Lecture 10: Ingredients for a convolutional neural network – Part II

Machine Learning and Imaging

BME 548L
Roarke Horstmeyer

Note: Much material borrowed from Stanford CS231n, Lectures 4 - 10

Important components of a CNN

deep imaging

CNN Architecture

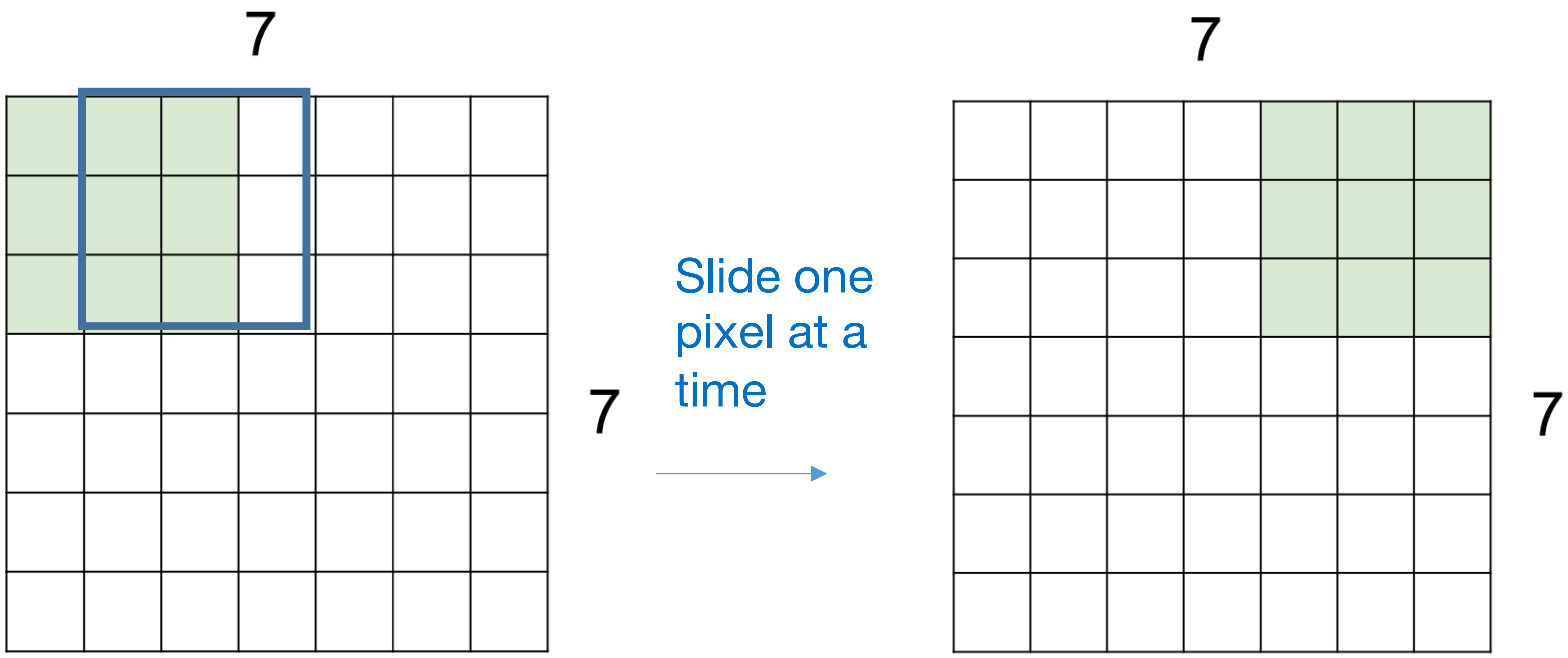
- CONV size, stride, pad, depth
- ReLU & other nonlinearities
- POOL methods
- Fully connected layers
- # of layers, dimensions per layer

Loss function & optimization

- Type of loss function
- Regularization
- Gradient descent method
- Gradient descent step size

Other specifics: Pre-processing, initialization, dropout, batch normalization, batch size

