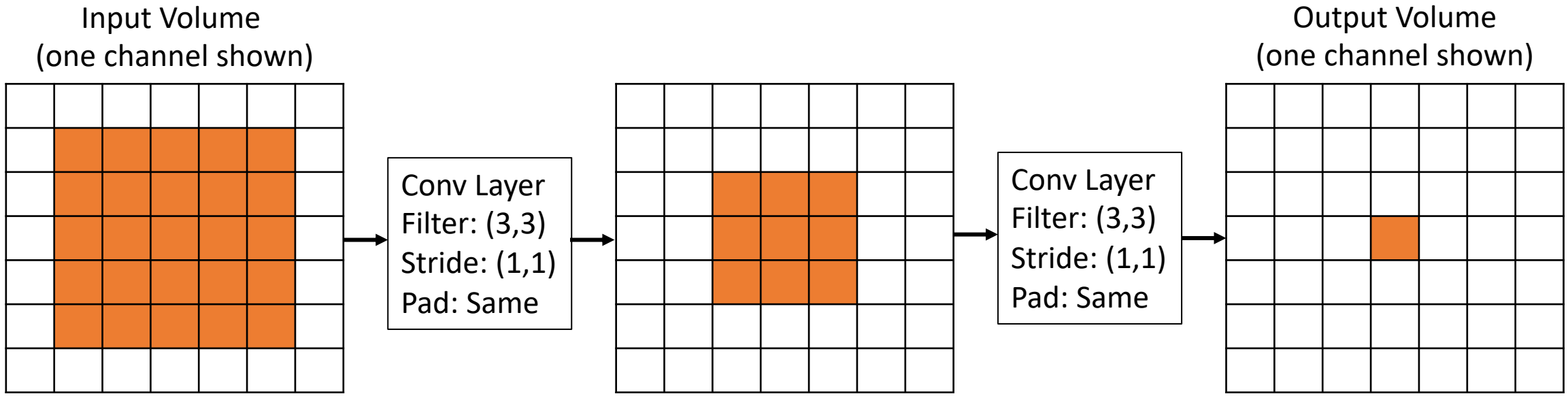
- Consider an input volume

# Only 3x3, stride 1, same padding conv layers




- Consider a single 5x5 convolution layer
- Each output activation can “see” a 5x5 spatial region of the input

# Only 3x3, stride 1, same padding conv layers




- Now consider using two stacked 3x3 conv layers
- Each output activation can **still** “see” a 5x5 spatial region of the input

# Single 5x5 layer vs 2 3x3 layers

- Parameters
  - Two (3x3) layers: $2 * K * (3 * 3 * c_{in} + 1)$
  - One (5x5) layer: $K * (5 * 5 * c_{in} + 1)$
  - → One (5x5) has ~ 25/18 ~ 1.4x more learned parameters

# Single 5x5 layer vs 2 3x3 layers

- Parameters
  - Two (3x3) layers: $2 * K * (3 * 3 * c_{in} + 1)$
  - One (5x5) layer: $K * (5 * 5 * c_{in} + 1)$
  - → One (5x5) has ~ 25/18 ~ 1.4x more learned parameters
- FLOPs:
  - Two (3x3) layers : $2 * h_{out} * w_{out} * c_{out} * 3 * 3 * c_{in}$
  - One (5x5) layer: $h_{out} * w_{out} * c_{out} * 5 * 5 * c_{in}$
  - → One (5x5) layer needs more flops (25/18 ~ 1.4x more)

# Single 5x5 layer vs 2 3x3 layers

- Parameters
  - Two (3x3) layers: $2 * K * (3 * 3 * c_{in} + 1)$
  - One (5x5) layer: $K * (5 * 5 * c_{in} + 1)$
  - → One (5x5) has ~ 25/18 ~ 1.4x more learned parameters
- FLOPs:
  - Two (3x3) layers : $2 * h_{out} * w_{out} * c_{out} * 3 * 3 * c_{in}$
  - One (5x5) layer: $h_{out} * w_{out} * c_{out} * 5 * 5 * c_{in}$
  - → One (5x5) layer needs more flops (25/18 ~ 1.4x more)
- Memory
  - Two (3x3) layers needs 2x memory because of intermediate activation maps

# In general, why stacking small filters is (thought to be) better

- Can achieve equivalent receptive field
- Fewer parameters to train
- Requires less computation
- Needs more memory, but not really a problem as long as you can fit into your GPU memory.  GPUs have high memory bandwidth

# In general, why stacking small filters is (thought to be) better

- Can achieve equivalent receptive field
- Fewer parameters to train
- Requires less computation
- Needs more memory, but not really a problem as long as you can fit into your GPU memory.  GPUs have high memory bandwidth
- Has multiple levels of non-linearities (ReLU)
- Less overfitting

# In general, why stacking small filters is (thought to be) better

- Can achieve equivalent receptive field
- Fewer parameters to train
- Requires less computation
- Needs more memory, but not really a problem as long as you can fit into your GPU memory. GPUs have high memory bandwidth
- Has multiple levels of non-linearities (ReLU)
- Less overfitting

**Stacking small filters seems like a clear win**

Note: 3x3 smallest size to still capture notion of center/up/down/left/right

# Maxpooling (2,2) stride 2

- Recall that pooling is discarding information
- But a necessity for practical reasons to control final volume size
- Don’t want to be too aggressive
- Non-overlapping stride follows intuition of doing a straight-forward downsampling

# Doubling Channel Size After Pooling

- A conv layer operating on a volume that has half of spatial dimensions and double channel size take the same number of flops.
- Keeps same compute time per layer (e.g. FLOPs)

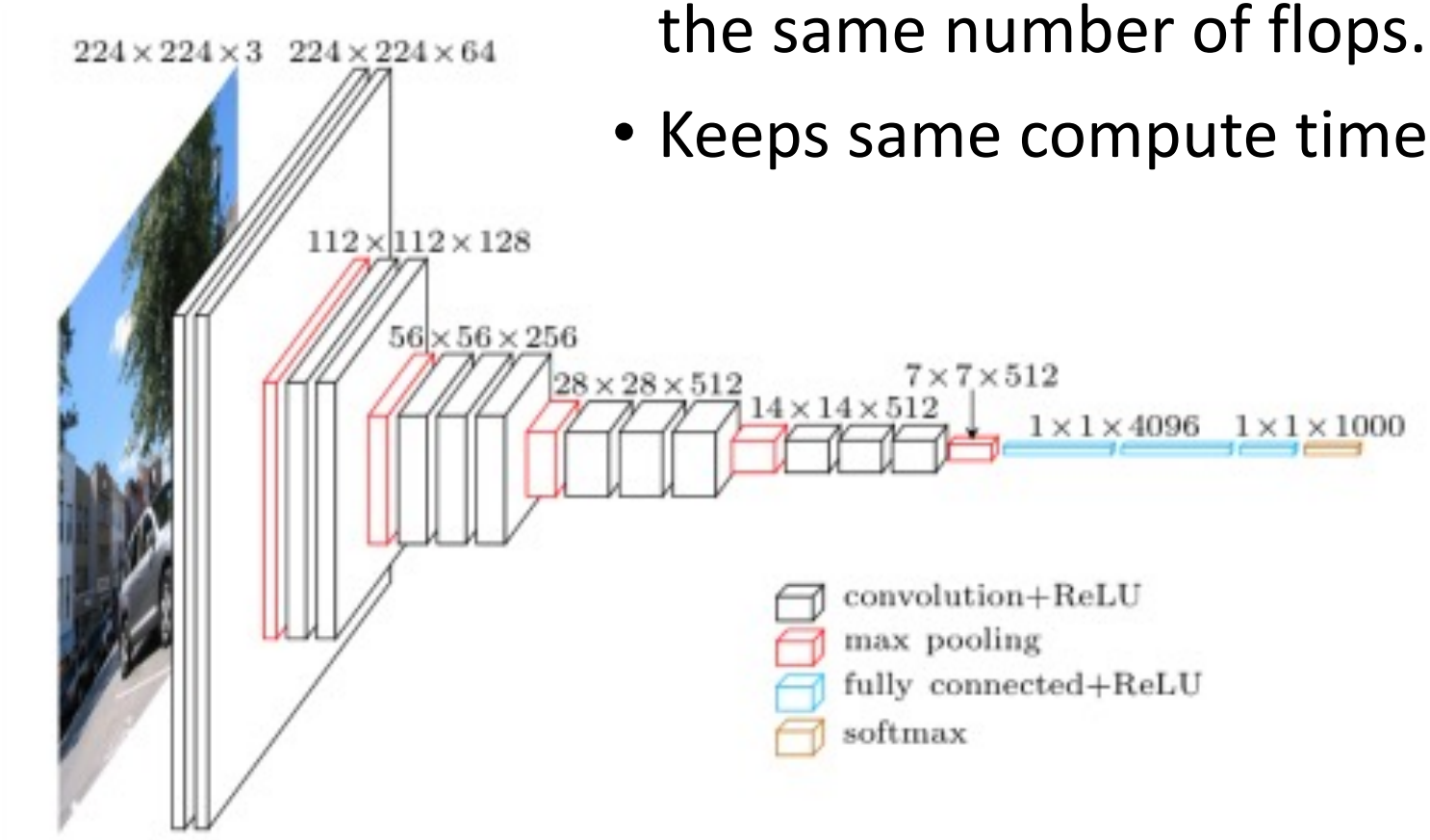



Image source: Davi Frossard https://www.cs.toronto.edu/~frossard/post/vgg16/

| Layer | HyperParams | Output | # Params | Mflops |
|---|---|---|---|---|
| Input | | (224,224,3) | | |
| Conv1 | K=64,f=(3,3),s=(1,1),p=same | (224,224,64) | 1792 | 86.7 |
| Conv2 | K=64,f=(3,3),s=(1,1),p=same | (224,224,64) | 36,928 | 1,849 |
| Max Pool | f=(2,2),s=(2,2) | (112,112,64) | 0 | 3.2 |
| Conv3 | K=128,f=(3,3),s=(1,1),p=same | (112,112,128) | 73,856 | 924 |
| Conv4 | K=128,f=(3,3),s=(1,1),p=same | (112,112,128) | 147,584 | 1,849 |
| Max Pool | f=(2,2),s=(2,2) | (56,56,128) | | 1.6 |
| Conv5 | K=256,f=(3,3),s=(1,1),p=same | (56,56,256) | 295,168 | 924 |
| Conv6 | K=256,f=(3,3),s=(1,1),p=same | (56,56,256) | 590,080 | 1,849 |
| Conv7 | K=256,f=(3,3),s=(1,1),p=same | (56,56,256) | 590,080 | 1,849 |
| Max Pool | f=z(2,2),s=(2,2) | (28,28,256) | 0 | 0.8 |

| Layer | HyperParams | Output | # Params | Mflops |
|---|---|---|---|---|
| Conv8 | K=512,f=(3,3),s=(1,1),p=same | (28,28,512) | 1,180,160 | 924 |
| Conv9 | K=512,f=(3,3),s=(1,1),p=same | (28,28,512) | 2,359,808 | 1,849 |
| Conv10 | K=512,f=(3,3),s=(1,1),p=same | (28,28,512) | 2,359,808 | 1,849 |
| Max Pool | f=(2,2),s=(2,2) | (14,14,512) | 0 | 0.4 |
| Conv11 | K=512,f=(3,3),s=(1,1),p=same | (14,14,512) | 2,359,808 | 462 |
| Conv12 | K=512,f=(3,3),s=(1,1),p=same | (14,14,512) | 2,359,808 | 462 |
| Conv13 | K=512,f=(3,3),s=(1,1),p=same | (14,14,512) | 2,359,808 | 462 |
| Max Pool | f=(2,2),s=(2,2) | (7,7,512) | 0 | 0.1 |
| Flatten | | (25088,) | 0 | |
| FC | n=4096 | (4096,) | 102,764,544 | 29 |
| FC | n=4096 | (4096,) | 16,781,312 | 16.8 |
| FC(softmax) | n=1000 | (1000,) | 4,097,000 | 4.1 |

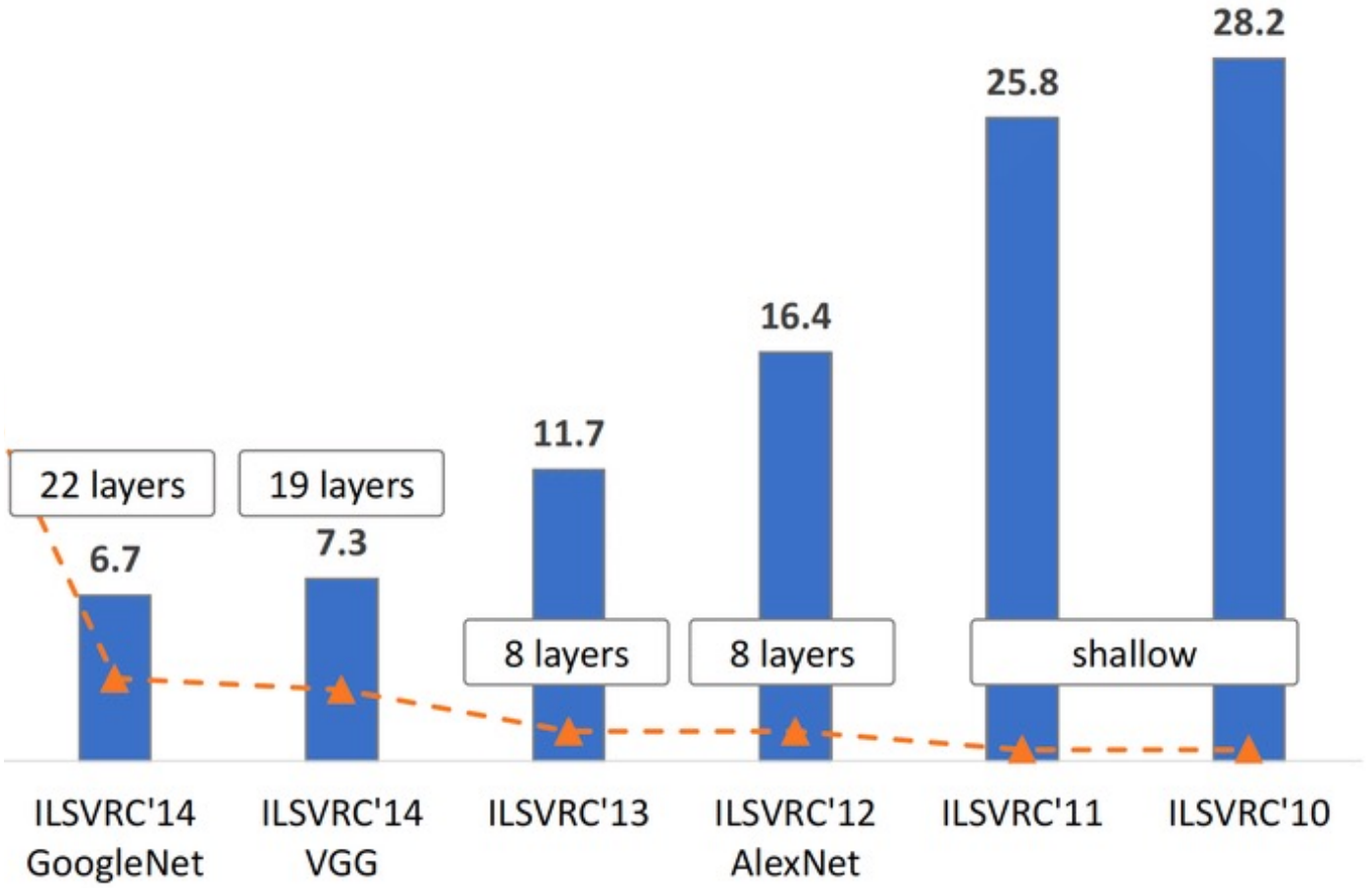
# Comparison so far

| Architecture | # Parameters (million | # GFLOPs | ImageNet top-5 error |
|---|---|---|---|
| AlexNet | 62 | 1 | 16.4 |
| ZFNet | 108 | 2.47 | 11.7 |
| VGG16 | 138 | 13.6 | 7.3 |

# VGG Summary

- Very uniform and straight forward architecture
- Has a really large number of parameters ~138million
- VGG19 only slightly better than VGG16
- Didn’t actually win the classification challenge (did win the localization challenge)

# Deep Learning at ImageNet



# GoogLeNet (Inception)

"Going Deeper with Convolutions", Szegedy et al., 2014, https://arxiv.org/abs/1409.4842


## References

[1] Know your meme: We need to go deeper. http://knowyourmeme.com/memes/we-need-to-go-deeper. Accessed: 2014-09-15.

[2] Sanjeev Arora, Aditya Bhaskara, Rong Ge, and Tengyu Ma. Provable bounds for learning some deep representations. *CoRR*, abs/1310.6343, 2013.

# Motivations

- Main motivation was on efficient use of compute resources

# Motivations

- Main motivation was on efficient use of compute resources
- Bigger architecture is (potentially) better, but …
  1. More parameters → more prone to overfitting → ok get more data → expensive

Siberian Husky

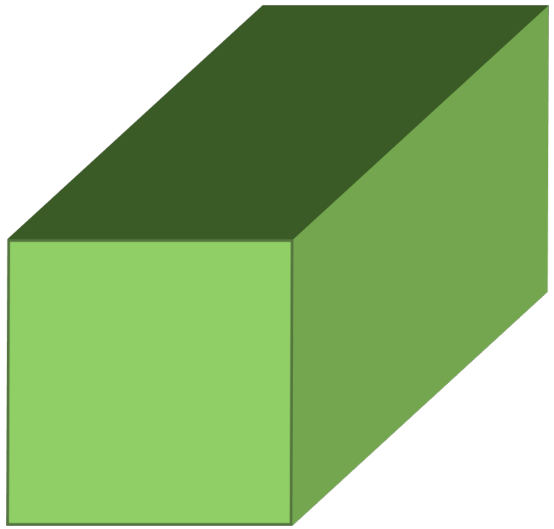
Eskimo Dog

# Motivations

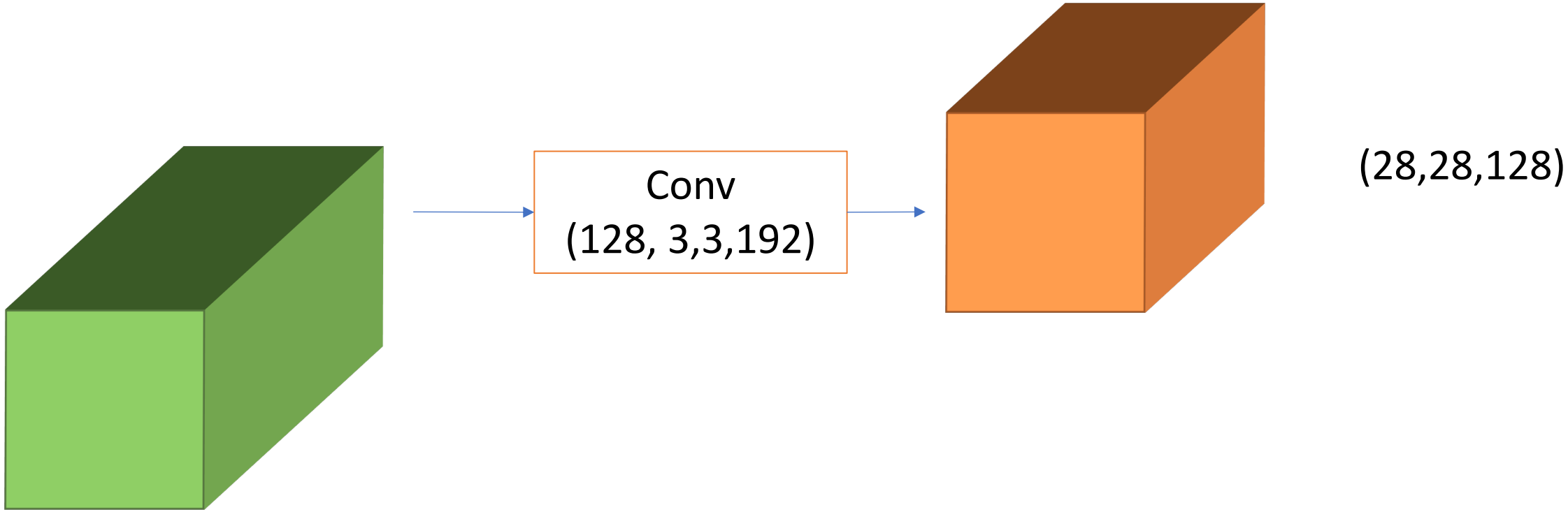
- Main motivation was on efficient use of compute resources
- Bigger architecture is (potentially) better, but …
  1. More parameters → more prone to overfitting → ok get more data → expensive
  2. Requires more computation → computation budget is finite → need to be more efficient with how you go bigger

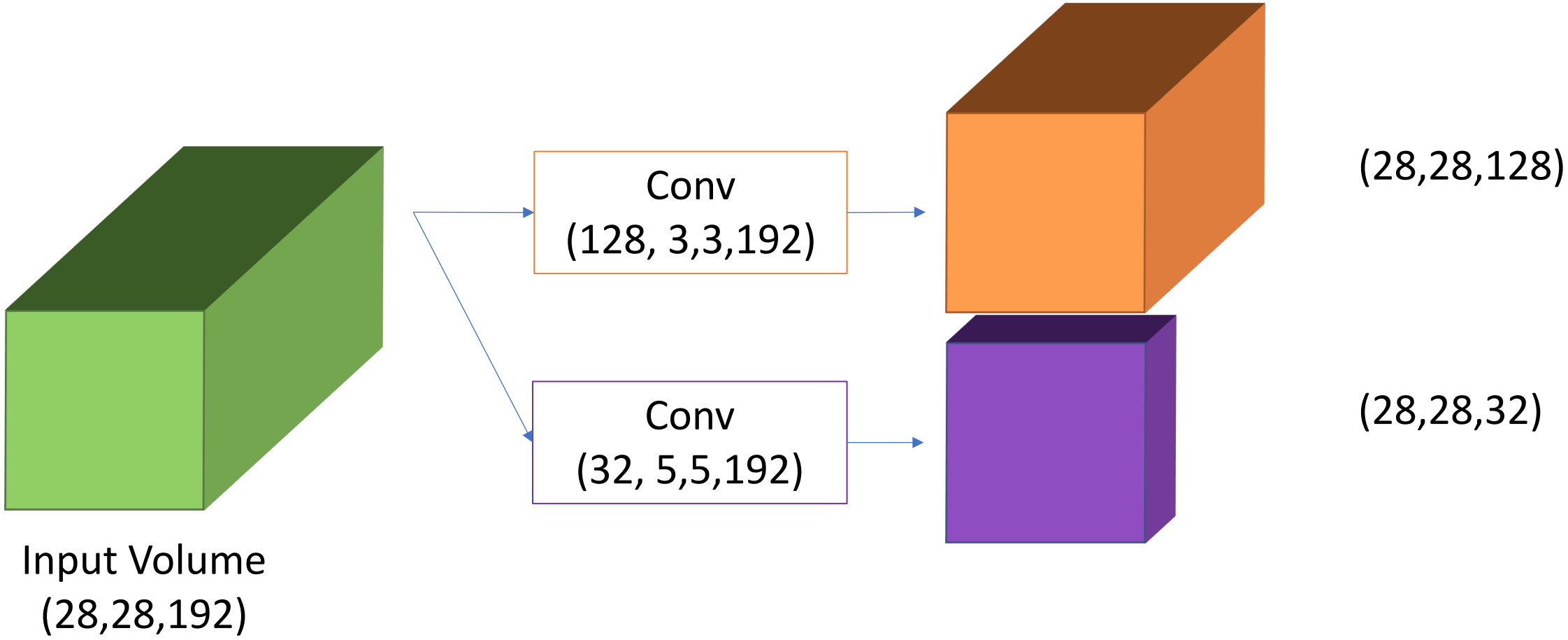
# Inception Module

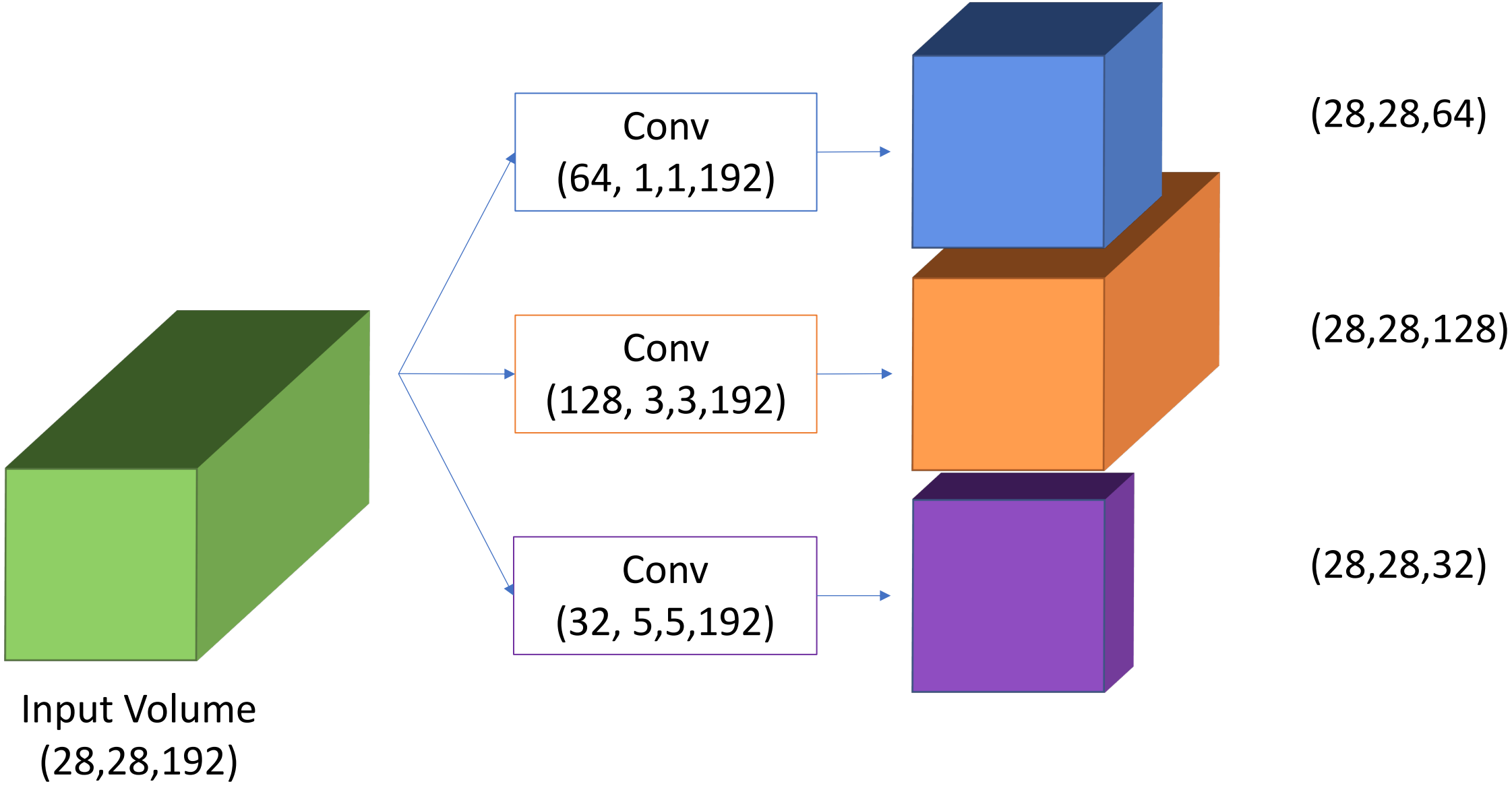
- Basic building block of the Inception Network
- VGGNet “eliminated” filter size as a hyperparameter by proposing to always use 3x3, and arguing that this has many benefits
- Inception Module ”eliminates” filter size as a hyperparameter by saying, “hey let’s just try them all!”
- Has filters of different sizes in a single layer
- Still Computationally Efficient

