# COMP [56]630– Machine Learning

Lecture 12 – Deep Learning (CNNs), Keras, Demo

# Key Ideas

- Take advantage of properties of natural signals
  - Local connections
  - Shared weights
  - Pooling
  - Use of many layers

# Comparison with Regular NNs

- Regular, Feed-forward NNs:
  - Need substantial number of training samples
  - Slow learning (convergence times)
  - Inadequate parameter selection techniques that lead to poor minima
- **Solution?**

# Comparison with Regular NNs

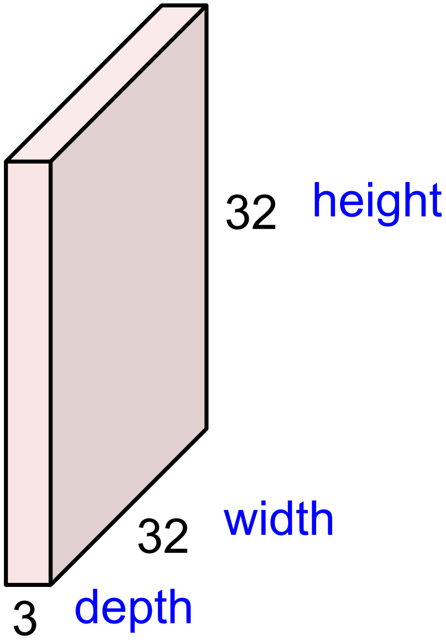
- Regular, Feed-forward NNs:
  - Need substantial number of training samples
  - Slow learning (convergence times)
  - Inadequate parameter selection techniques that lead to poor minima
- **Solution?**
- **Exploitation of Local Properties!**
- Network should exhibit invariance to translation, scaling and elastic deformations
  - A large training set can take care of this
- It ignores a key property of images
  - Nearby pixels are more strongly correlated than distant ones
  - Modern computer vision approaches exploit this property
- Information can be merged at later stages to get higher order features and about whole image

# Basic Mechanisms in CNNs

- Three Mechanisms of Convolutional Neural Networks:
  - Convolution Operation
  - Local Receptive Fields
  - Subsampling
  - Weight (Parameter) Sharing

# Convolution Layer

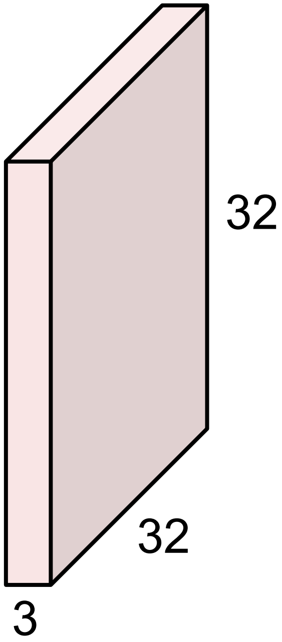
32x32x3 image -> preserve spatial structure



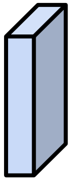
# Convolution Layer

Filters always extend the full depth of the input volume

32x32x3 image




5x5x3 filter

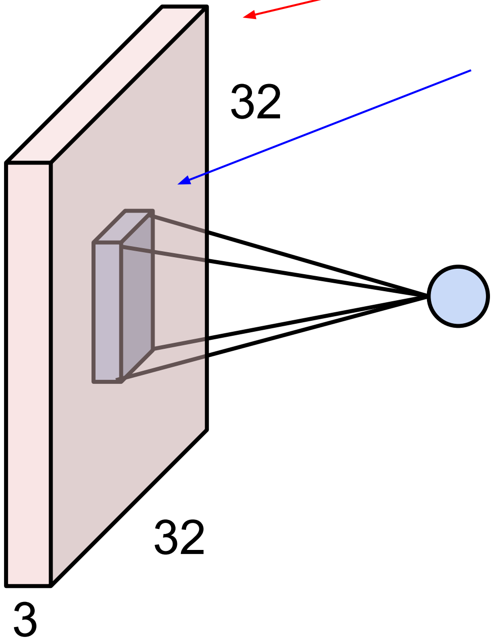


**Convolve** the filter with the image
i.e. "slide over the image spatially, computing dot products"

# Convolution Layer

32x32x3 image

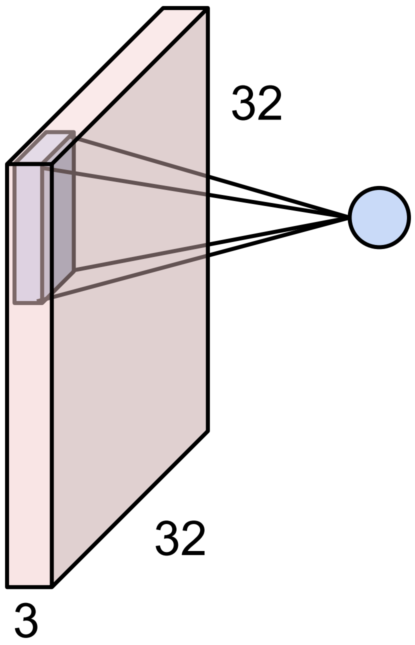
5x5x3 filter $w$



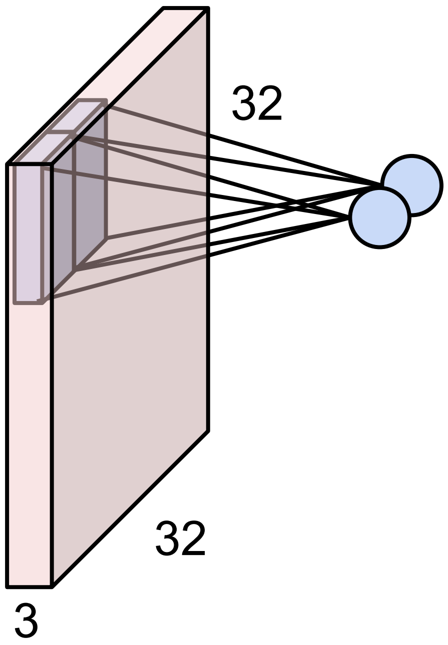

**1 number:**
the result of taking a dot product between the filter and a small 5x5x3 chunk of the image
(i.e. 5*5*3 = 75-dimensional dot product + bias)