Convolutions: size, stride and padding

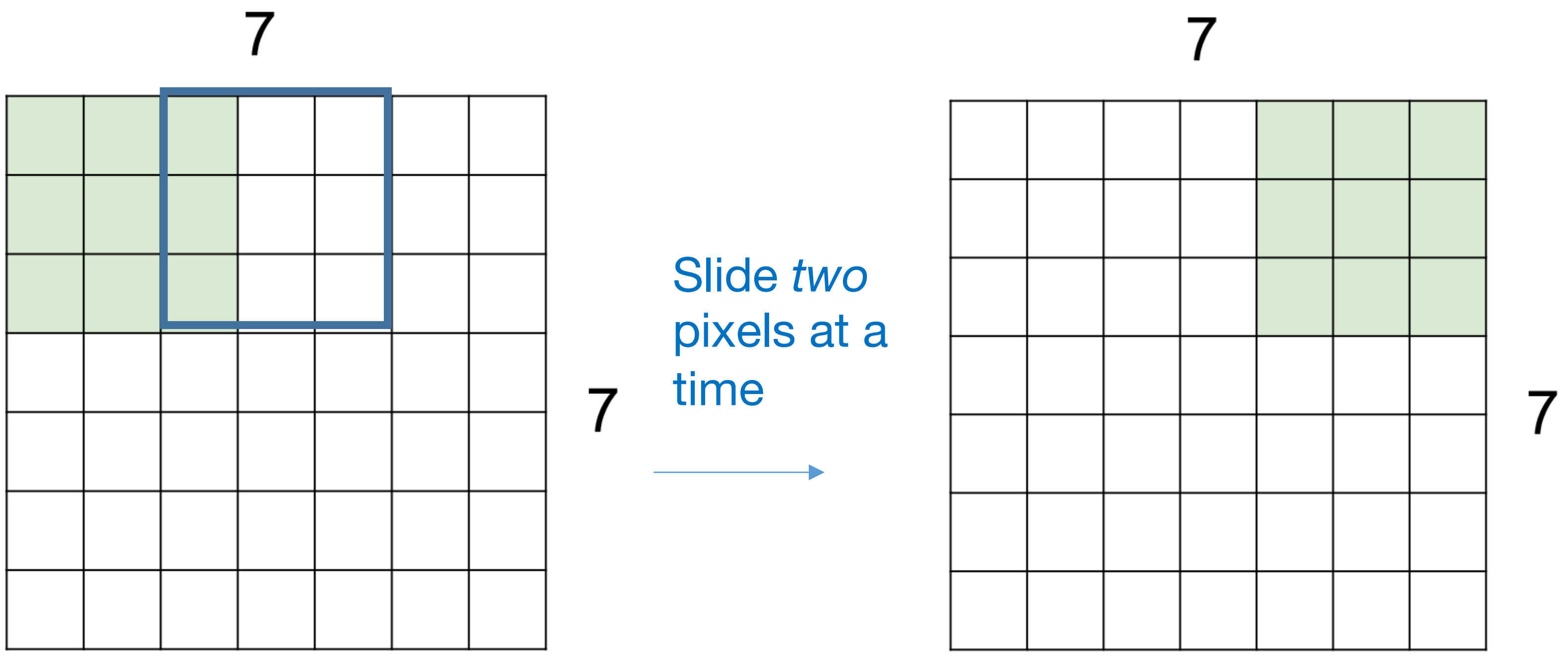


- 7x7 input image
- 3x3 filter

↓

- 5x5 output

Convolutions: size, stride and padding

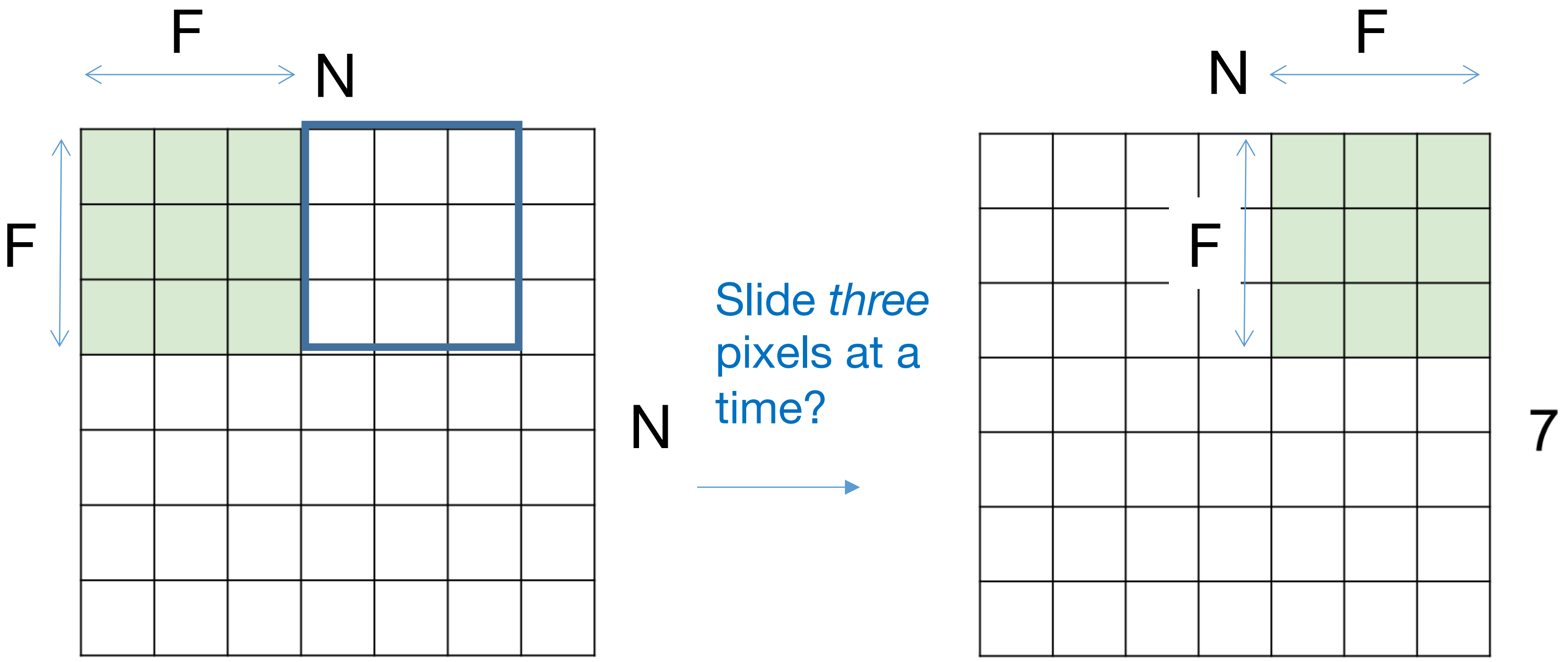


- 7x7 input image
- 3x3 filter

- 3x3 output!

This is called a “stride 2” convolution

Convolutions: size, stride and padding



Output matrix width W:

W = (N-F)/stride + 1

Example right: N=7, F=3

When stride = 1: W = 5

When stride = 2: W = 3

When stride = 3: W = 2.33???

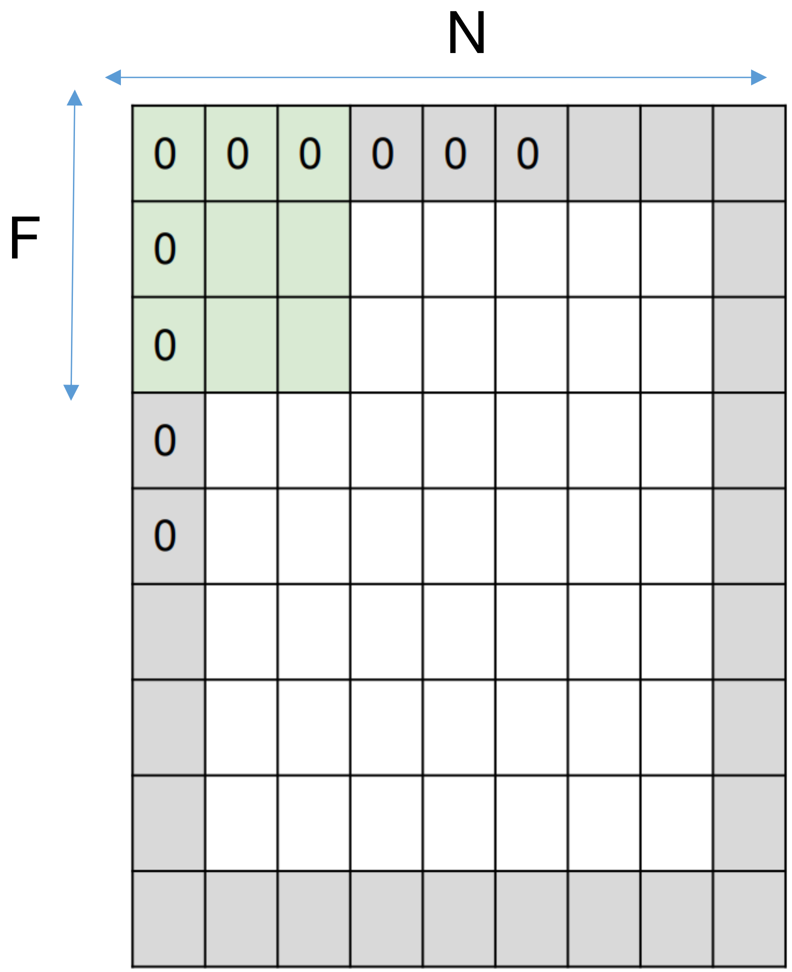
This is called a "stride 3" convolution

*Need to ensure integers work out!

Convolutions: size, stride and padding

Q: What if you really, really want to use a stride = 3 with N = 7 and F=3?

Convolutions: size, stride and padding



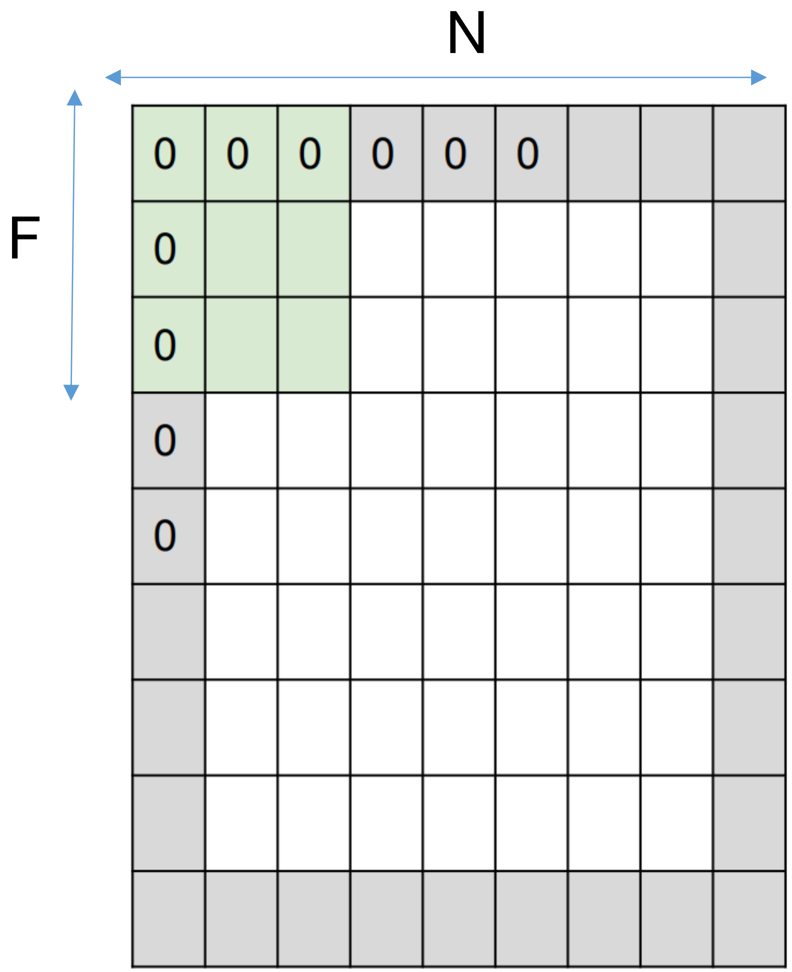
Q: What if you really, really want to use a stride = 3 with N = 7 and F=3?

A: Use *padding*

E.g., padding with 1 pixel around boarder makes N=9

Padding: add zeros around edge of image

Convolutions: size, stride and padding



Q: What if you really, really want to use a stride = 3 with N = 7 and F=3?