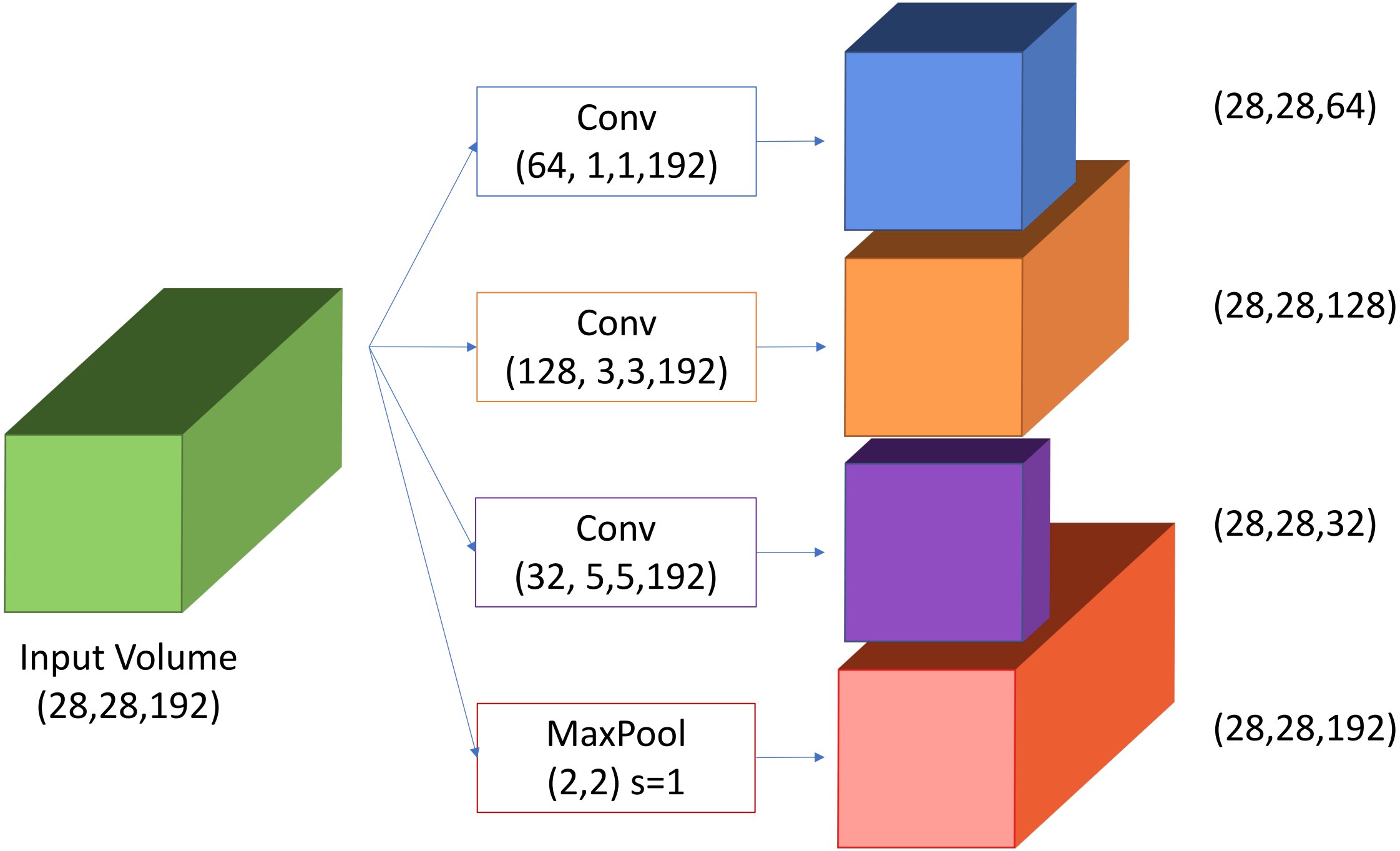


Input Volume
(28,28,192)

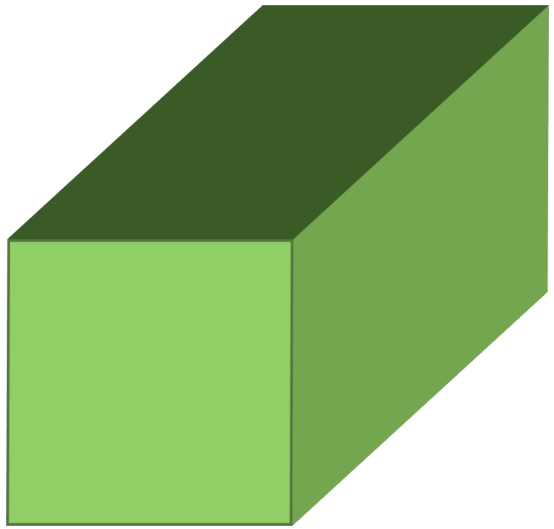

Conv
(128, 3,3,192)
(28,28,128)
Input Volume
(28,28,192)


Conv
(128, 3,3,192)
Conv
(32, 5,5,192)
(28,28,128)
(28,28,32)
Input Volume
(28,28,192)


Conv
(64, 1,1,192)
Conv
(128, 3,3,192)
Conv
(32, 5,5,192)
(28,28,64)
(28,28,128)
(28,28,32)
Input Volume
(28,28,192)


Input Volume
(28,28,192)
Conv
(64, 1,1,192)
Conv
(128, 3,3,192)
Conv
(32, 5,5,192)
MaxPool
(2,2) s=1
(28,28,64)
(28,28,128)
(28,28,32)
(28,28,192)

# Stack into a single volume



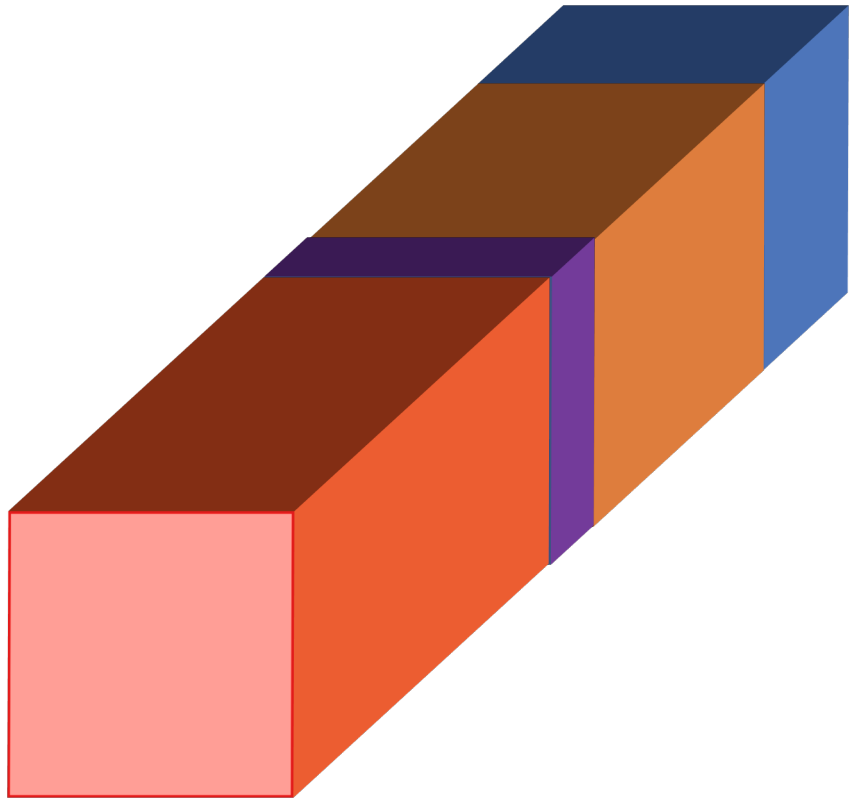
Input Volume
(28,28,192)

Conv
(64, 1,1,192)

Conv
(128, 3,3,192)

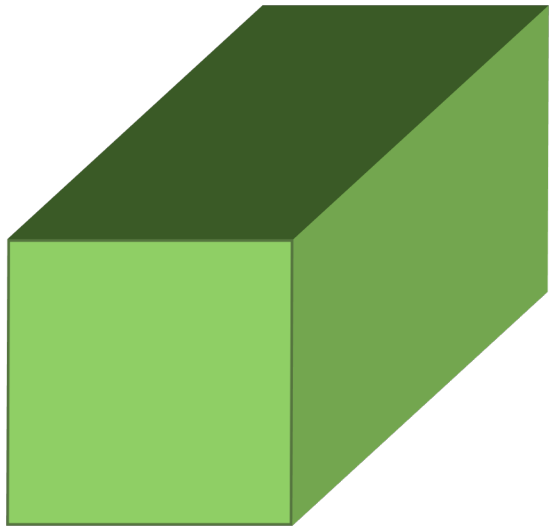
Conv
(32, 5,5,192)

MaxPool
(2,2) s=1



Output Volume
(28,28,416)

# Computationally Expensive!



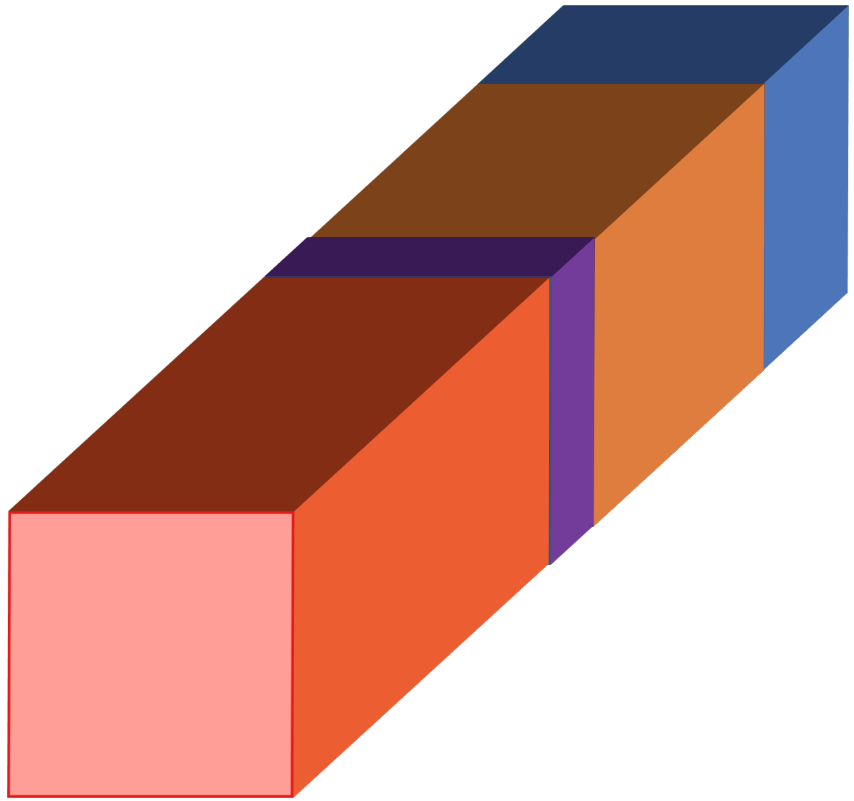
Input Volume
(28,28,192)

Conv
(64, 1,1,192)

Conv
(128, 3,3,192)

Conv
(32, 5,5,192)

MaxPool
(2,2) s=1



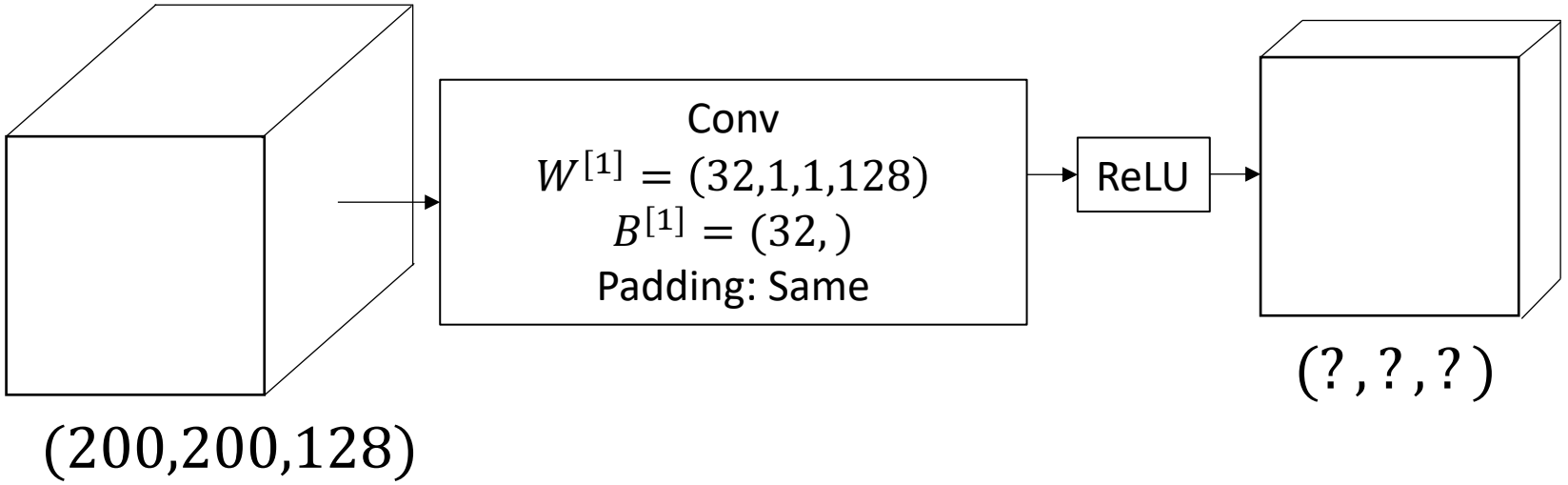
Output Volume
(28,28,416)

# 1x1 Convolutions

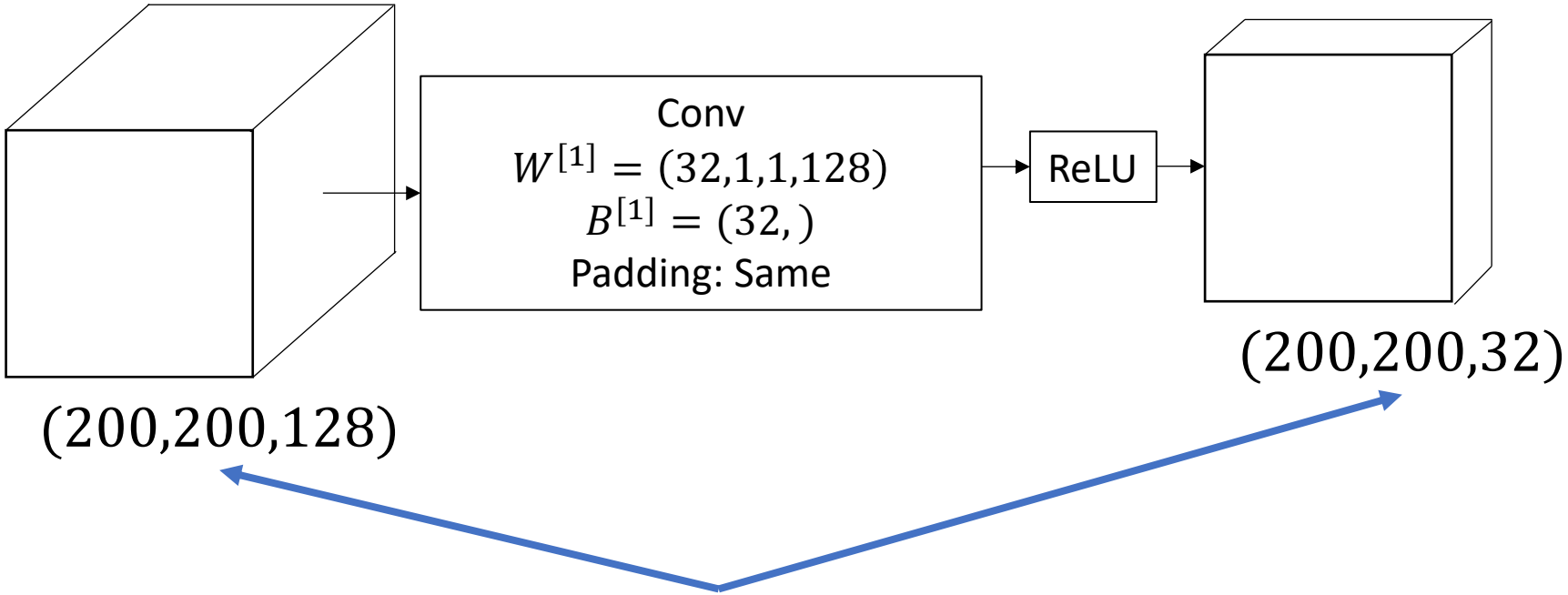
# 1x1 Convolutions

- Pooling allows us to downsample/reduce the **spatial dimensions**, but doesn't let us change the size of the channel dimension.
- We can reduce (or grow) the channel dimension using a convolution layer with 1x1 filters
- 1x1 filters may seem redundant, but remember filters have an implied third dimension equal to the input volume's number of channels.

# 1x1 Convolutions

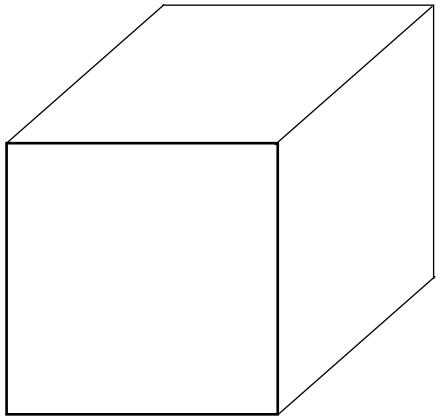


# 1x1 Convolutions



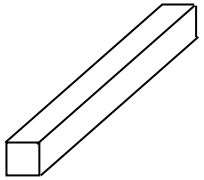

Channel size has reduced

# 1x1 Convolutions



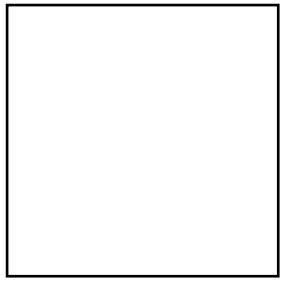
$(200,200,128)$

Input Volume



$(1,1,128)$

One Filter



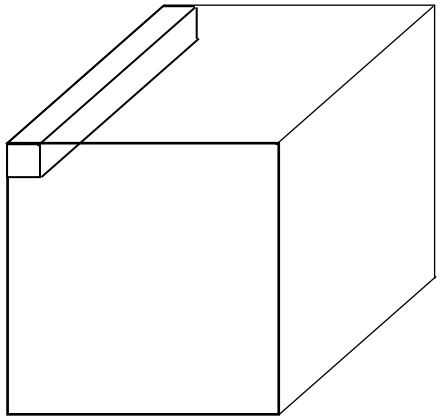
$(200,200,1)$

Activation Map for One Filter

For one (of the 32) filters,

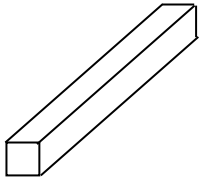
- Weighted-sum across all feature maps at each spatial location

# 1x1 Convolutions



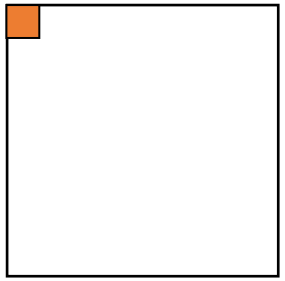
$(200,200,128)$

Input Volume



$(1,1,128)$

One Filter



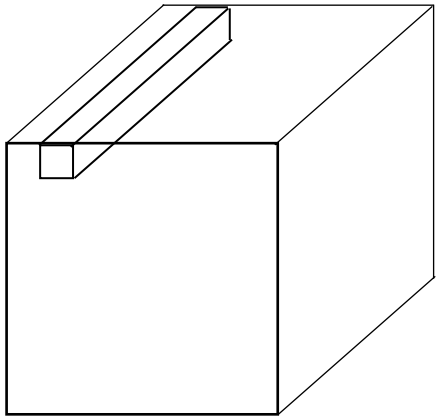
$(200,200,1)$

Activation Map for One Filter

For one (of the 32) filters,

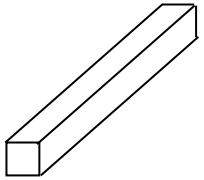
- Weighted-sum across all feature maps at each spatial location

# 1x1 Convolutions



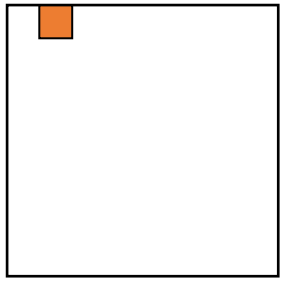
$(200,200,128)$

Input Volume



$(1,1,128)$

One Filter



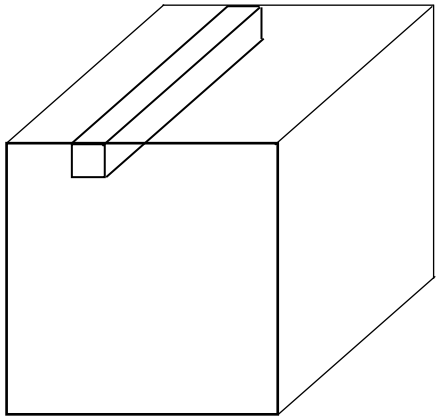
$(200,200,1)$

Activation Map for One Filter

For one (of the 32) filters,

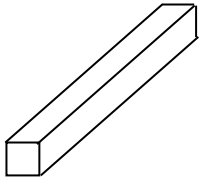
- Weighted-sum across all feature maps at each spatial location

# 1x1 Convolutions



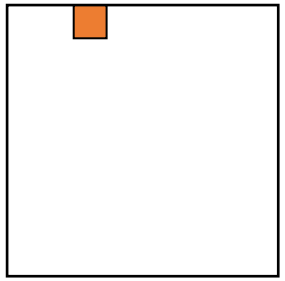
$(200,200,128)$

Input Volume



$(1,1,128)$

One Filter



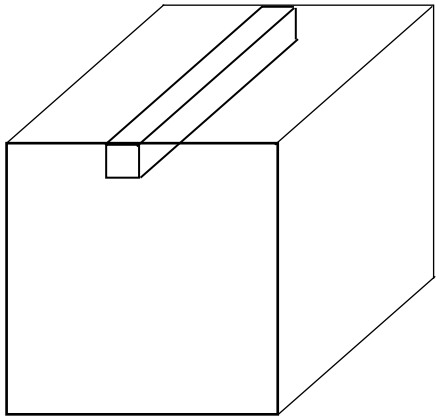
$(200,200,1)$

Activation Map for One Filter

For one (of the 32) filters,

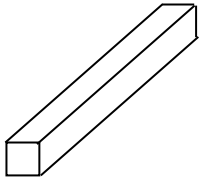
- Weighted-sum across all feature maps at each spatial location

# 1x1 Convolutions



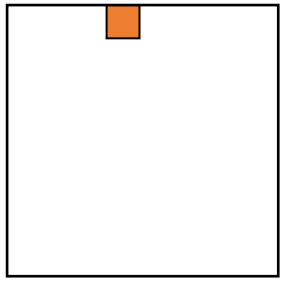
$(200,200,128)$

Input Volume



$(1,1,128)$

One Filter



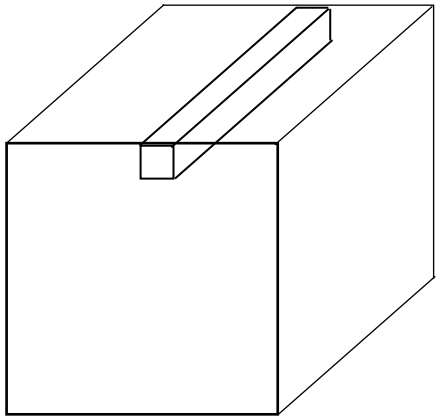
$(200,200,1)$

Activation Map for One Filter

For one (of the 32) filters,

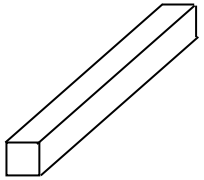
- Weighted-sum across all feature maps at each spatial location

# 1x1 Convolutions



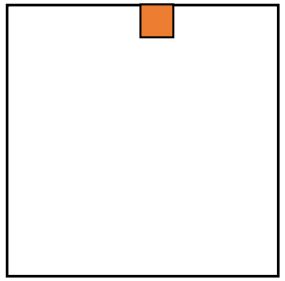
$(200,200,128)$

Input Volume



$(1,1,128)$

One Filter



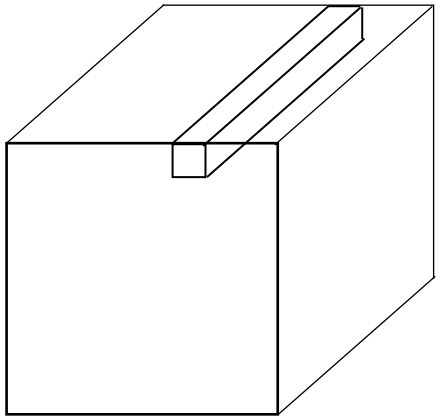
$(200,200,1)$

Activation Map for One Filter

For one (of the 32) filters,

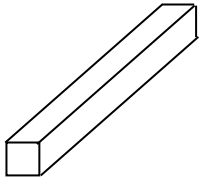
- Weighted-sum across all feature maps at each spatial location

# 1x1 Convolutions



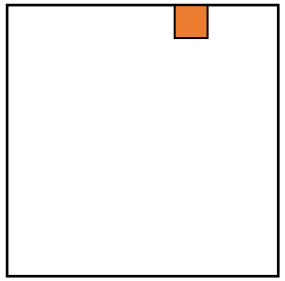
$(200,200,128)$

Input Volume



$(1,1,128)$

One Filter



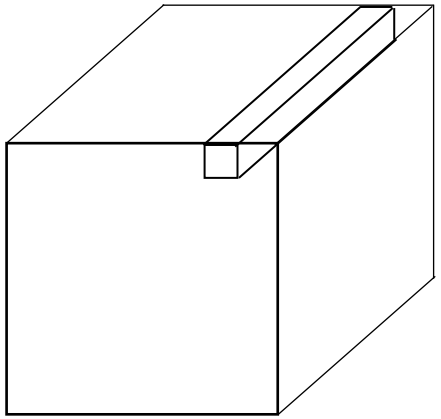
$(200,200,1)$

Activation Map for One Filter

For one (of the 32) filters,

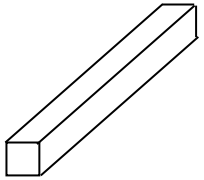
- Weighted-sum across all feature maps at each spatial location

