intel Nervana™ AI Academy

CONVOLUTIONAL NEURAL NET ARCHITECTURES

ALEXNET

Created in 2012 for the ImageNet Large Scale Visual Recognition Challenge (ILSVRC)

Task: predict the correct label from among 1000 classes

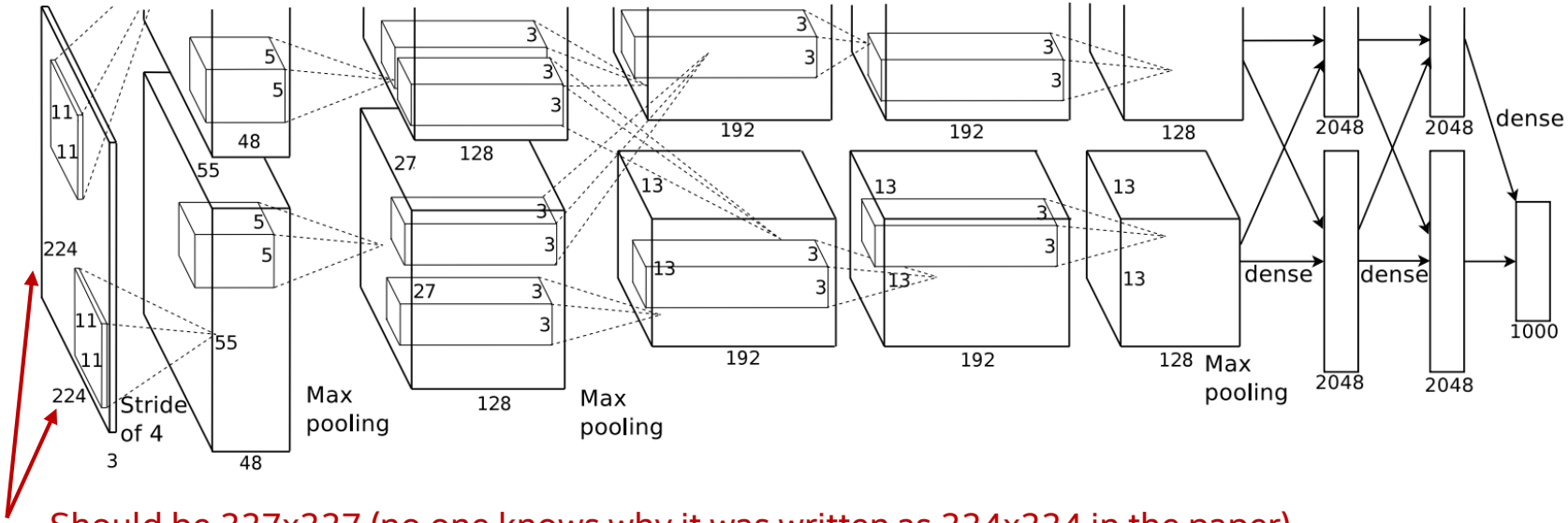
Dataset: around 1.2 million images

Considered the “flash point” for modern deep learning

Demolished the competition.

- Top 5 error rate of 15.4%
- Next best: 26.2%

MODEL DIAGRAM



Should be 227x227 (no one knows why it was written as 224x224 in the paper)

NOTES

They perform data augmentation for training

- Cropping, horizontal flipping, and more
- Useful to help make more use out of given training data

They split up the model across two GPUs, as illustrated in previous image

- This generally doesn't happen in modern CNN architectures
- We can replicate this effect by splitting Tensors in two

ALEXNET: MAIN TAKEAWAYS

CNNs are *very* powerful for image processing

Didn't change too much about LeNet-5

- Added extra depth, computation

Basic template:

- Convolutions with ReLUs
- Sometimes add maxpool after convolutional layer
- Fully connected layers at the end before a softmax classifier

GPUs are really good for this sort of computation!

ALEXNET—DETAILS

- They performed data augmentation for training
  - Cropping, horizontal flipping, and other manipulations
- Basic Template:
  - Convolutions with ReLUs
  - Sometimes add maxpool after convolutional layer
  - Fully connected layers at the end before a softmax classifier

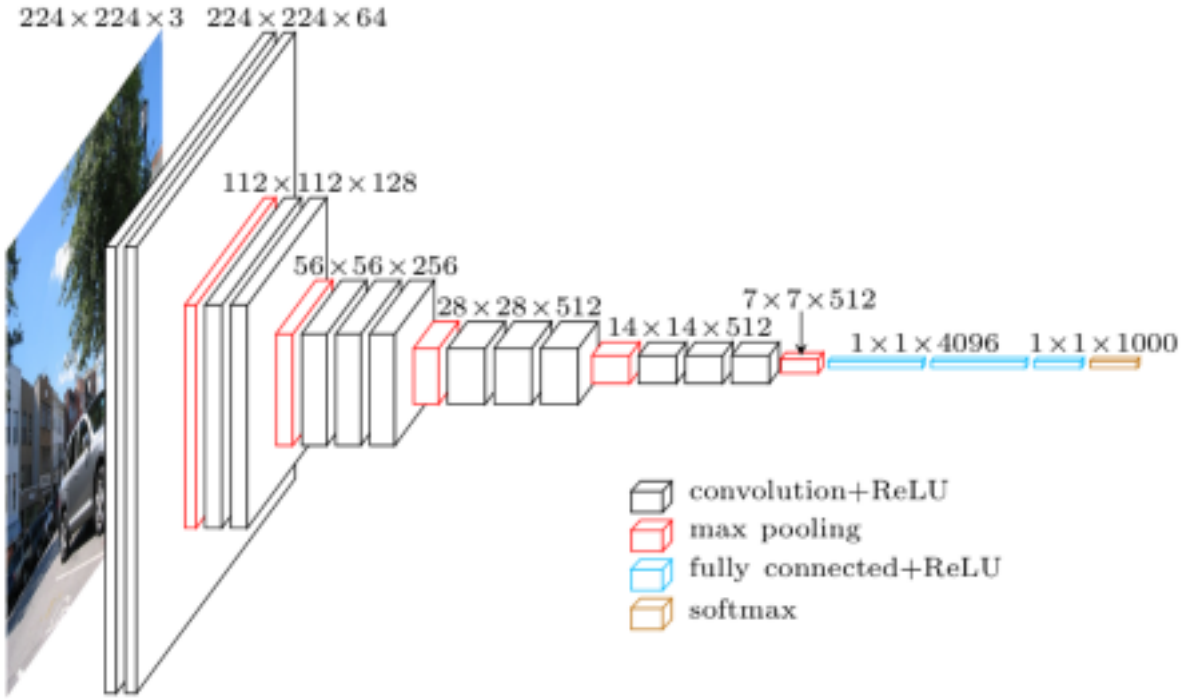
VGG

- Simplify Network Structure
- Avoid Manual Choices of Convolution Size
- Very Deep Network with 3x3 Convolutions
- These “effectively” give rise to larger convolutions

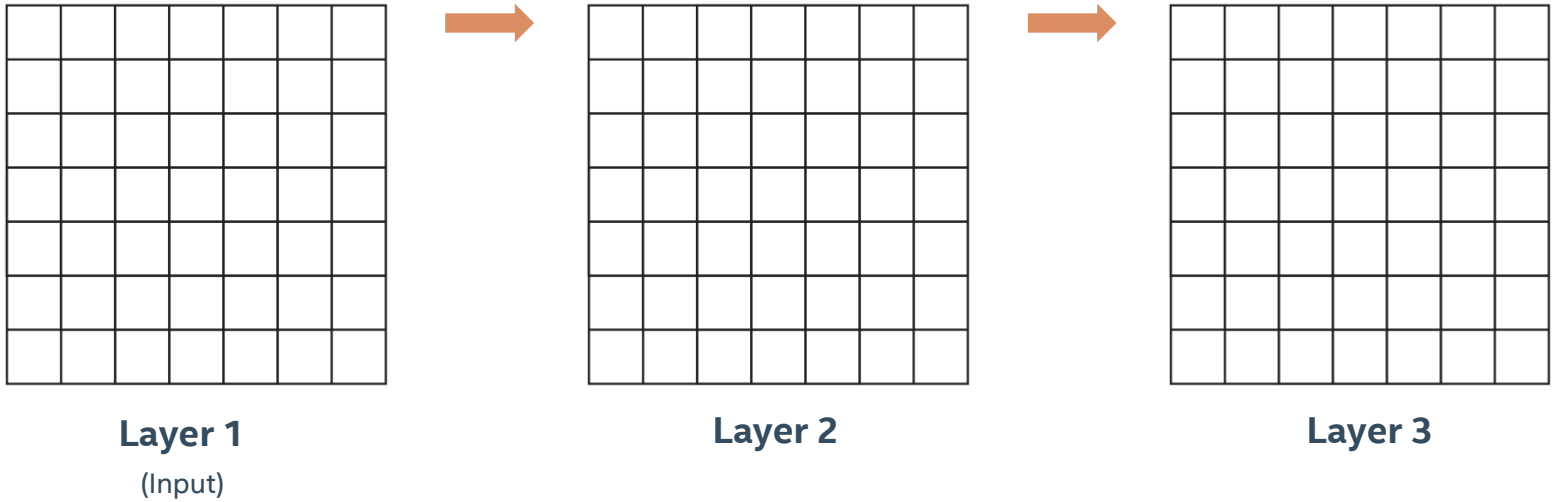
Reference: *Very Deep Convolutional Networks for Large-Scale Image Recognition*