A: Use *padding*

E.g., padding with 1 pixel around boarder makes N=9

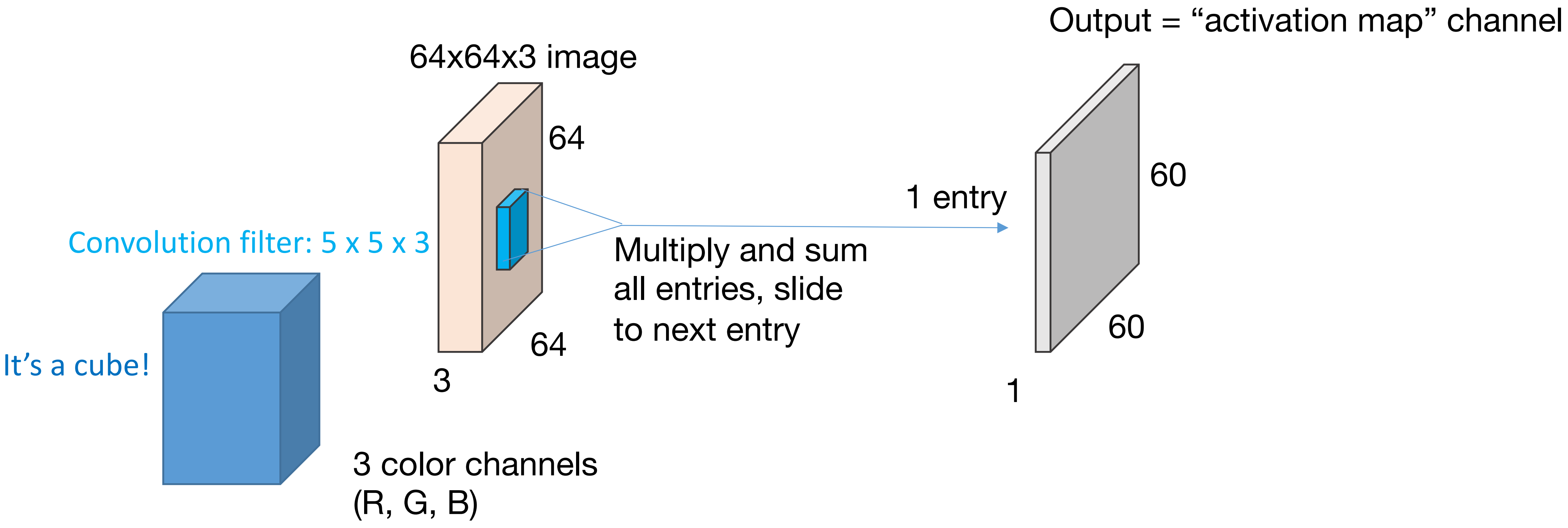
W = (N-F)/stride + 1

W = (9-3)/3 + 1 = 4

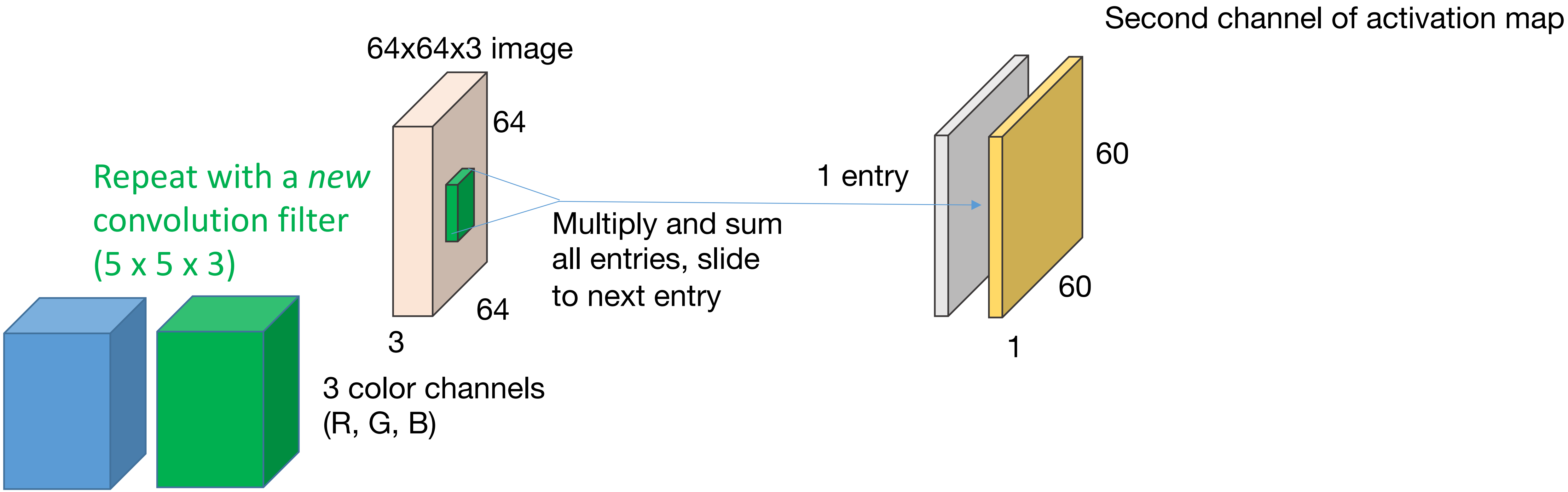
*Padding enables integer output!