$$w^T x + b$$

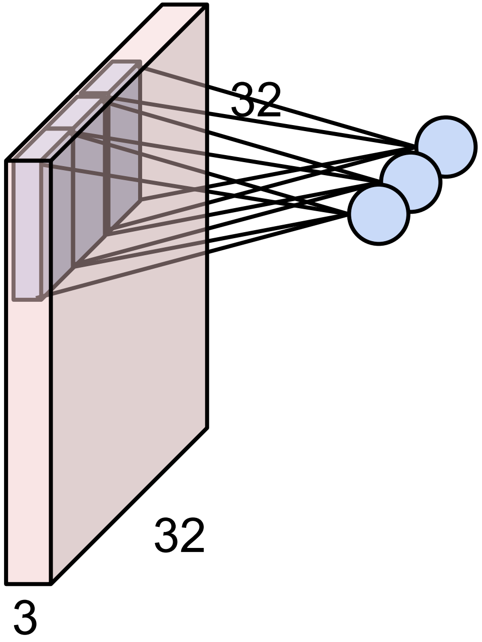
# Convolution Layer



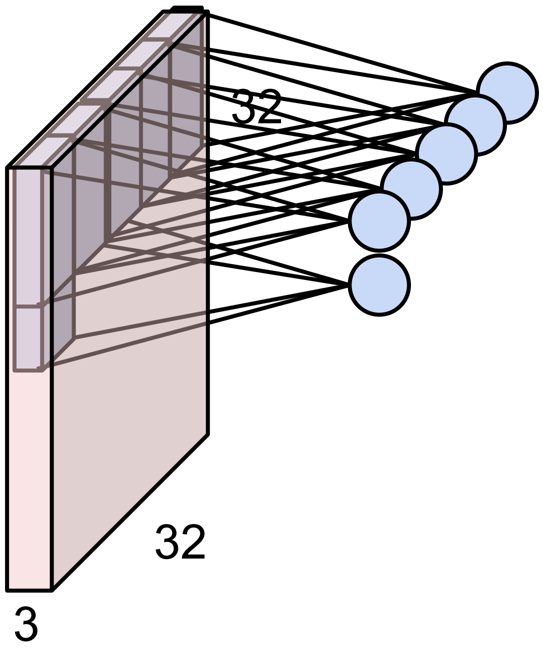
# Convolution Layer



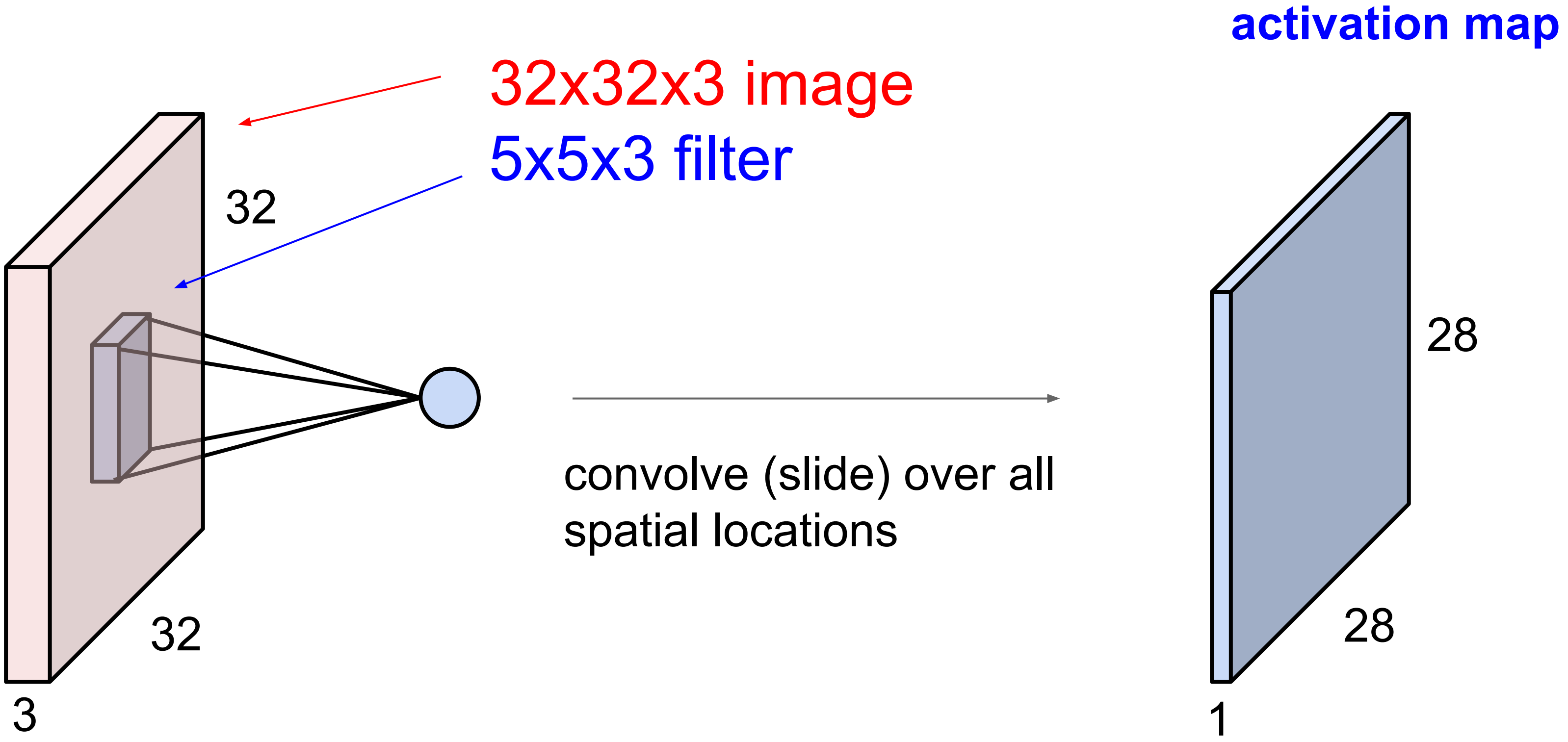
# Convolution Layer



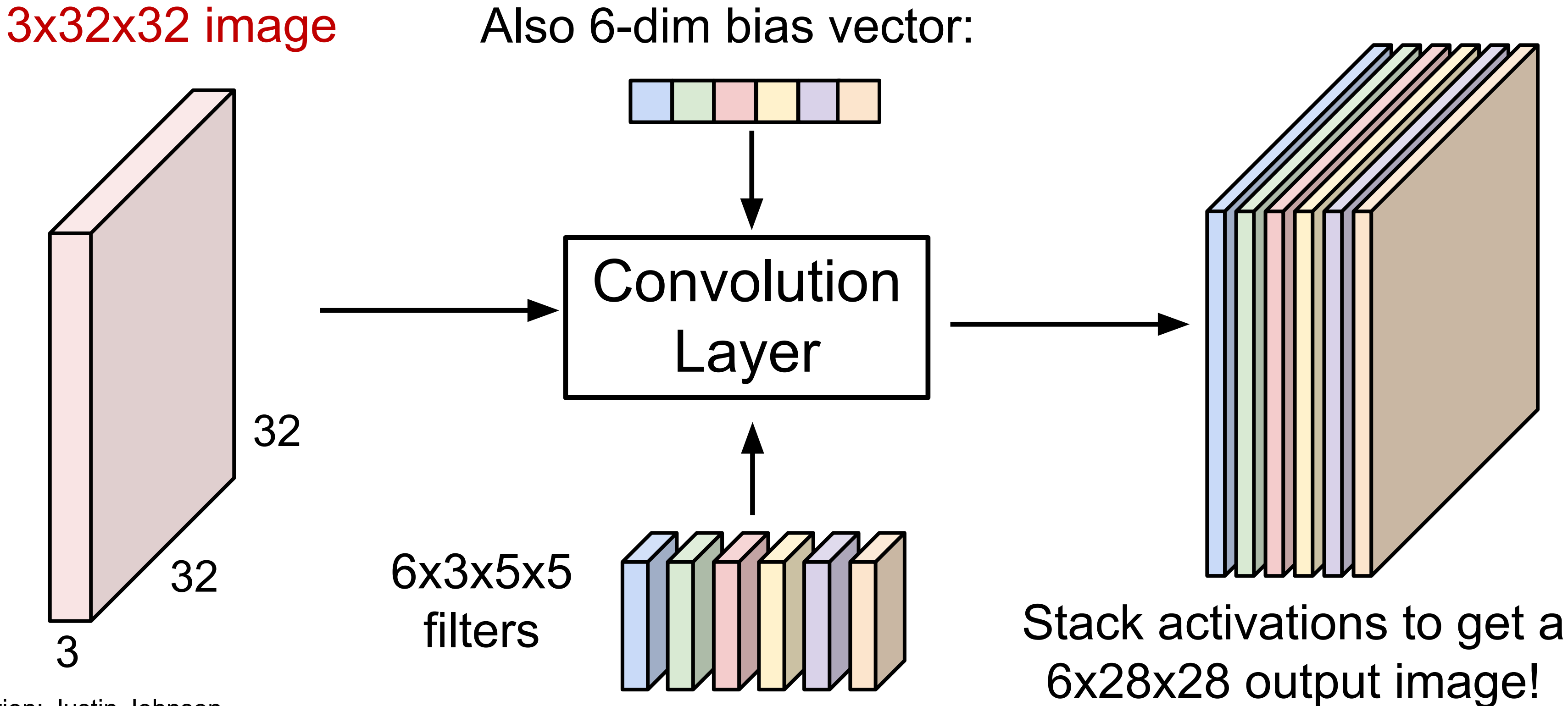
# Convolution Layer



# Convolution Layer

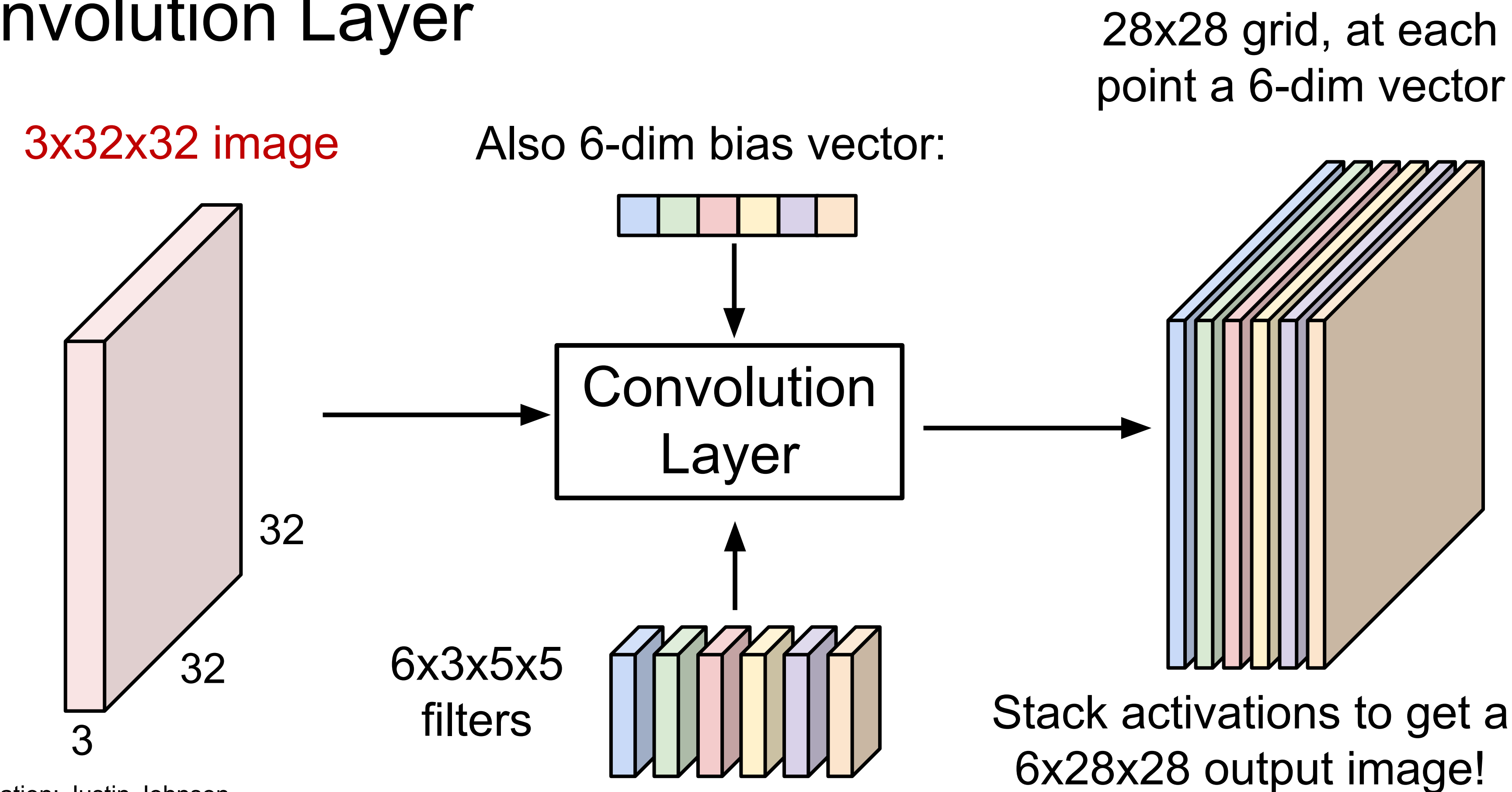


# Convolution Layer



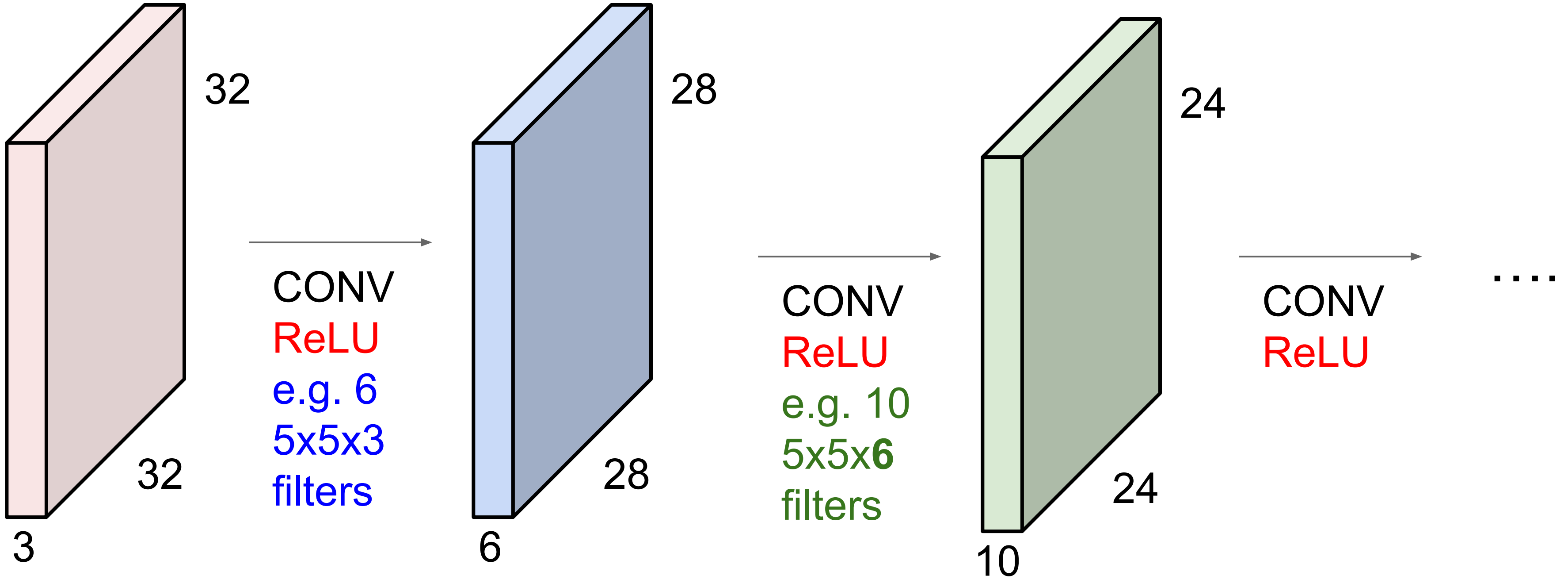

Slide inspiration: Justin Johnson

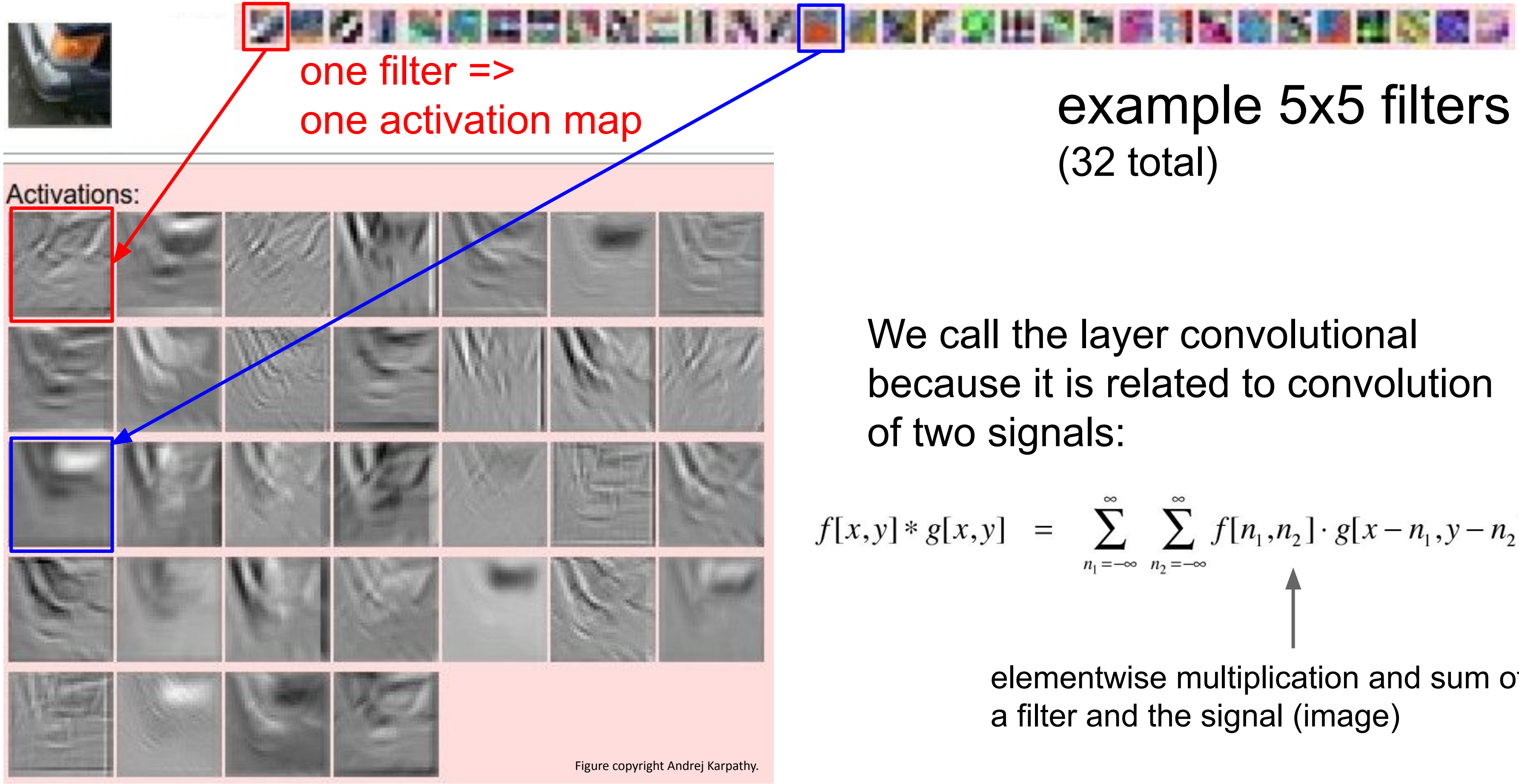
# Convolution Layer




Slide inspiration: Justin Johnson

**Preview:** ConvNet is a sequence of Convolution Layers, interspersed with activation functions





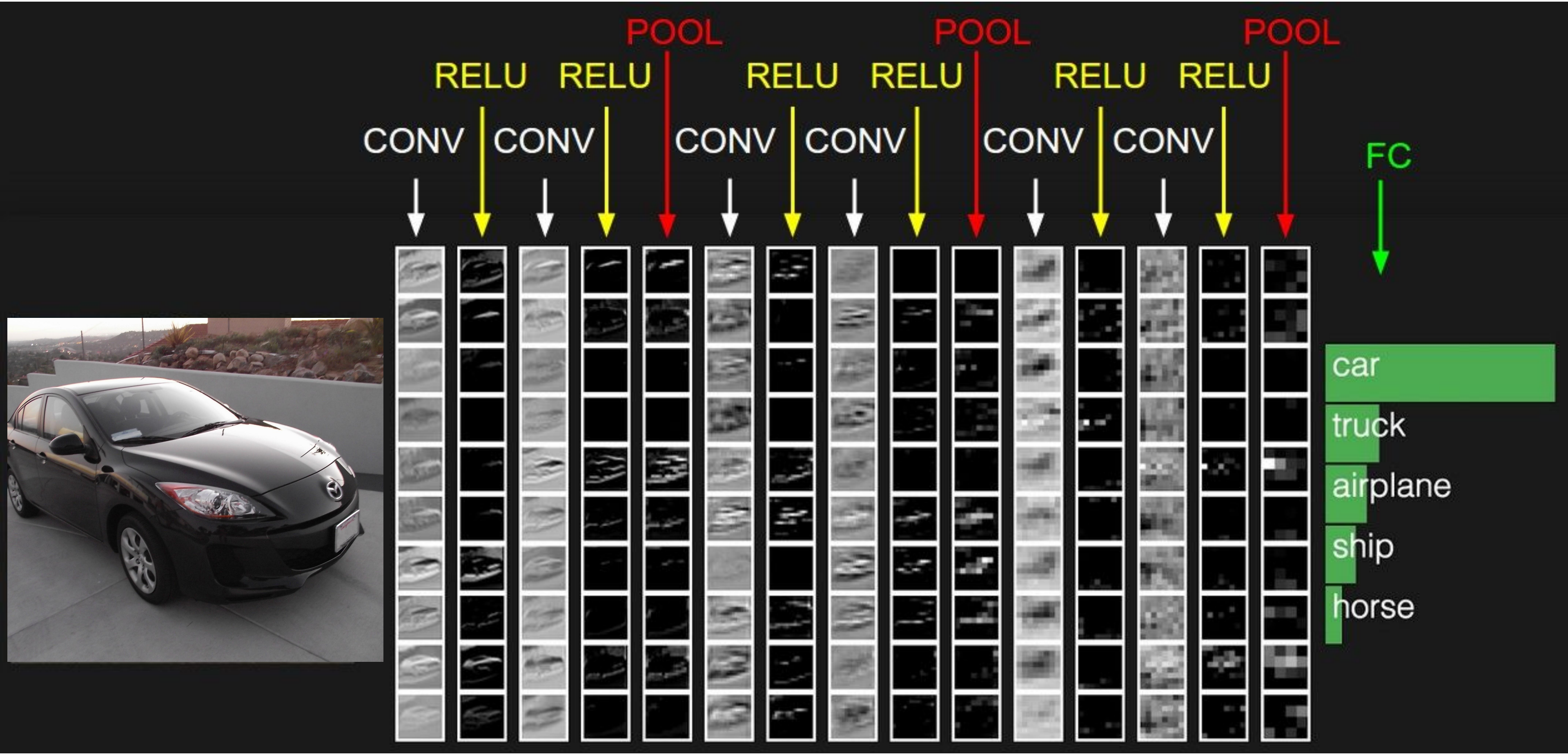



example 5x5 filters
(32 total)

We call the layer convolutional because it is related to convolution of two signals:

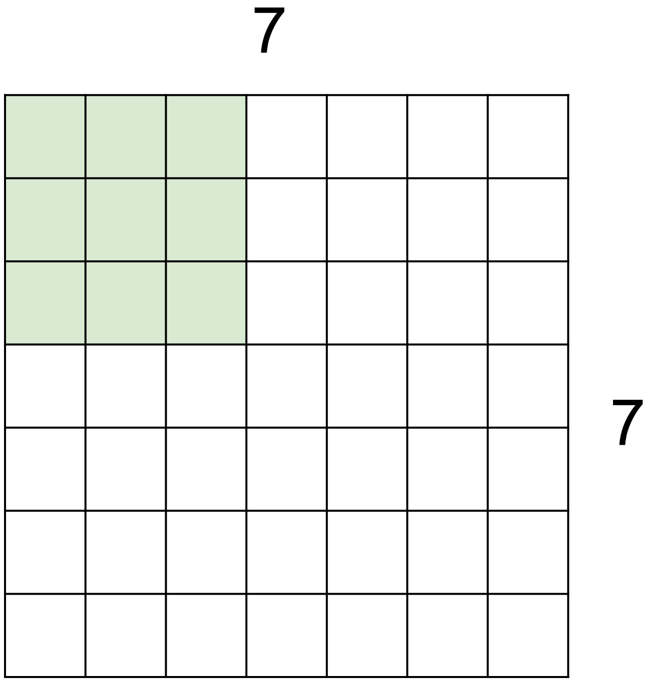
$$f[x,y]*g[x,y] = \sum_{n_1=-\infty}^{\infty}\sum_{n_2=-\infty}^{\infty} f[n_1,n_2]\cdot g[x-n_1,y-n_2]$$

elementwise multiplication and sum of a filter and the signal (image) preview:



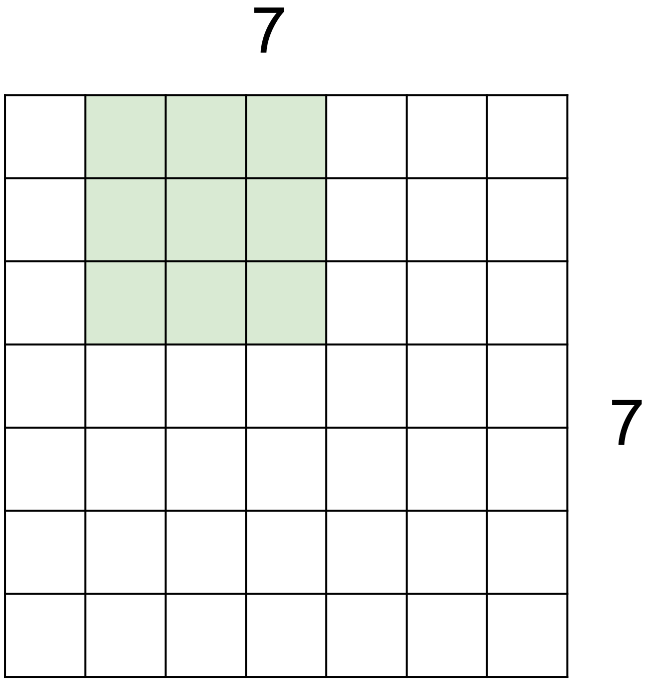
POOL
POOL
POOL
RELU
RELU
RELU
RELU
RELU
RELU
CONV
CONV
CONV
CONV
CONV
CONV
FC
car
truck
airplane
ship
horse