Karen Simonyan and Andrew Zisserman, 2014

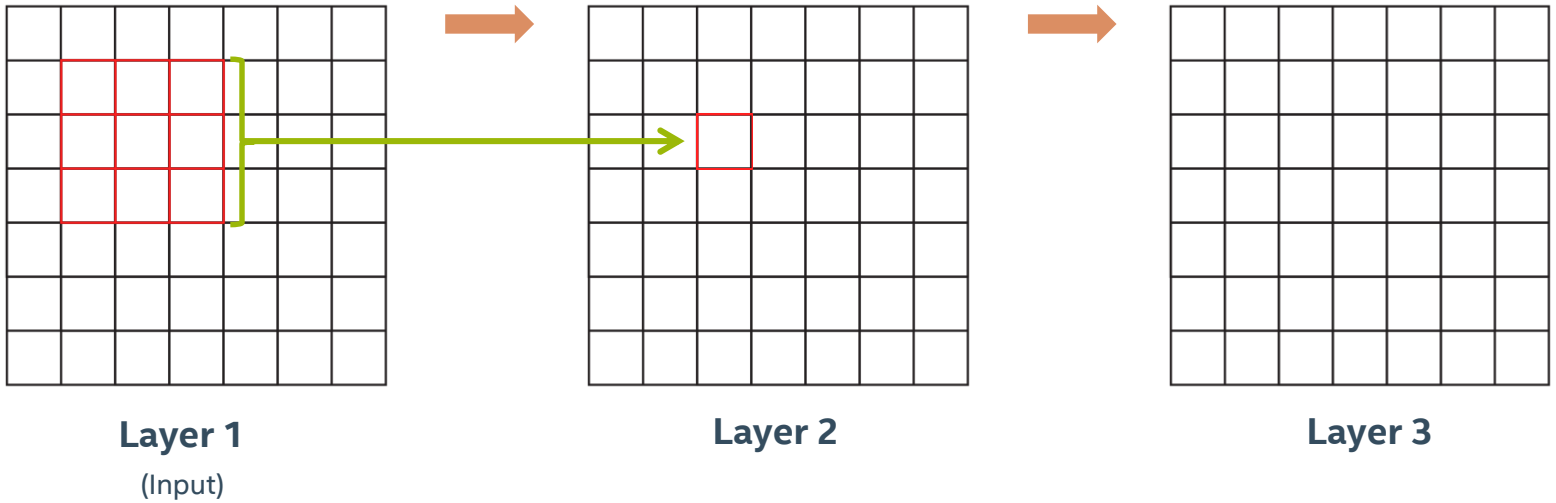
VGG16 DIAGRAM



VGG

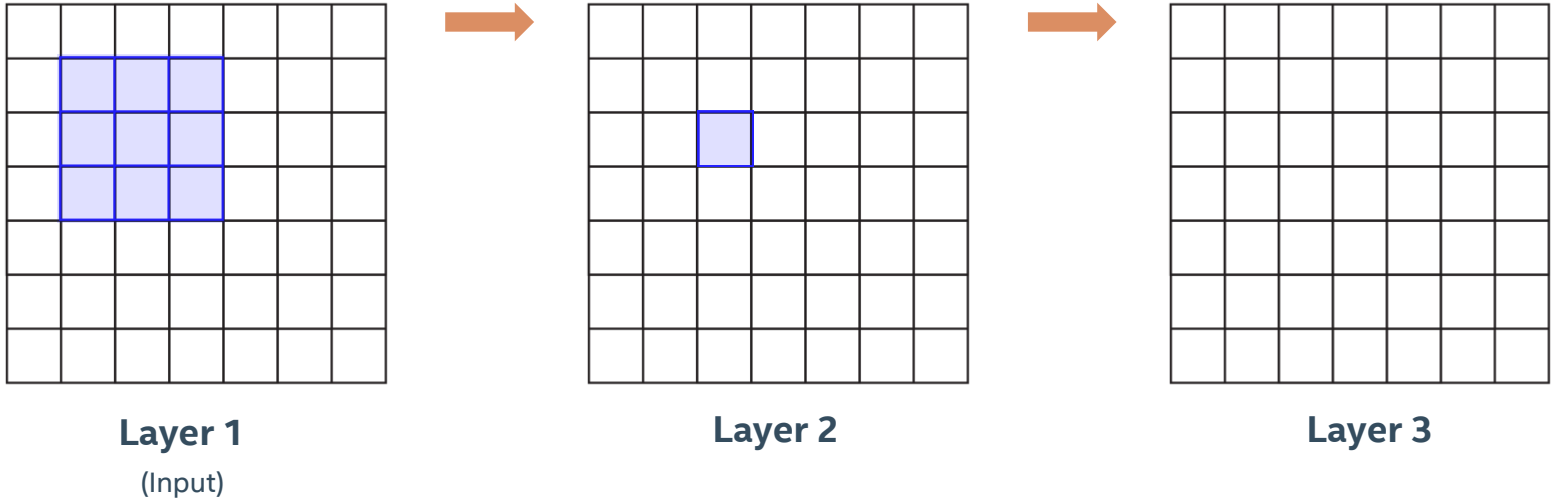


VGG



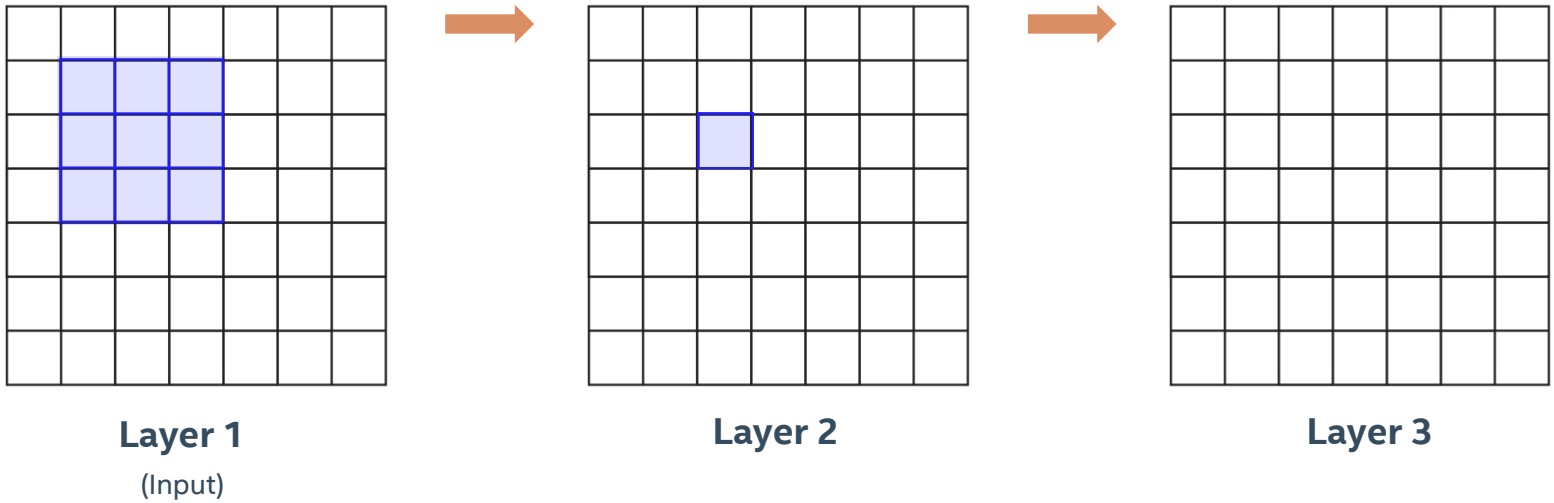
VGG

We can say that the “receptive field” of layer 2 is 3x3.
Each output has been influenced by a 3x3 patch of inputs.



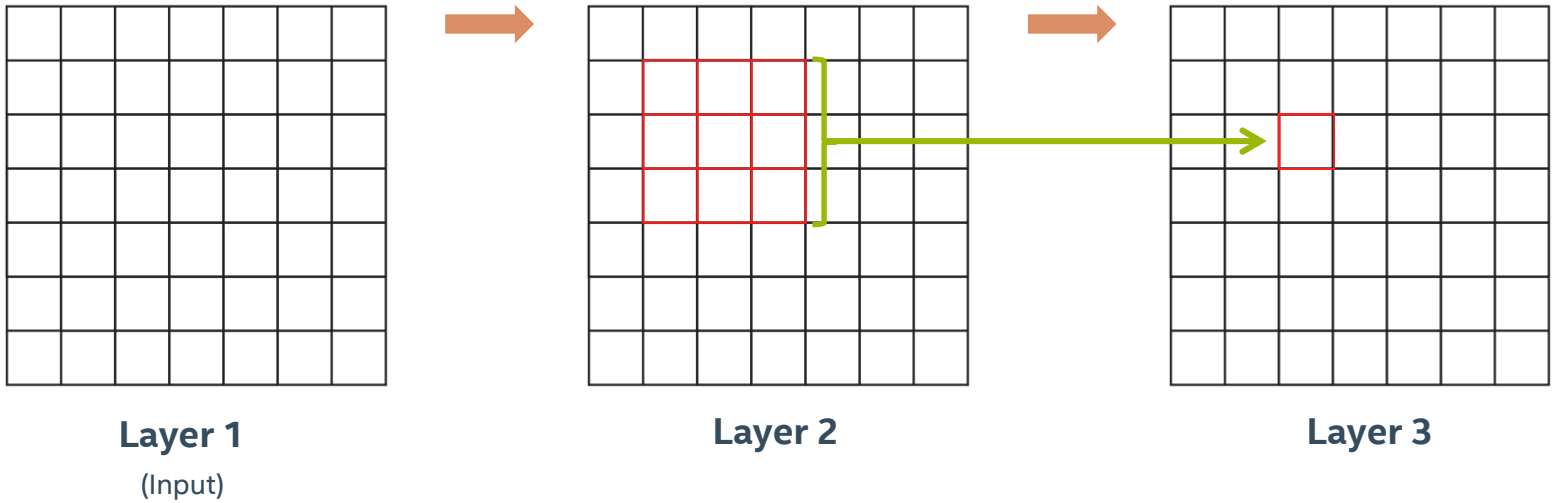
VGG

What about on layer 3?