Padding: add zeros around edge of image

Convolution layer: learn multiple filters

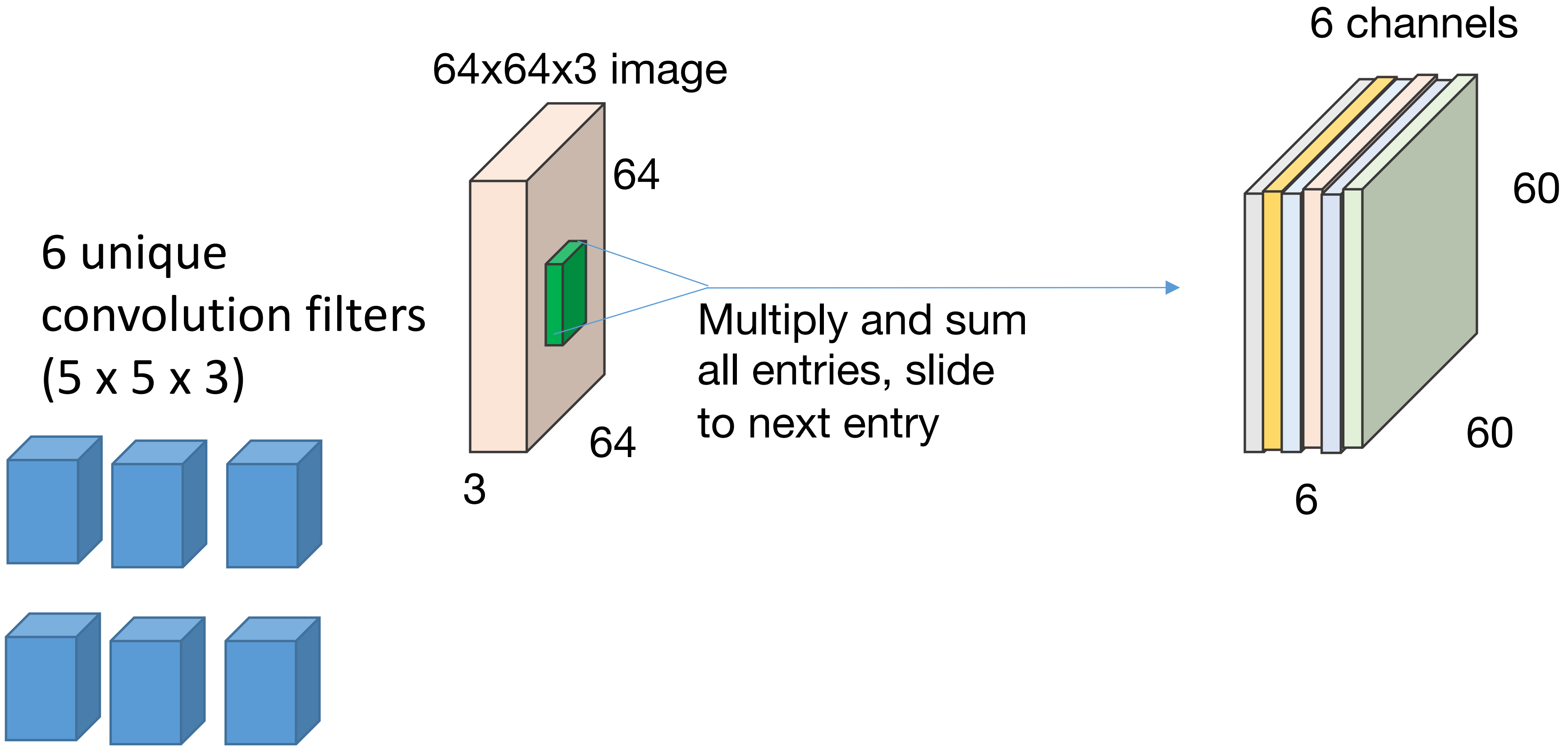


Convolution layer: learn multiple filters

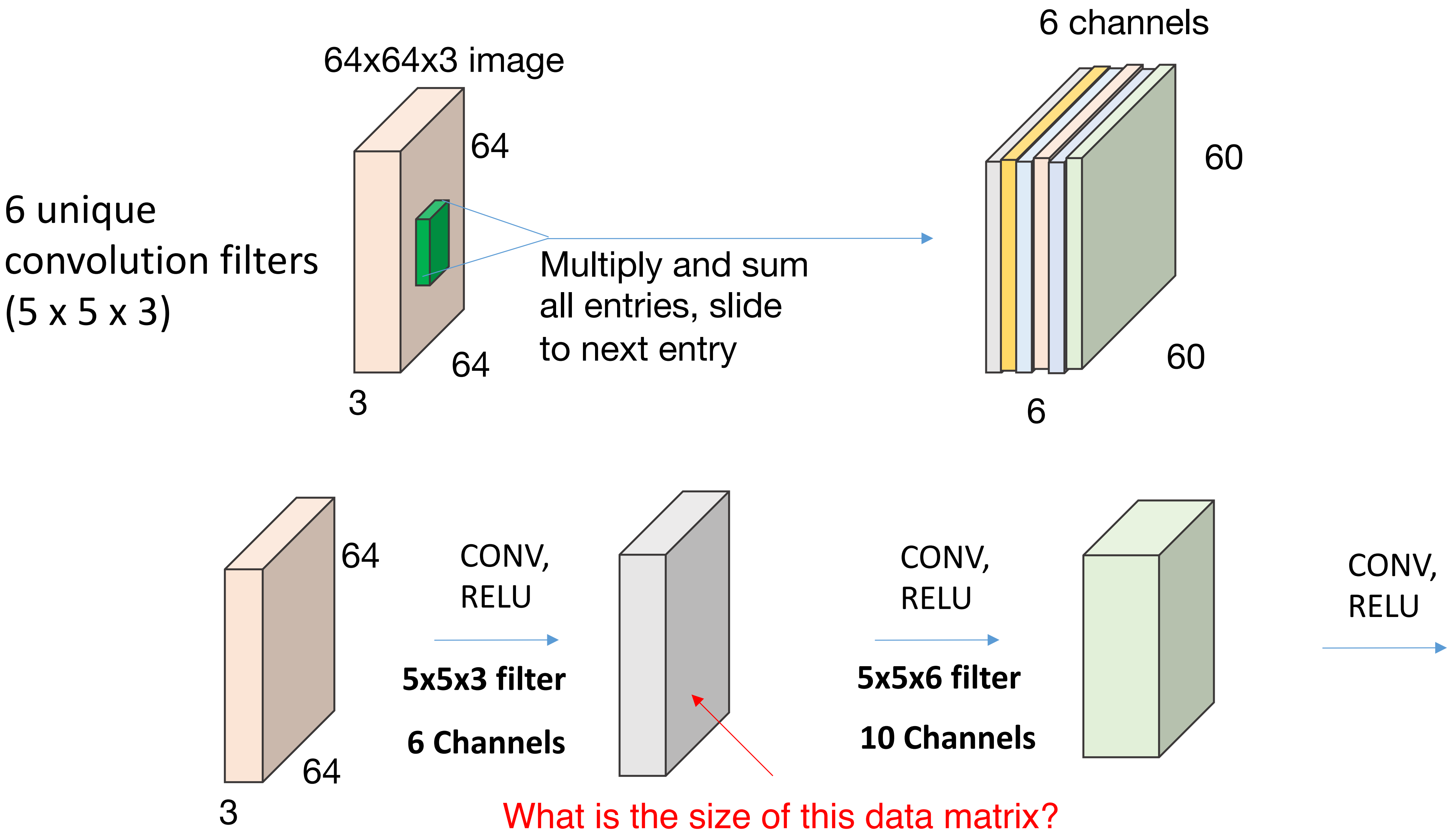


- Using more than one convolutional filter, with unknown weights that we will optimize for, creates more than one *channel*

Convolution layer: learn multiple filters

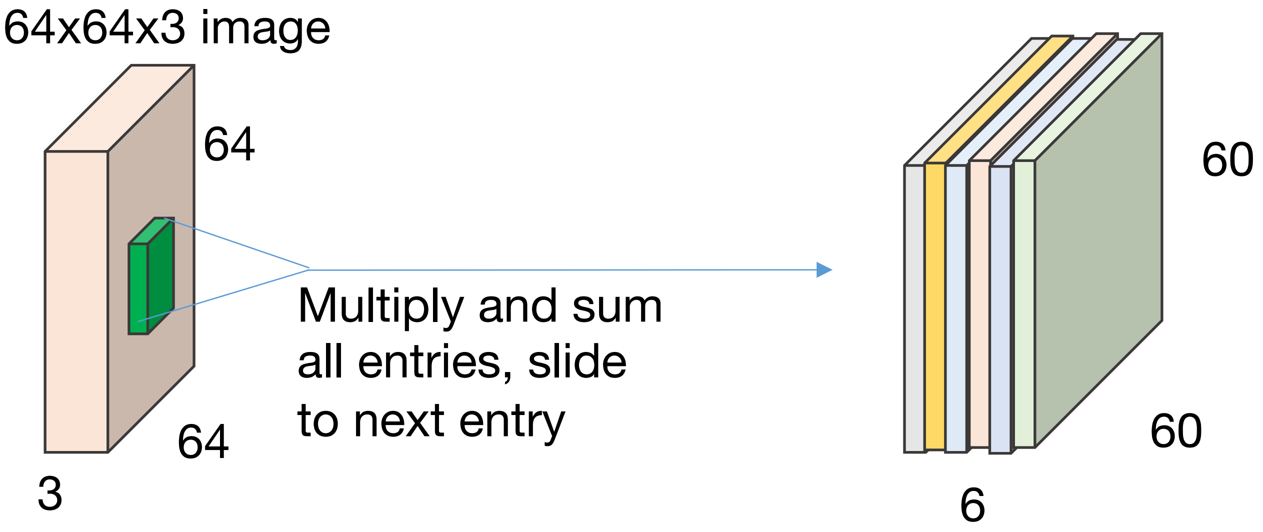


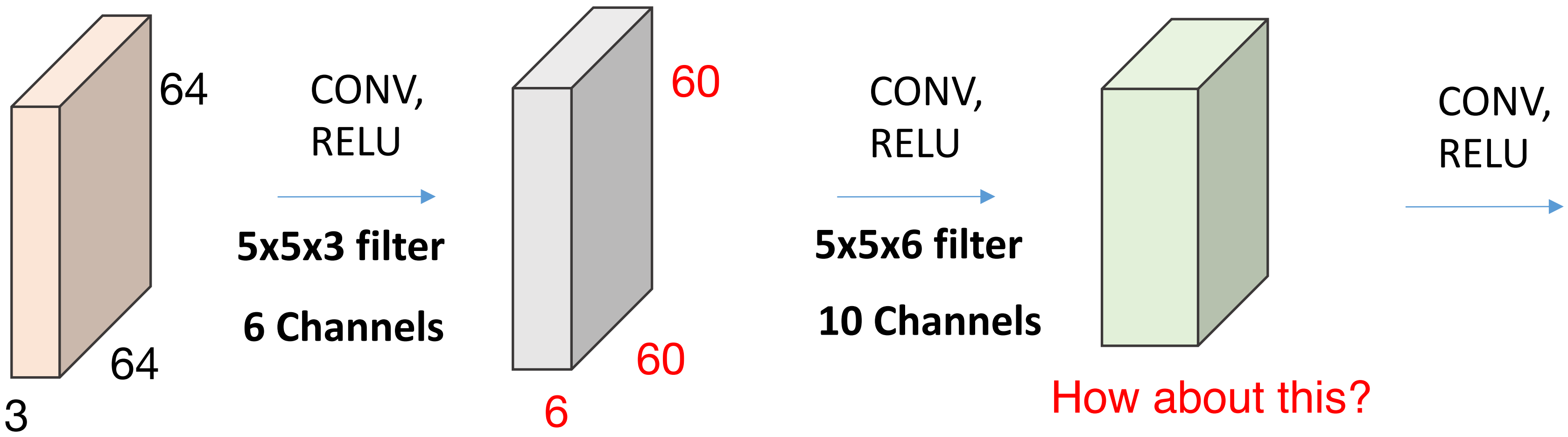
Convolution layer: learn multiple filters



Convolution layer: learn multiple filters

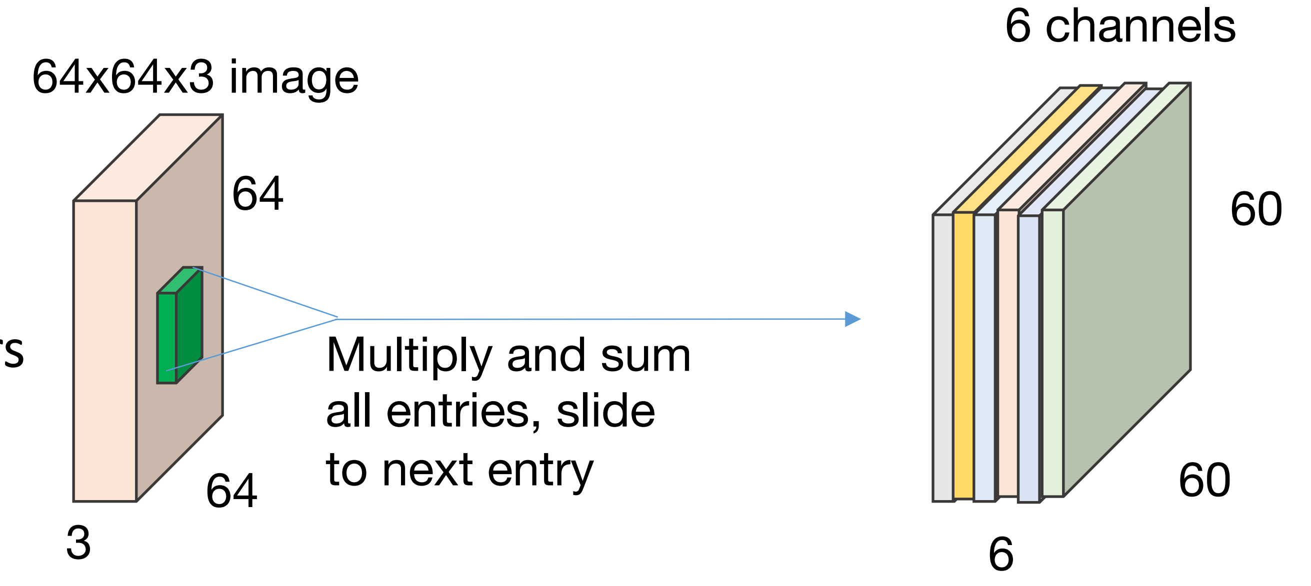
6 unique convolution filters (5 x 5 x 3)