# A closer look at spatial dimensions:



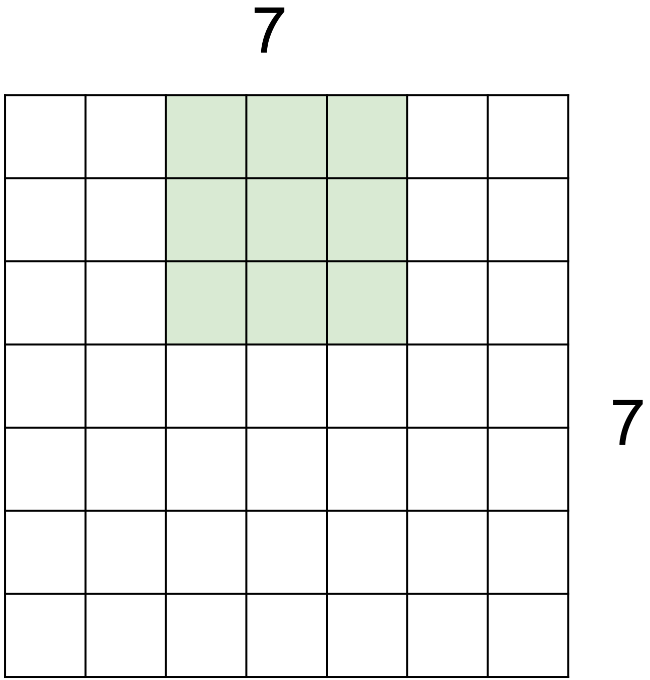

7x7 input (spatially)
assume 3x3 filter

# A closer look at spatial dimensions:



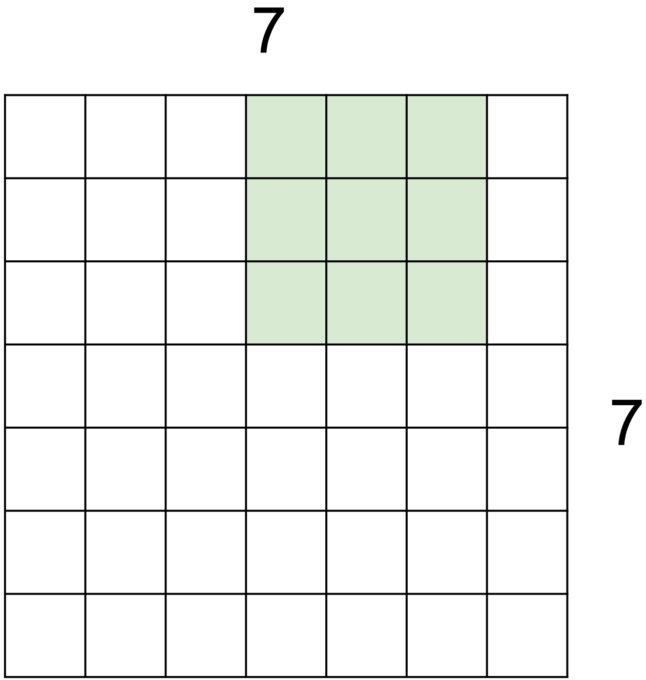

7x7 input (spatially)
assume 3x3 filter

# A closer look at spatial dimensions:



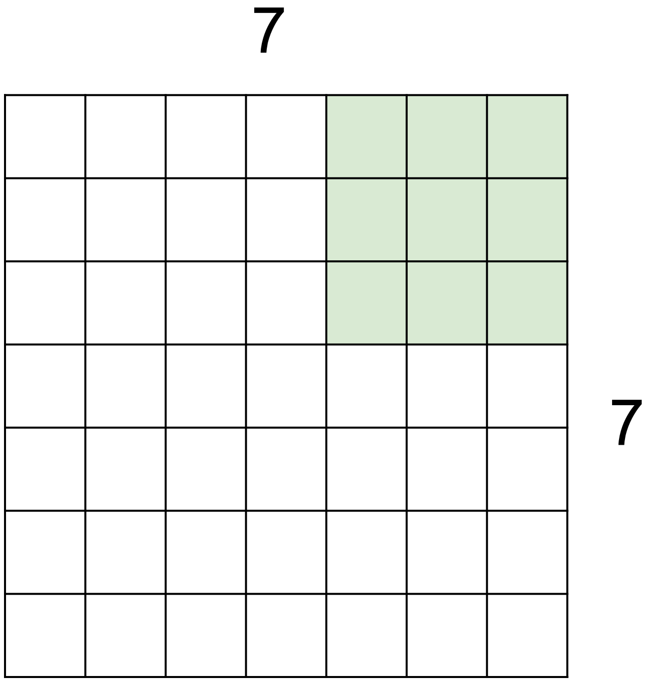

7x7 input (spatially)
assume 3x3 filter

A closer look at spatial dimensions:




7x7 input (spatially)
assume 3x3 filter

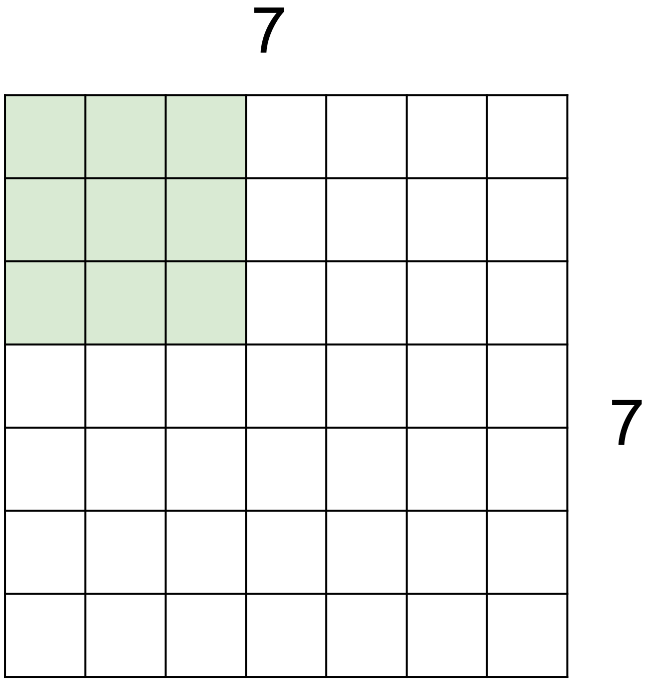
A closer look at spatial dimensions:




7x7 input (spatially)
assume 3x3 filter

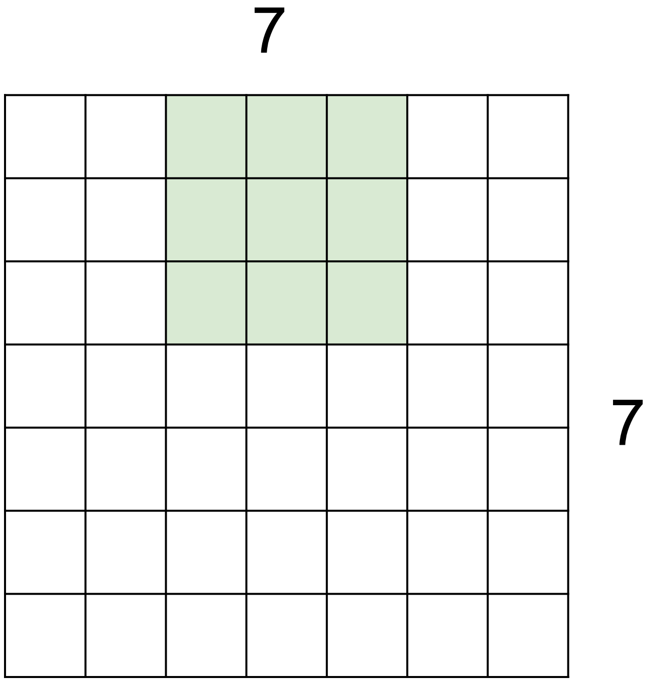
**=> 5x5 output**

# A closer look at spatial dimensions:



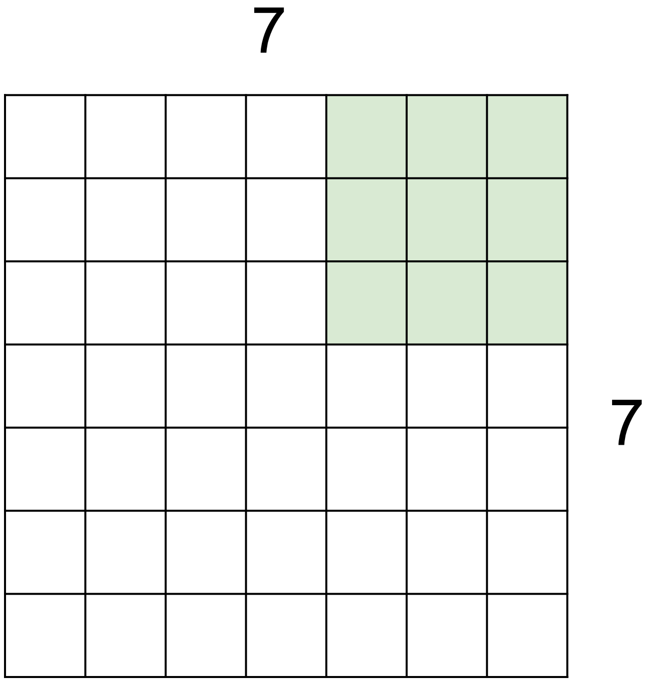

7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**

# A closer look at spatial dimensions:

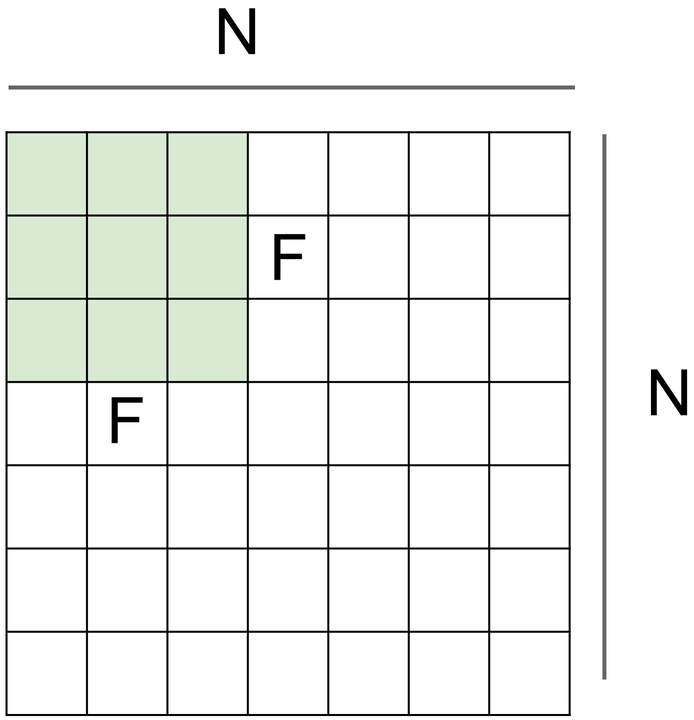



7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**

# A closer look at spatial dimensions:



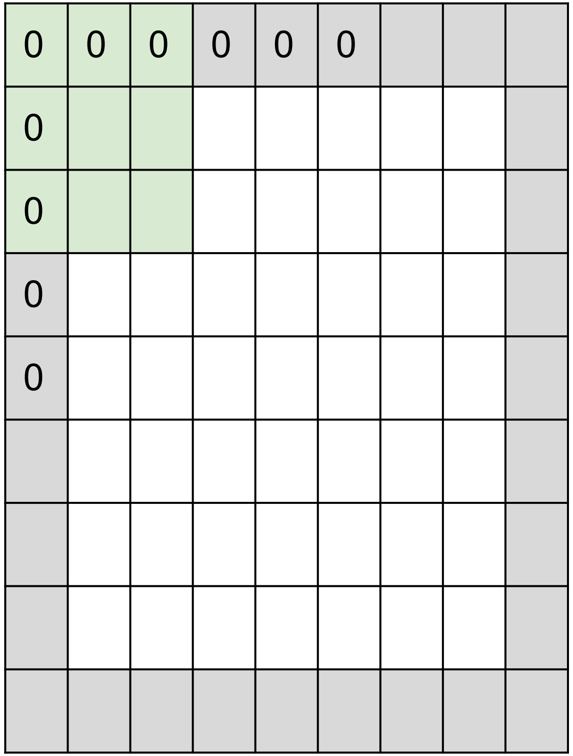

7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**
**=> 3x3 output!**




Output size:
**(N - F) / stride + 1**

e.g. N = 7, F = 3:
stride 1 => (7 - 3)/1 + 1 = 5
stride 2 => (7 - 3)/2 + 1 = 3
stride 3 => (7 - 3)/3 + 1 = 2.33 :\

# In practice: Common to zero pad the border

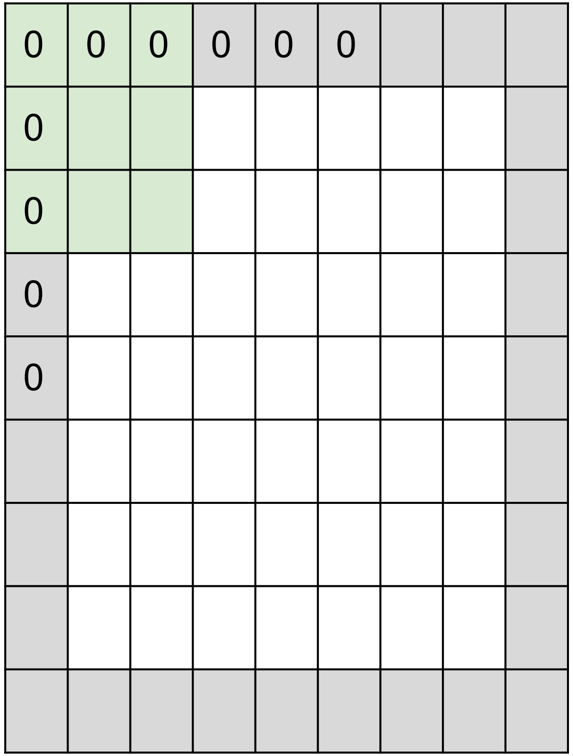



e.g. input 7x7
**3x3** filter, applied with **stride 1**
**pad with 1 pixel** border => what is the output?

**7x7 output!**

(recall:)
(N + 2P - F) / stride + 1

# In practice: Common to zero pad the border

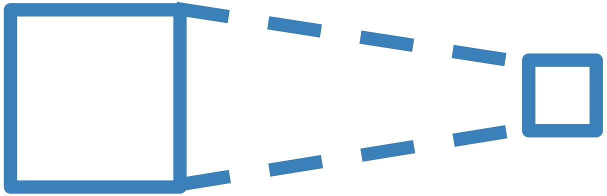



e.g. input 7x7
**3x3** filter, applied with **stride 1**
**pad with 1 pixel** border => what is the output?