# 1x1 Convolutions



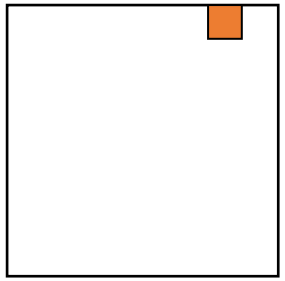
$(200,200,128)$

Input Volume



$(1,1,128)$

One Filter



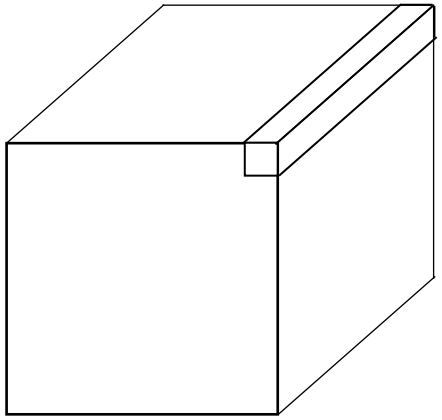
$(200,200,1)$

Activation Map for One Filter

For one (of the 32) filters,

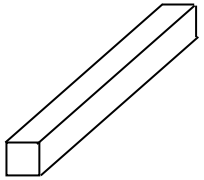
- Weighted-sum across all feature maps at each spatial location

# 1x1 Convolutions



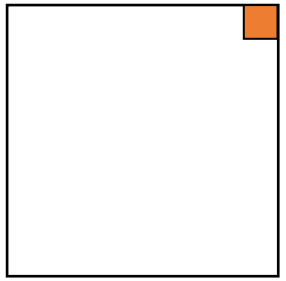
$(200,200,128)$

Input Volume



$(1,1,128)$

One Filter



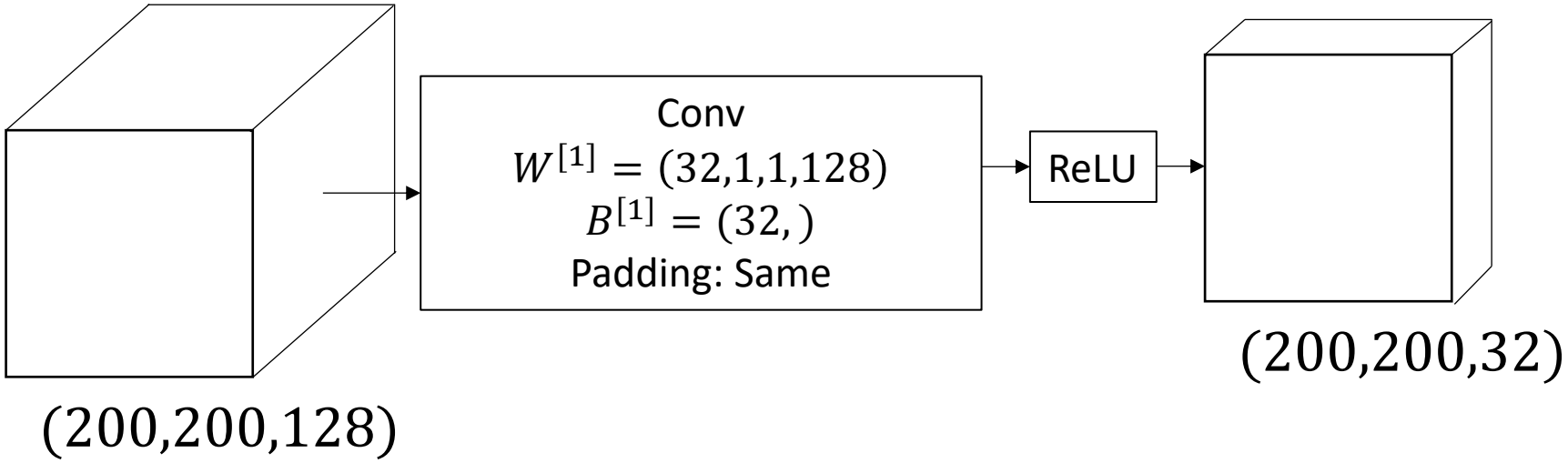
$(200,200,1)$

Activation Map for One Filter

For one (of the 32) filters,

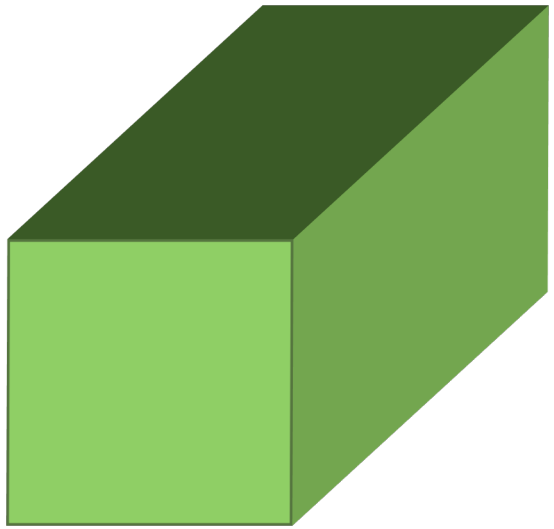
- Weighted-sum across all feature maps at each spatial location

# 1x1 Convolutions




- Conceptually like a form of compression where compression scheme is learned from the data
- Output features are a composition of the input features (potentially more complex)

# Computationally Expensive!



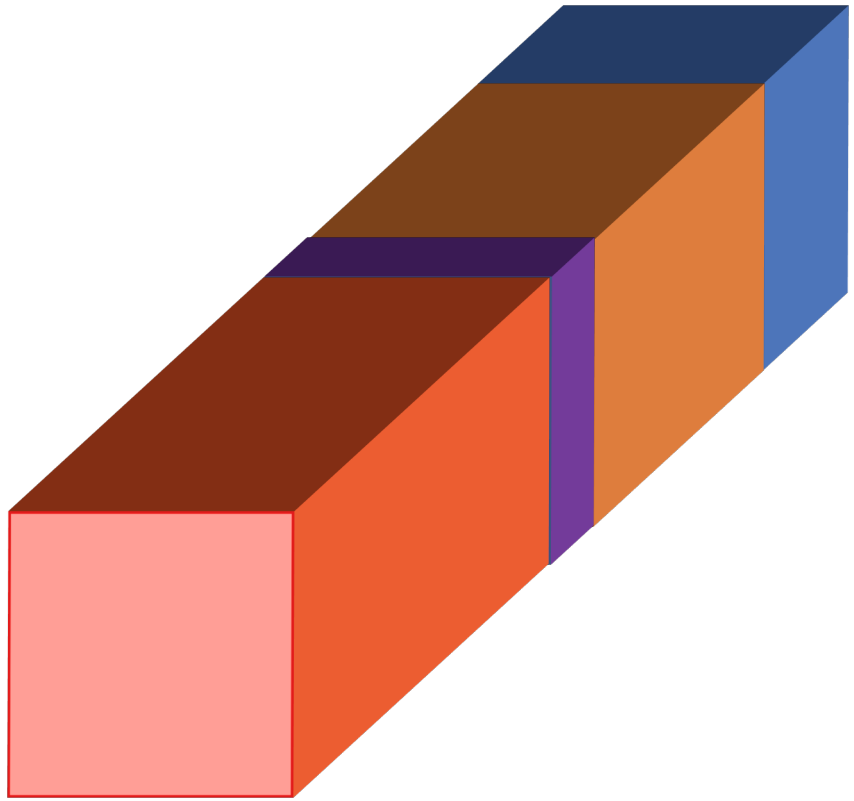
Input Volume
(28,28,192)

Conv
(64, 1,1,192)

Conv
(128, 3,3,192)

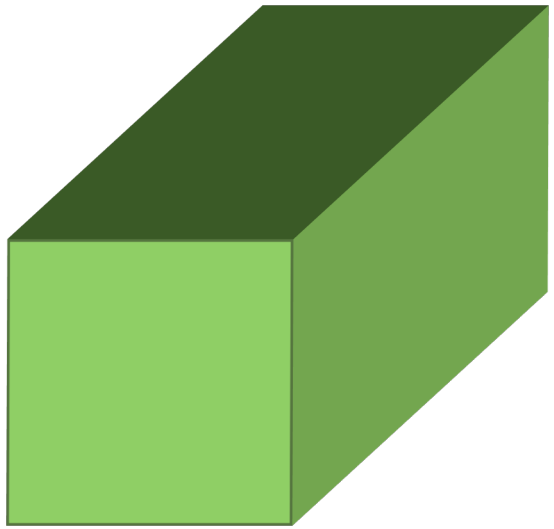
Conv
(32, 5,5,192)

MaxPool
(2,2) s=1



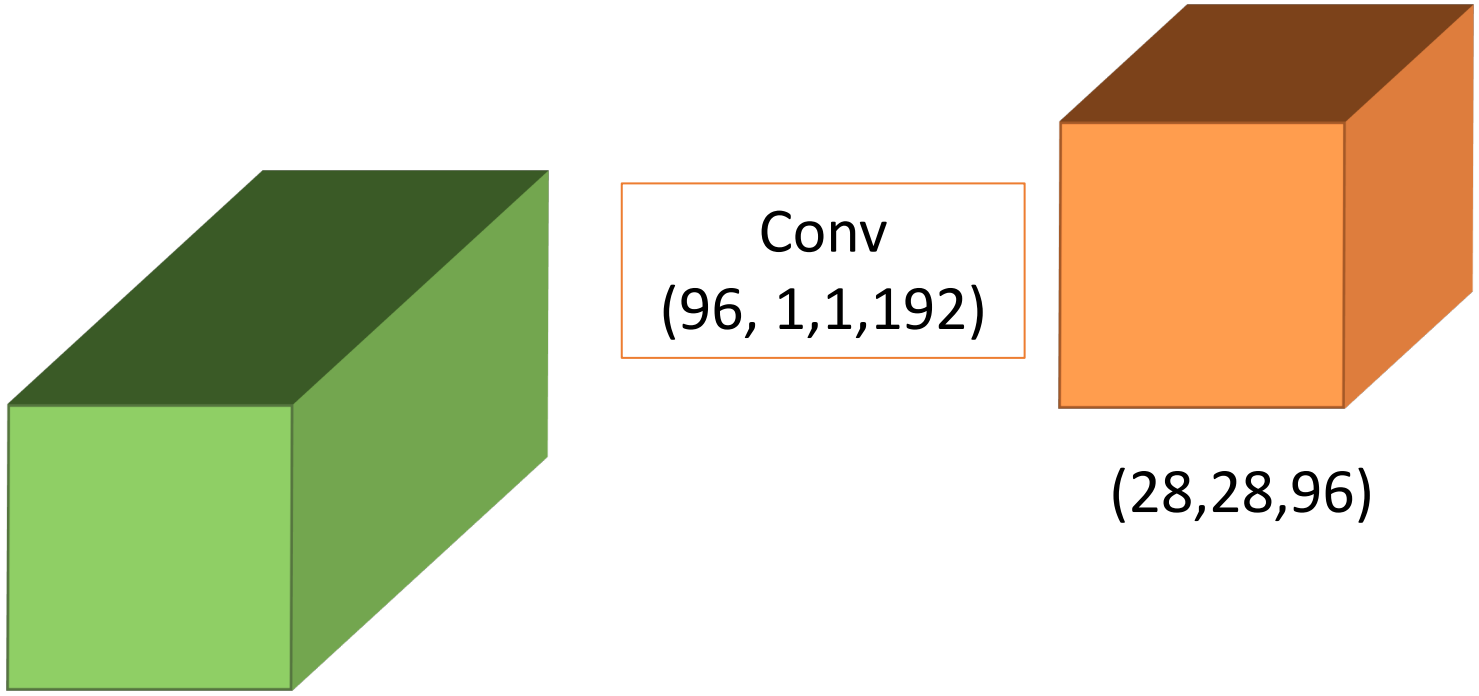
Output Volume
(28,28,416)

# Alternative: Use 1x1 Convolution to shrink input volume

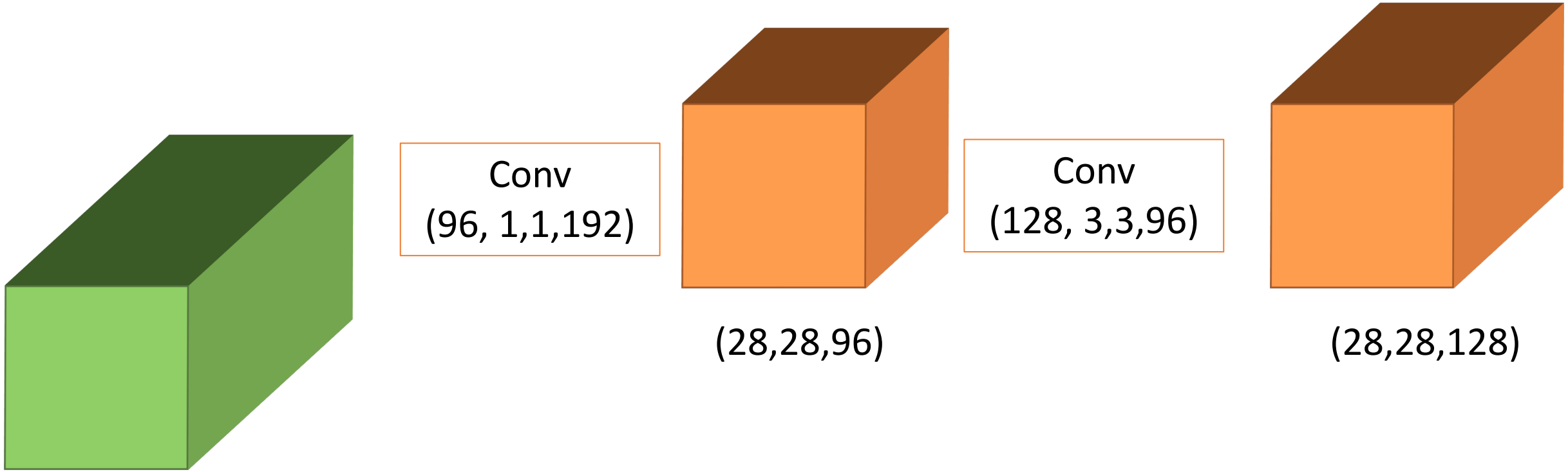


Input Volume
(28,28,192)

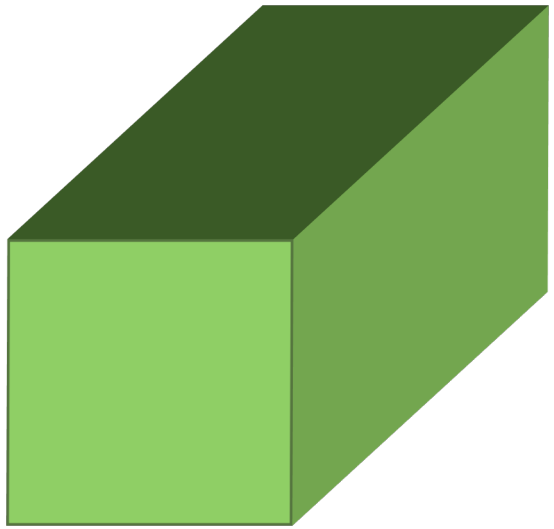
# Alternative: Use 1x1 Convolution to shrink input volume



# Alternative: Use 1x1 Convolution to shrink input volume

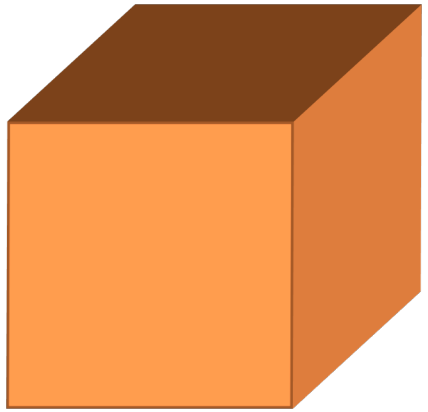


# Alternative: Use 1x1 Convolution to shrink input volume



Input Volume
(28,28,192)

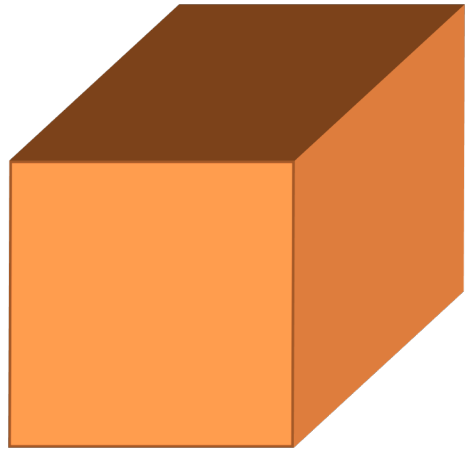
Conv
(96, 1,1,192)



(28,28,96)

14.4MFlops

Conv
(128, 3,3,96)



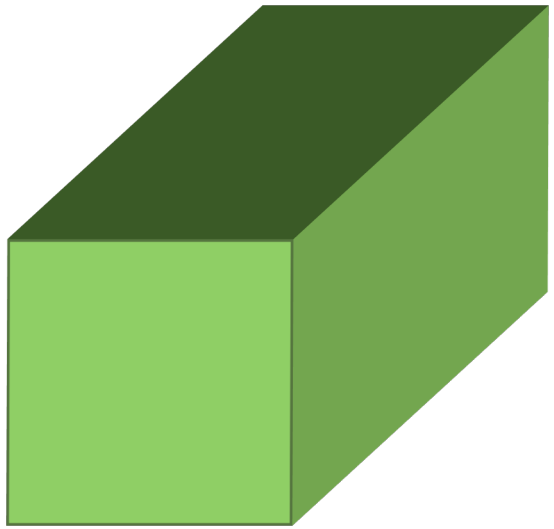
(28,28,128)

86.7MFlops

Total = 101MFlops

Before: 173.4MFlops

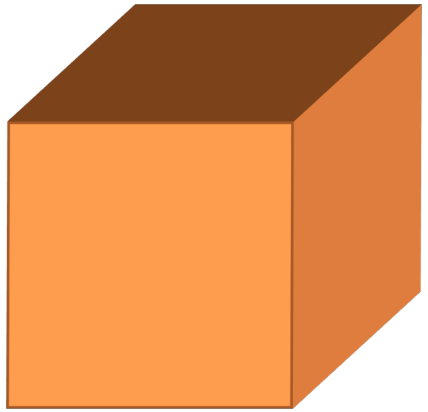
# Alternative: Use 1x1 Convolution to shrink input volume



Input Volume
(28,28,192)

Conv
(96, 1,1,192)

# Params:
18,528

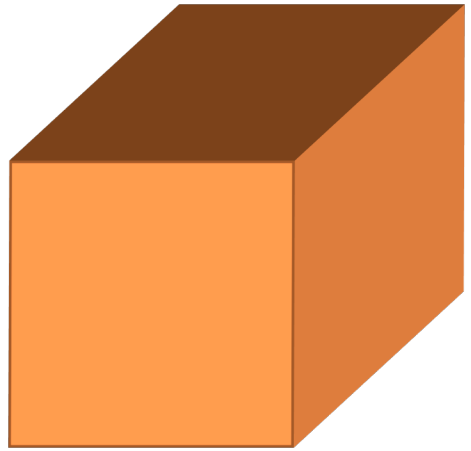


(28,28,96)

14.4MFlops

Conv
(128, 3,3,96)

# Params:
110,720



(28,28,128)