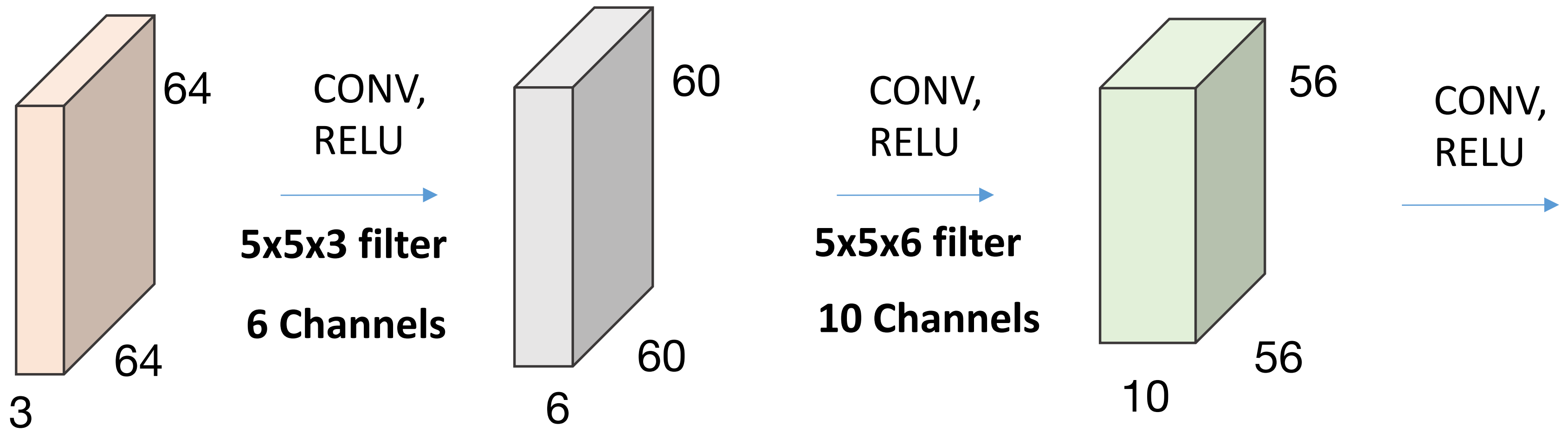




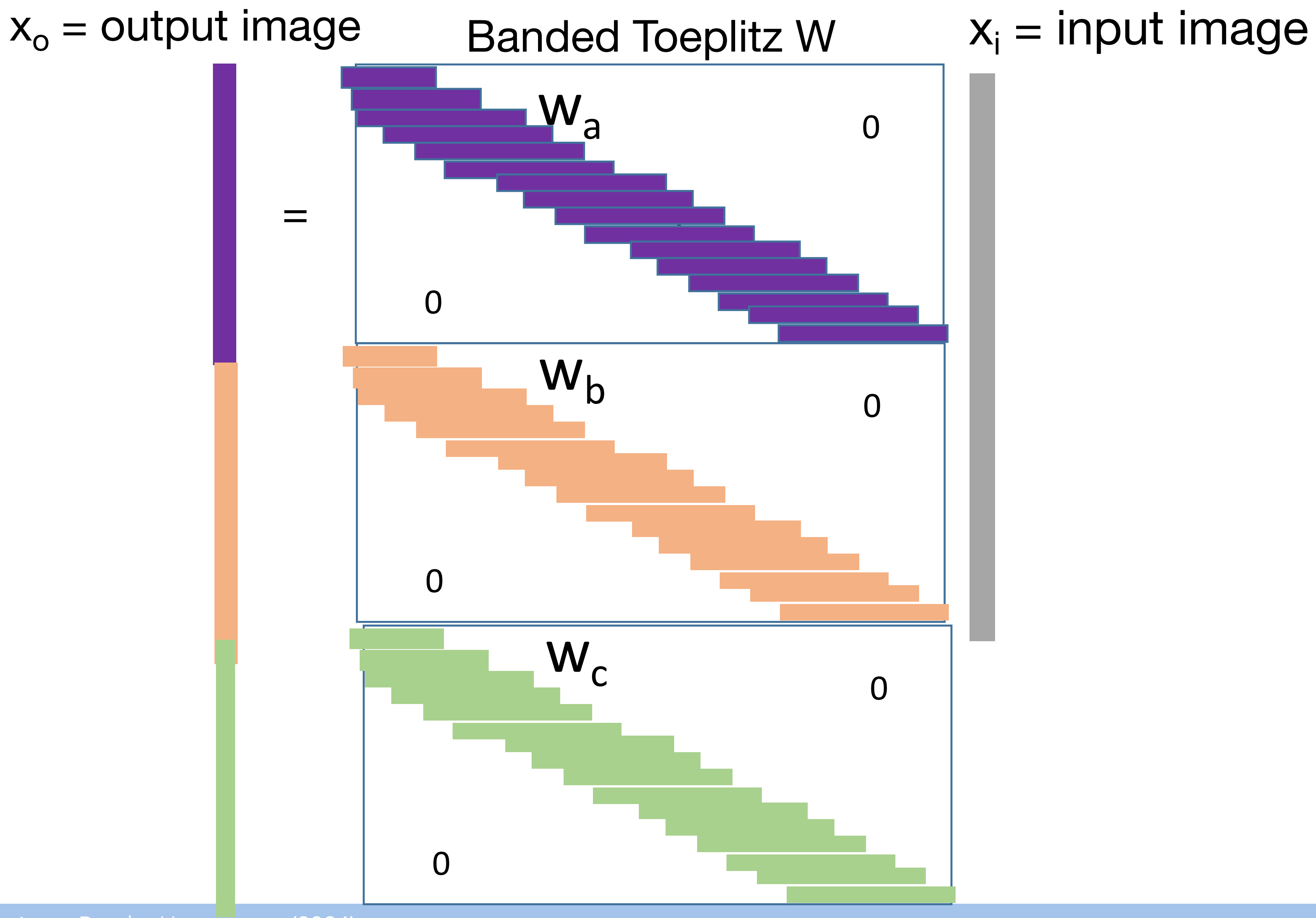
Convolution layer: learn multiple filters

6 unique convolution filters (5 x 5 x 3)





Summarize multiple filters with stacked matrices



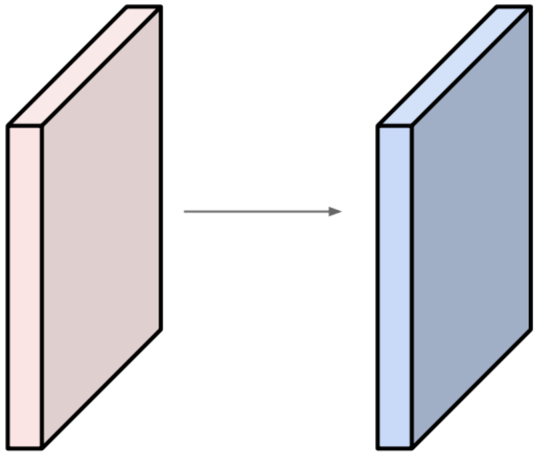
Convolution layer example mapping

deep imaging

Examples time:

Input volume: **32x32x3**
10 5x5x3 filters with stride 1, pad 2

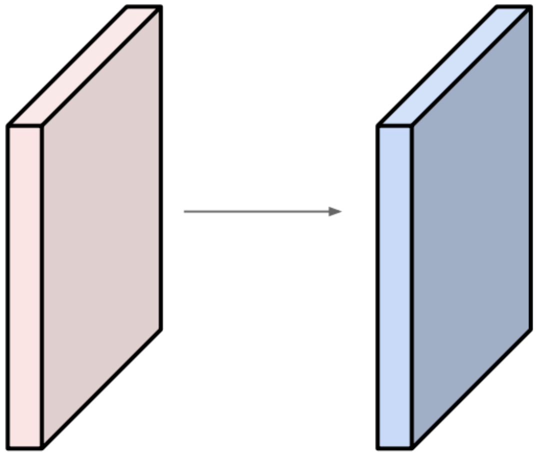
Output volume size: ?