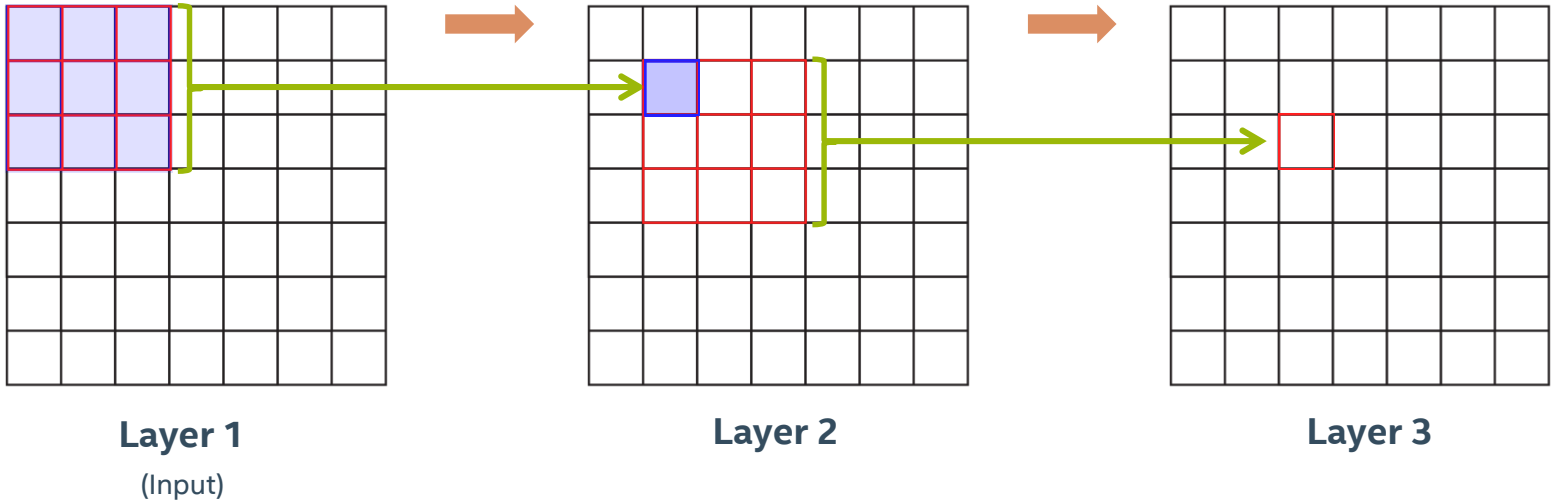


VGG

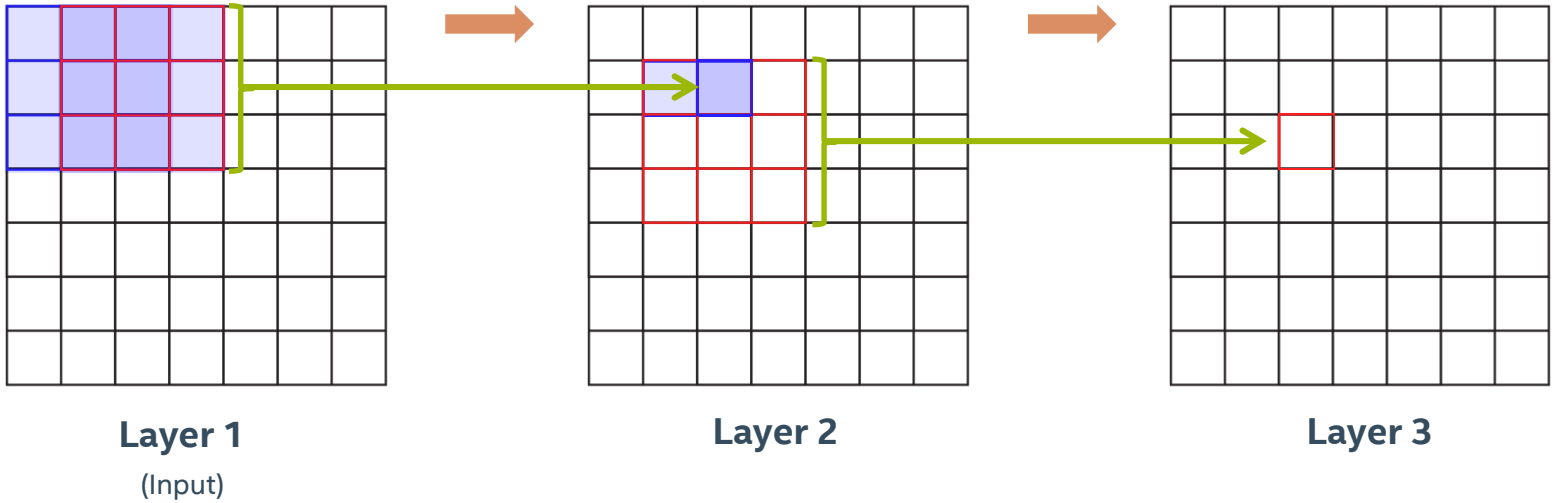
This output on Layer 3 uses a 3x3 patch from layer 2.
How much from layer 1 does it use?