**7x7 output!**
in general, common to see CONV layers with stride 1, filters of size FxF, and zero-padding with (F-1)/2. (will preserve size spatially)
e.g. F = 3 => zero pad with 1
F = 5 => zero pad with 2
F = 7 => zero pad with 3

# Receptive Fields

For convolution with kernel size K, each element in the output depends on a K x K **receptive field** in the input



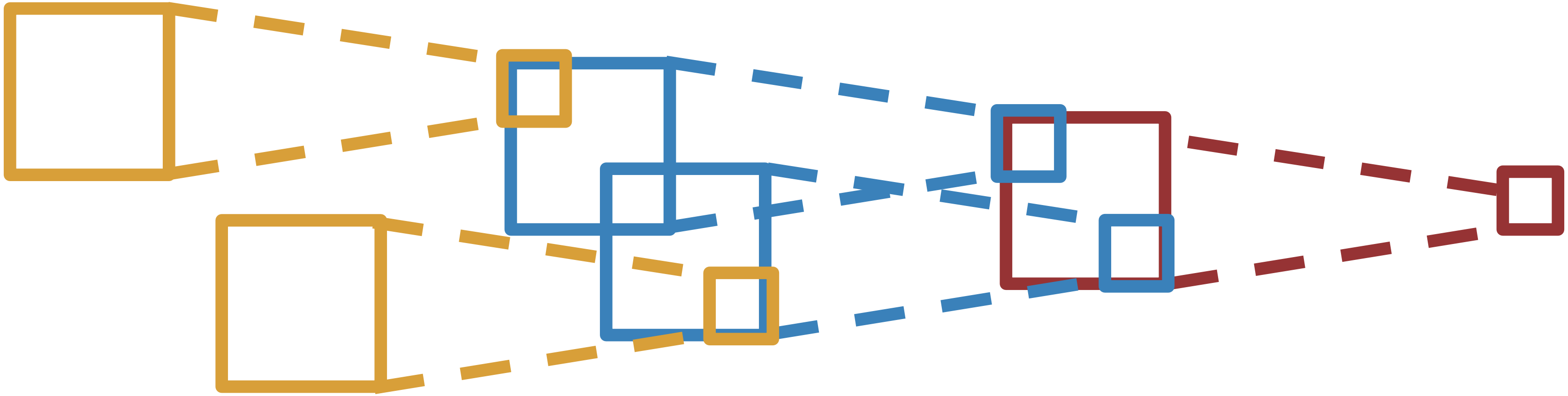
Input

Output

Slide inspiration: Justin Johnson

# Receptive Fields

Each successive convolution adds K – 1 to the receptive field size
With L layers the receptive field size is 1 + L * (K – 1)



Input

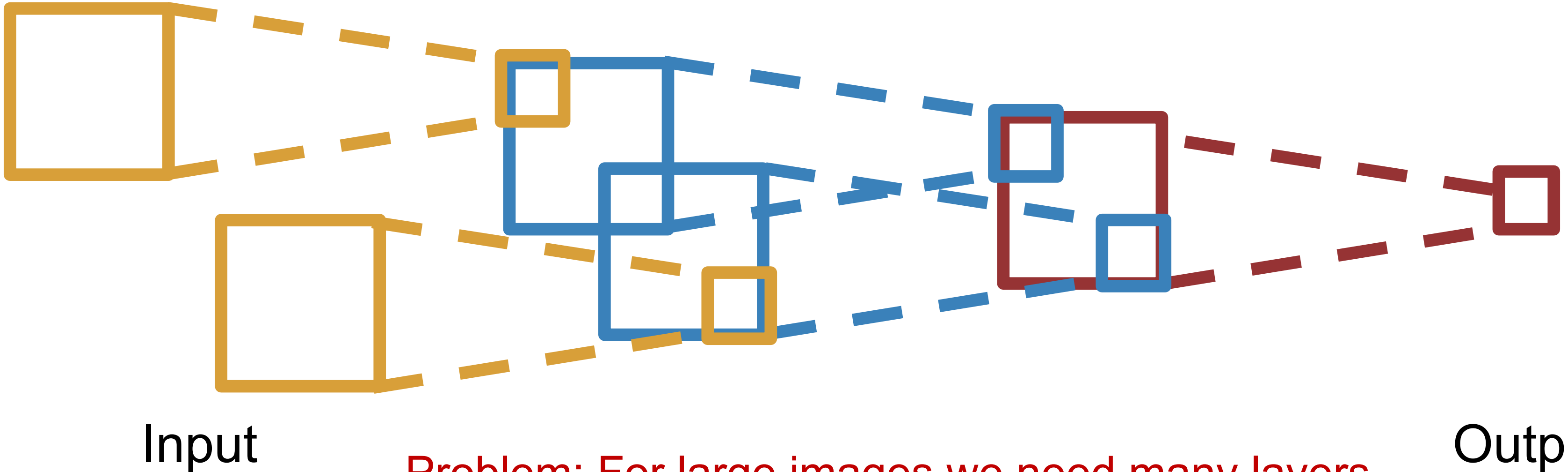
Output

Be careful – ”receptive field in the input” vs. “receptive field in the previous layer”

Slide inspiration: Justin Johnson

# Receptive Fields

Each successive convolution adds K – 1 to the receptive field size
With L layers the receptive field size is 1 + L * (K – 1)

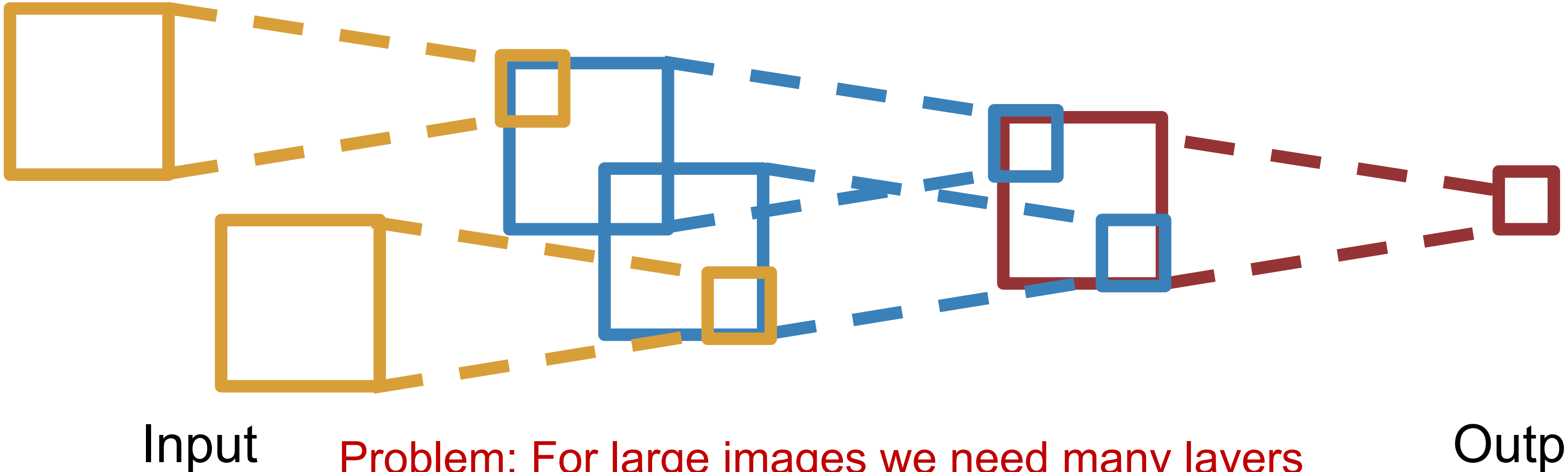



Problem: For large images we need many layers for each output to "see" the whole image image

Slide inspiration: Justin Johnson

# Receptive Fields

Each successive convolution adds K – 1 to the receptive field size
With L layers the receptive field size is 1 + L * (K – 1)

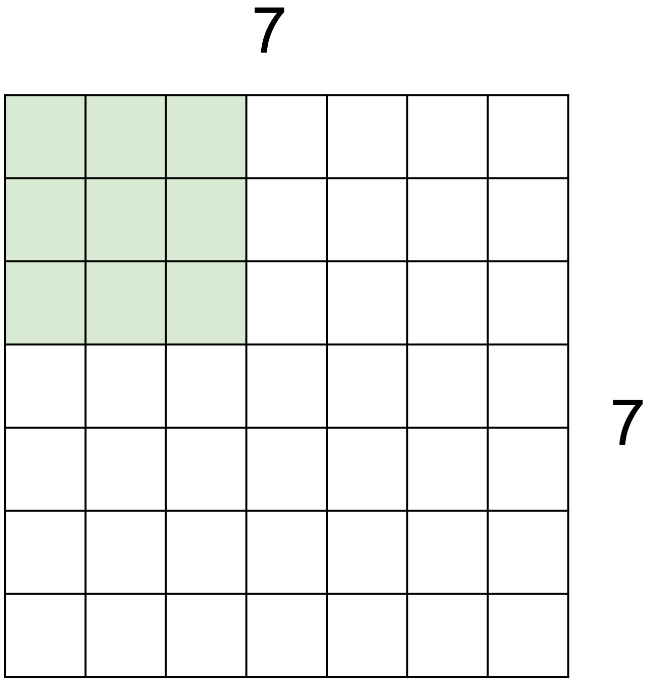



Problem: For large images we need many layers for each output to "see" the whole image image

Solution: Downsample inside the network

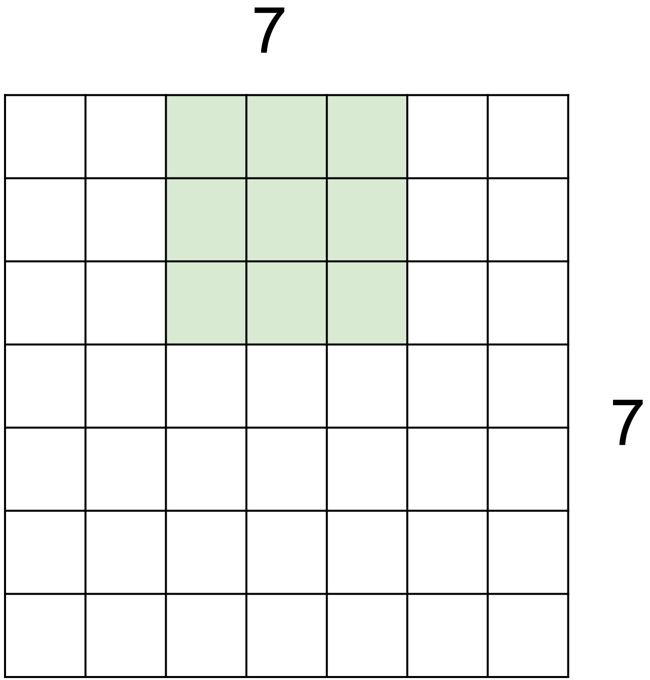
Slide inspiration: Justin Johnson

# Solution: Strided Convolution




7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**

# Solution: Strided Convolution




7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**

**=> 3x3 output!**

# Convolution layer: summary

Let's assume input is $W_1 \times H_1 \times C$
Conv layer needs 4 hyperparameters:

- Number of filters **K**
- The filter size **F**
- The stride **S**
- The zero padding **P**

This will produce an output of $W_2 \times H_2 \times K$
where:

- $W_2 = (W_1 - F + 2P)/S + 1$
- $H_2 = (H_1 - F + 2P)/S + 1$

Number of parameters: $F^2CK$ and K biases

# Convolution layer: summary

Let's assume input is $W_1 \times H_1 \times C$

Conv layer needs 4 hyperparameters:

- Number of filters **K**
- The filter size **F**
- The stride **S**
- The zero padding **P**

This will produce an output of $W_2 \times H_2 \times K$

where:

- $W_2 = (W_1 - F + 2P)/S + 1$
- $H_2 = (H_1 - F + 2P)/S + 1$

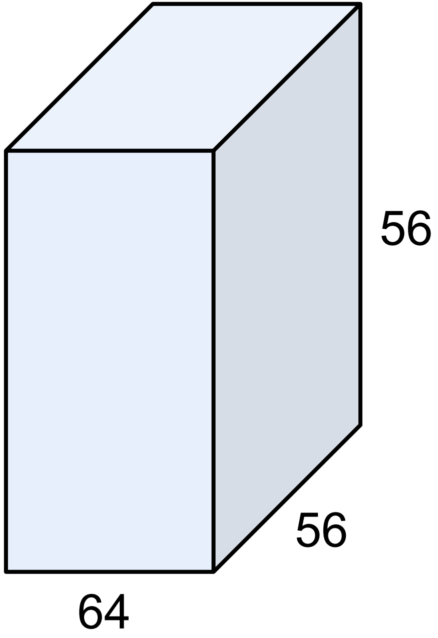
Number of parameters: $F^2CK$ and K biases

Common settings:

K = (powers of 2, e.g. 32, 64, 128, 512)

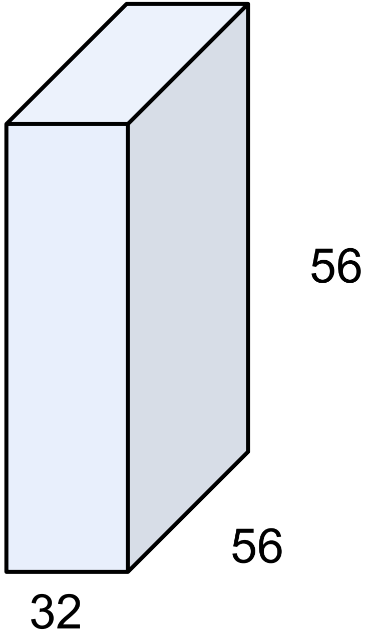
- F = 3, S = 1, P = 1
- F = 5, S = 1, P = 2
- F = 5, S = 2, P = ? (whatever fits)
- F = 1, S = 1, P = 0

# (btw, 1x1 convolution layers make perfect sense)

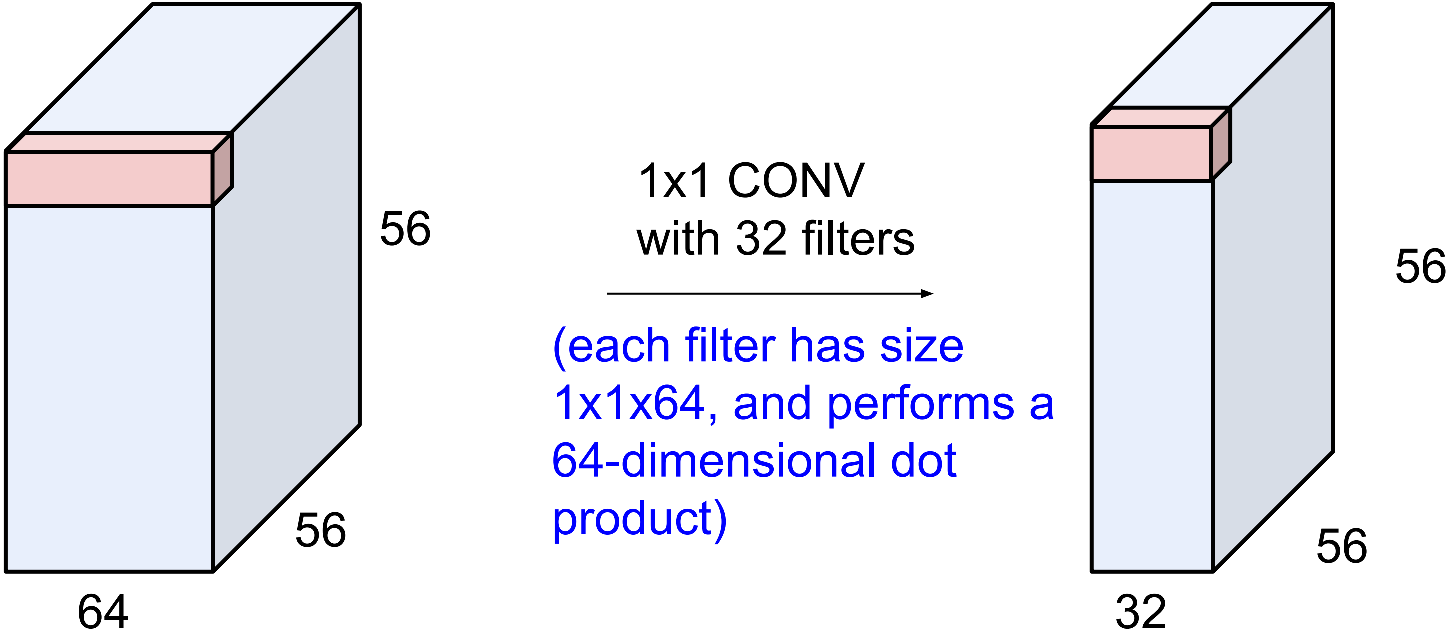



1x1 CONV
with 32 filters

(each filter has size 1x1x64, and performs a 64-dimensional dot product)



# (btw, 1x1 convolution layers make perfect sense)



# Example: CONV layer in PyTorch

Conv2d

CLASS `torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True)` [SOURCE]

Applies a 2D convolution over an input signal composed of several input planes.

In the simplest case, the output value of the layer with input size $(N, C_{\text{in}}, H, W)$ and output $(N, C_{\text{out}}, H_{\text{out}}, W_{\text{out}})$ can be precisely described as:

$$\text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) + \sum_{k=0}^{C_{\text{in}}-1} \text{weight}(C_{\text{out}_j}, k) \star \text{input}(N_i, k)$$

where $\star$ is the valid 2D cross-correlation operator, $N$ is a batch size, $C$ denotes a number of channels, $H$ is a height of input planes in pixels, and $W$ is width in pixels.