Convolution layer example mapping

Examples time:

Input volume: **32x32x3**
10 5x5x3 filters with stride 1, pad 2



Output volume size: ?

A: (N-F)/stride + 1 = (32+4-5)/1 + 1 = 32x32 spatial extent

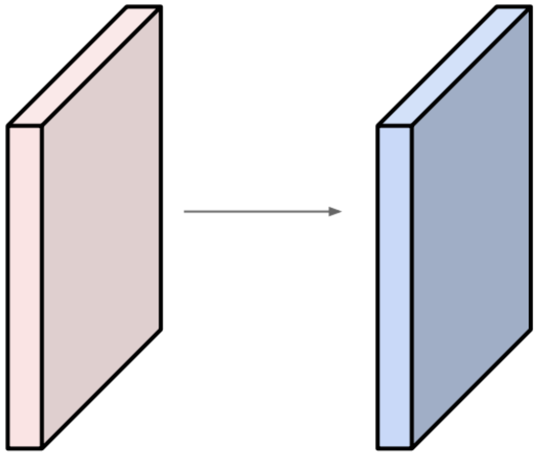
So, output is **32x32x10**

Convolution layer example mapping

Examples time:

Input volume: **32x32x3**

10 5x5x3 filters with stride 1, pad 2

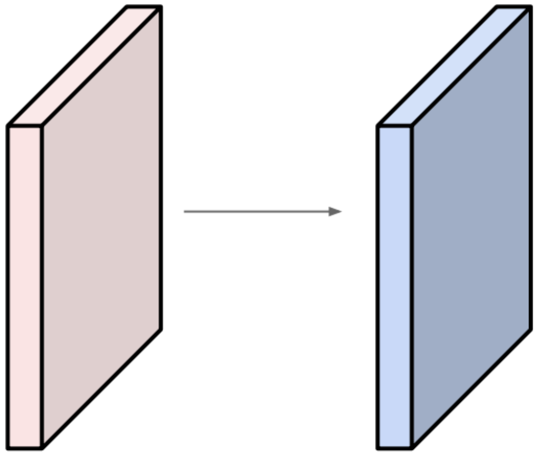


How many weights make up this transformation?

Convolution layer example mapping

Examples time:

Input volume: **32x32x3**
10 5x5x3 filters with stride 1, pad 2

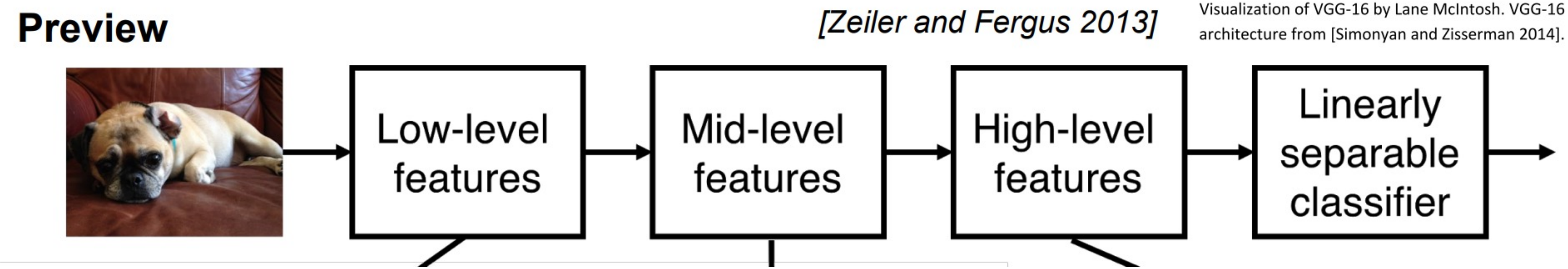


How many weights make up this transformation?

A: Each convolution filter: 5x5x3
1 offset parameter **b** per filter (**untied** biases)
Mapping to 10 output layers = 10 filters
Total: (5x5x3+1)*10 = **760**

What do these convolution filters look like after training?

Visualization of VGG-16 by Lane McIntosh. VGG-16 architecture from [Simonyan and Zisserman 2014].

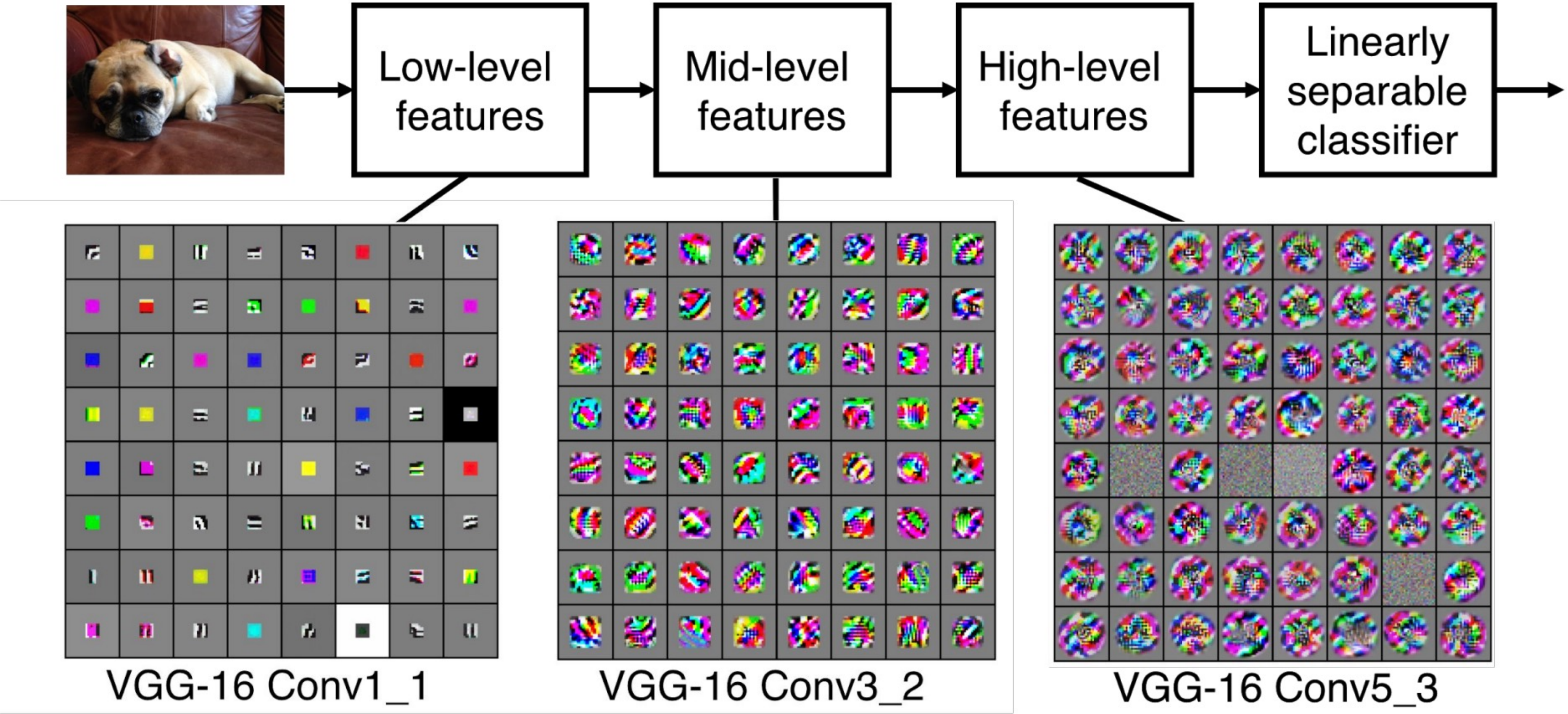


What do these convolution filters look like after training?