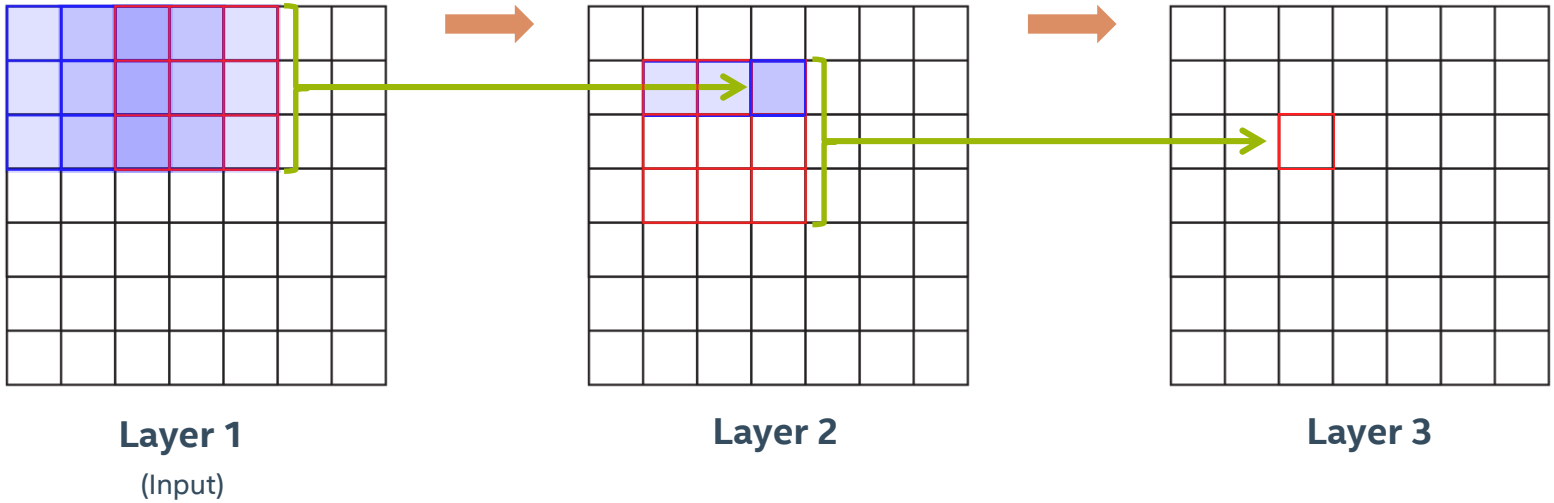
VGG



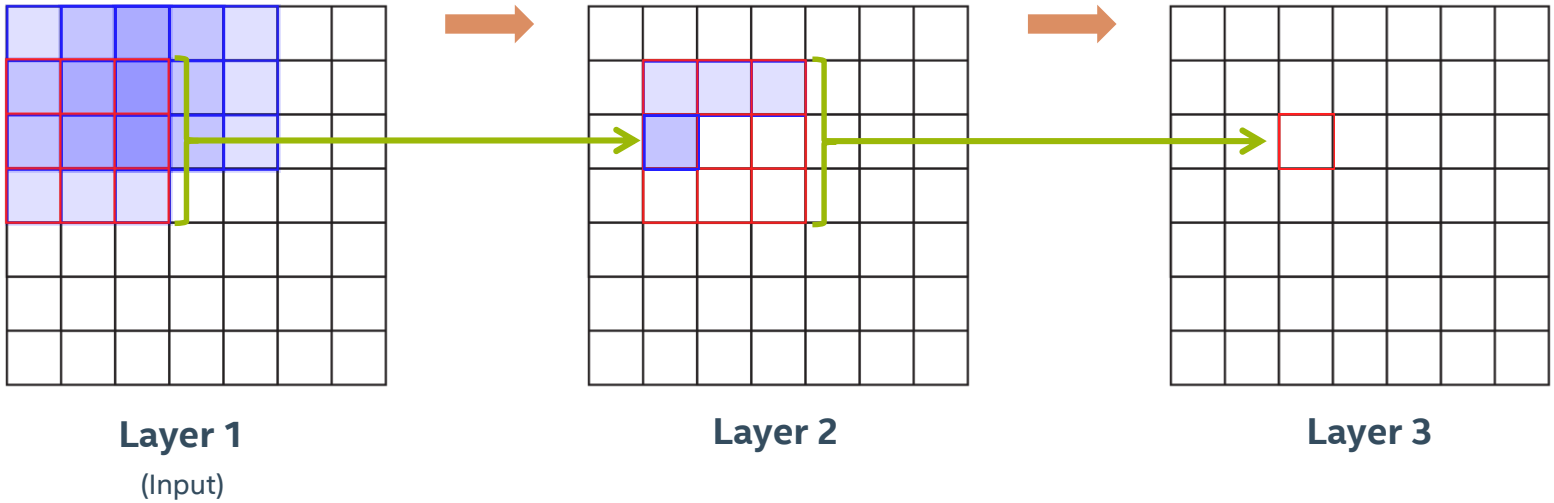
VGG



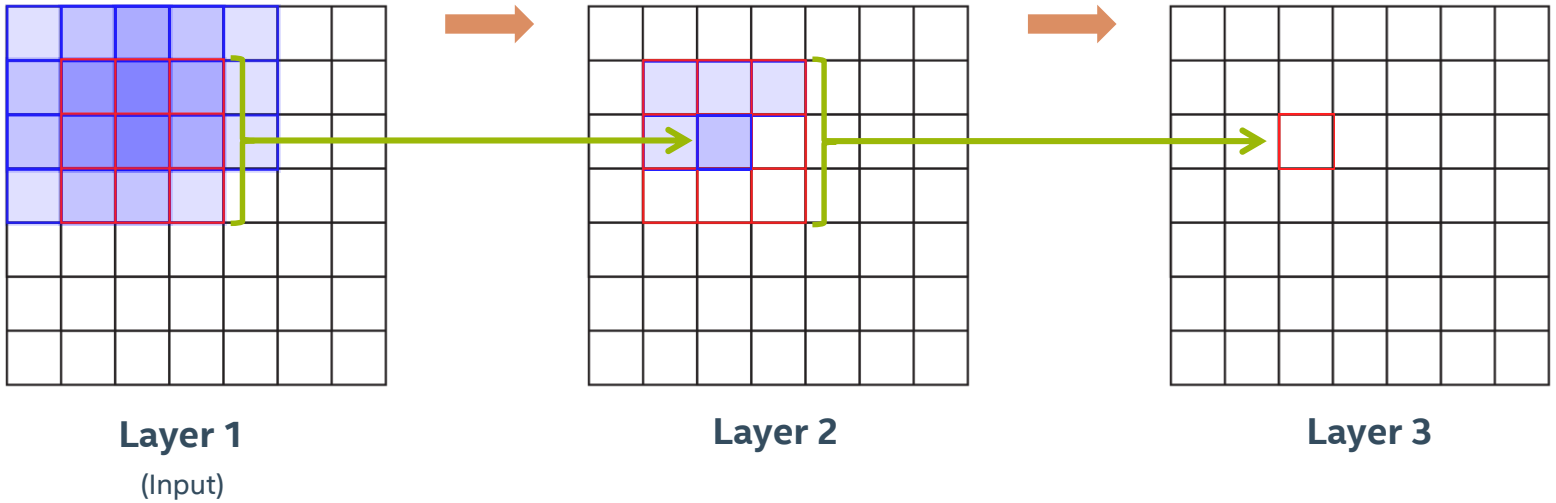
VGG



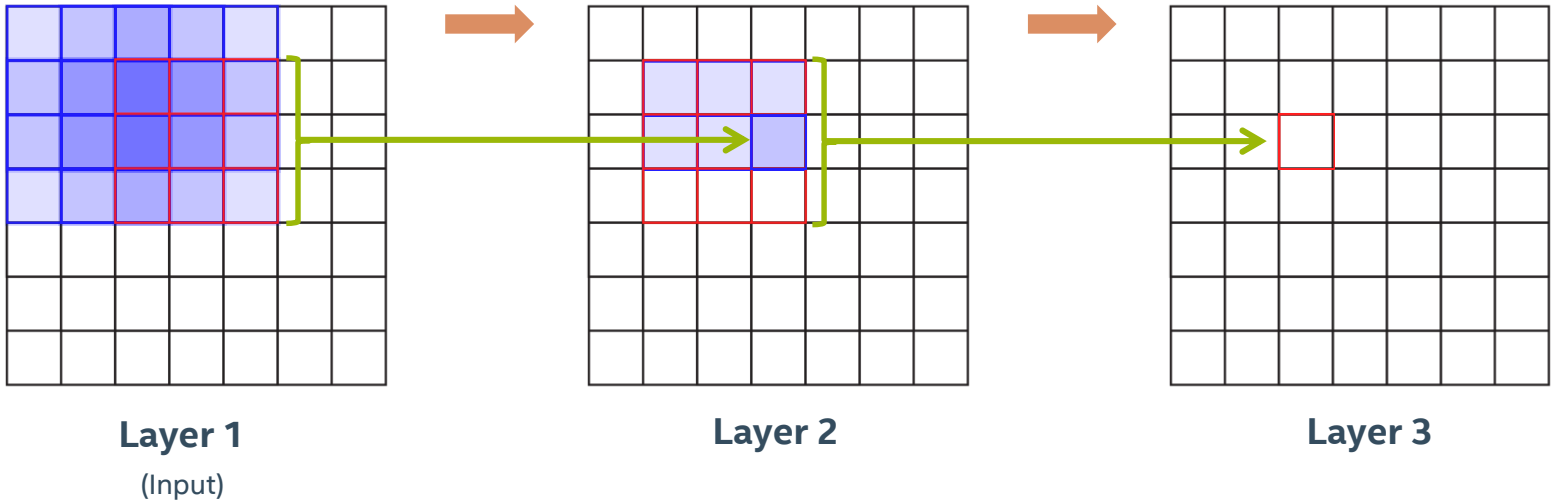
VGG



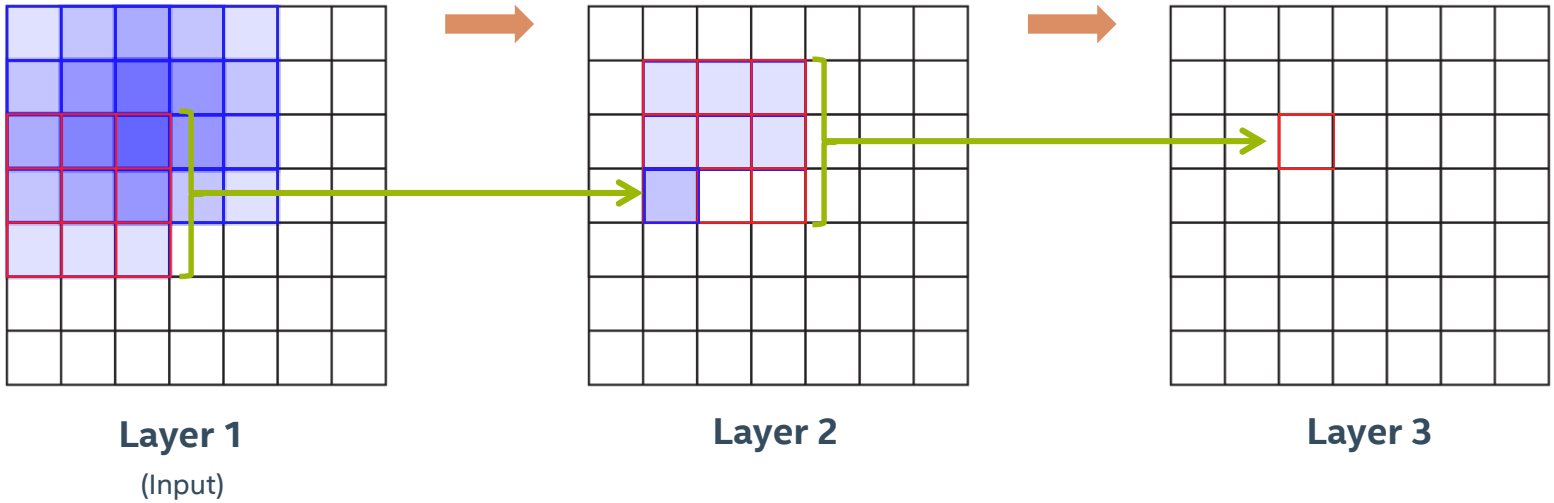
VGG



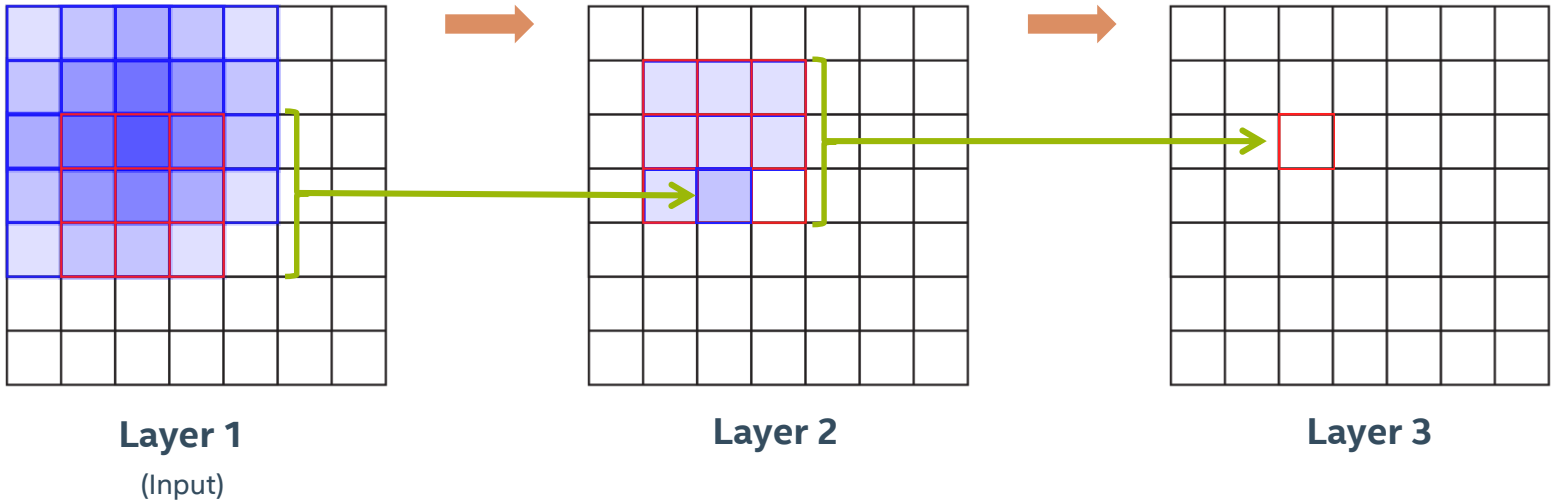
VGG



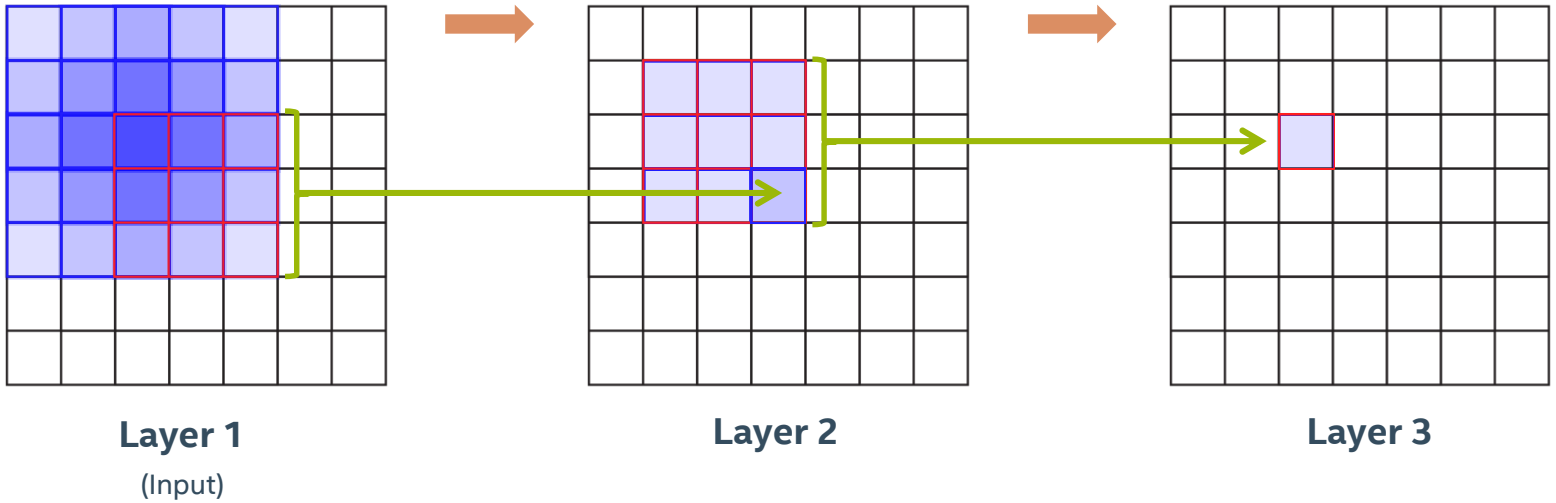
VGG



VGG

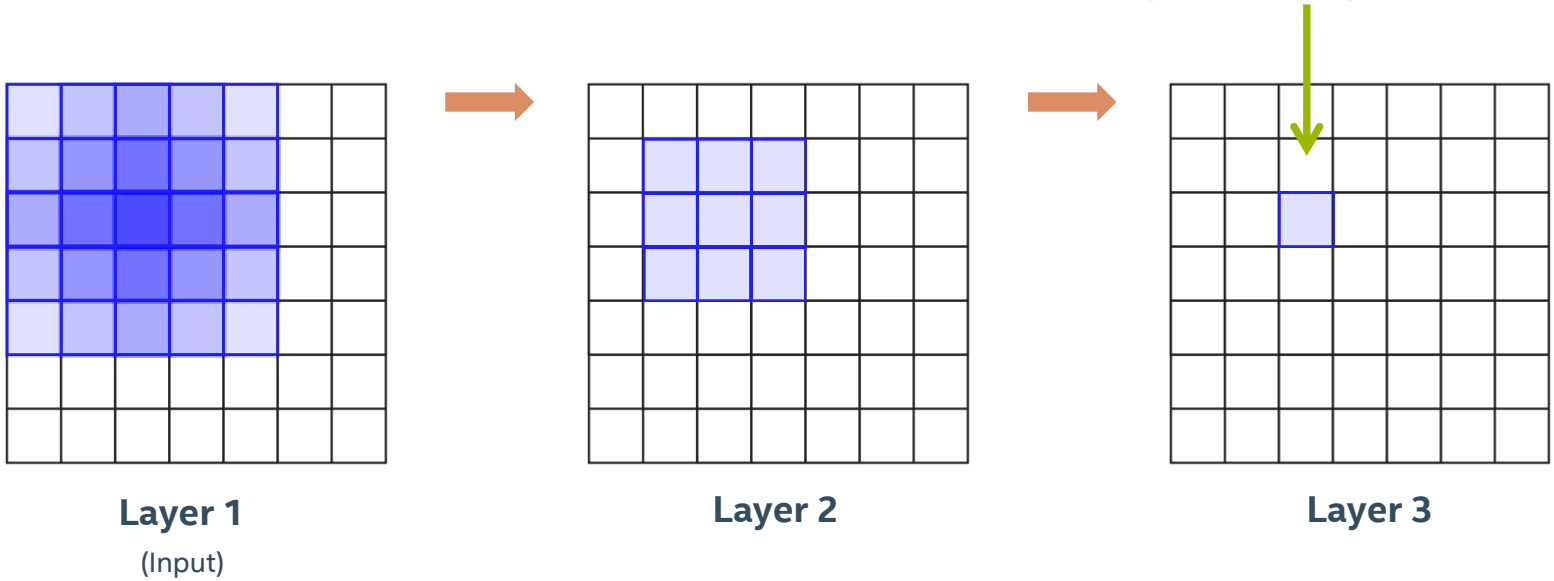


VGG



VGG

Each square in layer 3 “sees” a 5x5 grid from layer 1.



VGG

Two 3x3, stride 1 convolutions in a row → one 5x5.