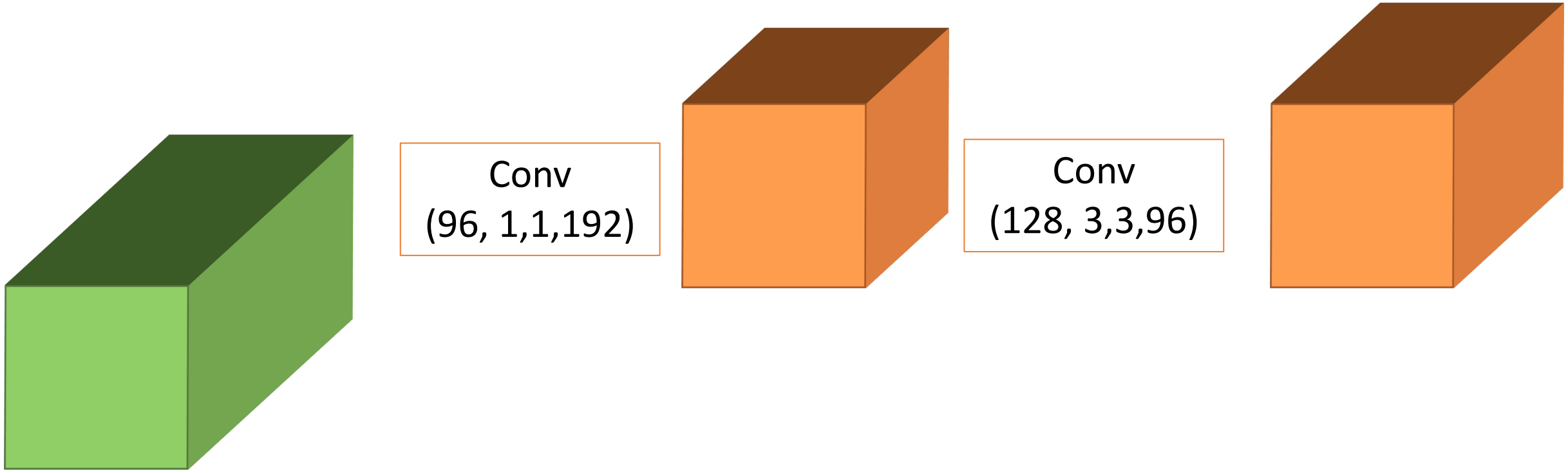
86.7MFlops

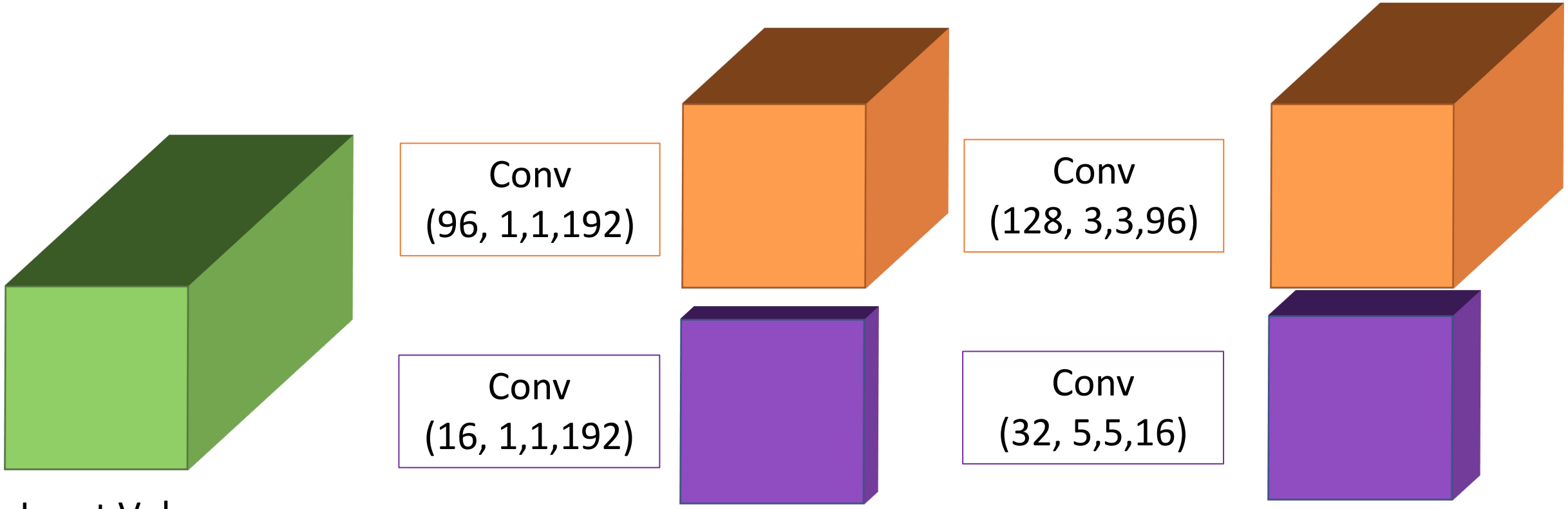
Total = 101MFlops

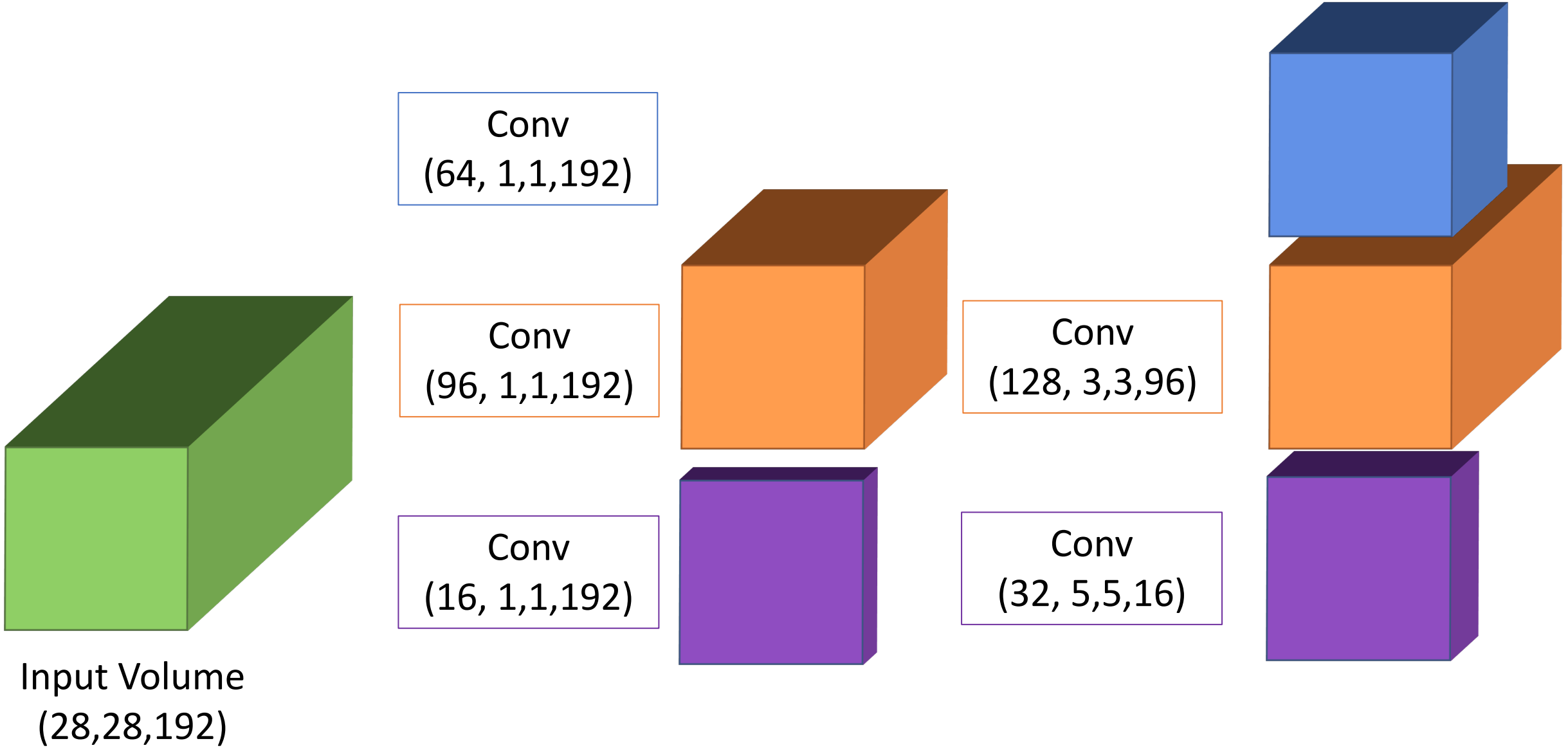
Before: 173.4MFlops

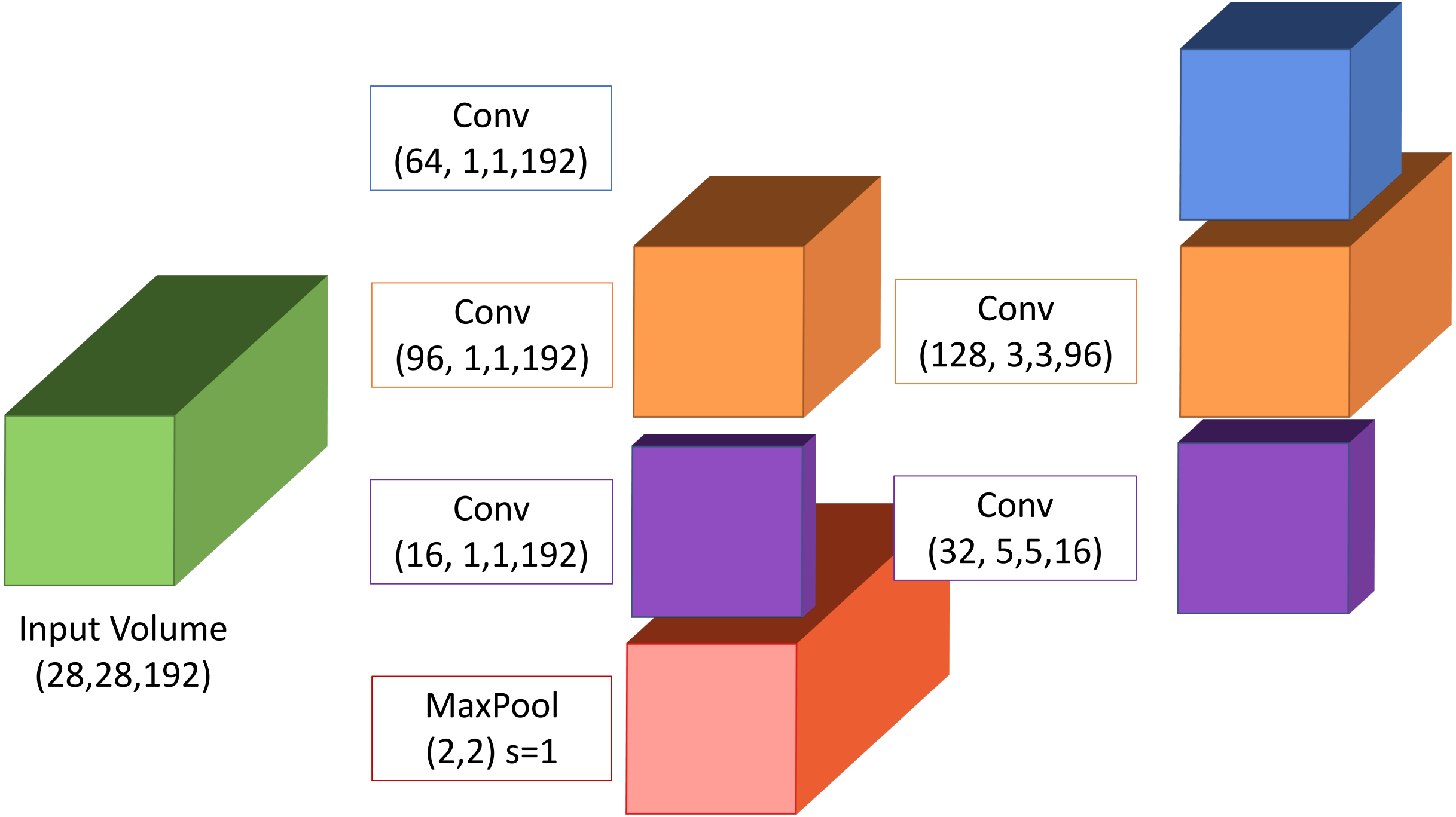
Total = 129,248 params

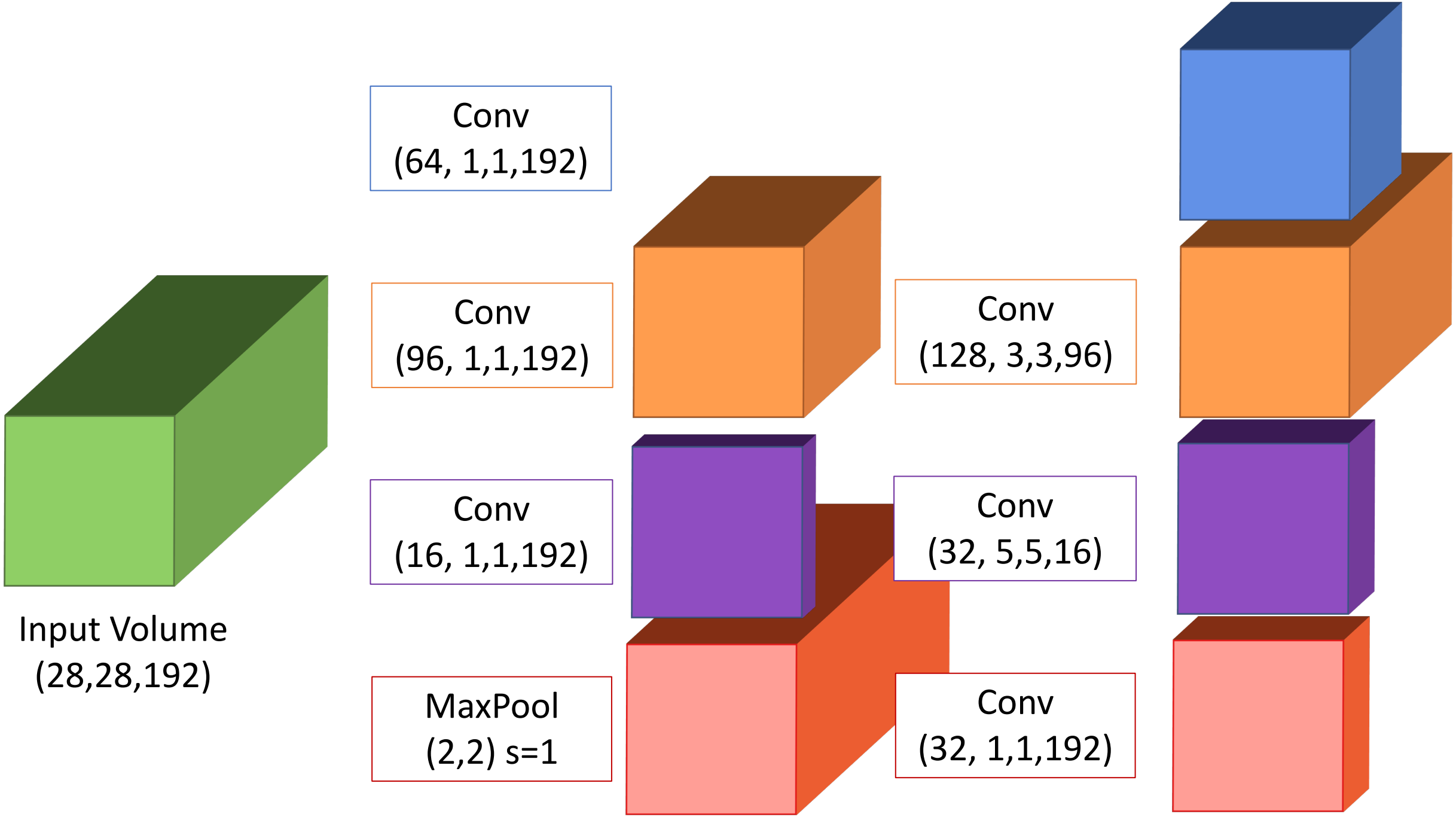
Before: 221,312 params


Conv
(96, 1,1,192)
Conv
(128, 3,3,96)
Input Volume
(28,28,192)


Conv
(96, 1,1,192)
Conv
(16, 1,1,192)
Conv
(128, 3,3,96)
Conv
(32, 5,5,16)
Input Volume
(28,28,192)

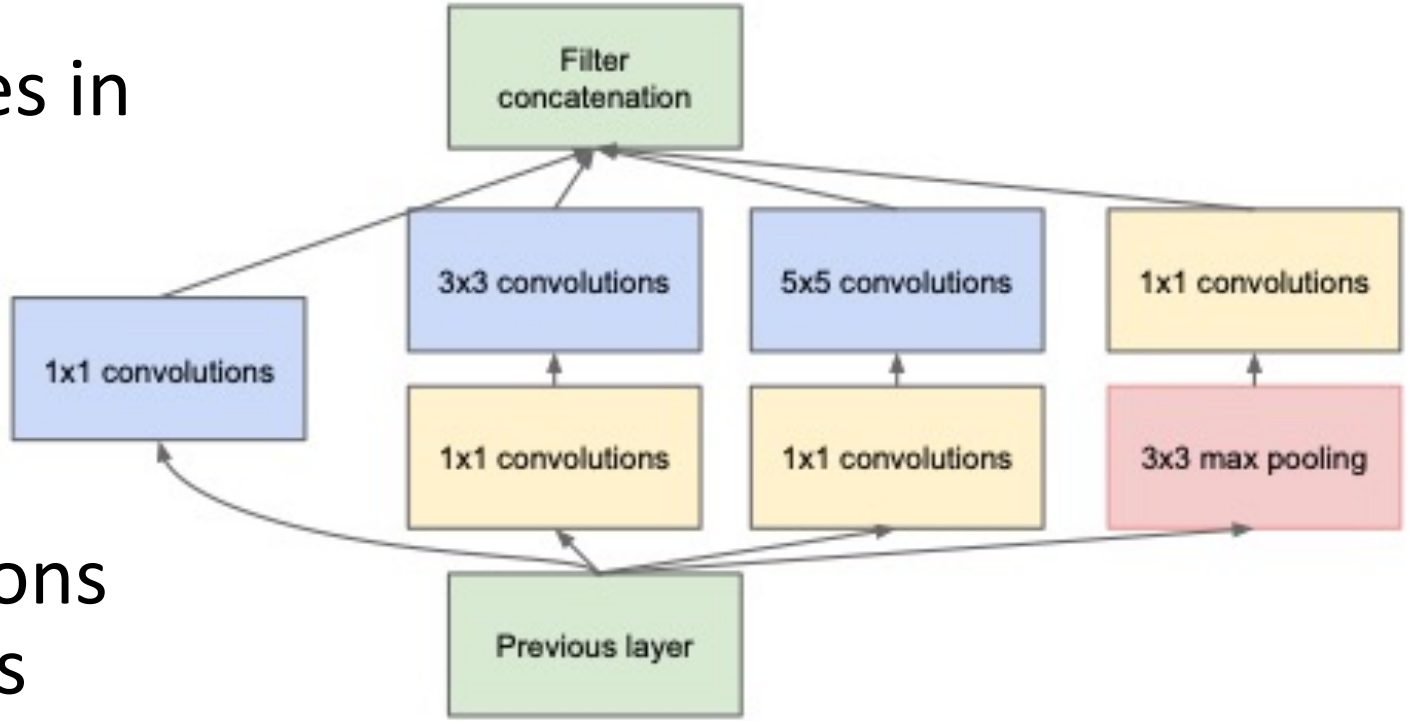

Conv
(64, 1,1,192)
Conv
(96, 1,1,192)
Conv
(16, 1,1,192)
Conv
(128, 3,3,96)
Conv
(32, 5,5,16)
Input Volume
(28,28,192)


Conv
(64, 1,1,192)
Conv
(96, 1,1,192)
Conv
(128, 3,3,96)
Conv
(16, 1,1,192)
Conv
(32, 5,5,16)
MaxPool
(2,2) s=1
Input Volume
(28,28,192)


Conv
(64, 1,1,192)
Conv
(96, 1,1,192)
Conv
(128, 3,3,96)
Conv
(16, 1,1,192)
Conv
(32, 5,5,16)
MaxPool
(2,2) s=1
Conv
(32, 1,1,192)
Input Volume
(28,28,192)

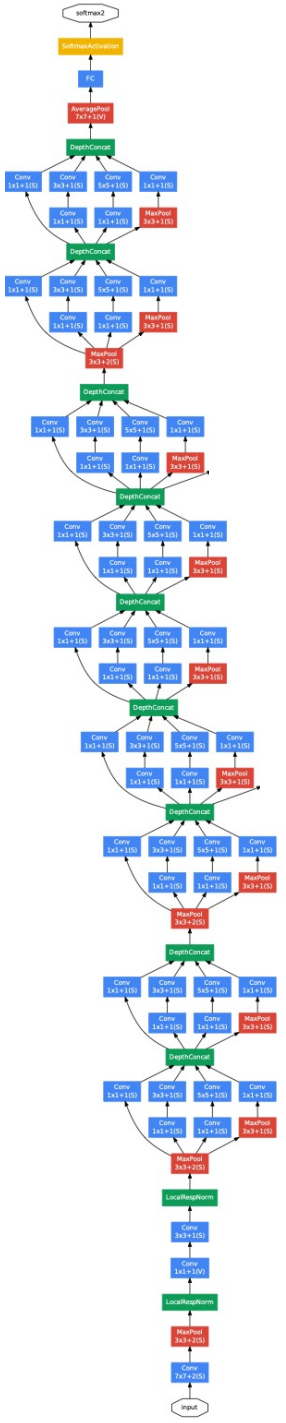
# Summary: Inception Module

- Has filters of different sizes in same layer
- Use 1x1 convolutions to improve computation efficiency
- Intuition of 1x1 convolutions is combining feature maps
- Doesn't hurt as long as not too aggressive




"Going Deeper with Convolutions", Szegedy et al., 2014, https://arxiv.org/abs/1409.4842

| type | patch size/ stride | output size | depth | #1×1 | #3×3 reduce | #3×3 | #5×5 reduce | #5×5 | pool proj | params | ops |
|---|---|---|---|---|---|---|---|---|---|---|---|
| convolution | 7×7/2 | 112×112×64 | 1 | | | | | | | 2.7K | 34M |
| max pool | 3×3/2 | 56×56×64 | 0 | | | | | | | | |
| convolution | 3×3/1 | 56×56×192 | 2 | | 64 | 192 | | | | 112K | 360M |
| max pool | 3×3/2 | 28×28×192 | 0 | | | | | | | | |
| inception (3a) | | 28×28×256 | 2 | 64 | 96 | 128 | 16 | 32 | 32 | 159K | 128M |
| inception (3b) | | 28×28×480 | 2 | 128 | 128 | 192 | 32 | 96 | 64 | 380K | 304M |
| max pool | 3×3/2 | 14×14×480 | 0 | | | | | | | | |
| inception (4a) | | 14×14×512 | 2 | 192 | 96 | 208 | 16 | 48 | 64 | 364K | 73M |
| inception (4b) | | 14×14×512 | 2 | 160 | 112 | 224 | 24 | 64 | 64 | 437K | 88M |
| inception (4c) | | 14×14×512 | 2 | 128 | 128 | 256 | 24 | 64 | 64 | 463K | 100M |
| inception (4d) | | 14×14×528 | 2 | 112 | 144 | 288 | 32 | 64 | 64 | 580K | 119M |
| inception (4e) | | 14×14×832 | 2 | 256 | 160 | 320 | 32 | 128 | 128 | 840K | 170M |
| max pool | 3×3/2 | 7×7×832 | 0 | | | | | | | | |
| inception (5a) | | 7×7×832 | 2 | 256 | 160 | 320 | 32 | 128 | 128 | 1072K | 54M |
| inception (5b) | | 7×7×1024 | 2 | 384 | 192 | 384 | 48 | 128 | 128 | 1388K | 71M |
| avg pool | 7×7/1 | 1×1×1024 | 0 | | | | | | | | |
| dropout (40%) | | 1×1×1024 | 0 | | | | | | | | |
| linear | | 1×1×1000 | 1 | | | | | | | 1000K | 1M |
| softmax | | 1×1×1000 | 0 | | | | | | | | |



"Going Deeper with Convolutions", Szegedy et al., 2014, https://arxiv.org/abs/1409.4842

# Global Average Pooling

- Traditionally, final layers is a flattening of the final volume into a vector, and sending this to one or more fully-connected layers.
  - Huge vector → large number of parameters for subsequent FC layer

Another approach

- Average pool across the entirety of each activation map →one number per activation map
- Resulting vector is fed to subsequent FC layers

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| inception (5b) | | $7\times7\times1024$ | 2 | 384 | 192 | 384 | 48 | 128 | 128 | 1388K | 71M |
| avg pool | $7\times7/1$ | $1\times1\times1024$ | 0 | | | | | | | | |
| dropout (40%) | | $1\times1\times1024$ | 0 | | | | | | | | |
| linear | | $1\times1\times1000$ | 1 | | | | | | | 1000K | 1M |
| softmax | | $1\times1\times1000$ | 0 | | | | | | | | |

# Global Average Pooling - Advantages

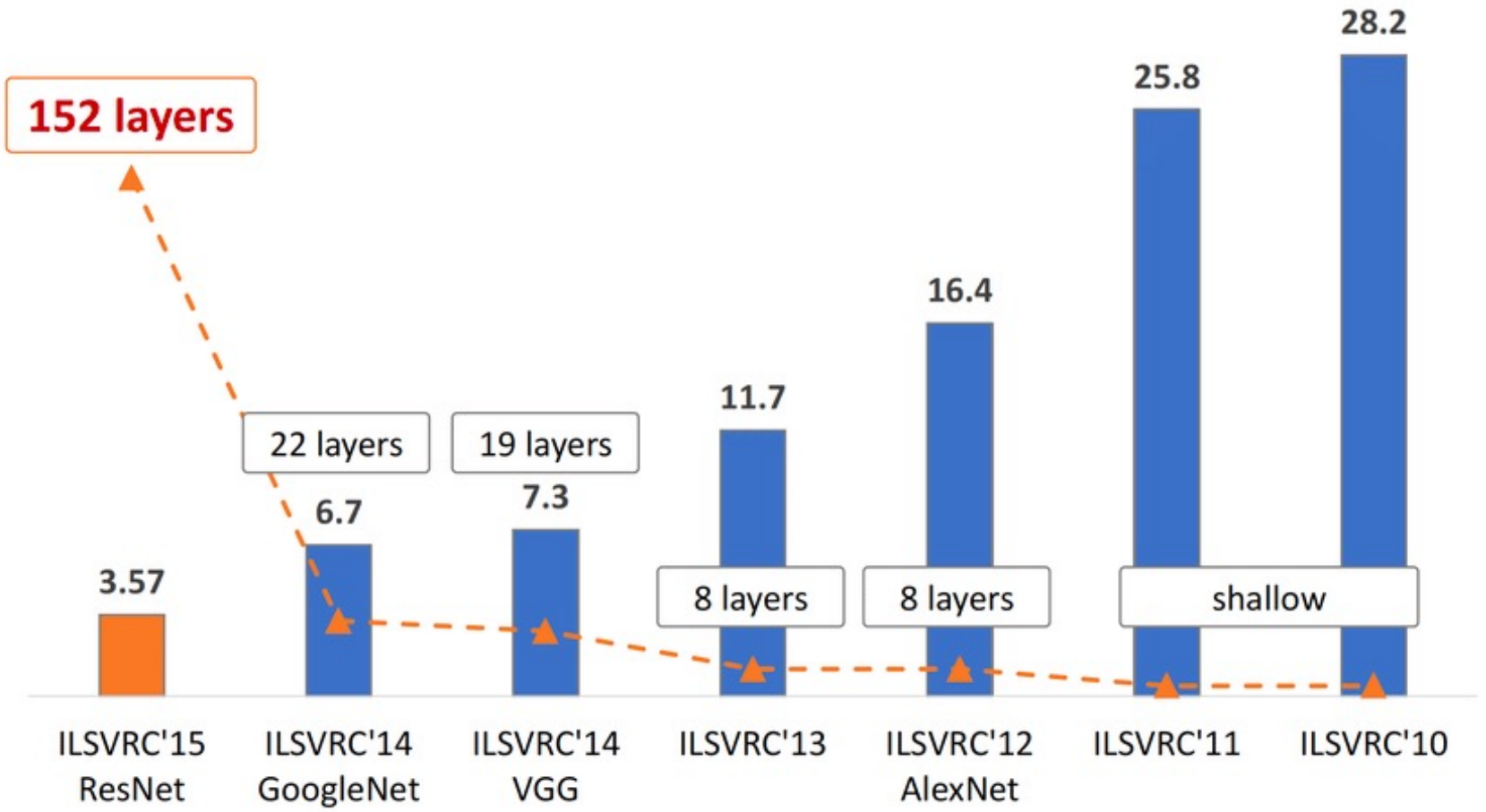
- Pooling operation is essentially ”free”
- No parameters to optimize so less prone to overfitting
- Since we are looking over the entire feature map, thus more robust to spatial translation of the final activations

“Network in Network”, Lin, Chen, Yan, 2013 https://arxiv.org/pdf/1312.4400.pdf

# Comparison so far

| Architecture | # Parameters (millions) | # GFLOPs | ImageNet top-5 error |
|---|---|---|---|
| AlexNet | 62 | 1 | 16.4 |
| ZFNet | 108 | 2.47 | 11.7 |
| VGG16 | 138 | 13.6 | 7.3 |
| Inception (GoogLeNet) | 6.8 | 1.5 | 6.7 |

# Deep Learning at ImageNet



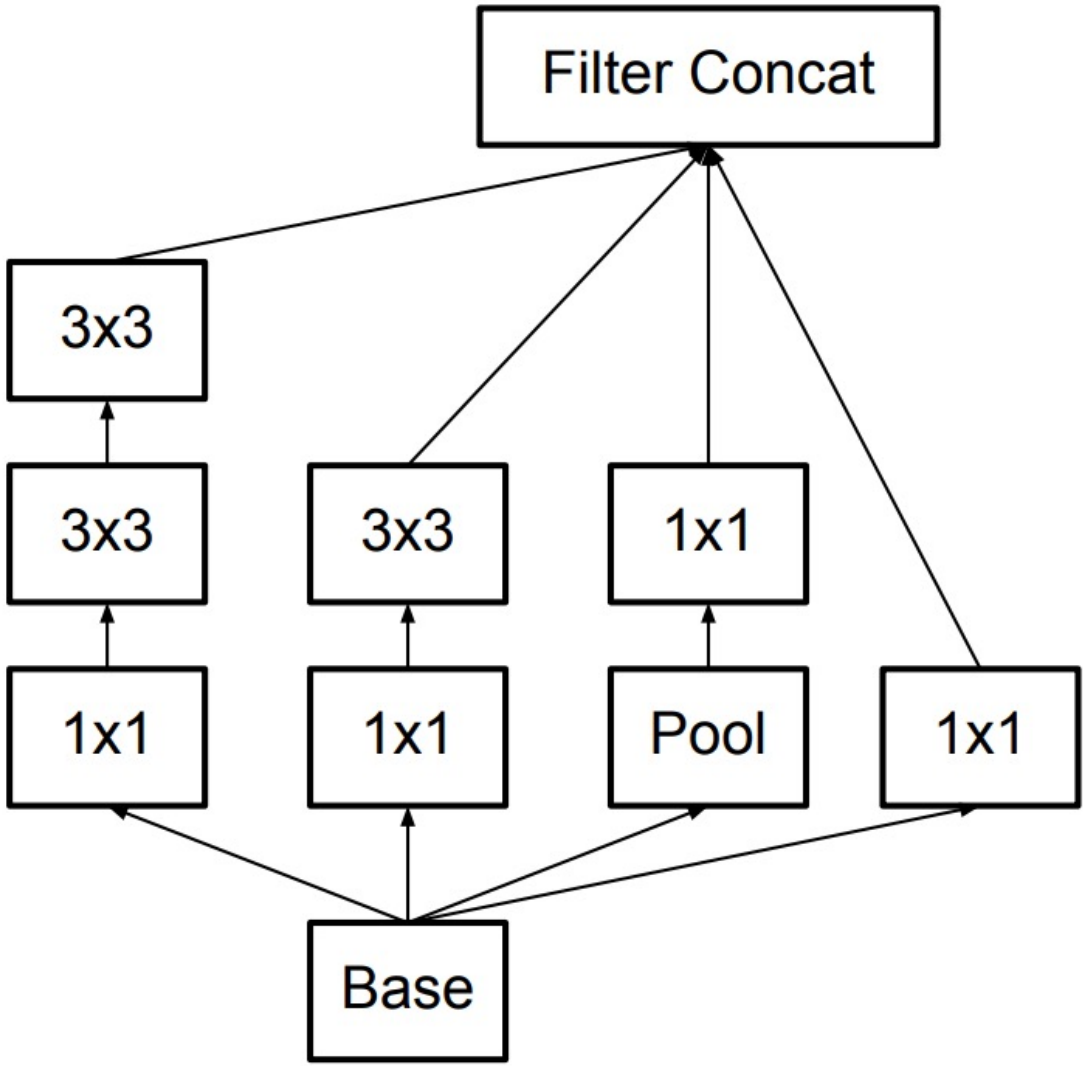



Brad Quinton, Scott Chin

# InceptionV3 (ReCeption)

“Rethinking the Inception Architecture for Computer Vision”, Szegedy et al., 2015, https://arxiv.org/abs/1512.00567