- `stride` controls the stride for the cross-correlation, a single number or a tuple.
- `padding` controls the amount of implicit zero-paddings on both sides for `padding` number of points for each dimension.
- `dilation` controls the spacing between the kernel points; also known as the à trous algorithm. It is harder to describe, but this link has a nice visualization of what `dilation` does.
- `groups` controls the connections between inputs and outputs. `in_channels` and `out_channels` must both be divisible by `groups`. For example,
    - At groups=1, all inputs are convolved to all outputs.
    - At groups=2, the operation becomes equivalent to having two conv layers side by side, each seeing half the input channels, and producing half the output channels, and both subsequently concatenated.
    - At groups= `in_channels`, each input channel is convolved with its own set of filters, of size: $\left\lfloor \frac{C_{\text{out}}}{C_{\text{in}}} \right\rfloor$.

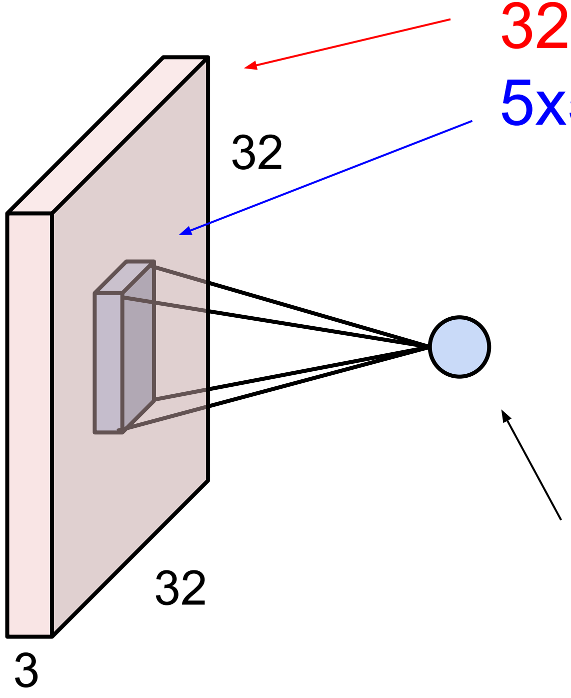
The parameters `kernel_size`, `stride`, `padding`, `dilation` can either be:

- a single `int` – in which case the same value is used for the height and width dimension
- a `tuple` of two ints – in which case, the first *int* is used for the height dimension, and the second *int* for the width dimension

Conv layer needs 4 hyperparameters:

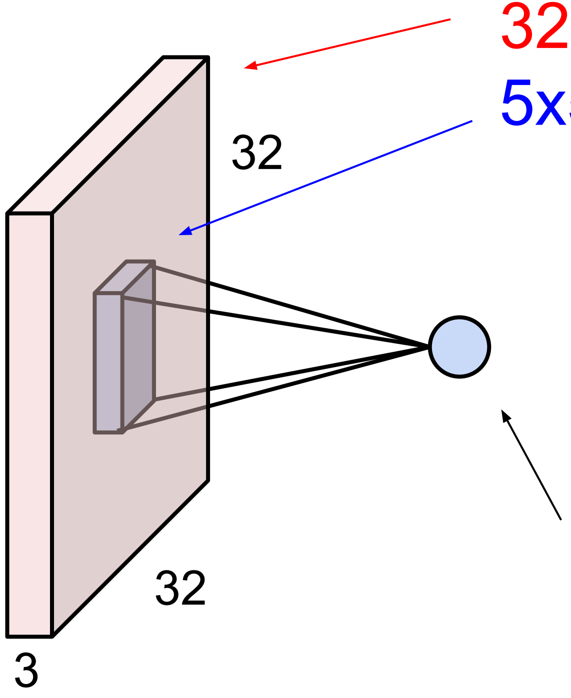
- Number of filters **K**
- The filter size **F**
- The stride **S**
- The zero padding **P**

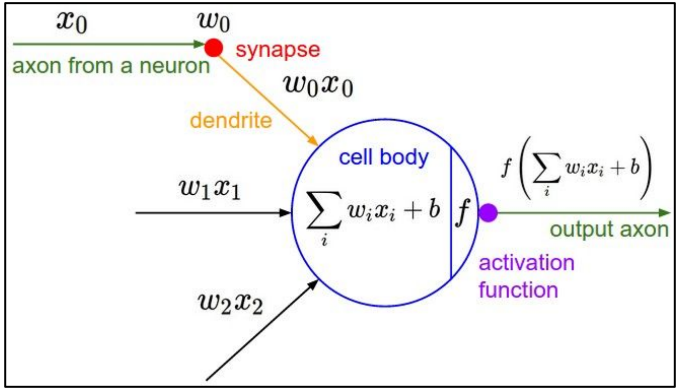
# The brain/neuron view of CONV Layer




**1 number:**
the result of taking a dot product between the filter and this part of the image
(i.e. 5*5*3 = 75-dimensional dot product)

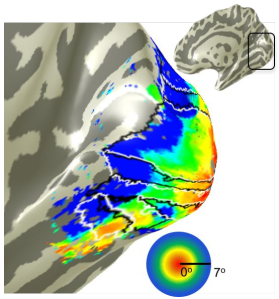
# The brain/neuron view of CONV Layer



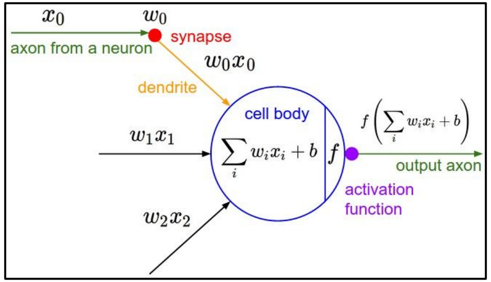



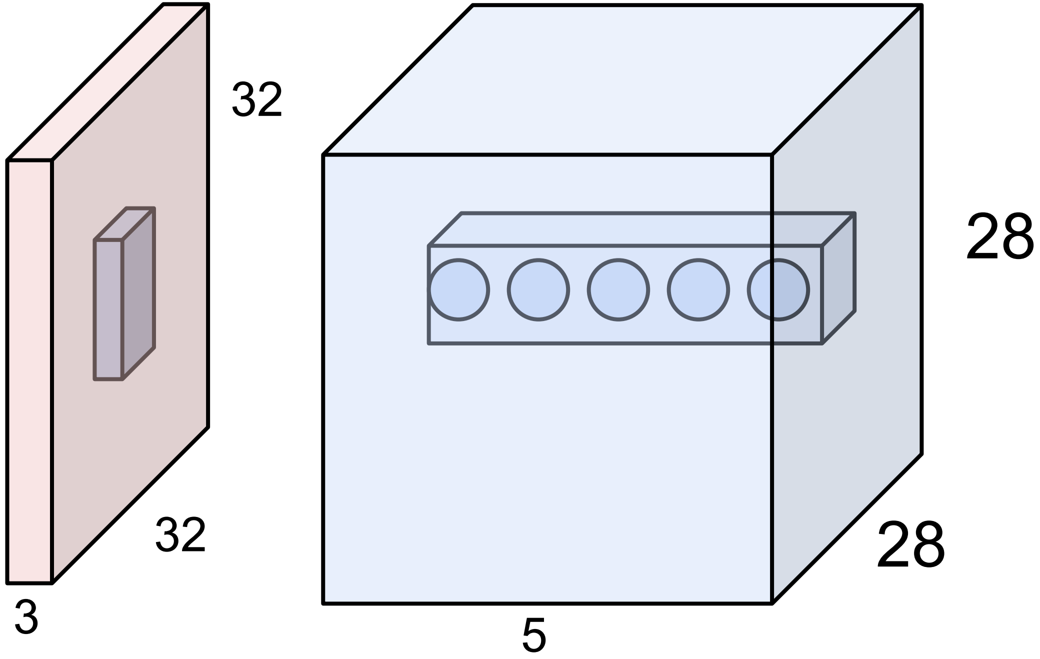

It's just a neuron with local connectivity...

**1 number:**
the result of taking a dot product between the filter and this part of the image
(i.e. 5*5*3 = 75-dimensional dot product)



# The brain/neuron view of CONV Layer



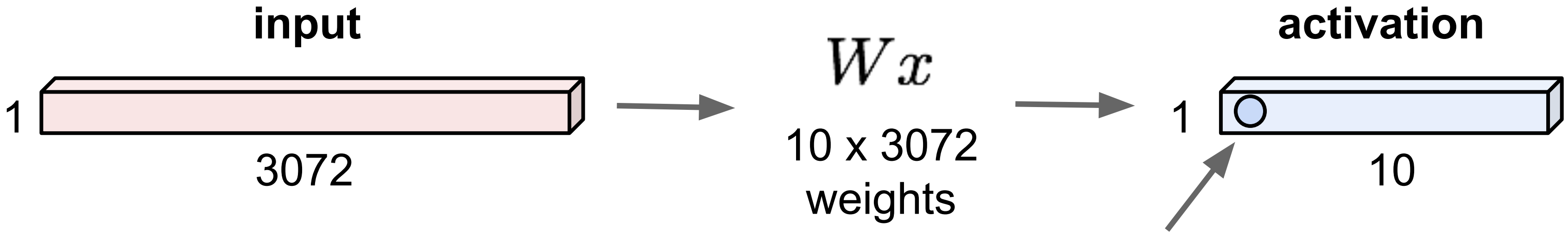

E.g. with 5 filters,
CONV layer consists of
neurons arranged in a 3D grid
(28x28x5)

There will be 5 different
neurons all looking at the same
region in the input volume