Three 3x3 convolutions → one 7x7 convolution.

Benefit: fewer parameters.

One 3x3 layer

$$3 \times 3 \times C \times C = 9C^2$$

One 7x7 layer

$$7 \times 7 \times C \times C = 49C^2$$

Three 3x3 layers

$$3 \times (9C^2) = 27C^2$$

$49C^2$ → $27C^2$ → ≈45% reduction!

VGG

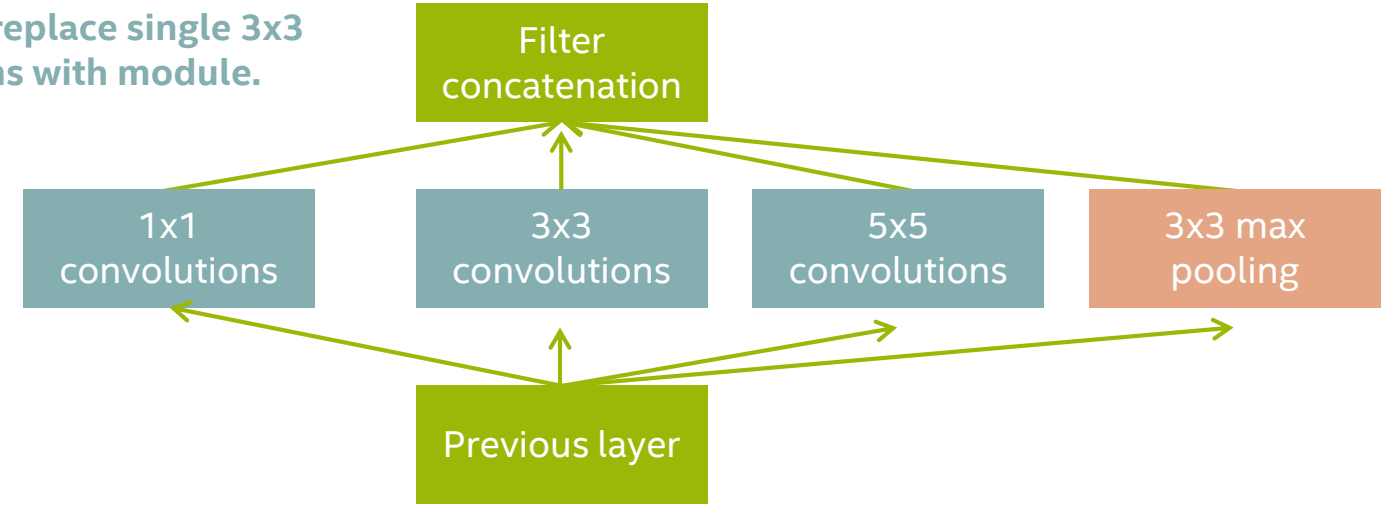
- One of the first architectures to experiment with many layers (More is better!)
- Can use multiple 3x3 convolutions to simulate larger kernels with fewer parameters
- Served as ”base model” for future works