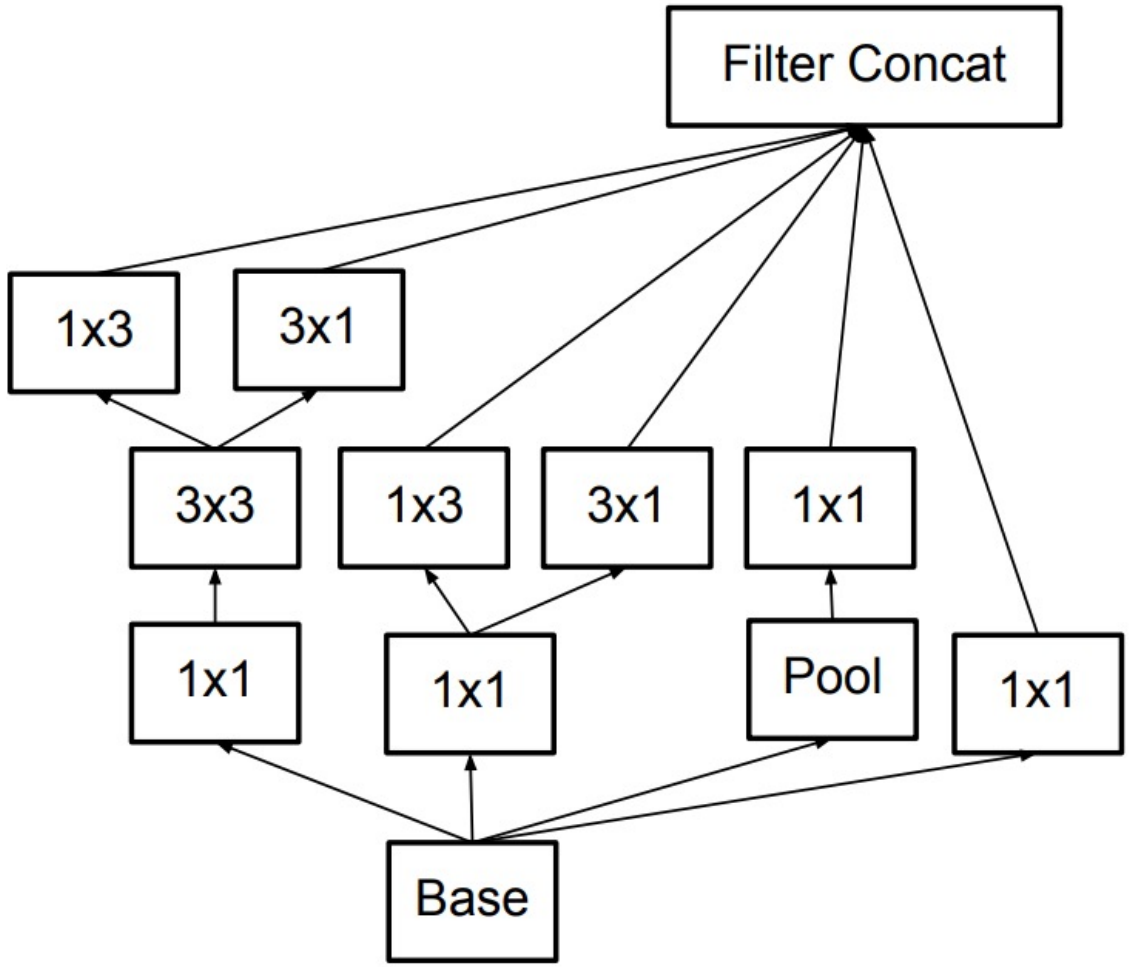
# Three types of Inception Modules

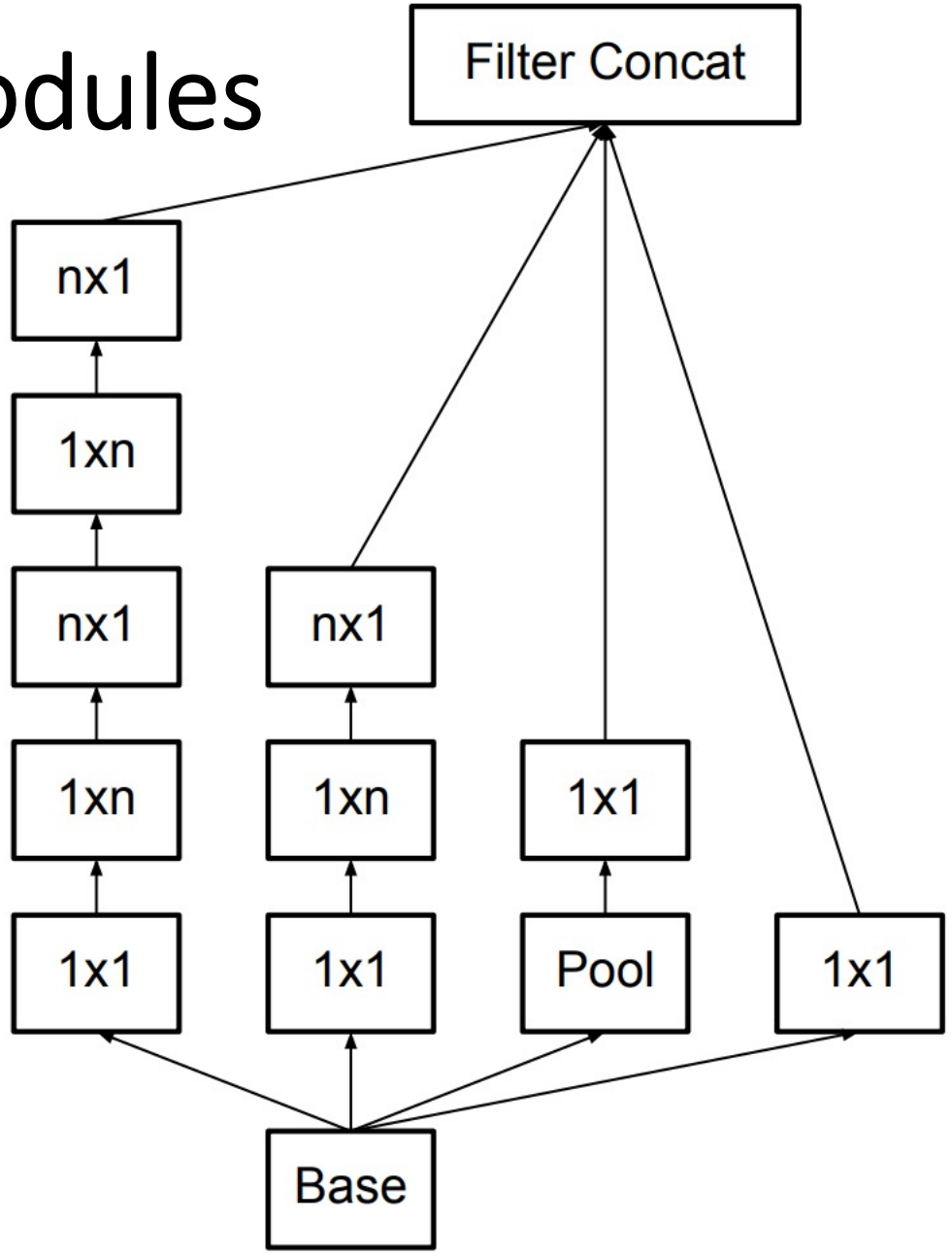



First Inception Module:

Same as GoogLeNet's Inception Module except 5x5 filters replaced by two layers of 3x3 filters

# Three types of Inception Modules



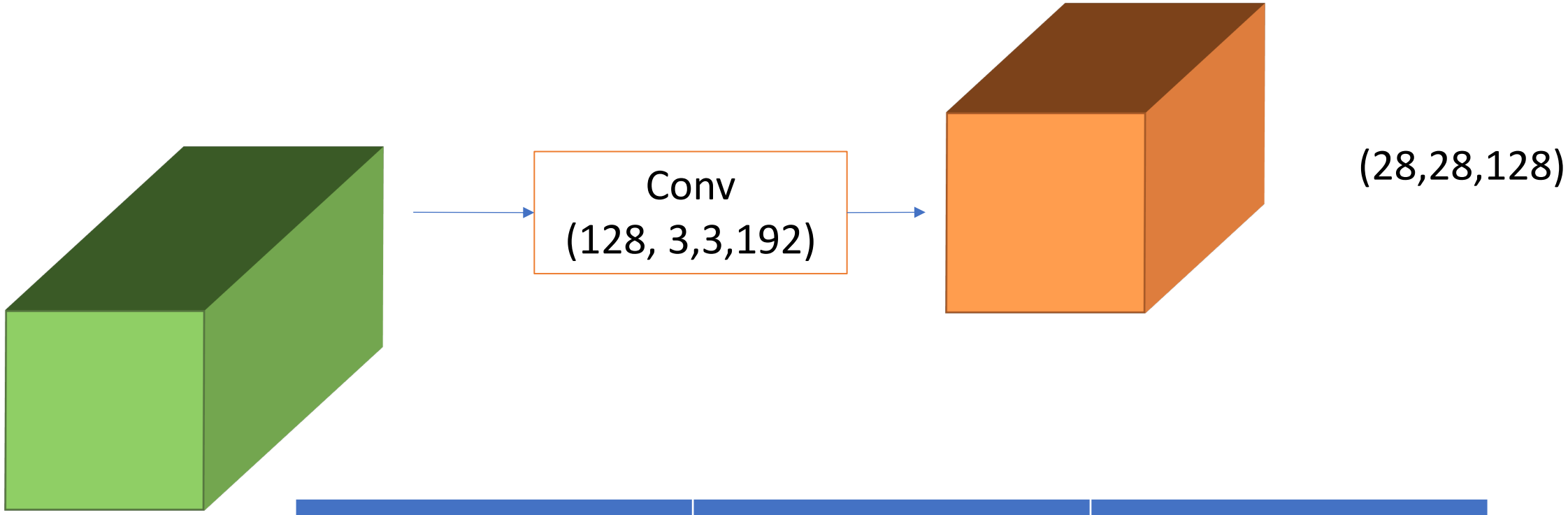



# Spatially Separable Convolutions

- Intuition is to decompose a convolution into two convolutions. For example Sobel 3x3 filter can be decomposed into a 3x1 and 1x3

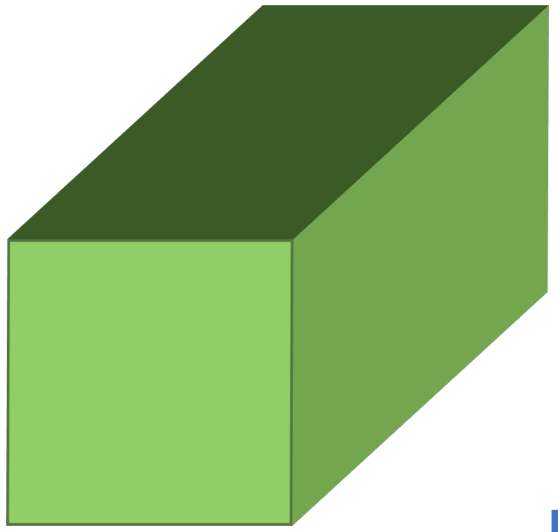
$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}, \begin{bmatrix} -1 & 0 & 1 \end{bmatrix}$$

# Recall: Can be more efficient than using one layer of 3x3s




| | # Params | MFLOPs |
|---|---|---|
| 3x3 | 221k | 173.4 |
| | | |
| | | |

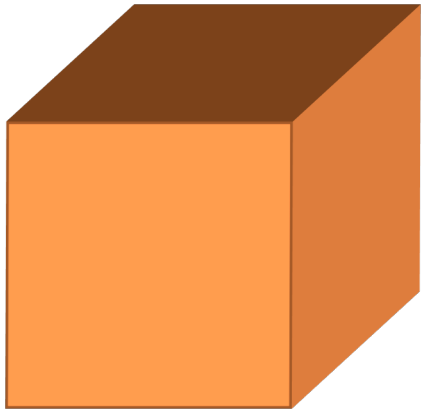
# Recall: Shrink channels with 1x1, then do 3x3



Input Volume
(28,28,192)

Conv
(96, 1,1,192)

# Params:
18,528

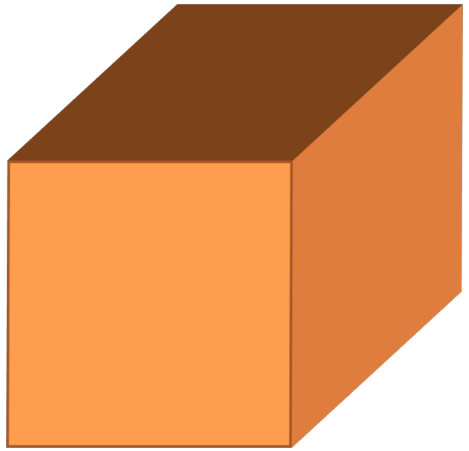


(28,28,96)

14.4MFlops

Conv
(128, 3,3,96)

# Params:
110,720

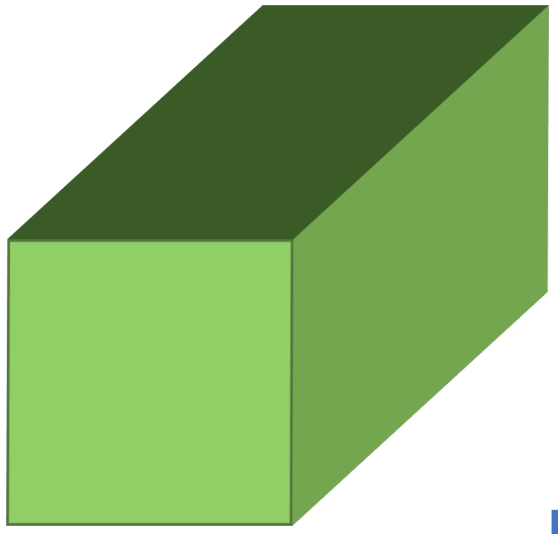


(28,28,128)

86.7MFlops

| | # Params | MFLOPs |
|---|---|---|
| 3x3 | 221k | 173.4 |
| 1x1 → 3x3 | 129k | 101 |
| | | |

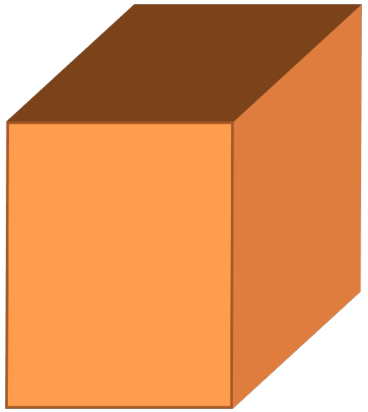
# Another Option: Spatial Factorization into Asymmetric Convolutions



Input Volume
(28,28,192)

Conv
(128, 1,3,192)

# Params:
50,816

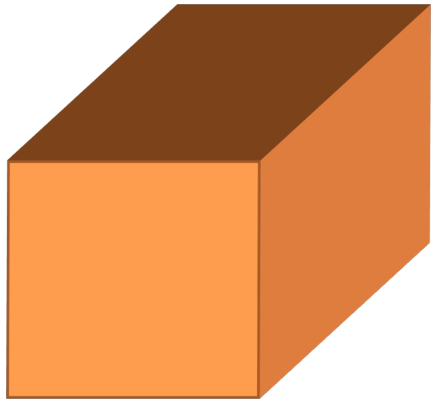


(28,26,128)

53.7MFlops

Conv
(128, 3,1,128)

# Params:
49,280

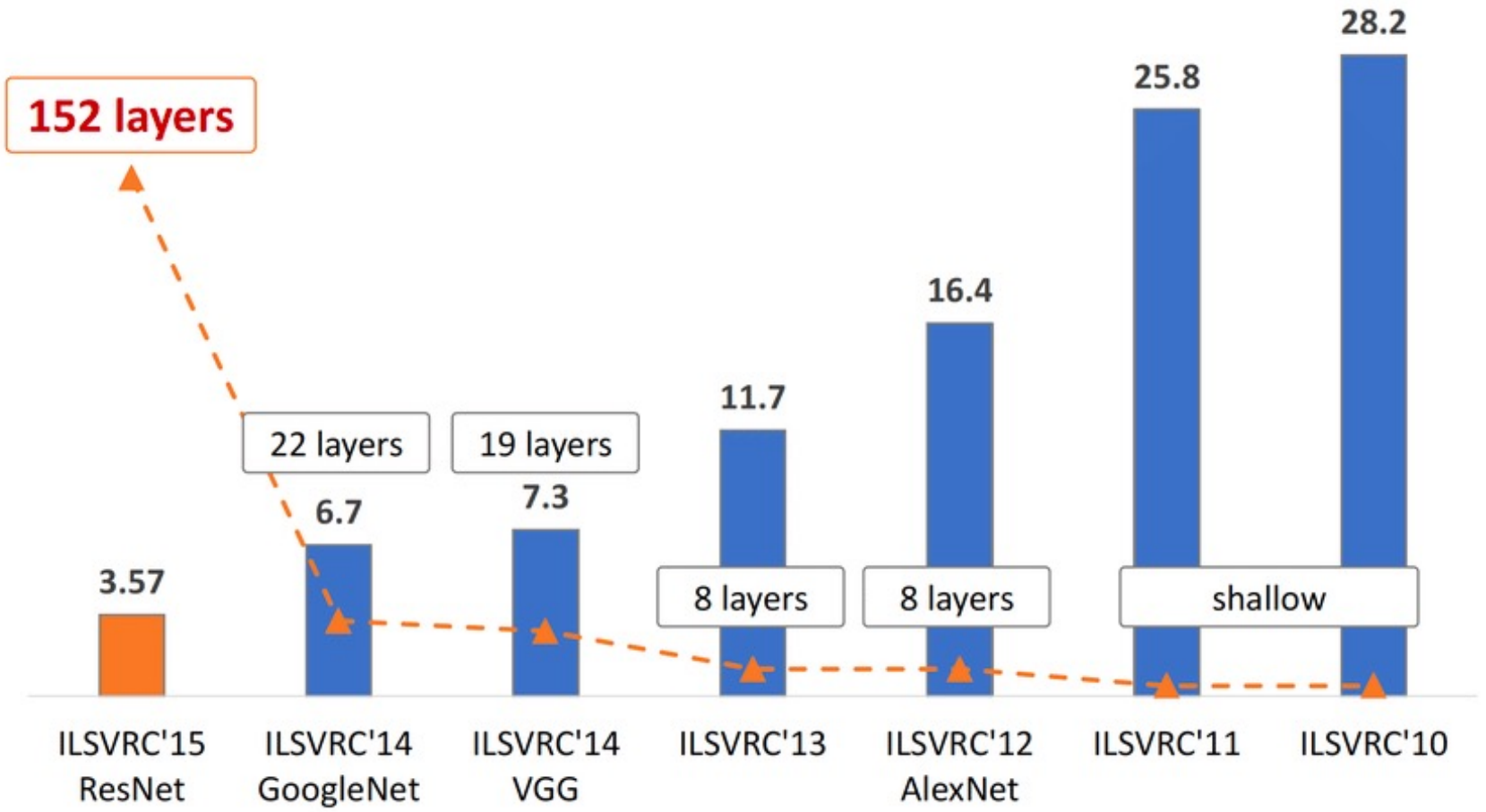


(26,26,128)

33.2MFlops

| | # Params | MFLOPs |
|---|---|---|
| 3x3 | 221k | 173.4 |
| 1x1 → 3x3 | 129k | 101 |
| 1x3 → 3x1 | 100k | 86.1 |

# Deep Learning at ImageNet



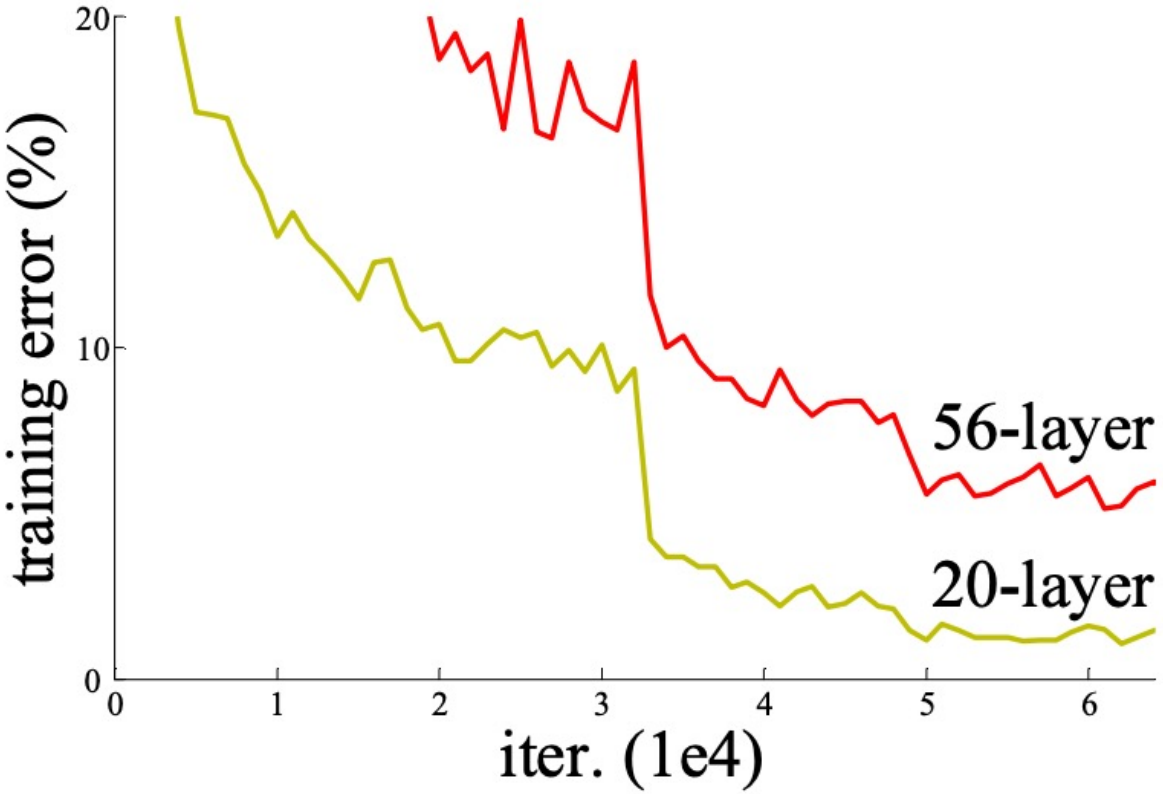


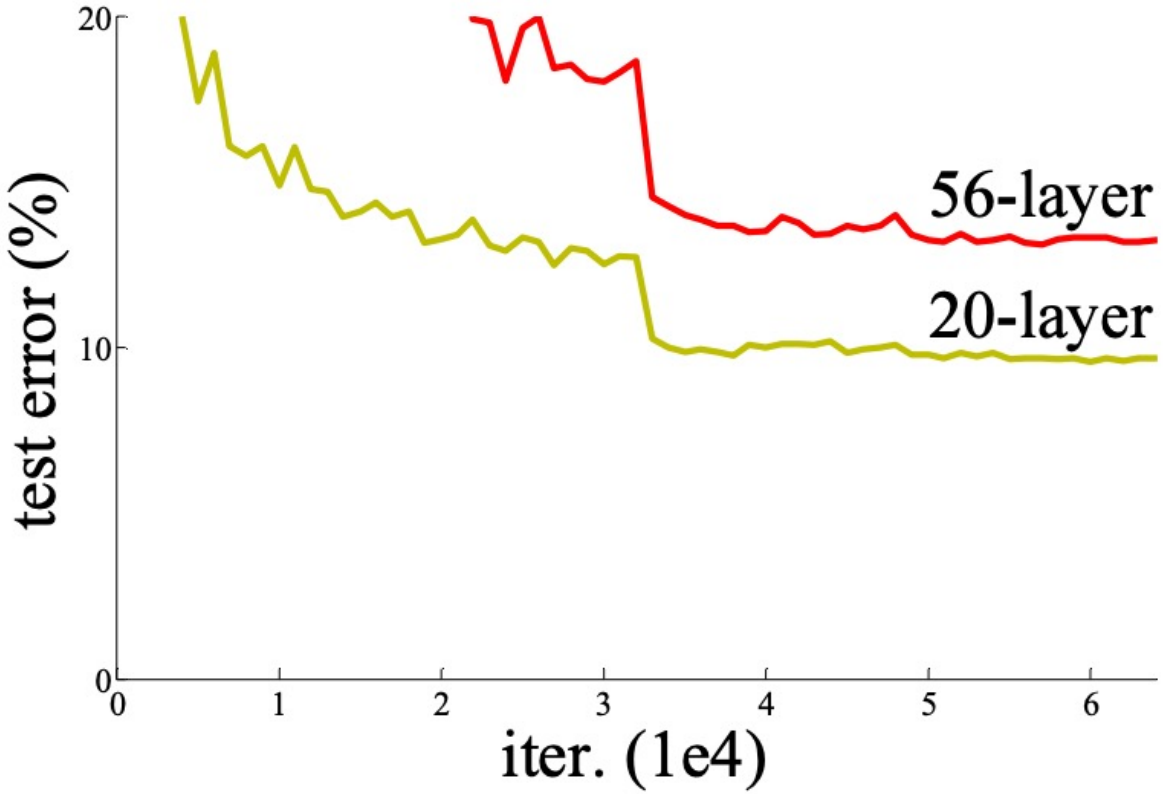

Brad Quinton, Scott Chin

# ResNet

“Deep Residual Learning for Image Recognition”, He, Zhang, Ren, Sun, 2015, https://arxiv.org/pdf/1512.03385.pdf

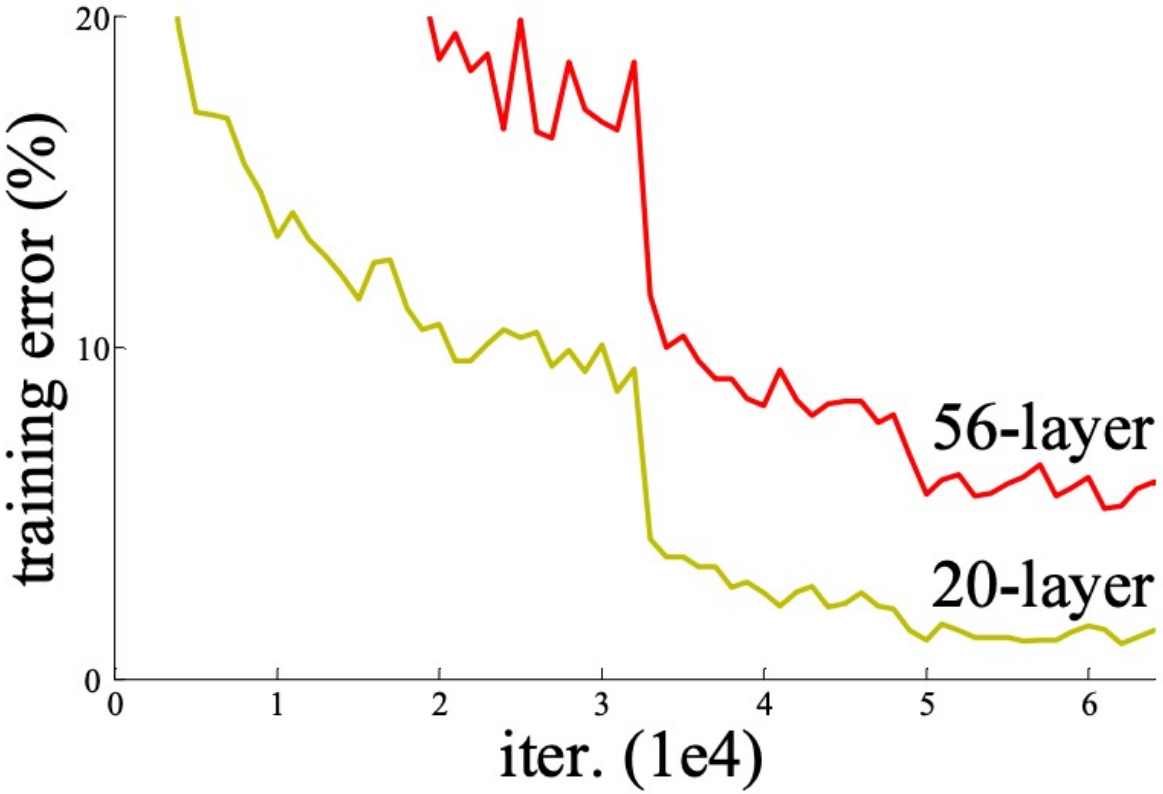
# Can we simply get better results by stacking more layers?