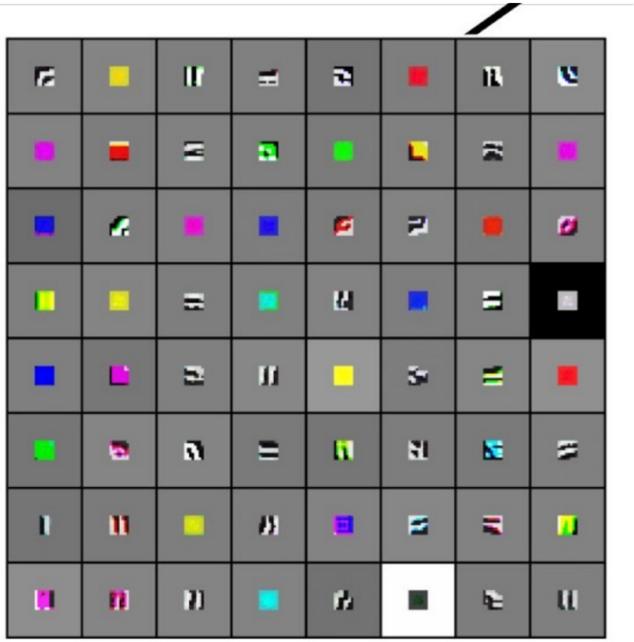
Preview

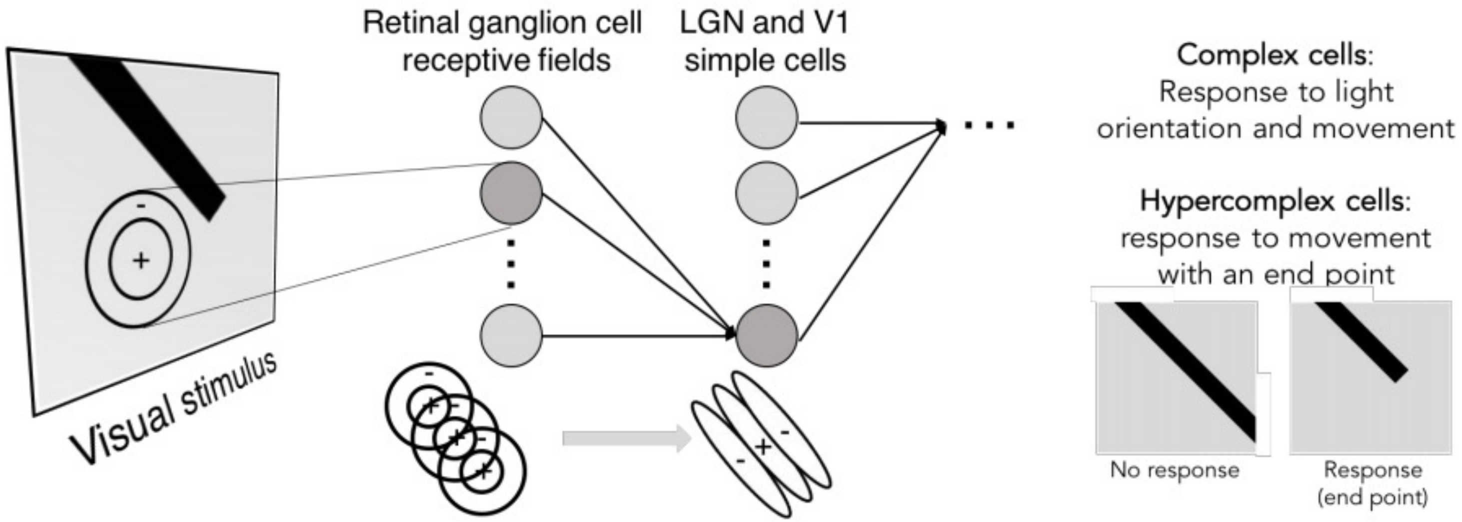
[Zeiler and Fergus 2013]

Visualization of VGG-16 by Lane McIntosh. VGG-16 architecture from [Simonyan and Zisserman 2014].

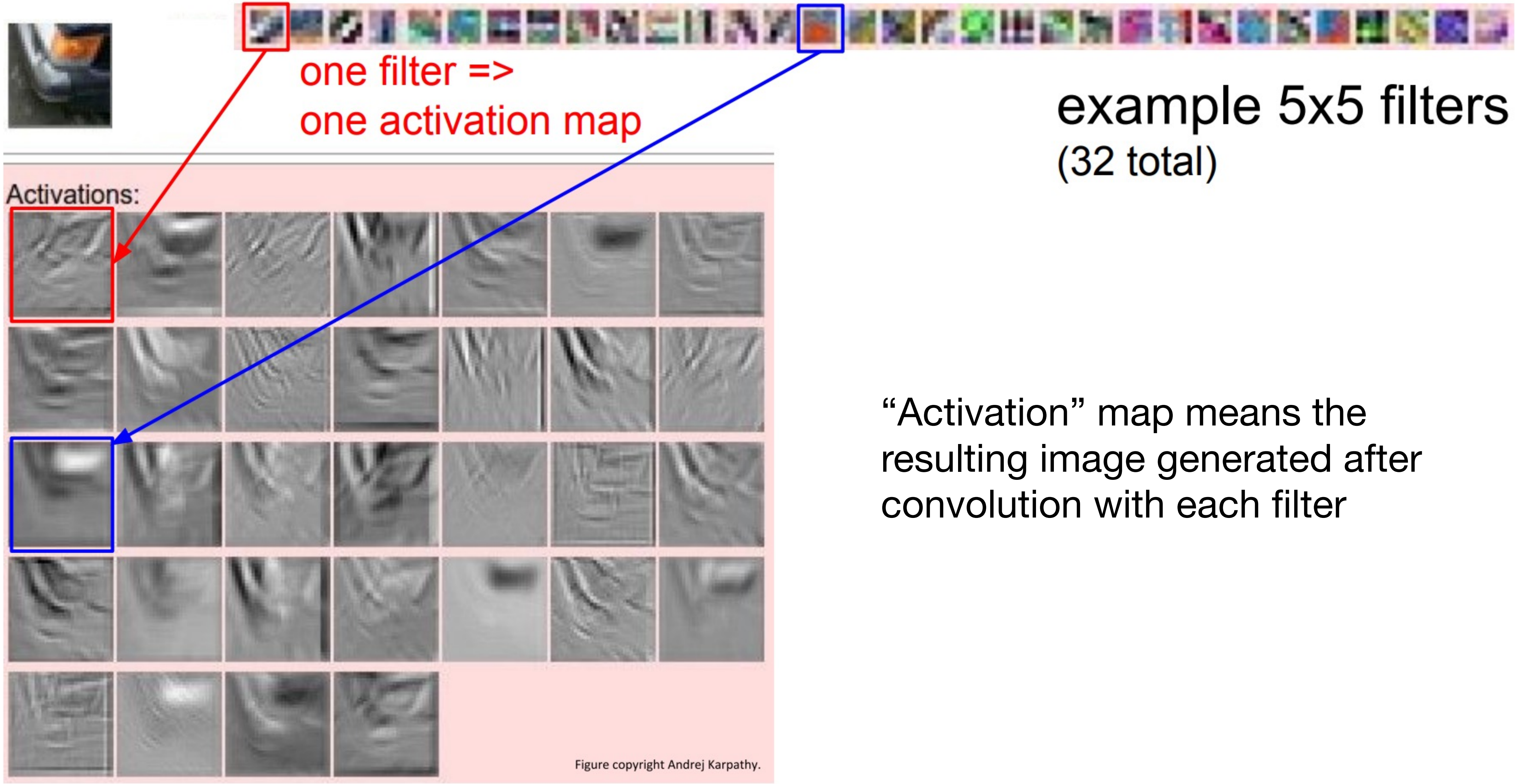


What do these convolution filters look like after training?





- ”Wavey” or wavelet like features are common in first layer
- Match how neurons within our eye map image data to our brain in an effective manner