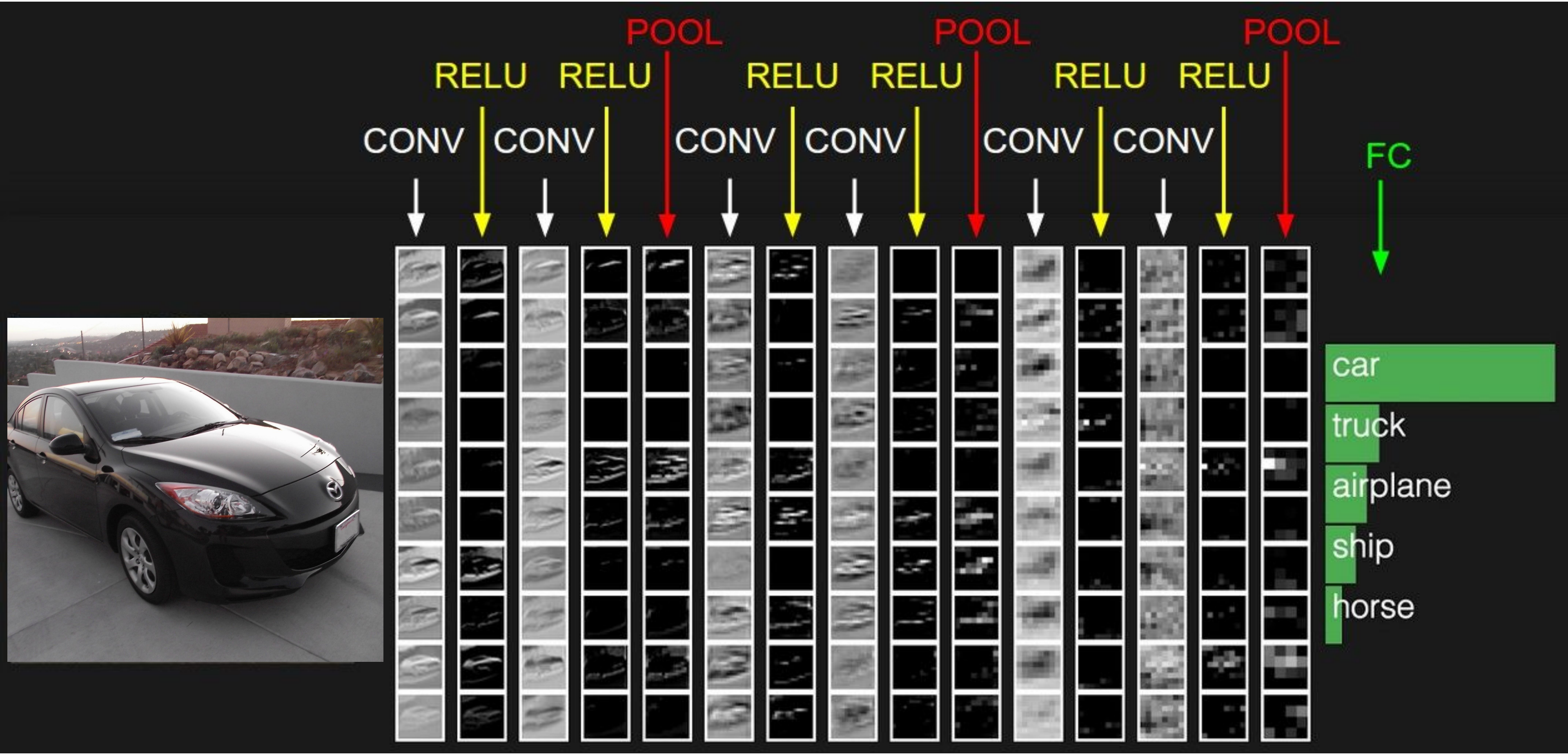
# Reminder: Fully Connected Layer

32x32x3 image -> stretch to 3072 x 1

Each neuron looks at the full input volume

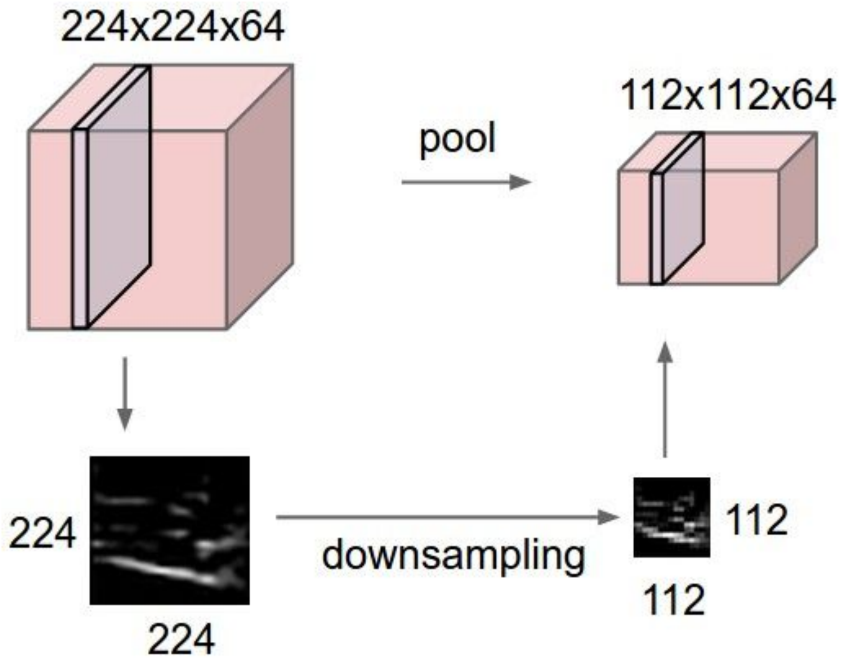



**1 number:**
the result of taking a dot product between a row of W and the input (a 3072-dimensional dot product)

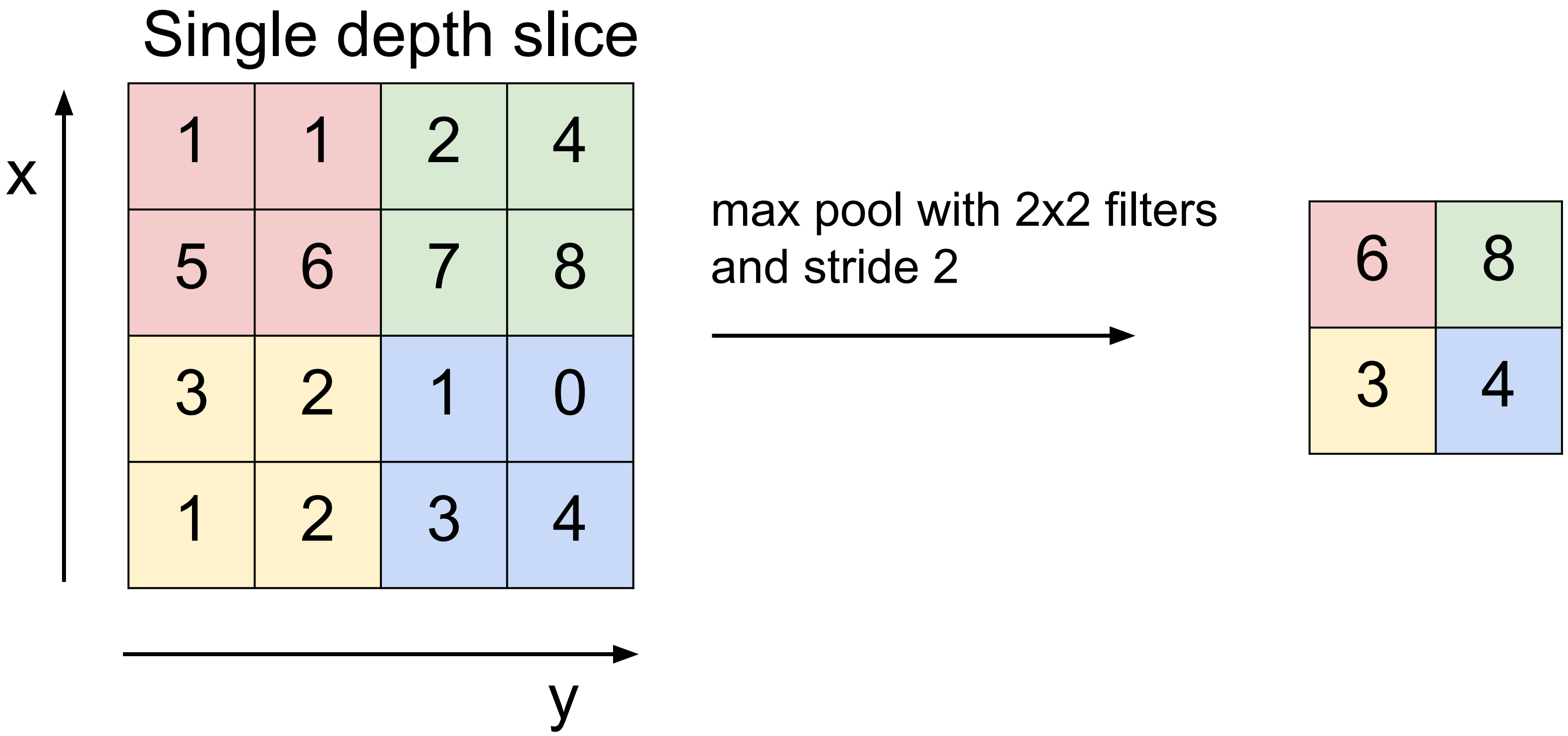

POOL
POOL
POOL
RELU
RELU
RELU
RELU
RELU
RELU
CONV
CONV
CONV
CONV
CONV
CONV
FC
car
truck
airplane
ship
horse

# Pooling layer

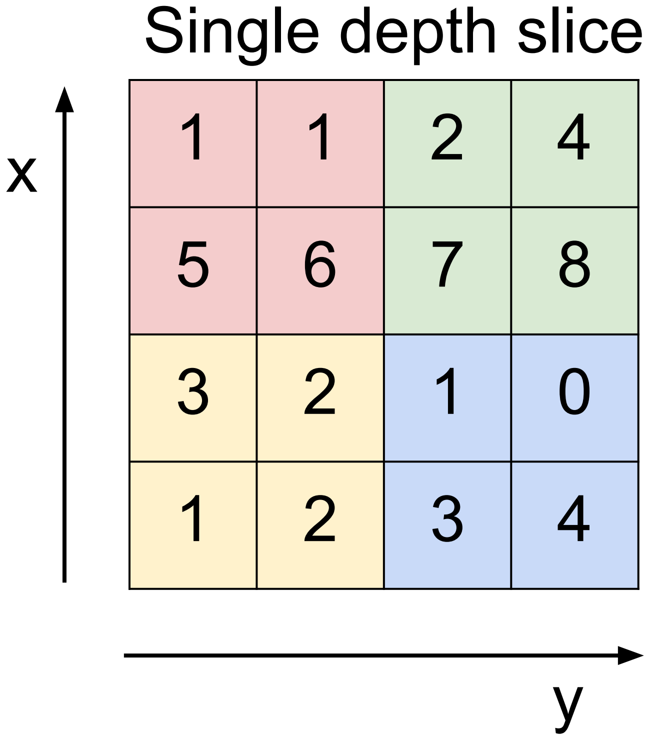
- makes the representations smaller and more manageable
- operates over each activation map independently

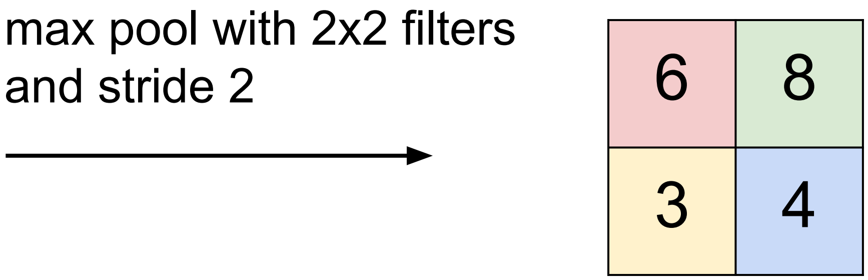


# MAX POOLING


Single depth slice
x
y
1 1 2 4
5 6 7 8
3 2 1 0
1 2 3 4
max pool with 2x2 filters
and stride 2
6 8
3 4

# MAX POOLING




- No learnable parameters
- Introduces spatial invariance

# Pooling layer: summary

Let's assume input is $W_1$ x $H_1$ x C
Conv layer needs 2 hyperparameters:

- The spatial extent **F**
- The stride **S**

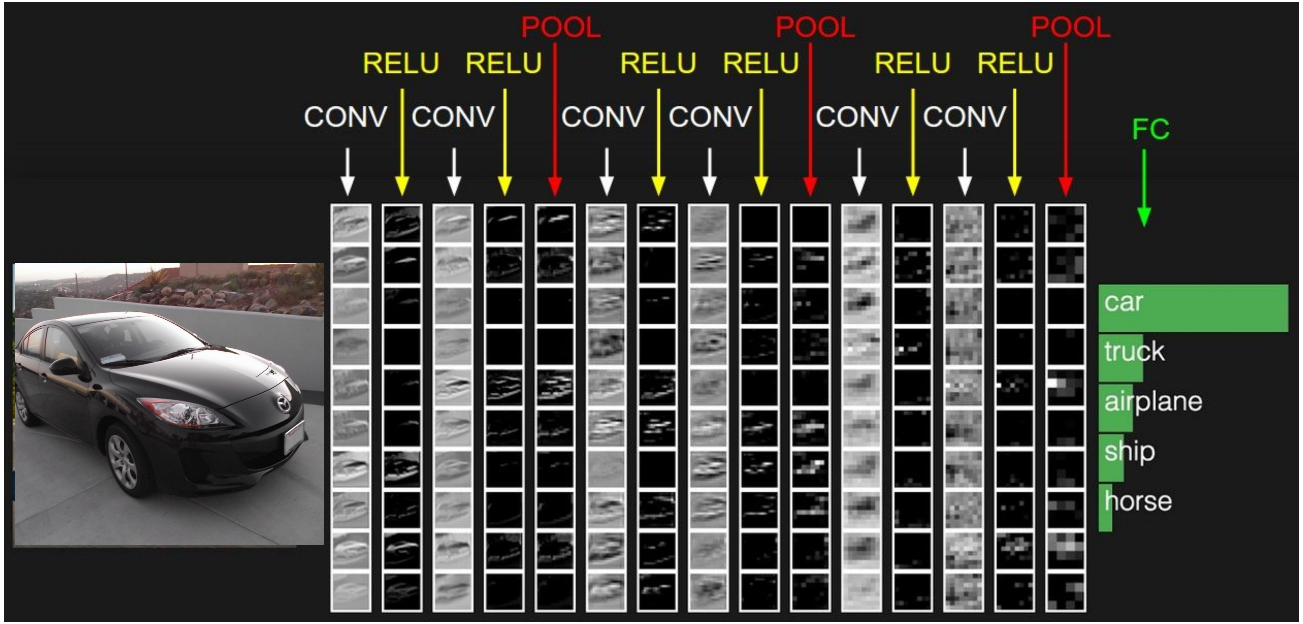
This will produce an output of $W_2$ x $H_2$ x C where:

- $W_2 = (W_1 - F)/S + 1$
- $H_2 = (H_1 - F)/S + 1$

Number of parameters: 0

# Fully Connected Layer (FC layer)

- Contains neurons that connect to the entire input volume, as in ordinary Neural Networks



# [ConvNetJS demo: training on CIFAR-10]

## ConvNetJS CIFAR-10 demo

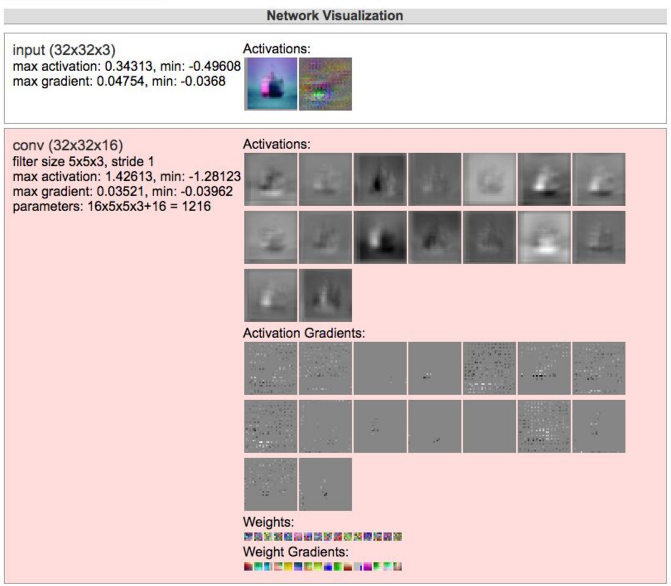
### Description

This demo trains a Convolutional Neural Network on the CIFAR-10 dataset in your browser, with nothing but Javascript. The state of the art on this dataset is about 90% accuracy and human performance is at about 94% (not perfect as the dataset can be a bit ambiguous). I used this python script to parse the original files (python version) into batches of images that can be easily loaded into page DOM with img tags.

This dataset is more difficult and it takes longer to train a network. Data augmentation includes random flipping and random image shifts by up to 2px horizontally and verically.

By default, in this demo we're using Adadelta which is one of per-parameter adaptive step size methods, so we don't have to worry about changing learning rates or momentum over time. However, I still included the text fields for changing these if you'd like to play around with SGD+Momentum trainer.

Report questions/bugs/suggestions to @karpathy.




http://cs.stanford.edu/people/karpathy/convnetjs/demo/cifar10.html

# Summary

- ConvNets stack CONV,POOL,FC layers
- Trend towards smaller filters and deeper architectures
- Trend towards getting rid of POOL/FC layers (just CONV)
- Historically architectures looked like

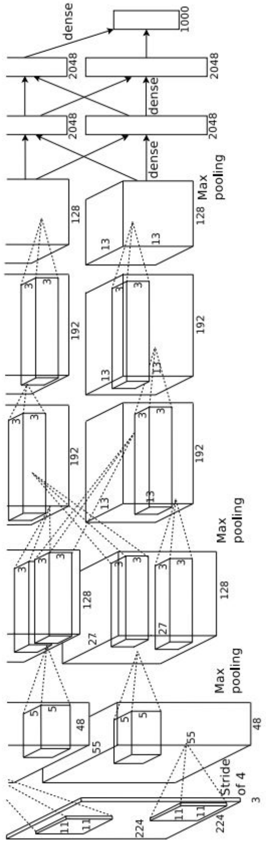
  **[(CONV-RELU)*N-POOL?]*M-(FC-RELU)*K,SOFTMAX**

  where N is usually up to ~5, M is large, 0 <= K <= 2.
- But recent advances such as ResNet/GoogLeNet have challenged this paradigm

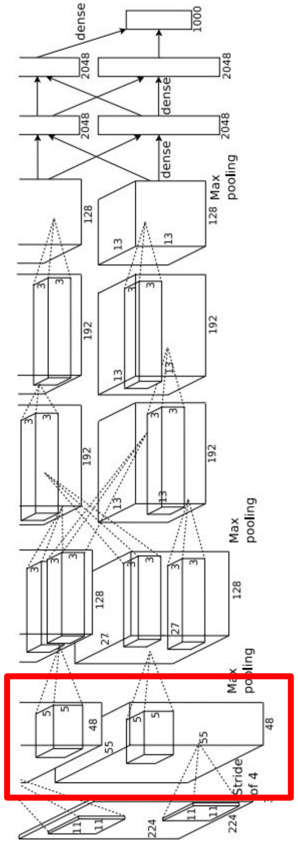
# Transfer learning

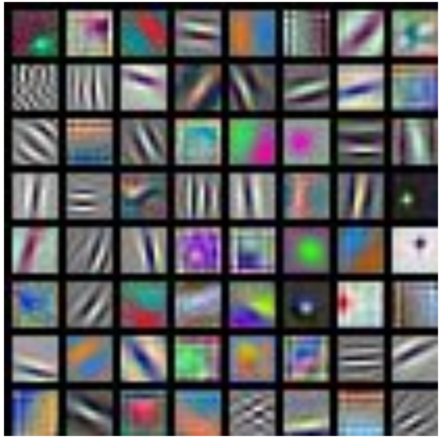
You need a lot of a data if you want to train/use CNNs?

# Transfer Learning with CNNs



# Transfer Learning with CNNs

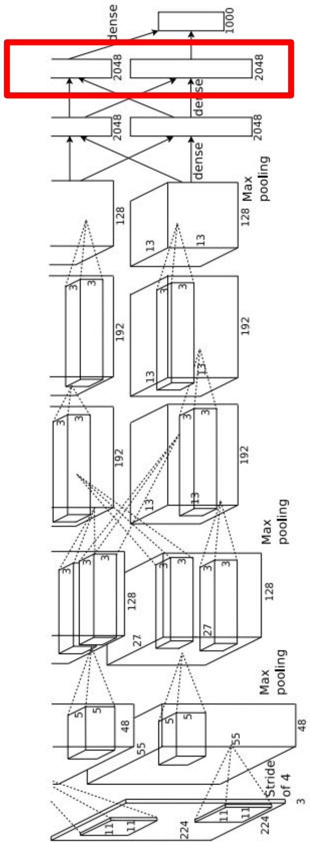


AlexNet:
64 x 3 x 11 x 11

(More on this in Lecture 13)

# Transfer Learning with CNNs



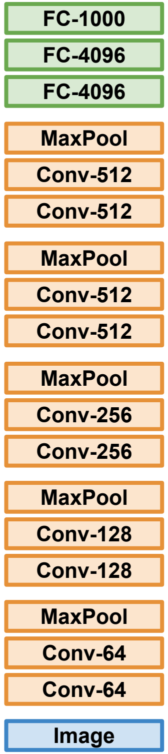
Test image    L2 Nearest neighbors in feature space

(More on this in Lecture 13)