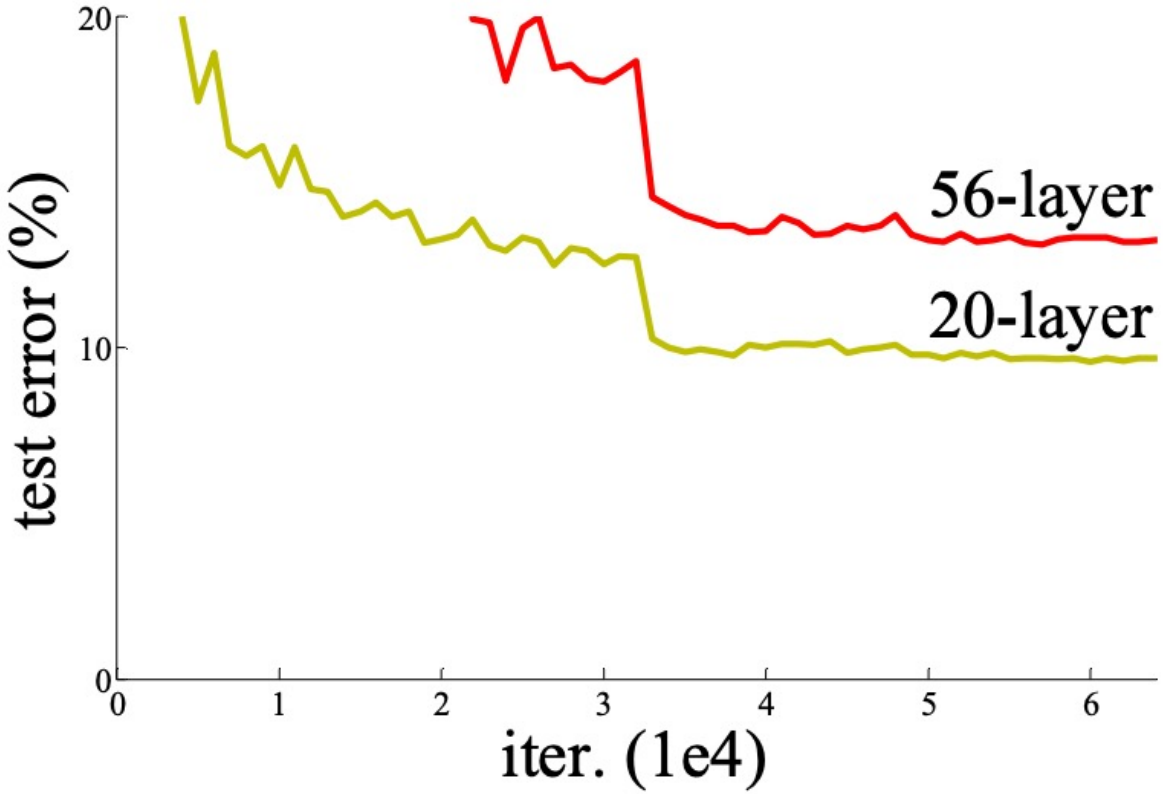


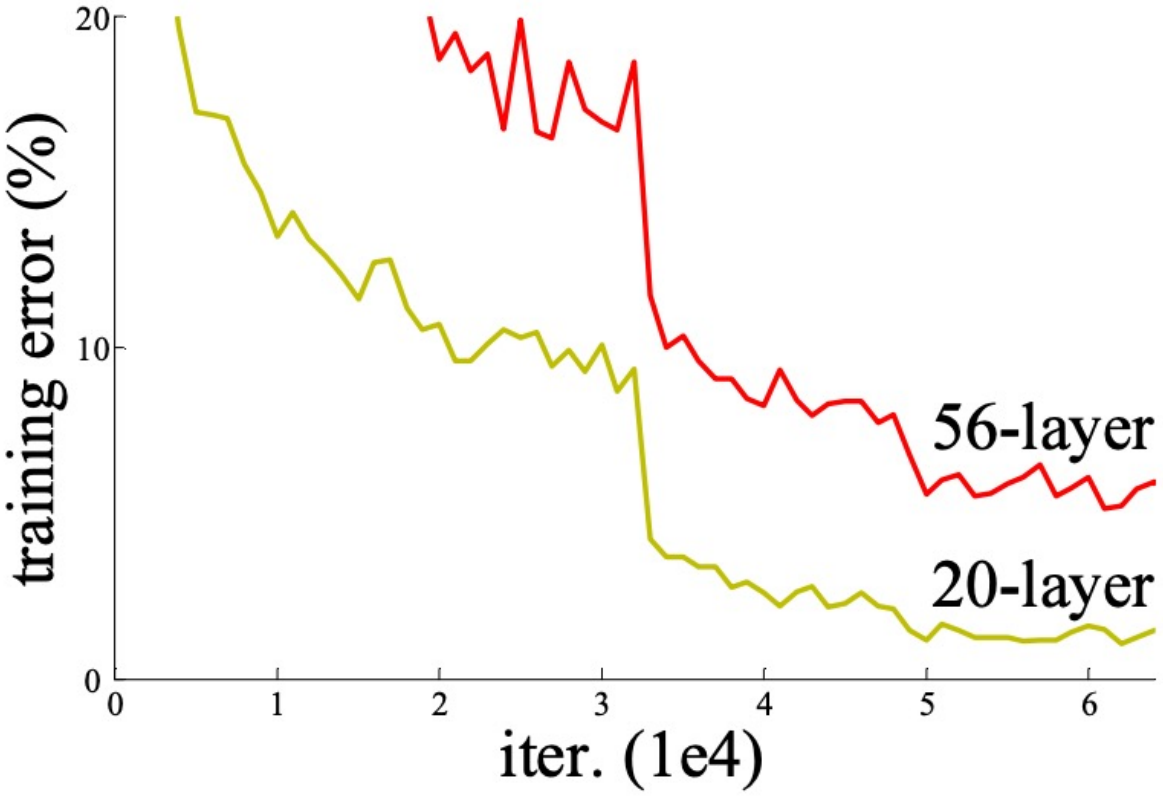


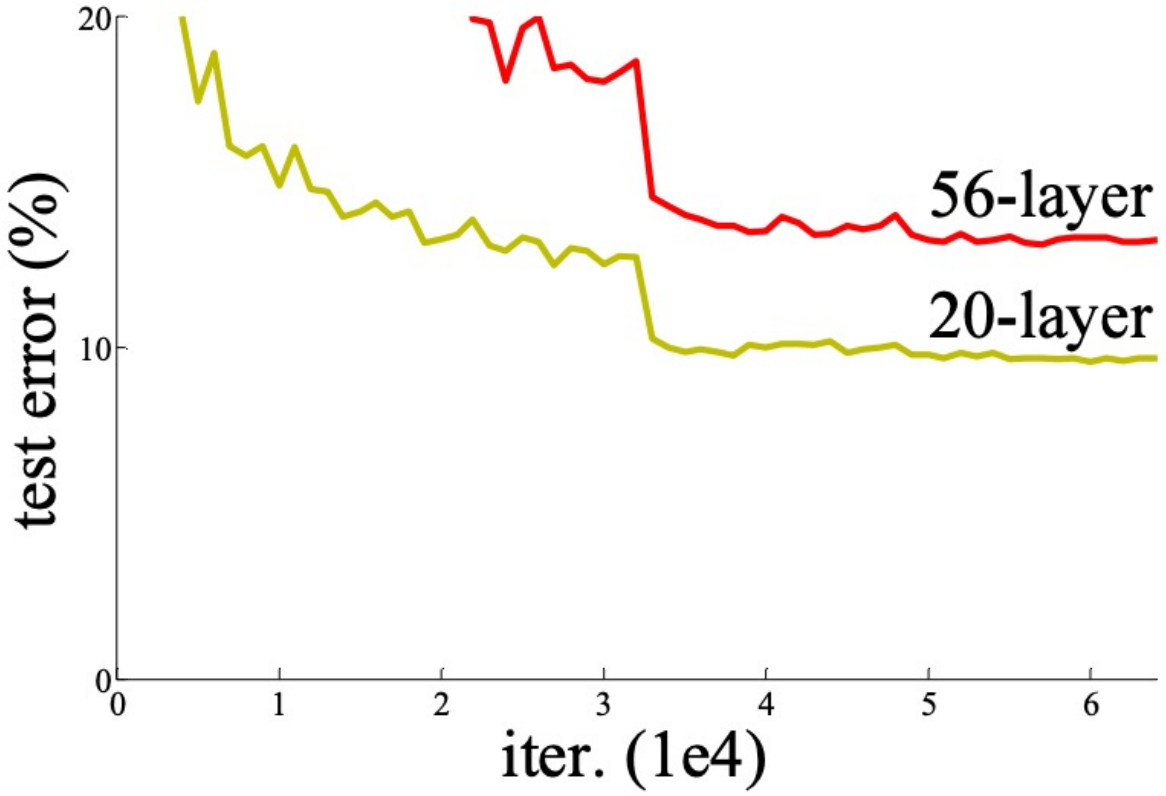

Brad Quinton, Scott Chin




- Test error increased with more layers. Overfitting?




- Test error increased with more layers. Overfitting?
- No! Training error also increased!



Brad Quinton, Scott Chin

# Deep Network should be at least as good as shallow network!

- Intuitively, given a shallow network, add more layers and
- Consider if these layers just learned the identity function (i.e. simply pass the inputs to the outputs of the layer)
- Then functionally, the deeper network is equivalent to the shallow network.
- Should achieve same accuracy!

# So the problem is an optimization problem

Hypothesis:

- Our current techniques makes it hard to find the identity function for a layer, let alone a function that improves the overall model

# So the problem is an optimization problem

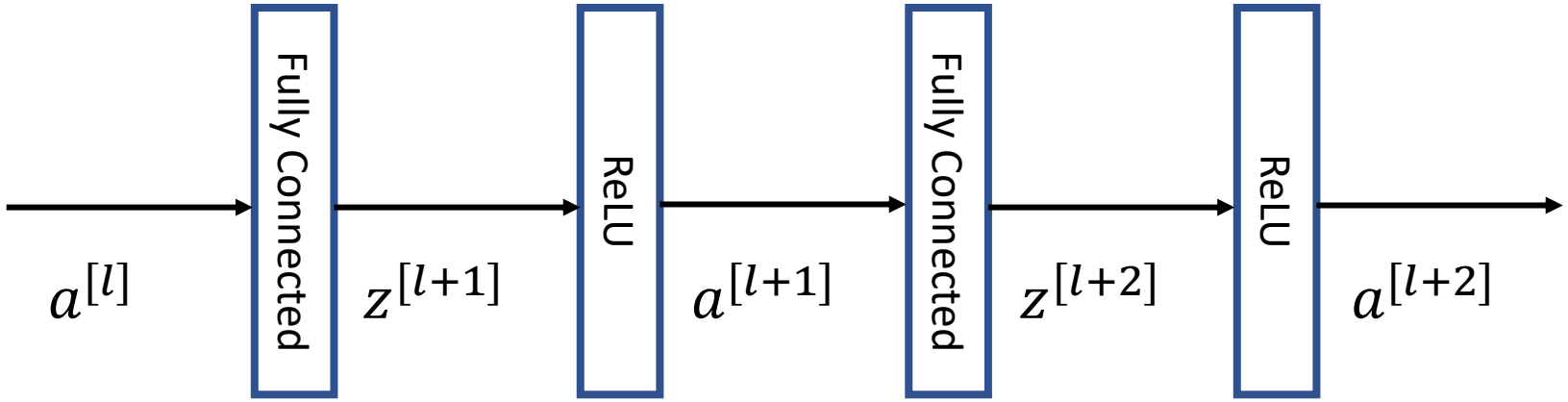
Hypothesis:

- Our current techniques makes it hard to find the identity function for a layer, let alone a function that improves the overall model

Proposed Solution:

- Augment architecture to start with the identify function, and then “learn” from there

# Residual Block (for fully connected layer)



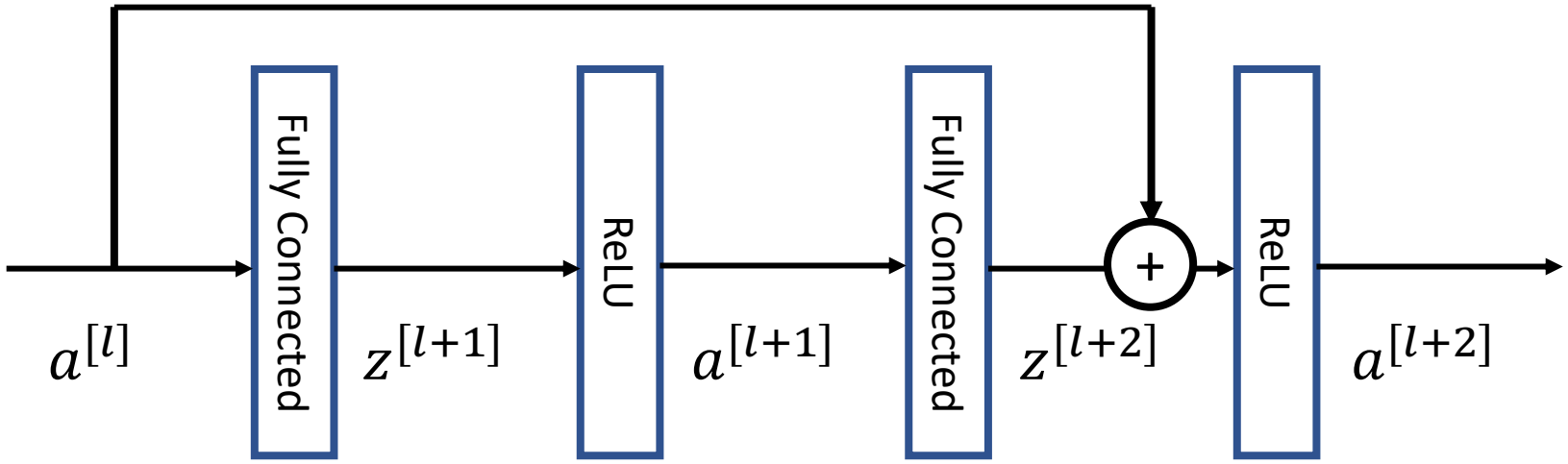

$$z^{[l+1]} = W^{[l+1]}a^{[l]} + b^{[l+1]}$$

$$a^{[l+1]} = ReLU\left(z^{[l+1]}\right)$$

$$z^{[l+2]} = W^{[l+2]}a^{[l+1]} + b^{[l+2]}$$

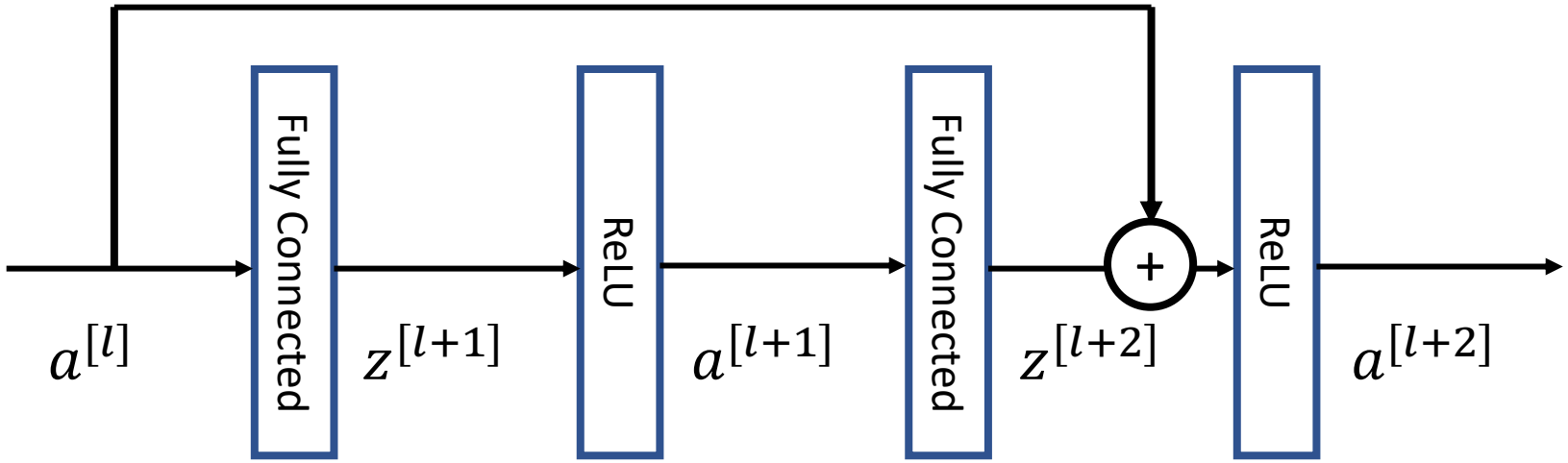
$$a^{[l+2]} = ReLU\left(z^{[l+2]}\right)$$

# Residual Block (for fully connected layer)




Add a “Shortcut” connection

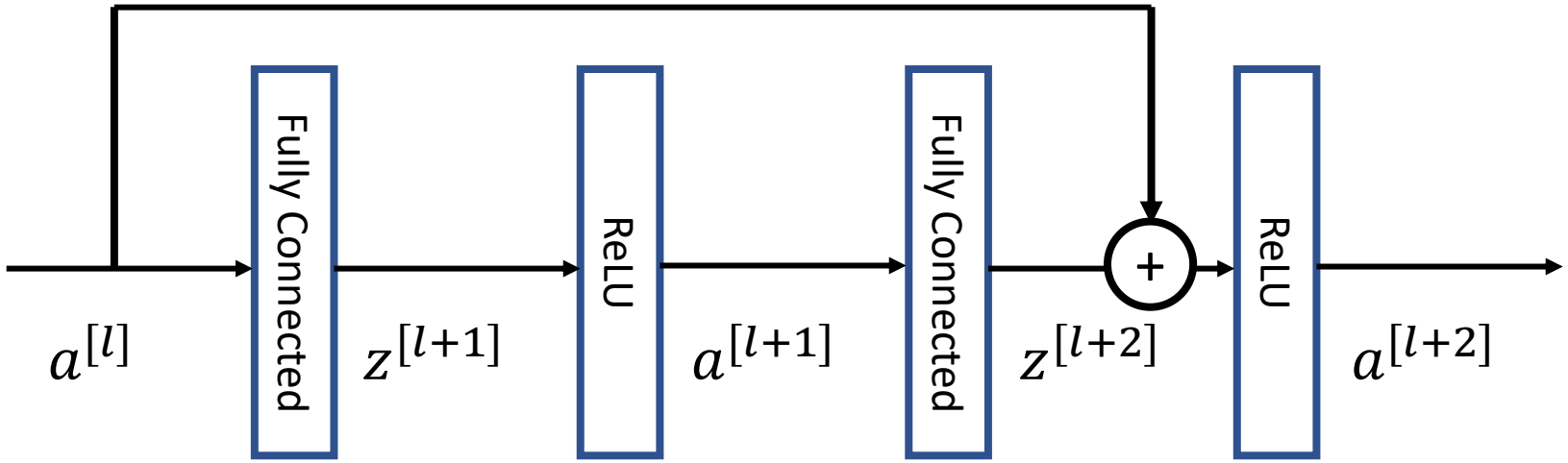
# Residual Block (for fully connected layer)




$$a^{[l+2]} = ReLU\left(z^{[l+2]} + a^{[l]}\right)$$

$$= ReLU\left(W^{[l+2]}a^{[l+1]} + b^{[l+2]} + a^{[l]}\right)$$

# Residual Block (for fully connected layer)



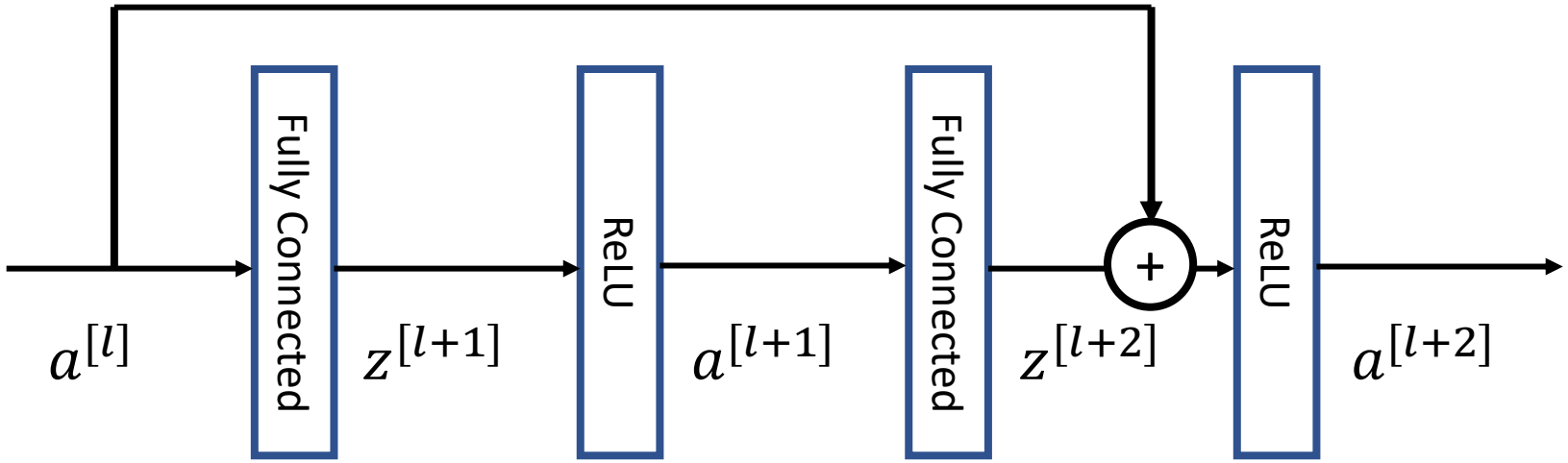

$$a^{[l+2]} = ReLU\left(z^{[l+2]} + a^{[l]}\right)$$

$$= ReLU\left(W^{[l+2]}a^{[l+1]} + b^{[l+2]} + a^{[l]}\right)$$

$$= ReLU\left(a^{[l]}\right)$$

If $W^{[l+2]}$ and $b^{[l+2]}$ approach 0, then

# Residual Block (for fully connected layer)



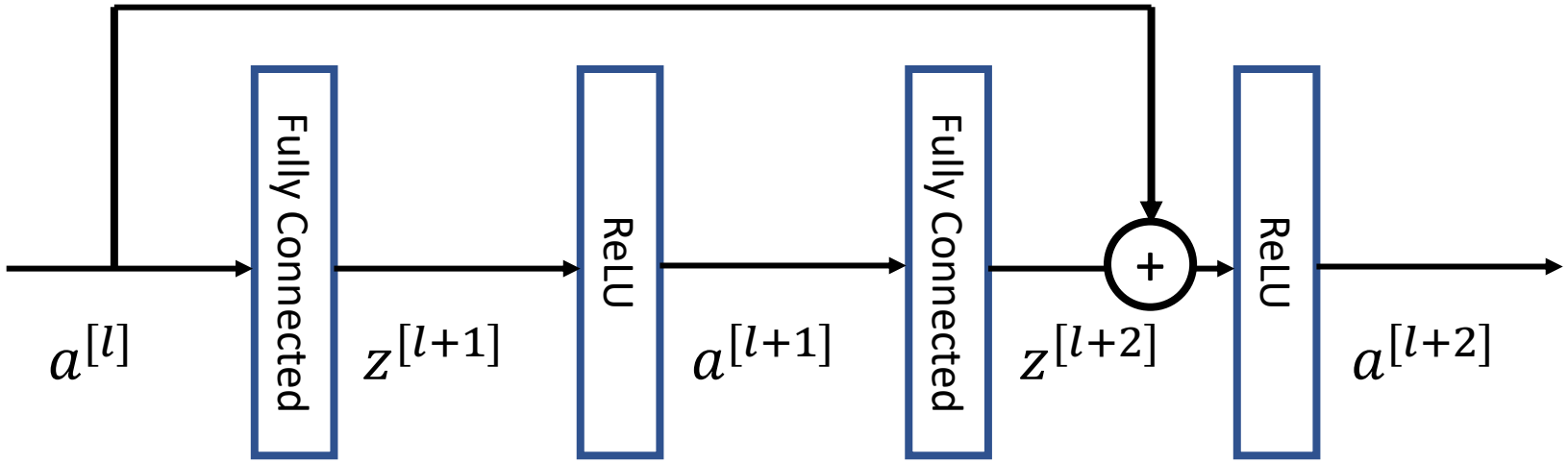

$$a^{[l+2]} = ReLU\left(z^{[l+2]} + a^{[l]}\right)$$

$$= ReLU\left(W^{[l+2]}a^{[l+1]} + b^{[l+2]} + a^{[l]}\right)$$

If $W^{[l+2]}$ and $b^{[l+2]}$ approach 0, then

$$= ReLU\left(a^{[l]}\right) = ReLU\left(ReLU\left(z^{[l]}\right)\right) = ReLU\left(z^{[l]}\right) = a^{[l]}$$

# Residual Block (for fully connected layer)

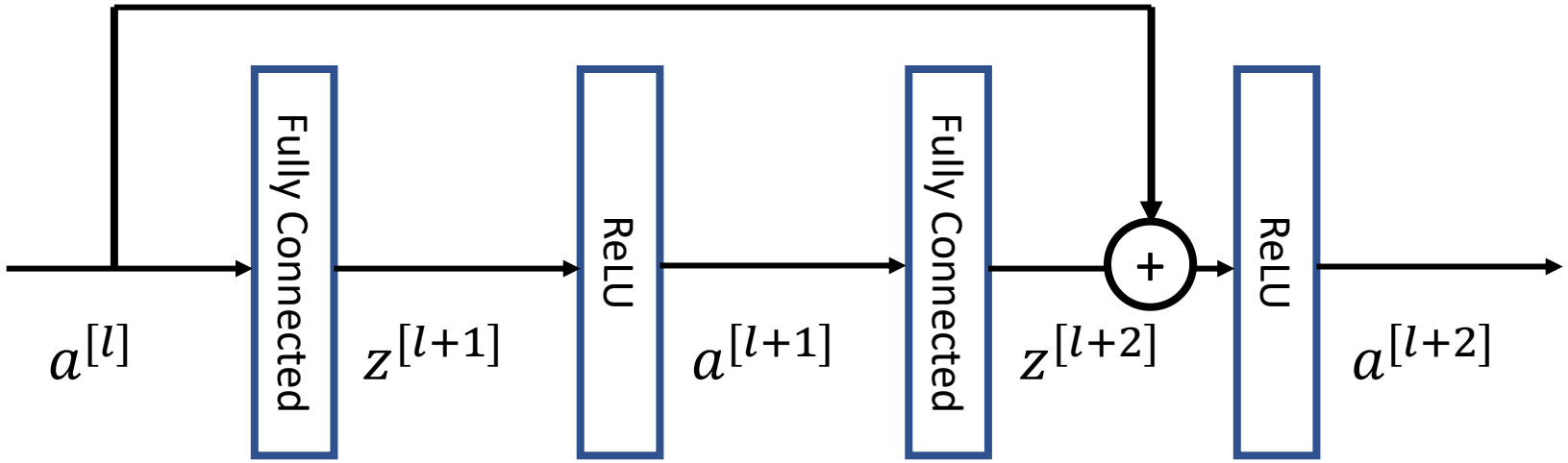



- Stacking these blocks to make a network deeper shouldn't hurt

Intuition

- The "residual" identity function gives a good baseline on which to try to improve