INCEPTION

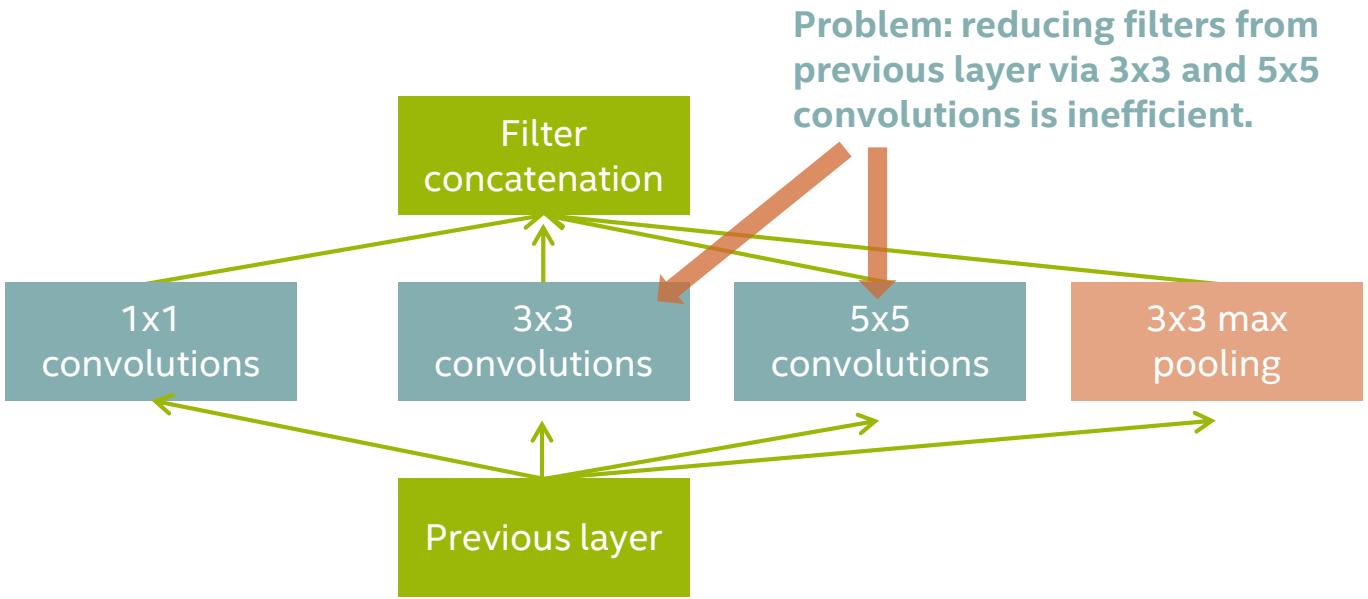
- Szegedy et al 2014
- Idea: network would want to use different receptive fields
- Want computational efficiency
- Also want to have sparse activations of groups of neurons
- Hebbian principle: “Fire together, wire together”
- Solution: Turn each layer into branches of convolutions
- Each branch handles smaller portion of workload
- Concatenate different branches at the end

INCEPTION

Basic idea: replace single 3x3 convolutions with module.



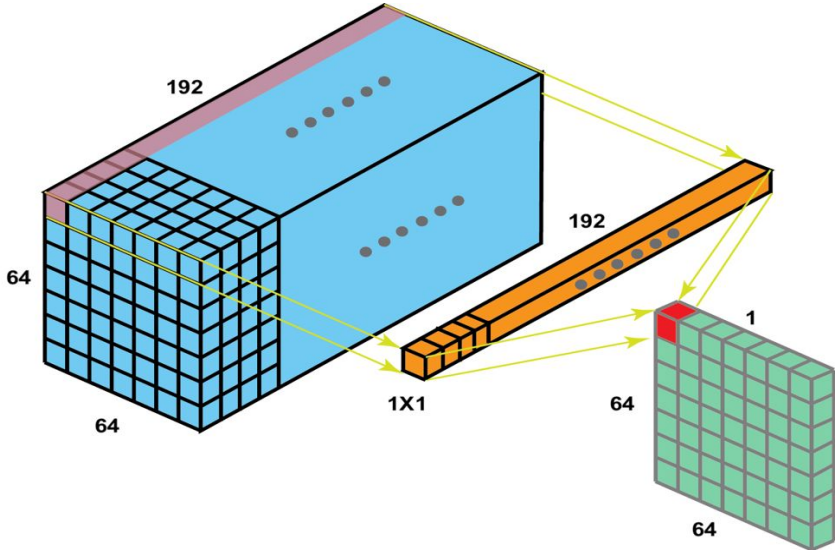
INCEPTION


Problem: reducing filters from previous layer via 3x3 and 5x5 convolutions is inefficient.
Filter concatenation
1x1 convolutions
3x3 convolutions
5x5 convolutions
3x3 max pooling
Previous layer

1x1 convolution

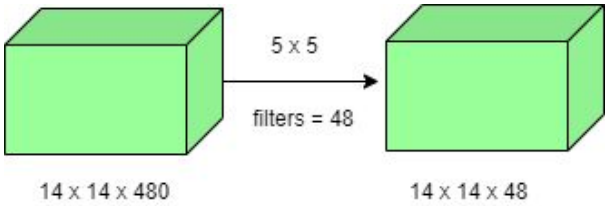
A 1x1 convolution simply maps an input pixel with all it's channels to an output pixel, not looking at anything around itself.

It is often used to reduce the number of depth channels, since it is often very slow to multiply volumes with extremely large depths.

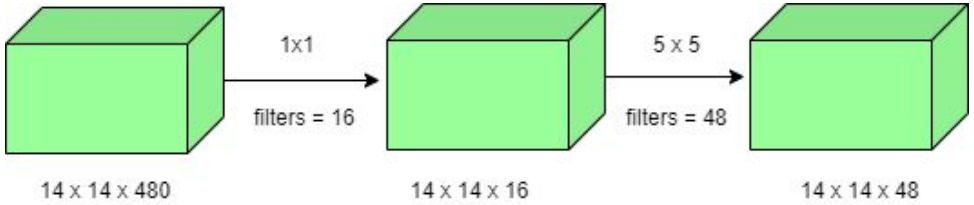


1x1 convolution - Example

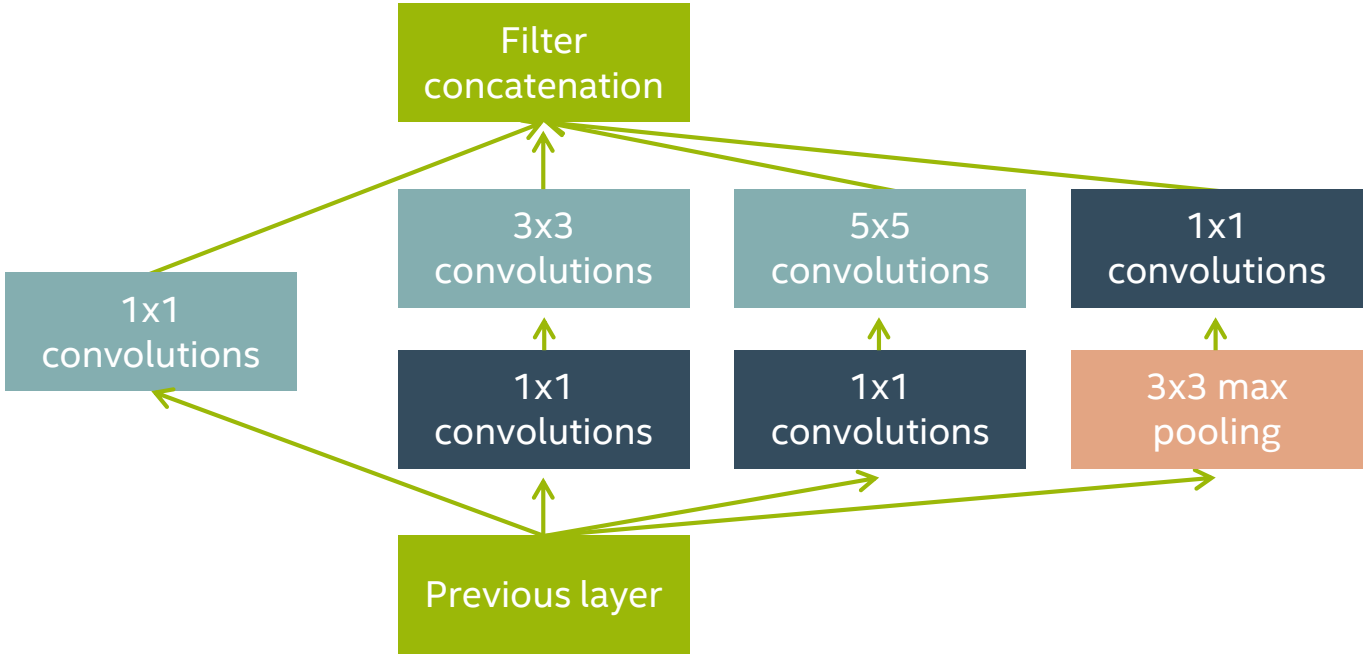
Number of operations involved here is *(14×14×48) × (5×5×480) = 112.9M*