# Transfer Learning with CNNs

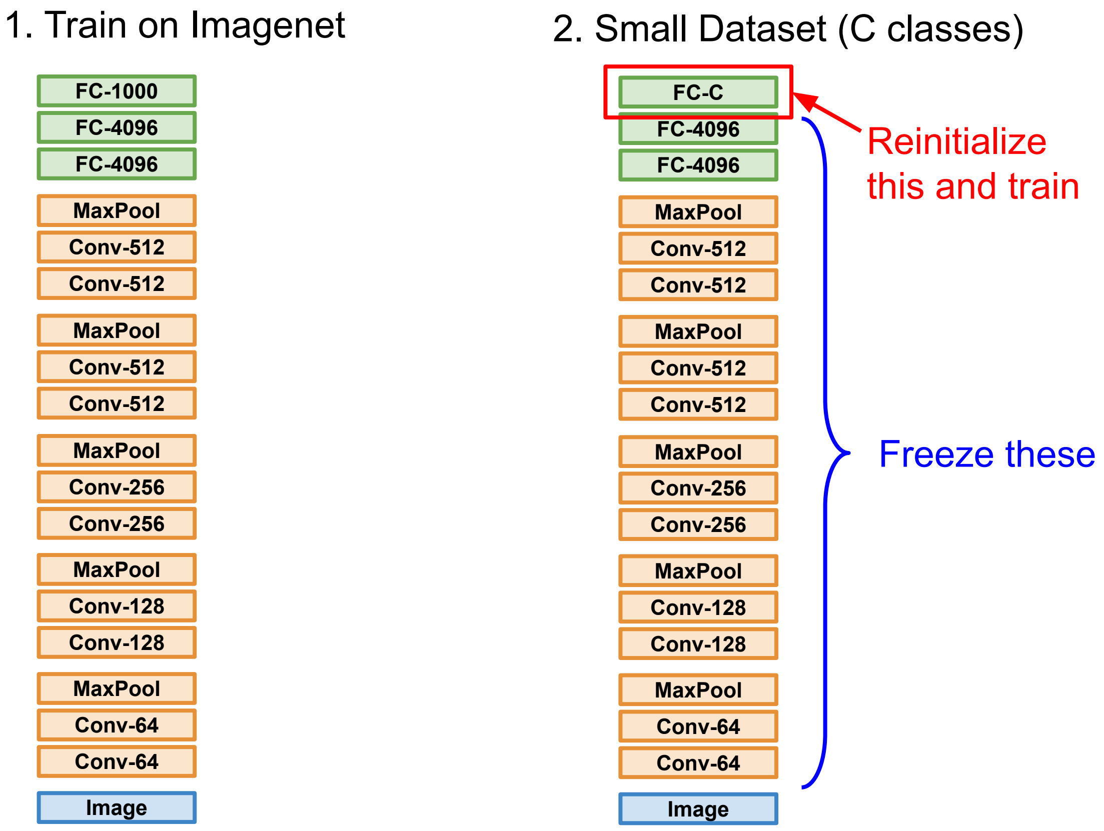
Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014
Razavian et al, "CNN Features Off-the-Shelf: An Astounding Baseline for Recognition", CVPR Workshops 2014

1. Train on Imagenet



# Transfer Learning with CNNs

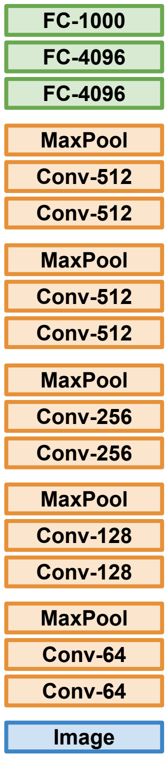
Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014
Razavian et al, "CNN Features Off-the-Shelf: An Astounding Baseline for Recognition", CVPR Workshops 2014



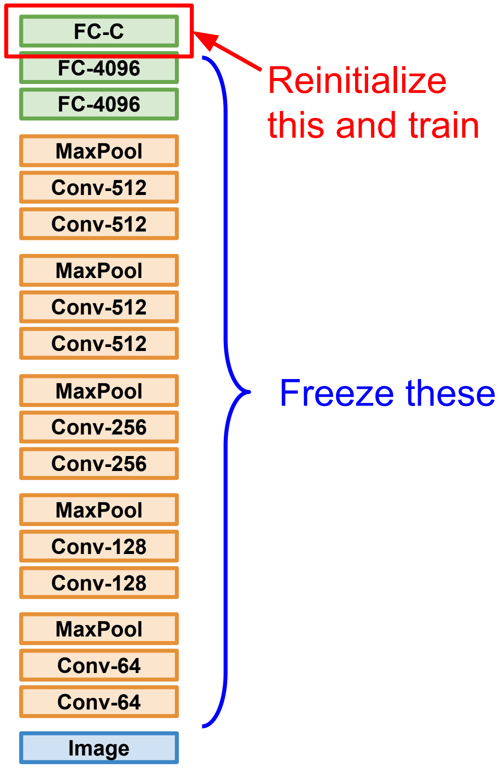
Donahue et al, “DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition”, ICML 2014
Razavian et al, “CNN Features Off-the-Shelf: An Astounding Baseline for Recognition”, CVPR Workshops 2014

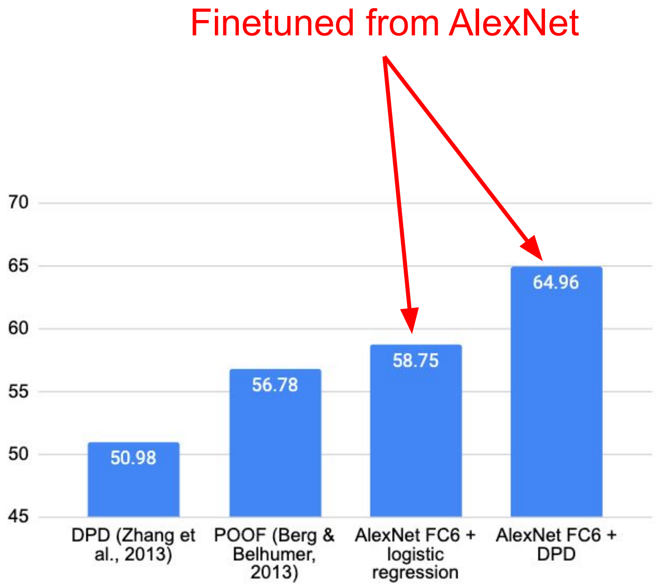
# Transfer Learning with CNNs

1. Train on Imagenet




2. Small Dataset (C classes)




Donahue et al, “DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition”, ICML 2014

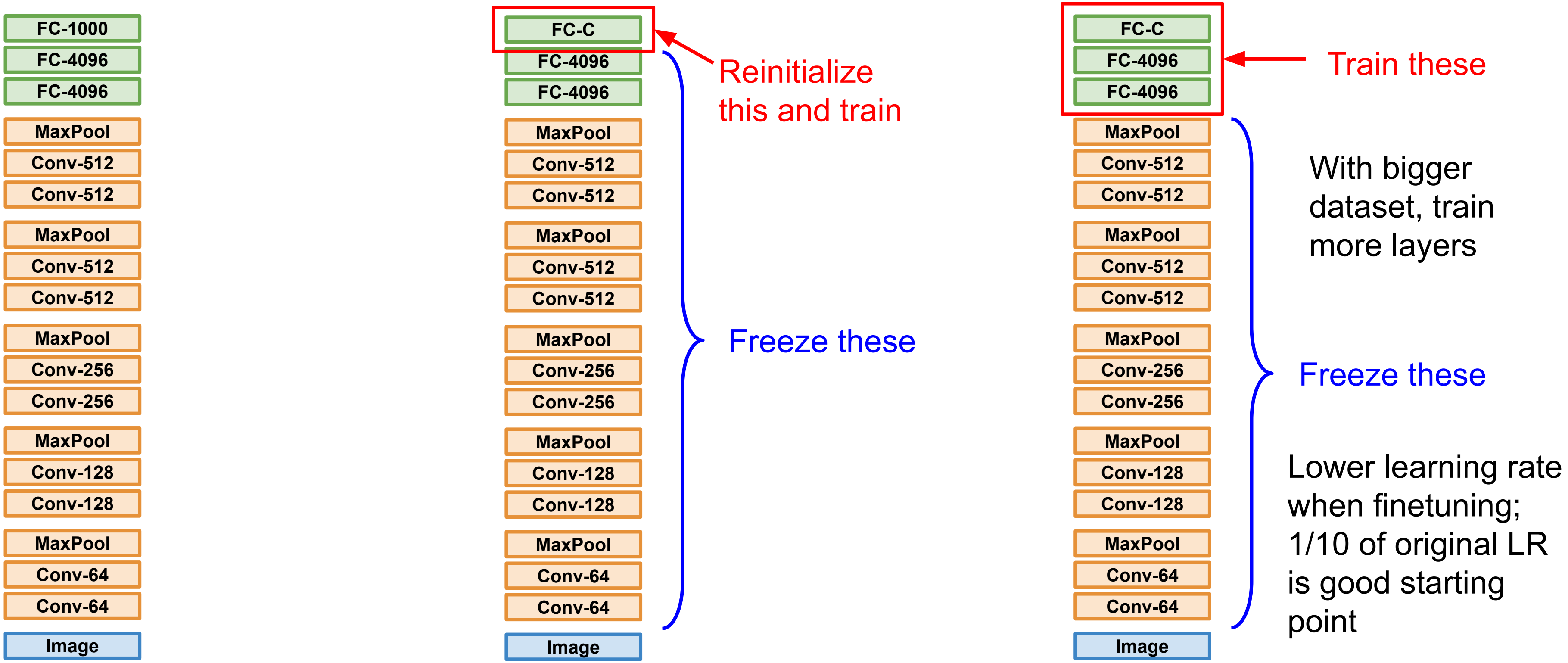
# Transfer Learning with CNNs

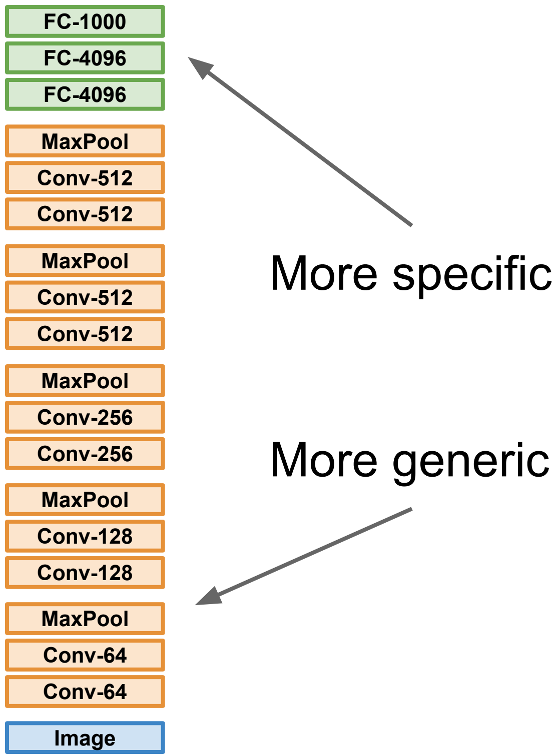
Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014
Razavian et al, "CNN Features Off-the-Shelf: An Astounding Baseline for Recognition", CVPR Workshops 2014

1. Train on Imagenet

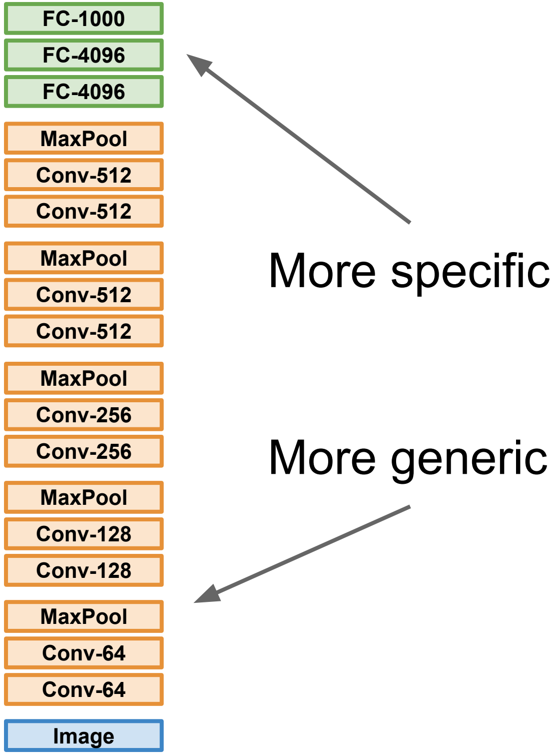
2. Small Dataset (C classes)

3. Bigger dataset

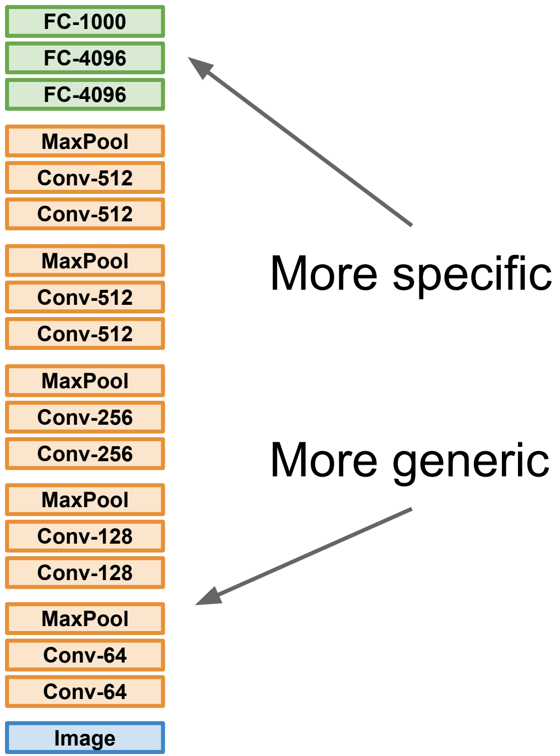





| | very similar dataset | very different dataset |
|---|---|---|
| **very little data** | ? | ? |
| **quite a lot of data** | ? | ? |




| | **very similar dataset** | **very different dataset** |
|---|---|---|
| **very little data** | Use Linear Classifier on top layer | ? |
| **quite a lot of data** | Finetune a few layers | ? |




| | **very similar dataset** | **very different dataset** |
|---|---|---|
| **very little data** | Use Linear Classifier on top layer | You're in trouble… Try linear classifier from different stages |
| **quite a lot of data** | Finetune a few layers | Finetune a larger number of layers |

# Transfer learning with CNNs is pervasive…
## (it's the norm, not an exception)

Object Detection
(Fast R-CNN)

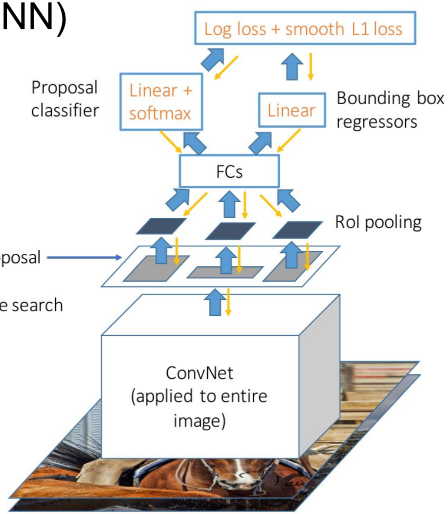
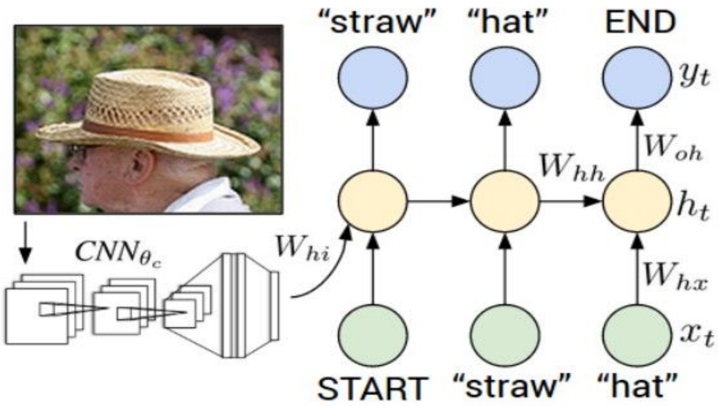



Image Captioning: CNN + RNN

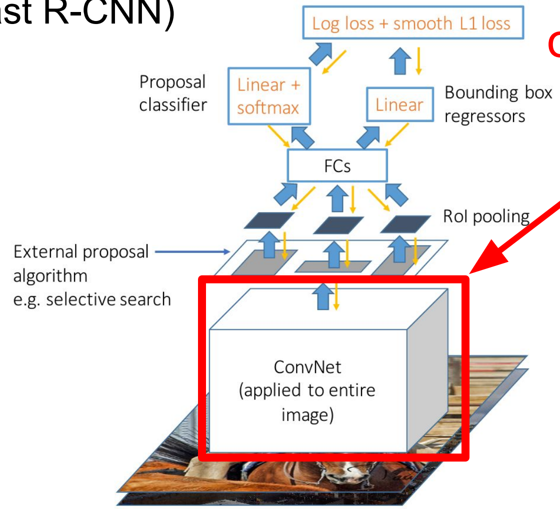



Girshick, "Fast R-CNN", ICCV 2015
Figure copyright Ross Girshick, 2015. Reproduced with permission.

Karpathy and Fei-Fei, "Deep Visual-Semantic Alignments for Generating Image Descriptions", CVPR 2015
Figure copyright IEEE, 2015. Reproduced for educational purposes.