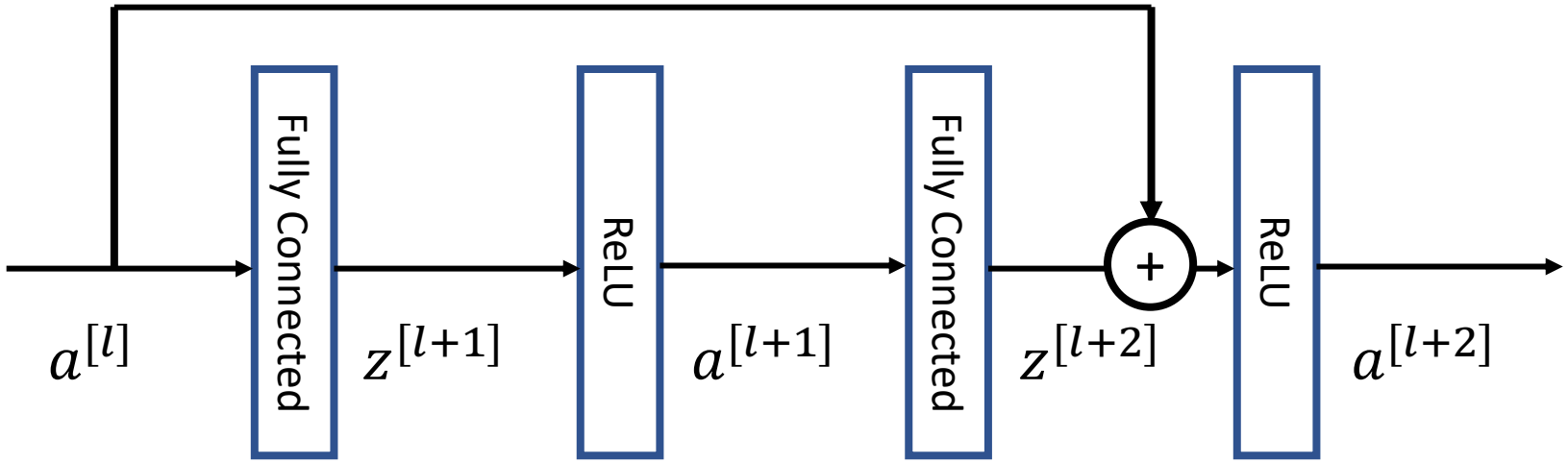
# Residual Block (for fully connected layer)




Also

- Doesn't add any learned parameters
- Doesn't increase computational complexity significantly
- Shortcut paths provide another path for backprop gradient flow

# Residual Block (for fully connected layer)

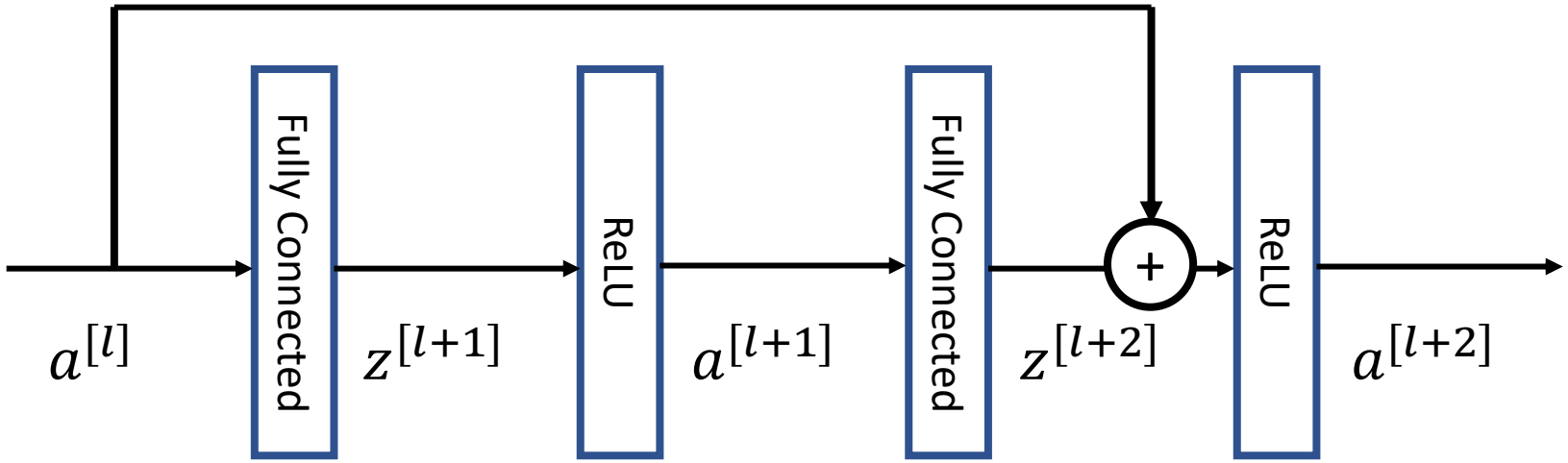



One more detail:

- Shape of $z^{[l+2]}$ and $a^{[l]}$ must match

$$a^{[l+2]} = ReLU\left(z^{[l+2]} + a^{[l]}\right)$$

# Residual Block (for fully connected layer)

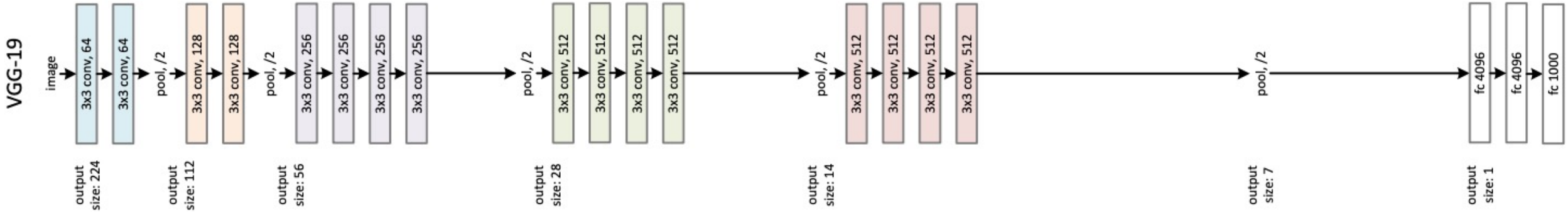



One more detail:

- Shape of $z^{[l+2]}$ and $a^{[l]}$ must match
- If not, either use a projection matrix $W_s$
- Or pad with zeroes

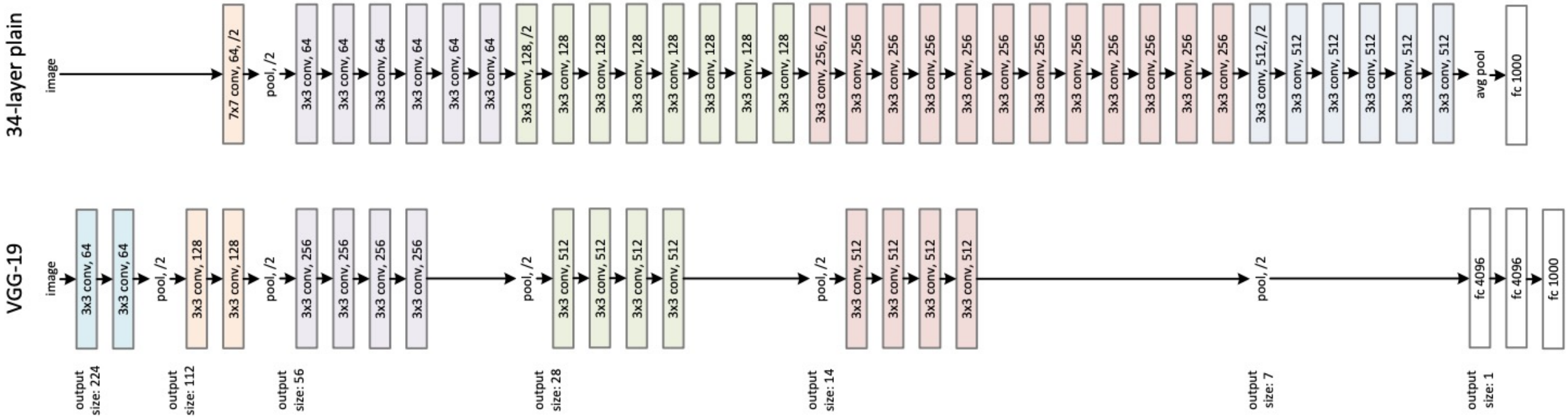
$$a^{[l+2]} = ReLU\left(z^{[l+2]} + W_s a^{[l]}\right)$$

# ResNet Architecture



# ResNet Architecture

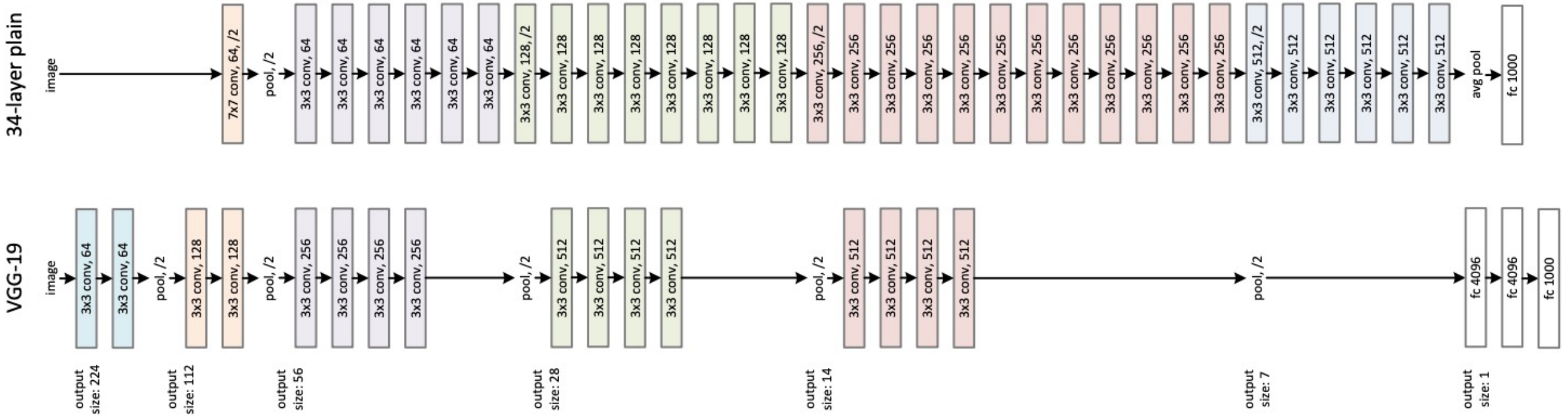
- 34 Parameter Layers
- No pooling layers.  Use stride=2 in conv layer to shrink volumes
- Use Global Average Pooling instead of FC layers at the end



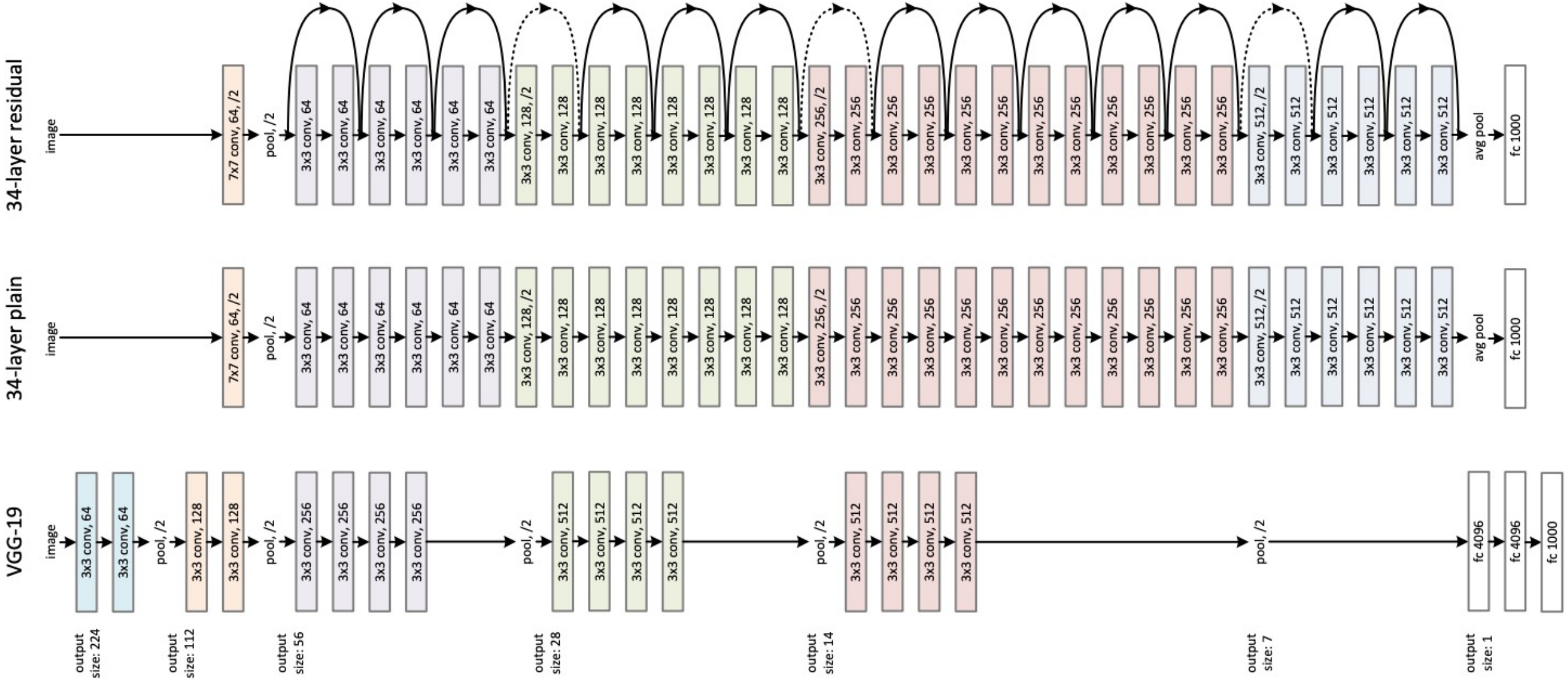
# ResNet Architecture

Plain Network Compared to VGG19

- 3.6 GFLOPs vs 19.6 GFLOPs
- 30-35M params vs 144M params



# ResNet Architecture



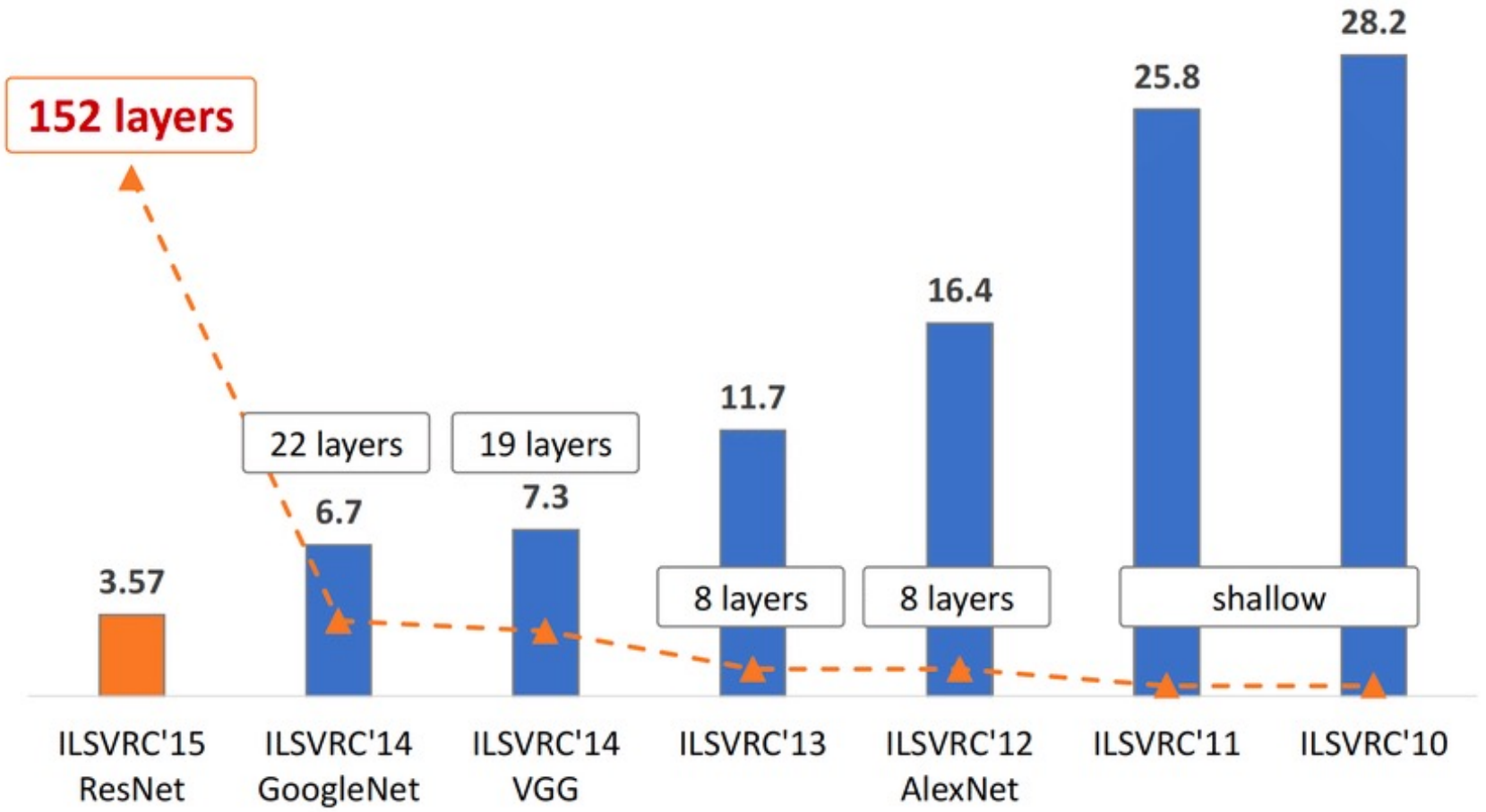



Brad Quinton, Scott Chin

# Comparison so far

| Architecture | # Parameters (millions) | # GFLOPs | ImageNet top-5 error |
|---|---|---|---|
| AlexNet | 62 | 1 | 16.4 |
| ZFNet | 108 | 2.47 | 11.7 |
| VGG16 | 138 | 13.6 | 7.3 |
| GoogLeNet | 6.8 | 1.5 | 6.7 |
| ResNet152 | ~60 | 11.3 | 3.57 |

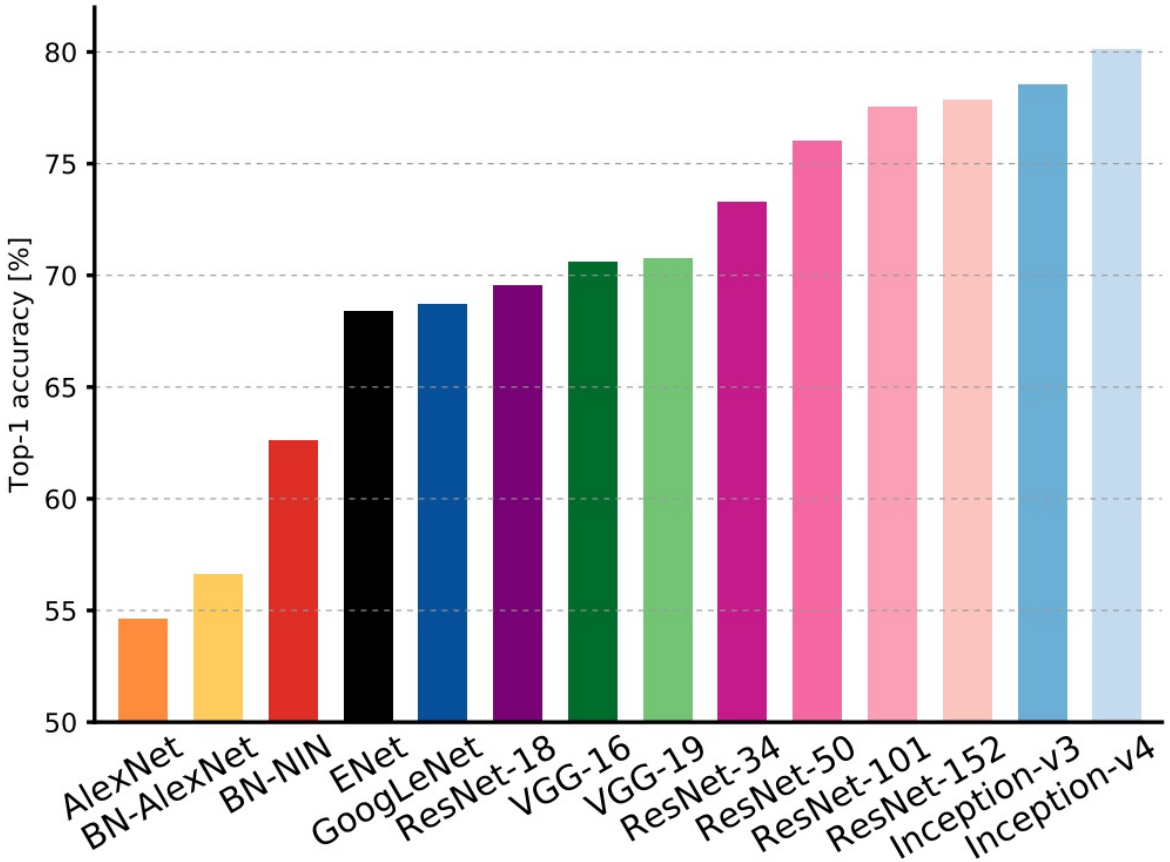
# Deep Learning at ImageNet

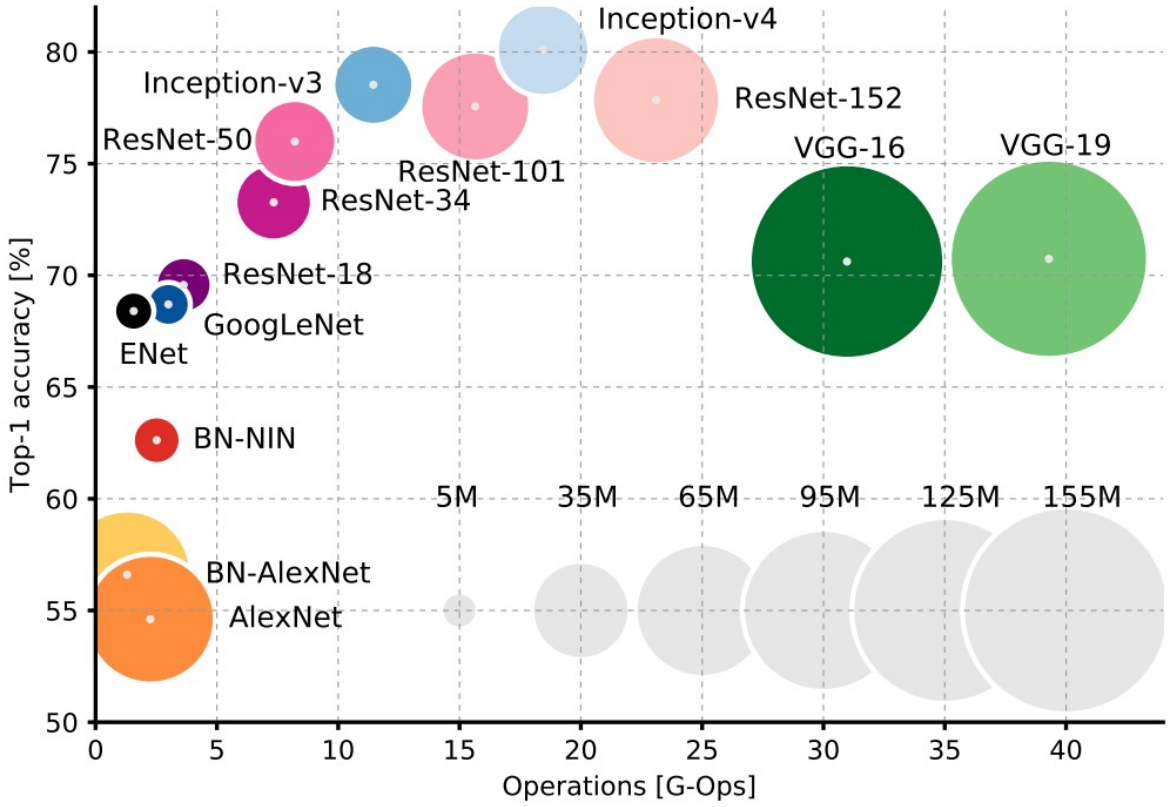





Brad Quinton, Scott Chin

# Summary Comparison






"An Analysis of Deep Neural Network Models For Practical Applications", Canziani, Culurciello, Paszke, 2017, https://arxiv.org/pdf/1605.07678.pdf

# Memory Usage

Sources:

- Activations – i.e. the intermediate volumes and their gradients
- Parameters - Parameter values and their gradients
- Training Data – the batch currently being processed

For training, you need to fit everything into the GPU memory, or else you take massive runtime hit.