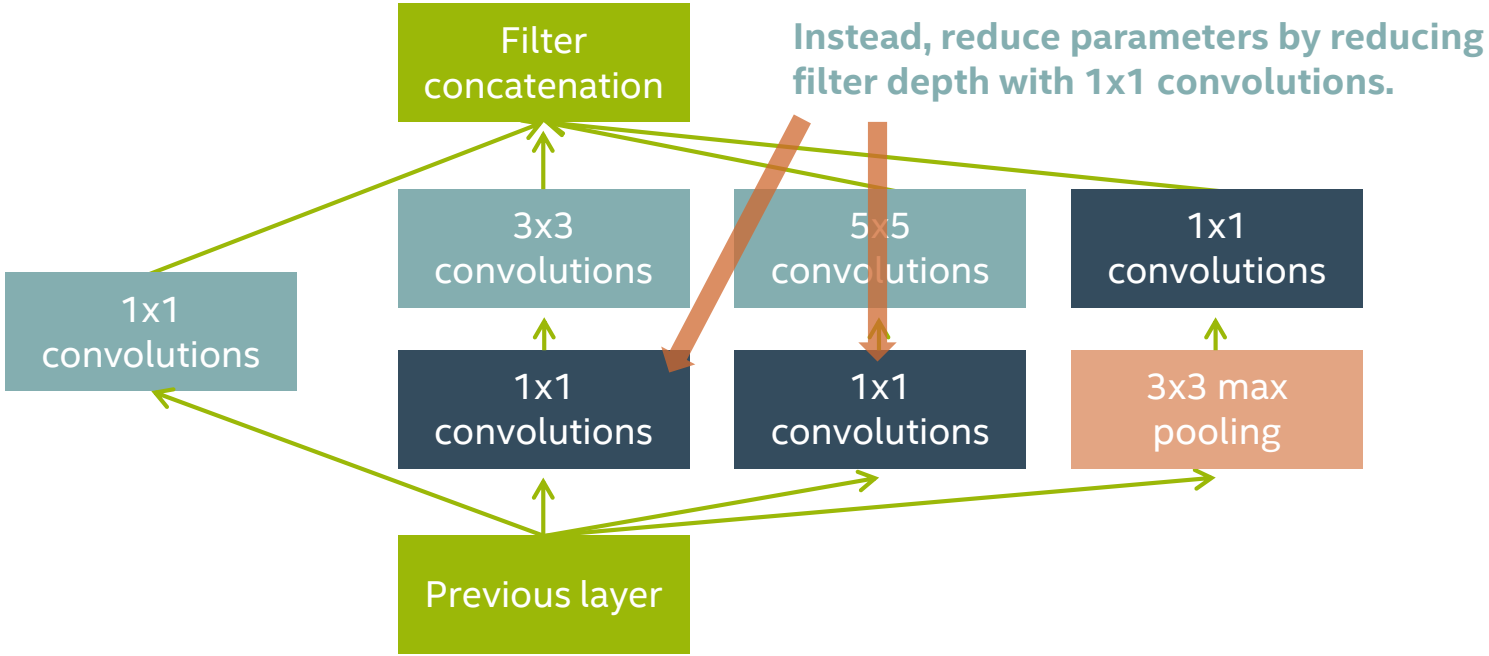
Number of operations for 1×1 convolution = *(14×14×16) × (1×1×480) = 1.5M*
Number of operations for 5×5 convolution = *(14×14×48) × (5×5×16) = 3.8M*
After addition we get, *1.5M + 3.8M = 5.3M*



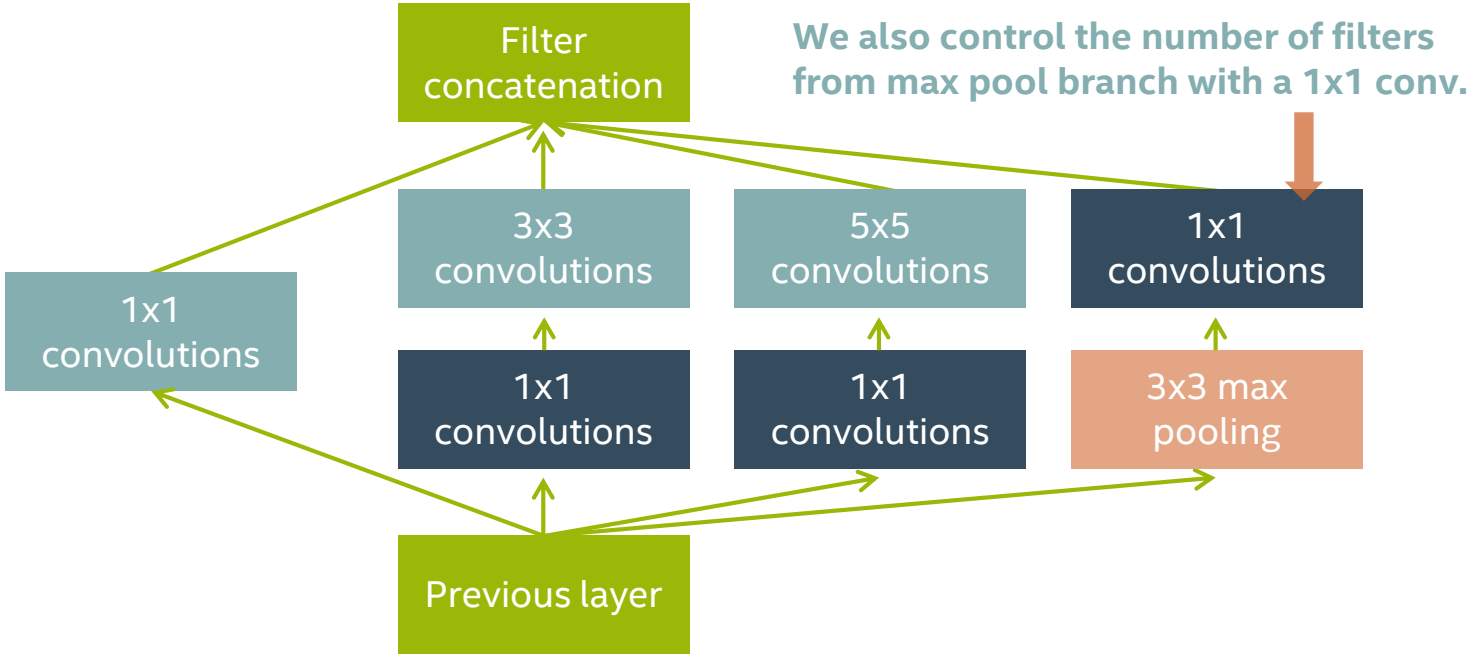
INCEPTION


Filter concatenation
1x1 convolutions
3x3 convolutions
5x5 convolutions
1x1 convolutions
1x1 convolutions
1x1 convolutions
3x3 max pooling
Previous layer

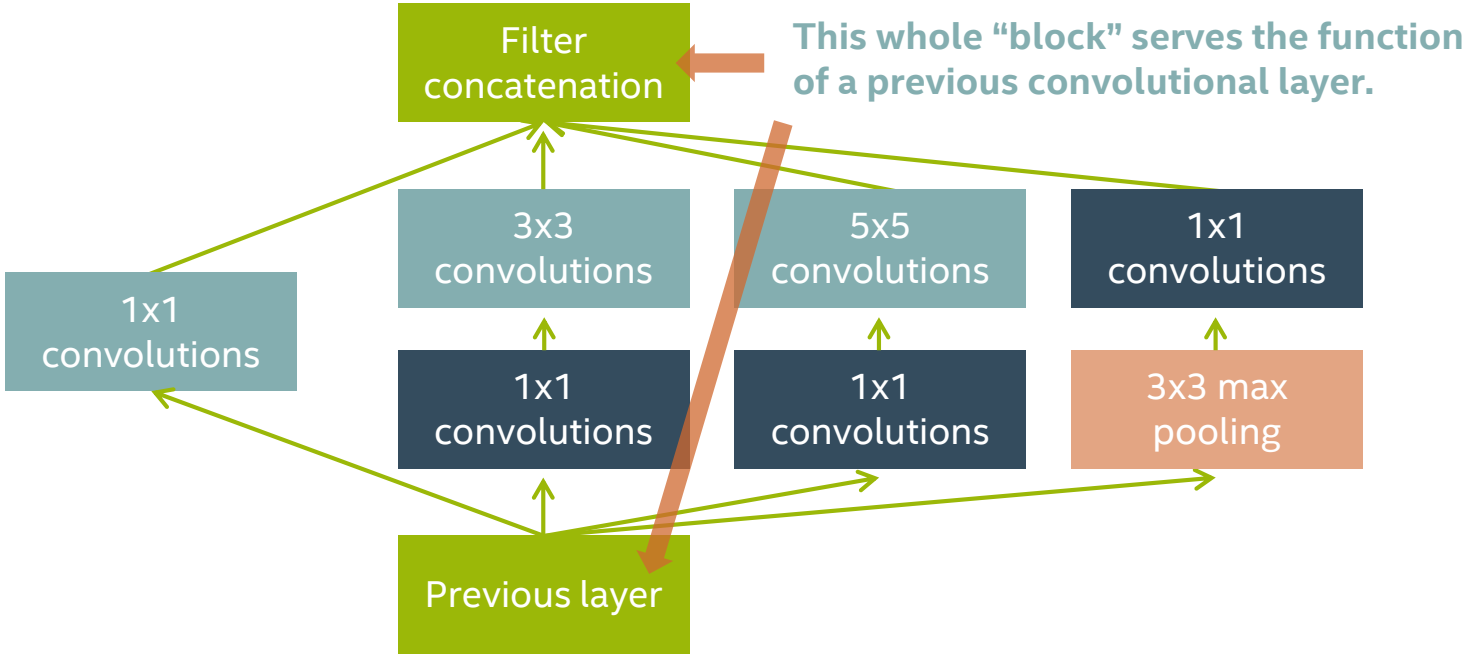
INCEPTION


Filter concatenation
Instead, reduce parameters by reducing filter depth with 1x1 convolutions.
1x1 convolutions
3x3 convolutions
5x5 convolutions
1x1 convolutions
1x1 convolutions
1x1 convolutions
3x3 max pooling
Previous layer