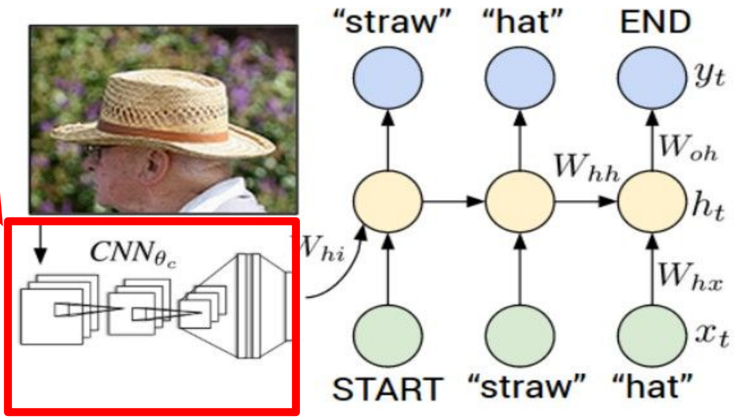
# Transfer learning with CNNs is pervasive…
## (it's the norm, not an exception)

Object Detection
(Fast R-CNN)




CNN pretrained on ImageNet

Image Captioning: CNN + RNN

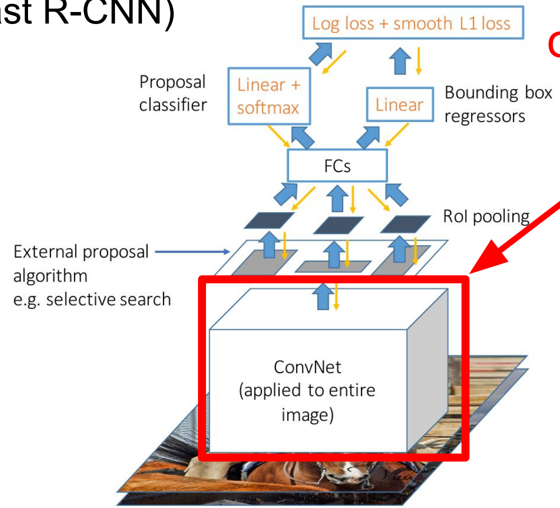



Girshick, "Fast R-CNN", ICCV 2015


Karpathy and Fei-Fei, "Deep Visual-Semantic Alignments for Generating Image Descriptions", CVPR 2015

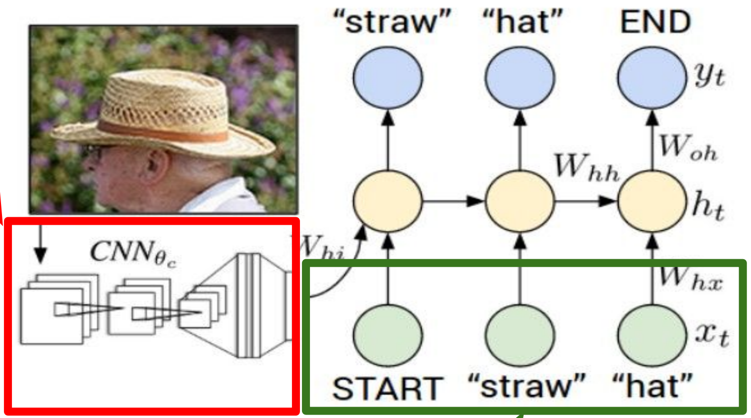
# Transfer learning with CNNs is pervasive…
## (it’s the norm, not an exception)

Object Detection
(Fast R-CNN)




CNN pretrained on ImageNet

Image Captioning: CNN + RNN

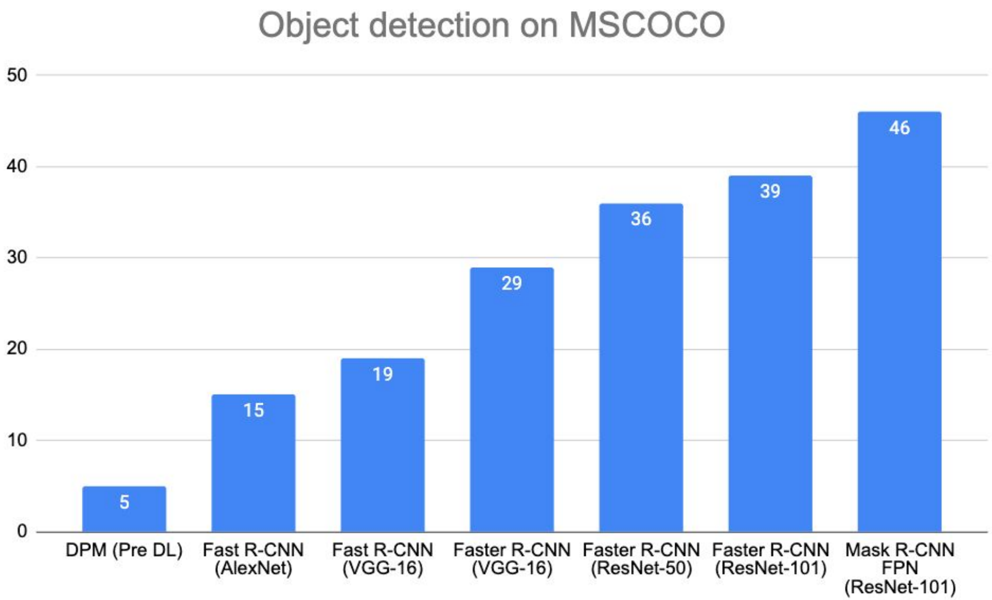



Word vectors pretrained with word2vec

Girshick, “Fast R-CNN”, ICCV 2015


Karpathy and Fei-Fei, “Deep Visual-Semantic Alignments for Generating Image Descriptions”, CVPR 2015

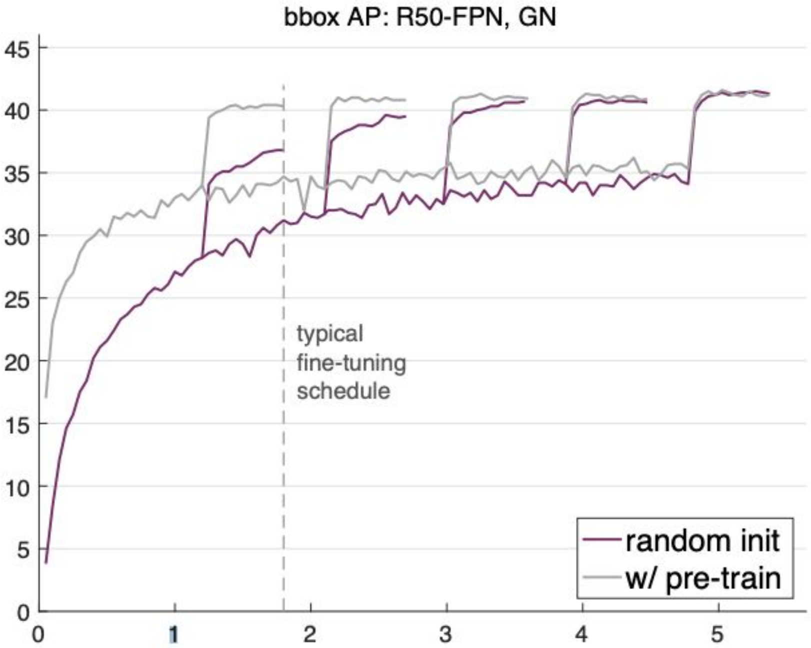
# Transfer learning with CNNs - Architecture matters




Girshick, “The Generalized R-CNN Framework for Object Detection”, ICCV 2017 Tutorial on Instance-Level Visual Recognition

# Transfer learning with CNNs is pervasive…

## But recent results show it might not always be necessary!




He et al, "Rethinking ImageNet Pre-training", ICCV 2019


Training from scratch can work just as well as training from a pretrained ImageNet model for object detection

But it takes 2-3x as long to train.

They also find that collecting more data is better than finetuning on a related task

# Takeaway for your projects and beyond:

Have some dataset of interest but it has < ~1M images?

1. Find a very large dataset that has similar data, train a big ConvNet there
2. Transfer learn to your dataset

Deep learning frameworks provide a "Model Zoo" of pretrained models so you don't need to train your own

TensorFlow: https://github.com/tensorflow/models
PyTorch: https://github.com/pytorch/vision

# Deep learning packages

- **TensorFlow** – Google
- **PyTorch** – Facebook AI research
- **Keras** – Francois Chollet (now at Google)
- **Chainer** – Company in Japan
- **Caffe** - Berkeley Vision and Learning Center
- **CNTK** - Microsoft

# Overview of the tutorial

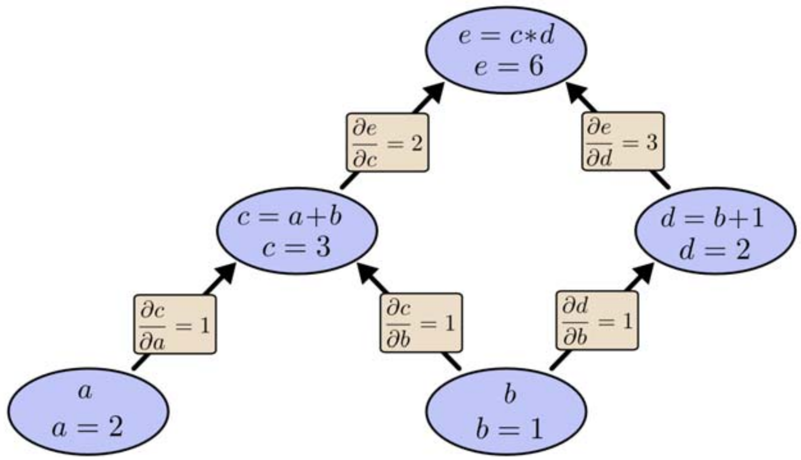
- What is Keras ?
- Basics of Keras environment
- Building Convolutional neural networks
- Building Recurrent neural networks
- Introduction to other types of layers
- Introduction to Loss functions and Optimizers in Keras
- Using Pre-trained models in Keras
- Saving and loading weights and models
- Popular architectures in Deep Learning

# What is Keras ?

- **Deep neural network library in Python**
  - High-level neural networks API
  - Modular – Building model is just stacking layers and connecting computational graphs
  - Runs on top of either TensorFlow or Theano or CNTK
- **Why use Keras ?**
  - Useful for fast prototyping, ignoring the details of implementing backprop or writing optimization procedure
  - Supports Convolution, Recurrent layer and combination of both.
  - Runs seamlessly on CPU and GPU
  - Almost any architecture can be designed using this framework
  - Open Source code – Large community support

# Working principle - Backend

- **Computational Graphs**
  - Expressing complex expressions as a combination of simple operations
  - Useful for calculating derivatives during backpropagation
  - Easier to implement distributed computation
  - Just specify the inputs, outputs and make sure the graph is connected




*e = c*d*
where, *"c = a+b"* and *"d = b+1"*
So, *e = (a+b)*(b+1)*
Here *"a"*, *"b"* are inputs

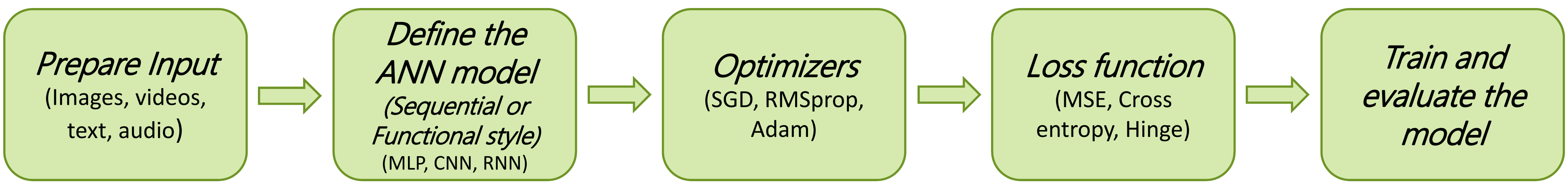
# General pipeline for implementing an ANN

- Design and define the neural network architecture
- Select the optimizer that performs optimization (gradient descent)
- Select the loss function and train it
- Select the appropriate evaluation metric for the given problem

# Implementing a neural network in Keras

- **Five major steps**
  - Preparing the input and specify the input dimension (size)
  - Define the model architecture and build the computational graph
  - Specify the optimizer and configure the learning process
  - Specify the Inputs, Outputs of the computational graph (model) and the Loss function
  - Train and test the model on the dataset

**Note:** Gradient calculations are taken care by Auto – Differentiation and parameter updates are done automatically in the backend



# Procedure to implement an ANN in Keras

- Importing *Sequential class* from keras.models

```
from keras.models import Sequential

model = Sequential()
```

- Stacking layers using .add() method

```
model.add(Dense(units=64, input_dim=100))
model.add(Activation('relu'))
model.add(Dense(units=10))
model.add(Activation('softmax'))
```

- Configure learning process using .compile() method

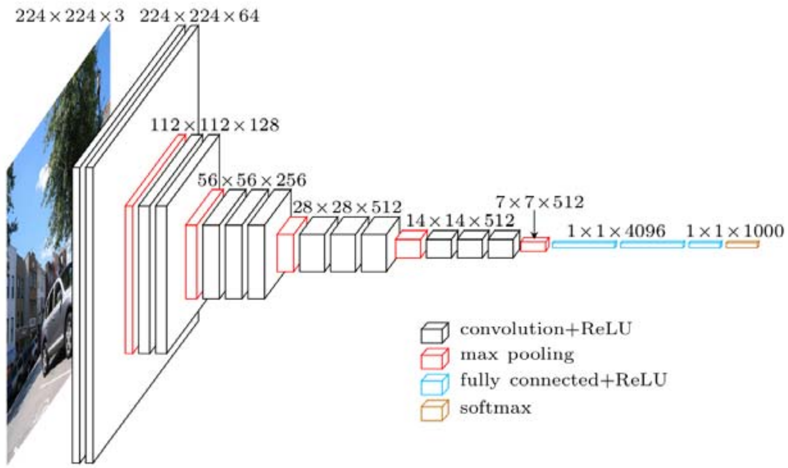
```
model.compile(loss='categorical_crossentropy',
              optimizer='sgd',
              metrics=['accuracy'])
```

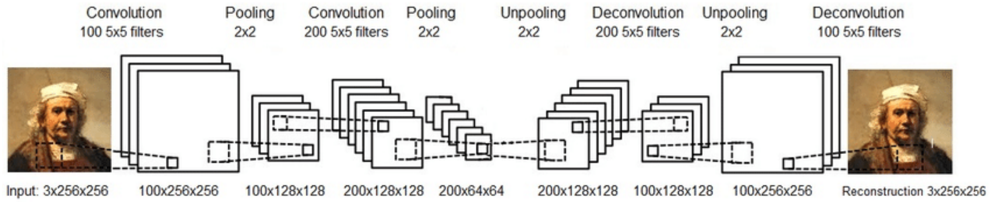
- Train the model on train dataset using .fit() method

```
model.fit(x_train, y_train, epochs=5, batch_size=32)
```

# Keras models – Sequential

- Sequential model
- Linear stack of layers
- Useful for building simple models
  - Simple classification network
  - Encoder – Decoder models

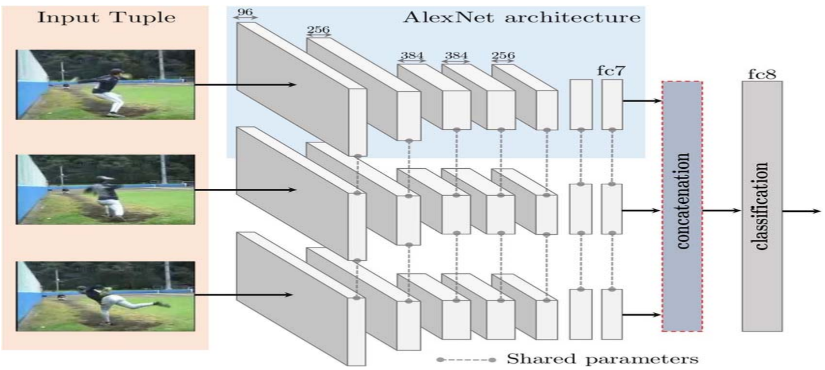



[1] https://blog.heuritech.com/2016/02/29/a-brief-report-of-the-heuritech-deep-learning-meetup-5/vgg16/
[2] https://www.cc.gatech.edu/~hays/7476/projects/Avery_Wenchen/

# Keras models – Functional

- Functional Model
  - Multi – input and Multi – output models
  - Complex models which forks into 2 or more branches
  - Models with shared (Weights) layers





[1] https://www.sciencedirect.com/science/article/pii/S0263224117304517
[2] Unsupervised Domain Adaptation by Backpropagation, https://arxiv.org/abs/1409.7495