Can tune optimizer batch size (more on this in next lecture)

# MobileNet

- Another way of using 1x1 convolutions to create a factorized convolution which in turn further improves compute efficiency
- Hyperparameter to trade off accuracy and FLOPs/Params

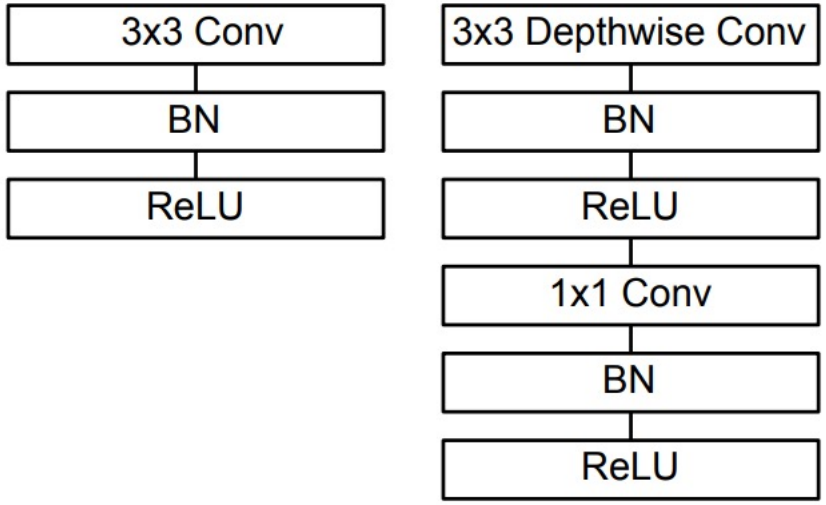
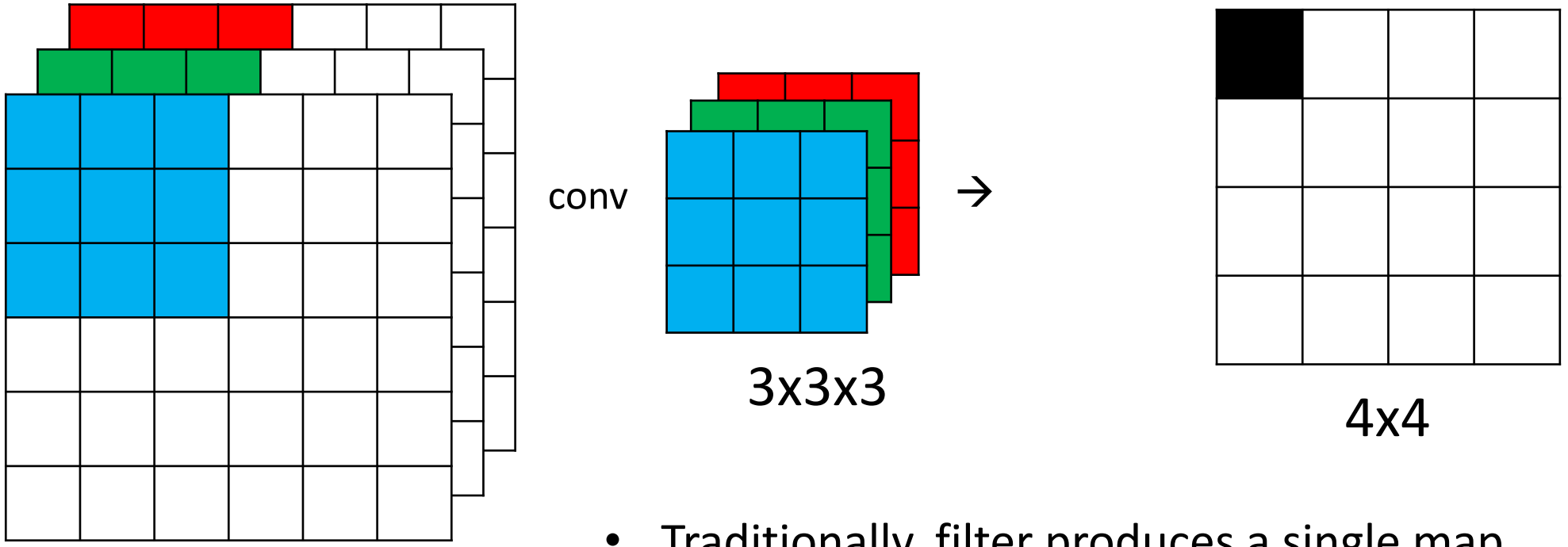



Table 6. MobileNet Width Multiplier

| Width Multiplier | ImageNet Accuracy | Million Mult-Adds | Million Parameters |
|---|---|---|---|
| 1.0 MobileNet-224 | 70.6% | 569 | 4.2 |
| 0.75 MobileNet-224 | 68.4% | 325 | 2.6 |
| 0.5 MobileNet-224 | 63.7% | 149 | 1.3 |
| 0.25 MobileNet-224 | 50.6% | 41 | 0.5 |

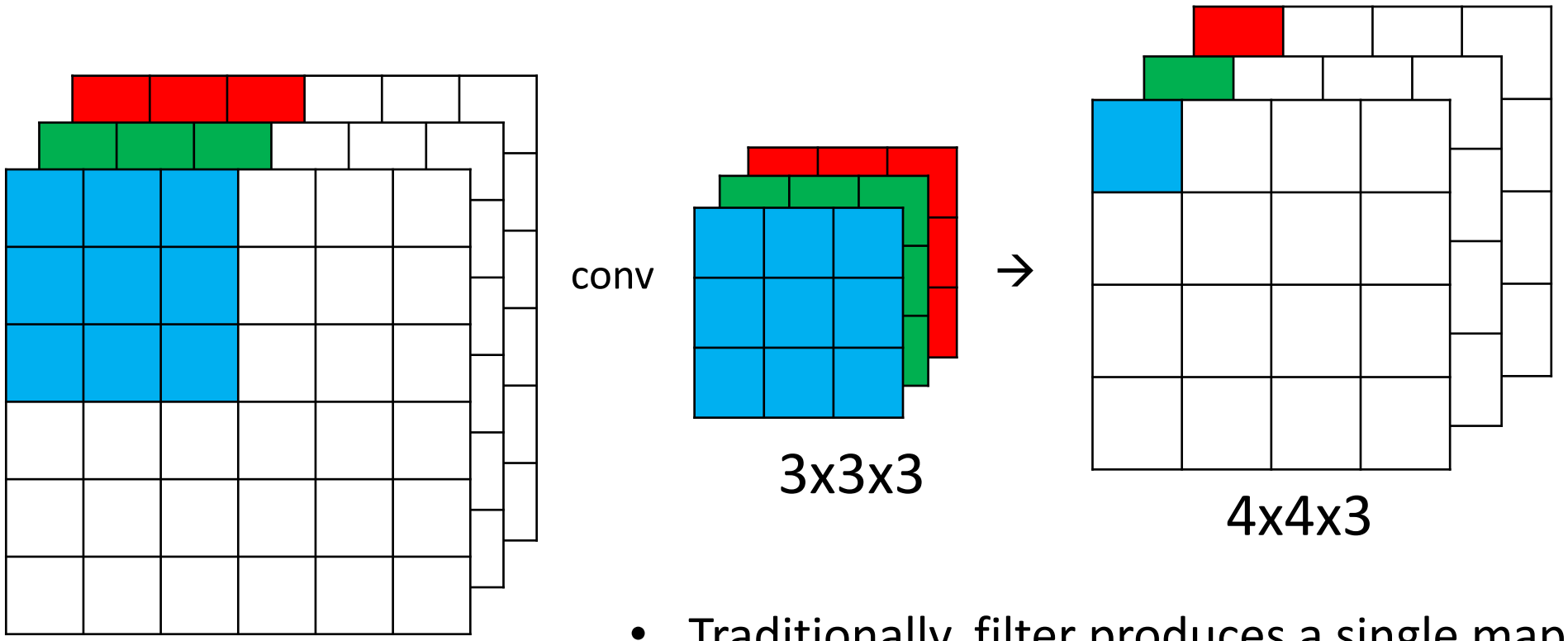
"MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications", Howard et al., 2017, https://arxiv.org/pdf/1704.04861.pdf

# Traditional Convolution



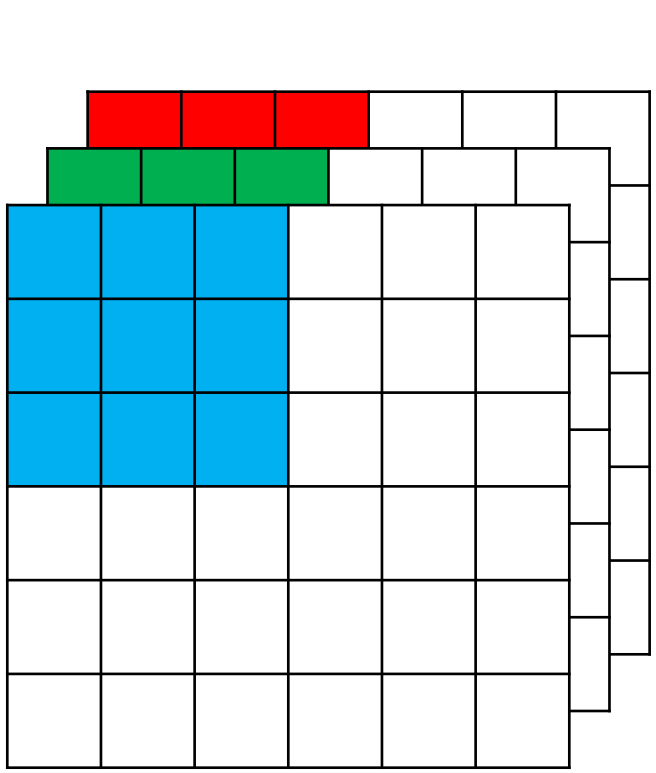

- Traditionally, filter produces a single map

# Traditional Convolution



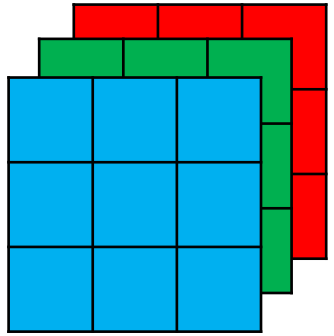

- Traditionally, filter produces a single map
- Can also think of it as two steps
  1. Channel independent convolution

# Traditional Convolution



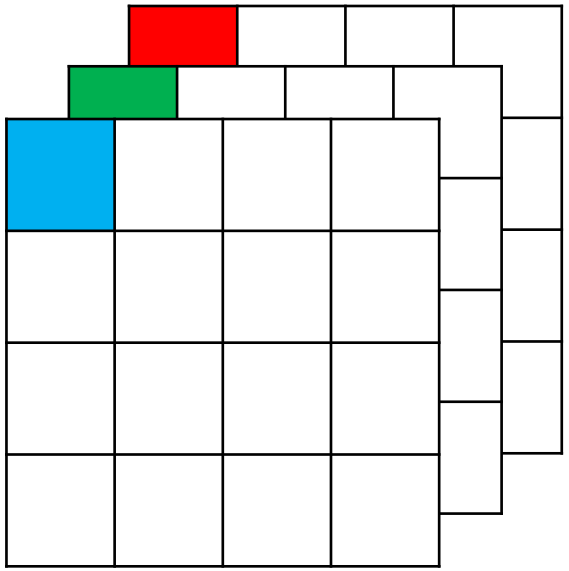
6x6x3

conv



3x3x3

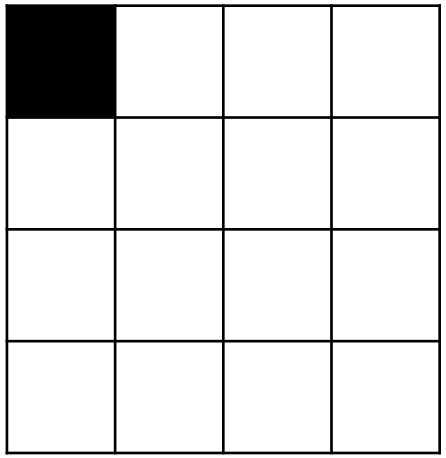
→



4x4x3

sum

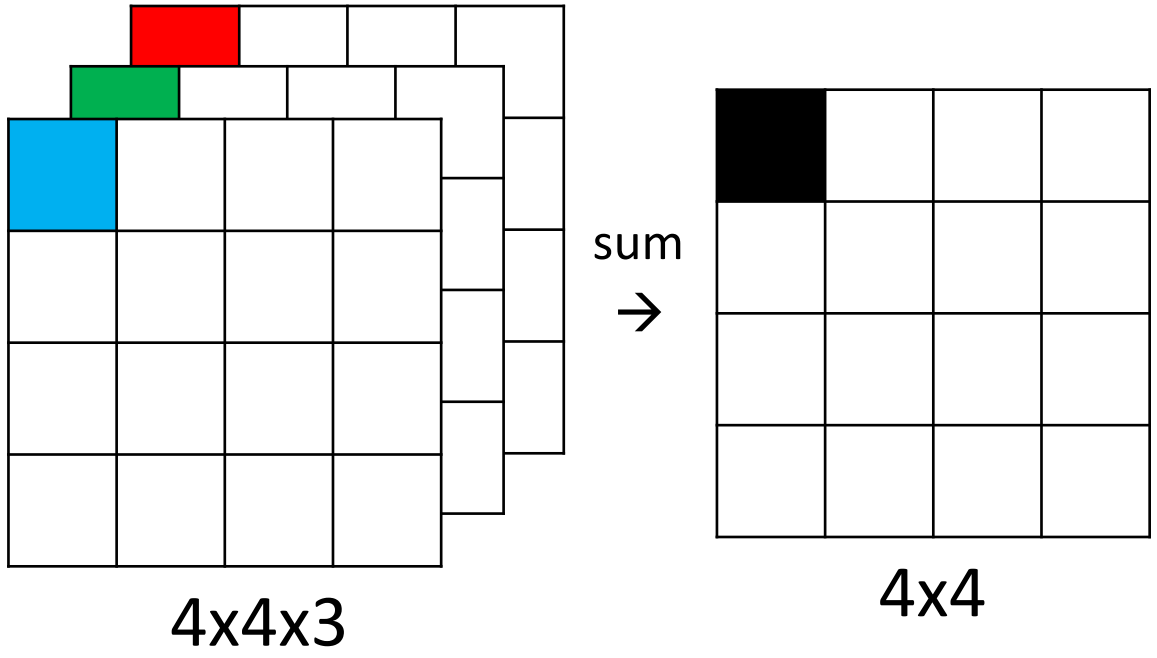
→



4x4

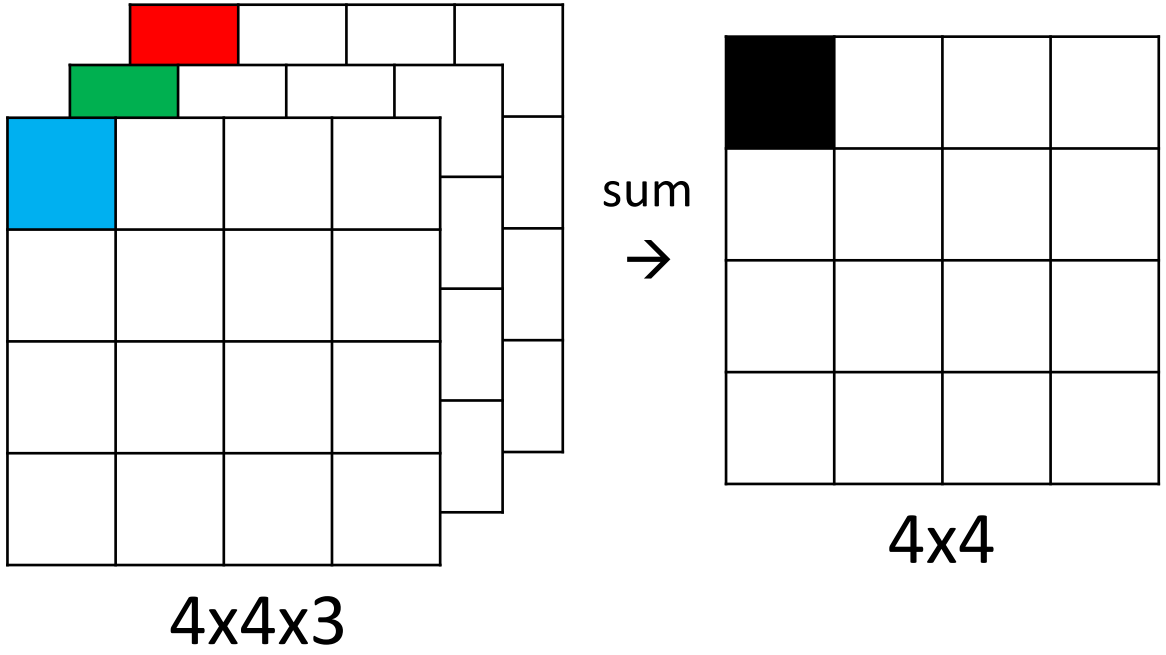
- Traditionally, filter produces a single map
- Can also think of it as two steps
  1. Channel independent convolution
  2. Summing across channels

# Traditional Convolution



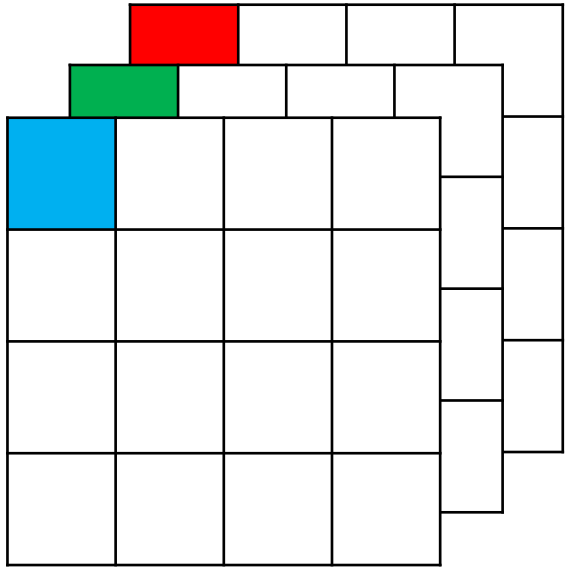

- Traditionally, filter produces a single map
- Can also think of it as two steps
  1. Channel independent convolution
  2. Summing across channels

# Traditional Convolution



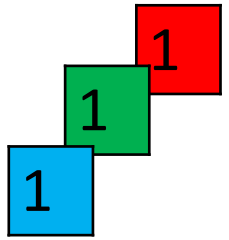

- Traditionally, filter produces a single map
- Can also think of it as two steps
  1. Channel independent convolution
  2. Summing across channels

# Traditional Convolution



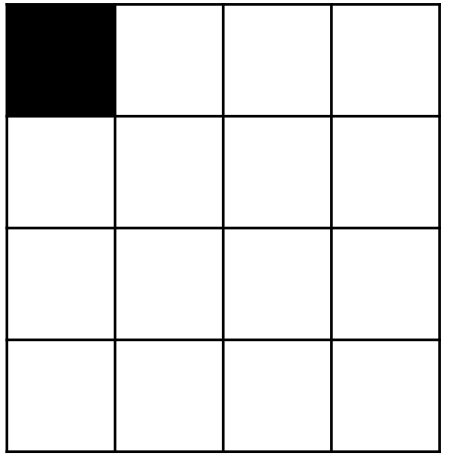
4x4x3

conv




1x1x3

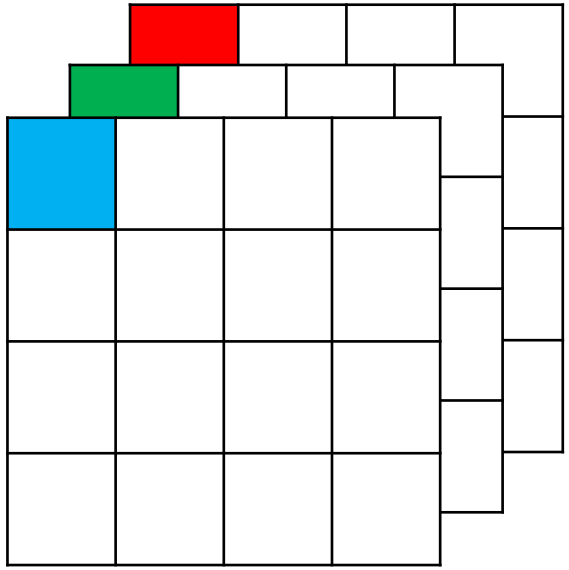
→



4x4

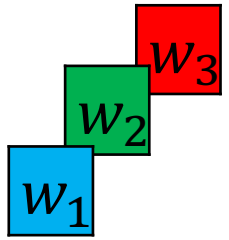
- Traditionally, filter produces a single map
- Can also think of it as two steps
  1. Channel independent convolution
  2. 1x1 convolution with fixed filter values (all 1's)

# Instead, use learned weights for the 1x1 conv



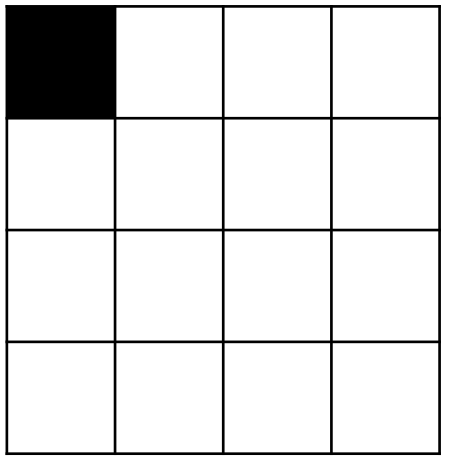
4x4x3

conv

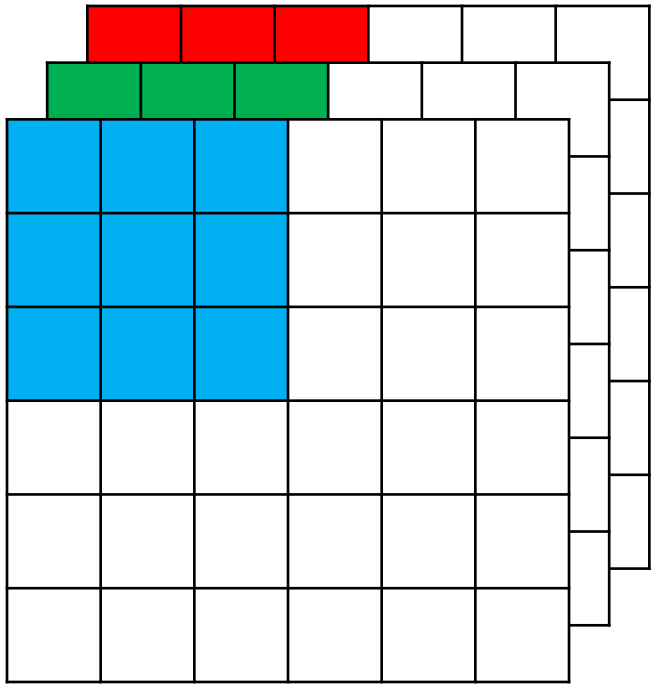



1x1x3

→

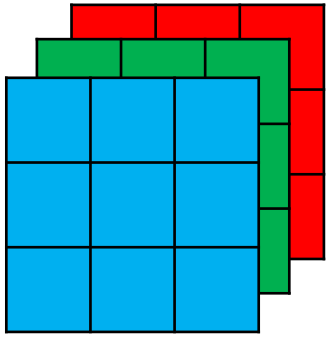


4x4



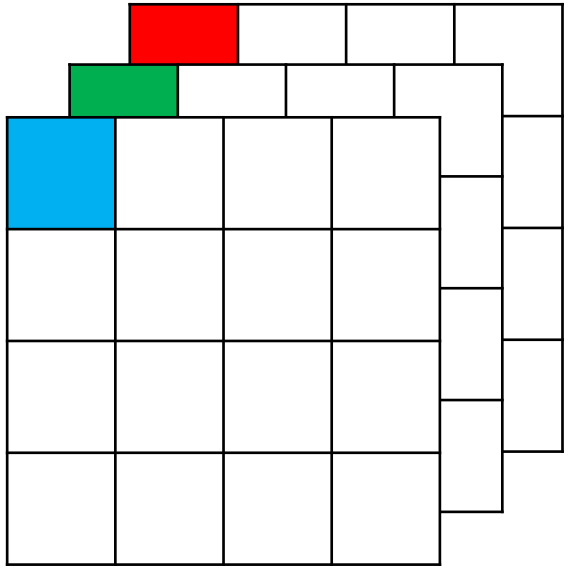
6x6x3

conv



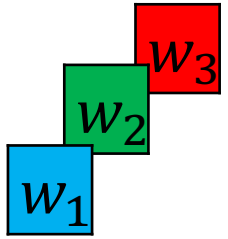
3x3x3

Depthwise Convolution



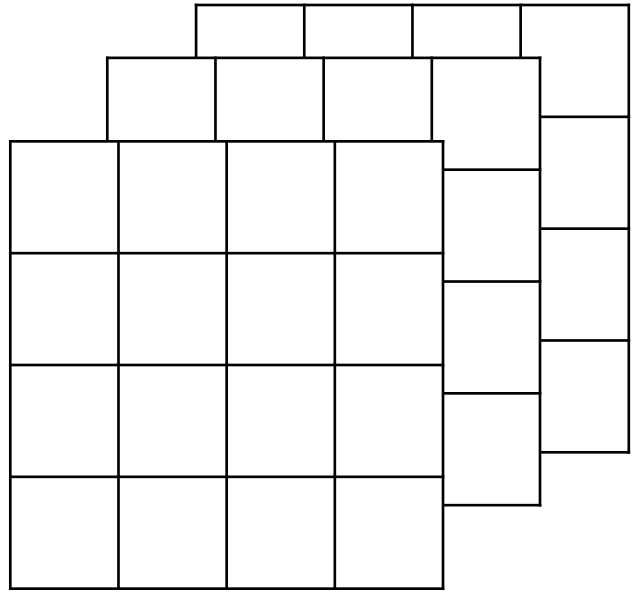
4x4x3

↓ conv




$K$ 1x1x3

Pointwise Convolution



4x4xK

**Depthwise Separable Convolutions** has two stages

1. Depthwise Convolution
   - One $(f, f, c_{in})$ filter
   - Each channel convolved independantly
2. Pointwise Convolution
   - $K$ number of $(1,1, c_{in})$ filters

# EfficientNet

"EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks", Tan and Le, 2019, https://arxiv.org/pdf/1905.11946.pdf

https://ai.googleblog.com/2019/05/efficientnet-improving-accuracy-and.html

# What Architecture Should I Use?

- Start with one of the mainstream architectures
- Have a good reason to invent a new architecture

Keras:

- Xception
- VGG16, VGG19
- ResNet, ResNetV2
- InceptionV3
- InceptionResNetV2
- MobileNet, MobileNetV2
- DenseNet
- NASNet
- EfficientNet

# Keras – Use ResNet Imagenet Weights

```python
import tensorflow
from tensorflow.keras.applications.resnet50 import ResNet50
from tensorflow.keras.preprocessing import image
from tensorflow.keras.applications.resnet50 import preprocess_input, deco
import numpy as np

# Load the model with weights
model = ResNet50(weights='imagenet')

# Prepare the image
img_path = 'frog.jpg'
img = image.load_img(img_path, target_size=(224, 224))
x = image.img_to_array(img)
x = np.expand_dims(x, axis=0)
x = preprocess_input(x)

# Make a prediction
preds = model.predict(x)

# decode the results into a list of tuples (class, description, probability)
# (one such list for each sample in the batch, but we are only supplying one s
print('Prediction:')
for _, label, prob in decode_predictions(preds, top=10)[0]:
    print('{0:0.10f}: {1}'.format(prob, label))
```

```
Prediction:
0.9893396497: tree_frog
0.0102263978: tailed_frog
0.0001932405: bullfrog
0.0000632278: African_chameleon
0.0000347402: leaf_beetle
0.0000215269: mantis
0.0000150923: grasshopper
0.0000142750: green_lizard
0.0000133581: leafhopper
0.0000115666: weevil
```

# Learning Objectives

- Look at successful CNN architectures to see how people tend to combine conv, pool, fully-connected layers to build CNNs and other innovationss
- Learn some back-of-the-envelope metrics for comparing two architectures (# of FLOPs, # of parameters)
- Learn about Convolutional Layers that use 1x1 filters
- Get a little bit of a history lesson in Computer Vision Deep Learning