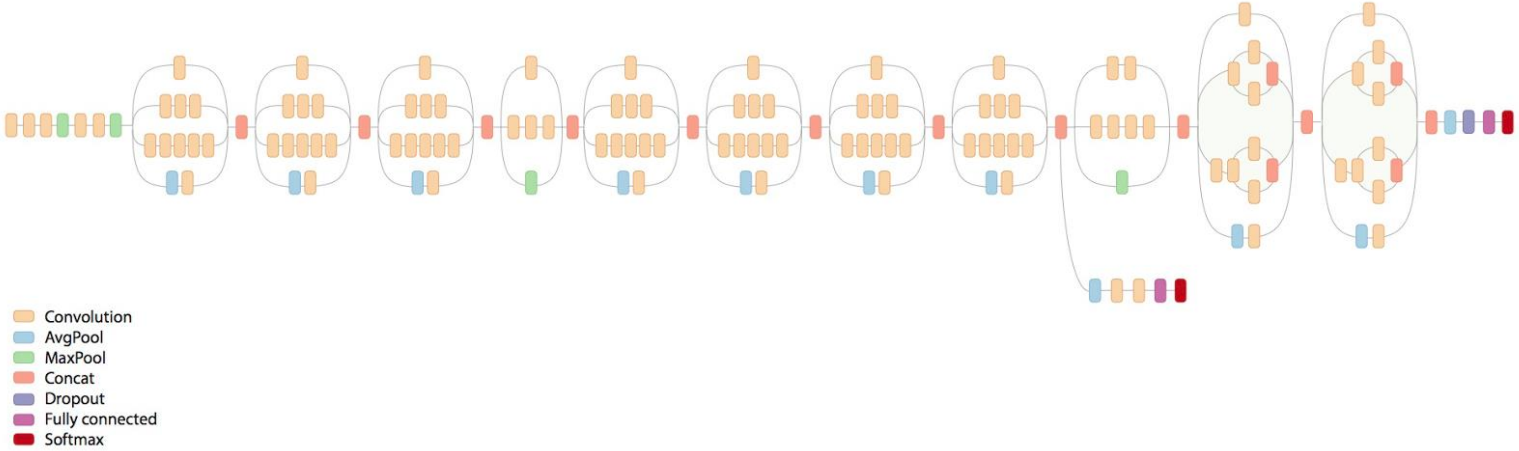
INCEPTION


Filter concatenation
We also control the number of filters from max pool branch with a 1x1 conv.
1x1 convolutions
3x3 convolutions
5x5 convolutions
1x1 convolutions
1x1 convolutions
1x1 convolutions
3x3 max pooling
Previous layer

INCEPTION

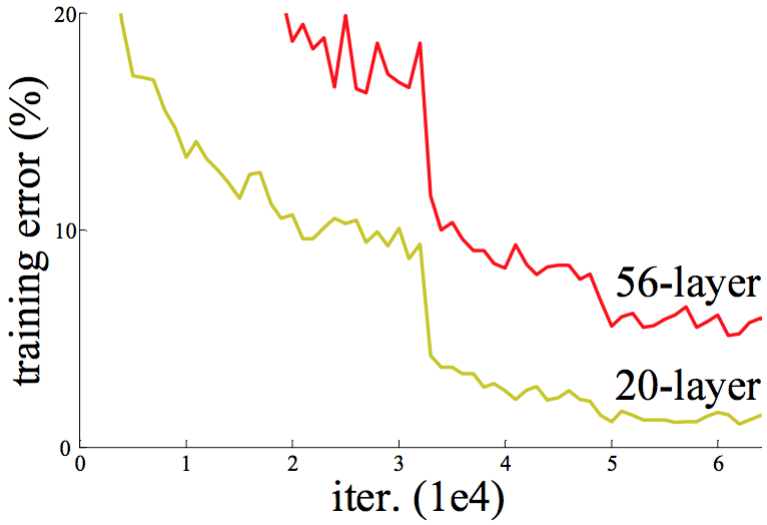

Filter concatenation
This whole "block" serves the function of a previous convolutional layer.
1x1 convolutions
3x3 convolutions
5x5 convolutions
1x1 convolutions
1x1 convolutions
1x1 convolutions
3x3 max pooling
Previous layer

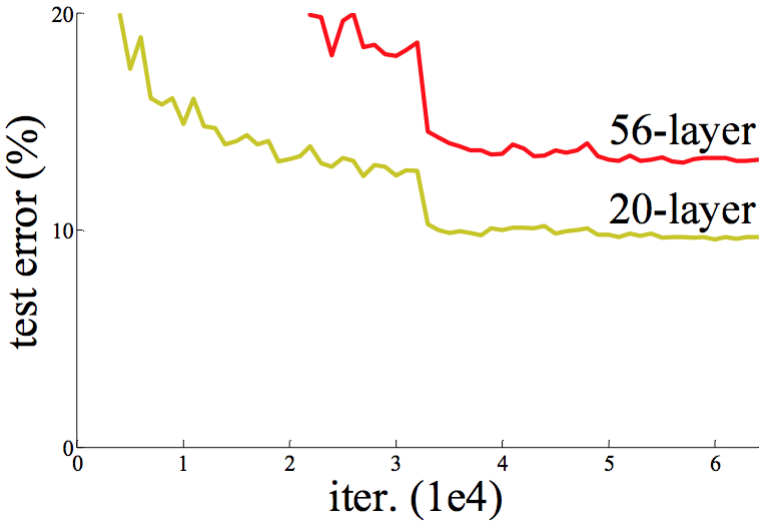
INCEPTION V3 SCHEMATIC



RESNET—MOTIVATION

Issue: Deeper Networks performing worse on training data! (as well as test data)



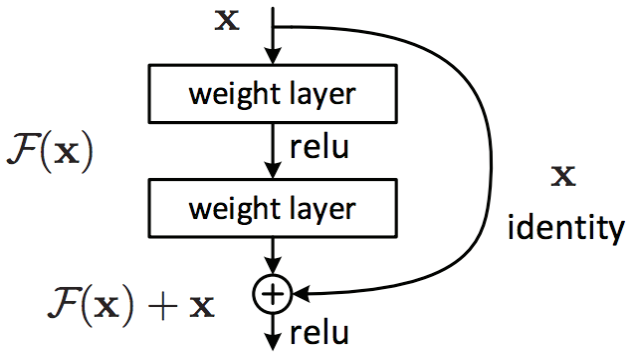


RESNET

- Surprising because deeper networks should overfit more
- So what's happening?
- Early layers of Deep Networks are very slow to adjust
- Analogous to "Vanishing Gradient" issue
- In theory, should be able to just have an "identity" transformation that makes the deeper network behave like a shallower one

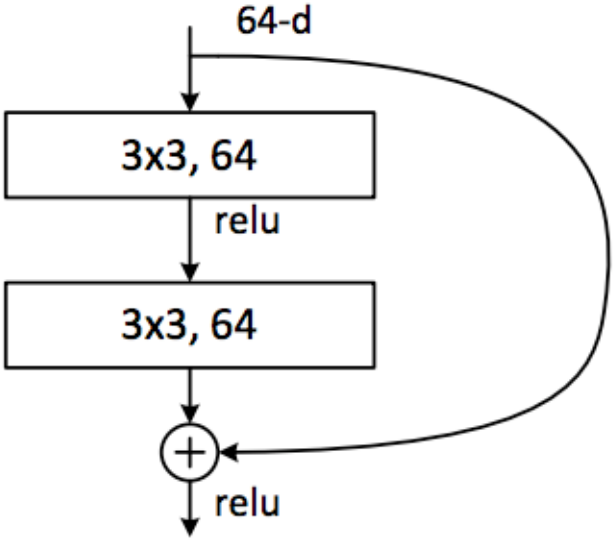
RESNET

- Assumption: best transformation over multiple layers is close to $\mathcal{F}(x)+x$
- x → input to series of layers
- $\mathcal{F}(x)$ → function represented by several layers (such as convs)
- Enforce this by adding "shortcut connections"
- Add the inputs from an earlier layer to the output of current layer



RESNET

- Add previous layer back in to current layer!
- Similar idea to “boosting”



KERAS FUNCTIONAL API

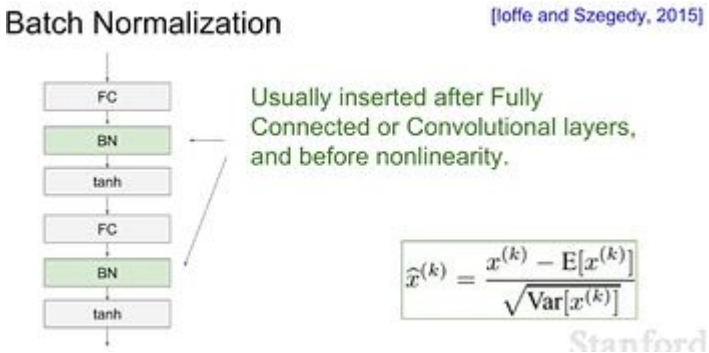
- So far we have primarily used the Keras Sequential method.
- Very convenient when each additional layer takes the results of the previous layer.
- However, suppose we have more than one input to a particular layer.
- Suppose we have multiple outputs, and want different loss functions for these outputs.
- More complicated graph structures require the Functional API.

KERAS FUNCTIONAL API—PRINCIPLES

- Every layer is “callable” on a tensor, and returns a tensor.
- Specifically designate inputs using the Input layer.
- Merge outputs into a single output using `keras.layers.concatentate`
- Stack layers in a similar fashion to the Sequential model.
- Use Model to specify inputs and outputs of your model.
- When compiling, can specify different losses for different outputs, and specify how they should be weighted.

Batch Normalization

- The distribution of the inputs to layers deep in the network may change after each mini-batch when the weights are updated.
- Therefore data should be normalized not only at input but after each transformation (layer).
- Batch normalization is a technique for training very deep neural networks that standardizes the inputs to a layer for each mini-batch.
- Helps to coordinate the update of multiple layers in the model.



intel Nervana™ AI Academy