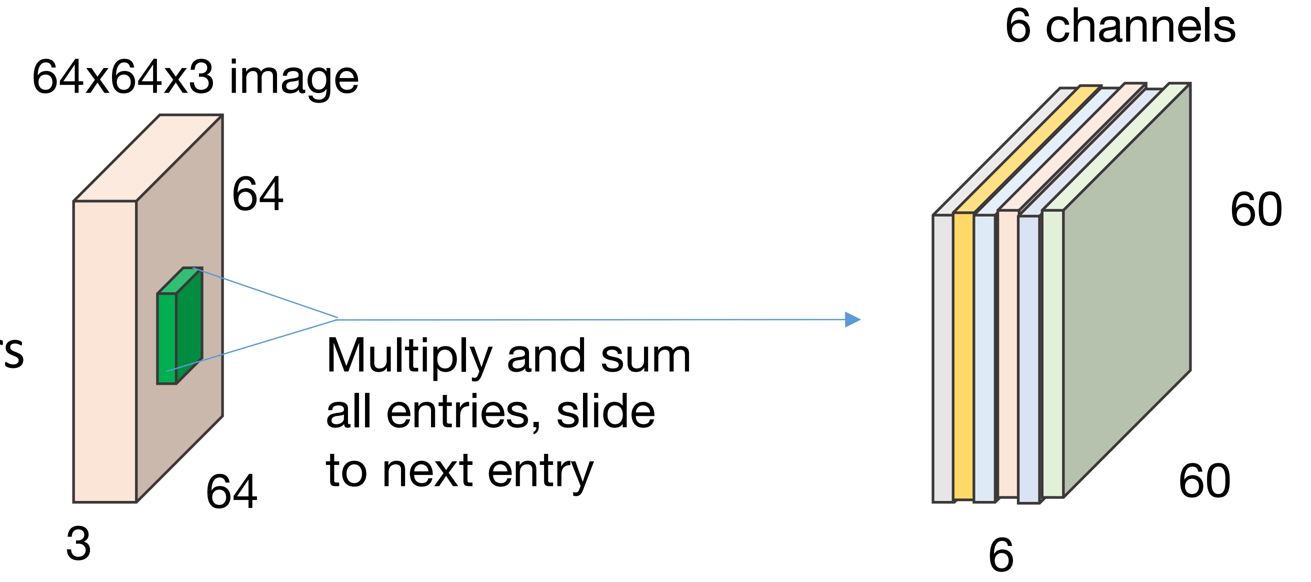


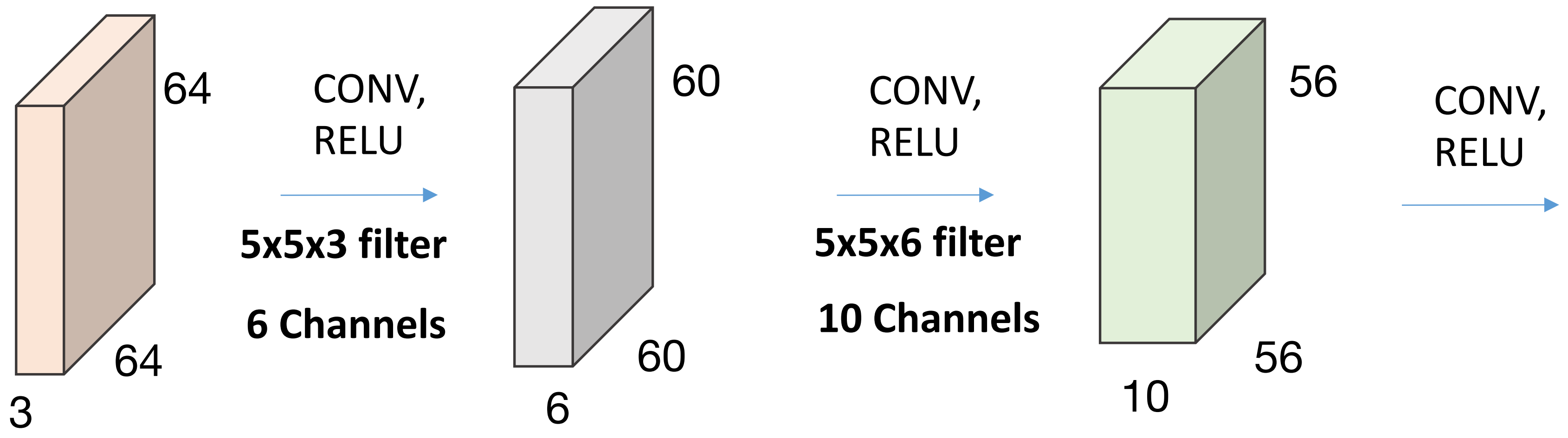
example 5x5 filters
(32 total)

“Activation” map means the resulting image generated after convolution with each filter

Convolution layer: learn multiple filters

6 unique convolution filters (5 x 5 x 3)





Important components of a CNN

CNN Architecture

- CONV size, stride, pad, depth
- ReLU & other nonlinearities
- POOL methods
- # of layers, dimensions per layer
- Fully connected layers

Loss function & optimization

- Type of loss function
- Regularization
- Gradient descent method
- Gradient descent step size

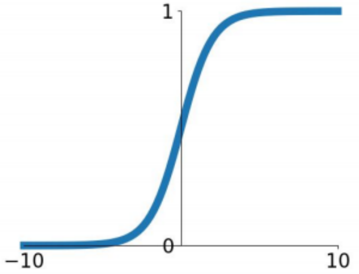
Other specifics: Pre-processing, initialization, dropout, batch normalization, batch size

deep imaging

Non-linear “activation” functions

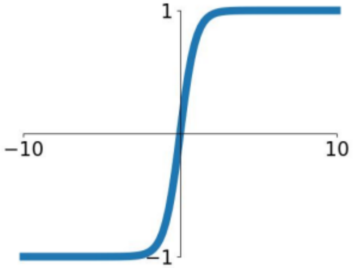
Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



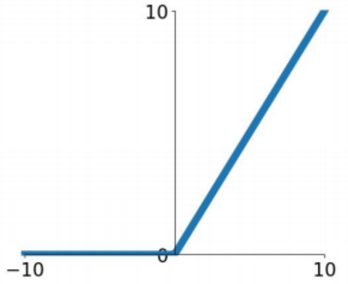
tanh

$$\tanh(x)$$



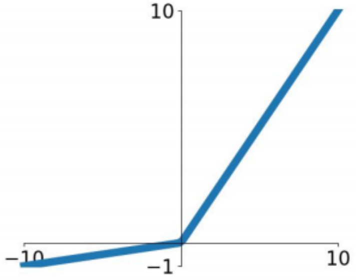
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

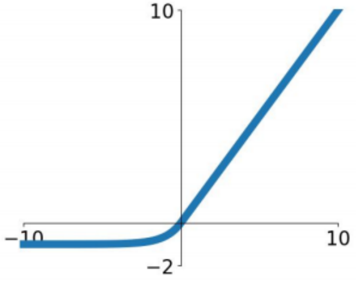


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

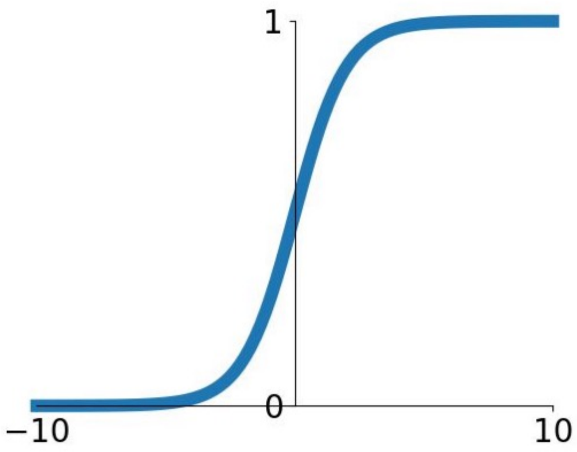
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



From Stanford CS231

Non-linear “activation” functions

$$\sigma(x) = 1/(1 + e^{-x})$$

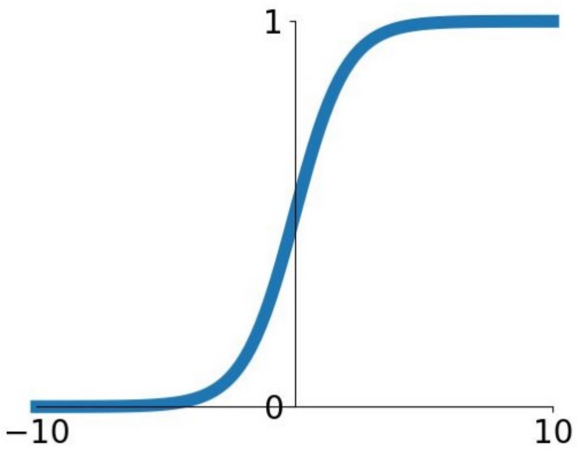


Sigmoid

- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

From Stanford CS231

Non-linear "activation" functions



Sigmoid

$$\sigma(x) = 1/(1+e^{-x})$$

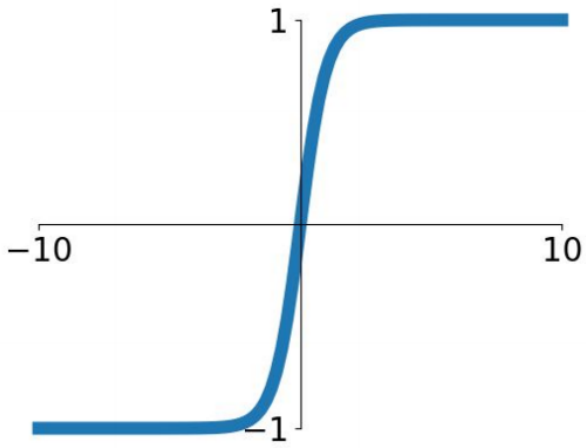
- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating "firing rate" of a neuron

3 problems:

1. Saturated neurons "kill" the gradients
2. Sigmoid outputs are not zero-centered
3. exp() is a bit compute expensive

From Stanford CS231

Non-linear "activation" functions

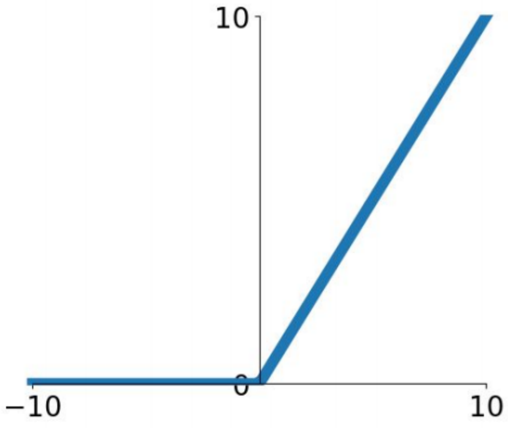


tanh(x)

- Squashes numbers to range [-1,1]
- zero centered (nice)
- still kills gradients when saturated :(

From Stanford CS231

Non-linear "activation" functions



ReLU
(Rectified Linear Unit)

Computes **f(x) = max(0,x)**

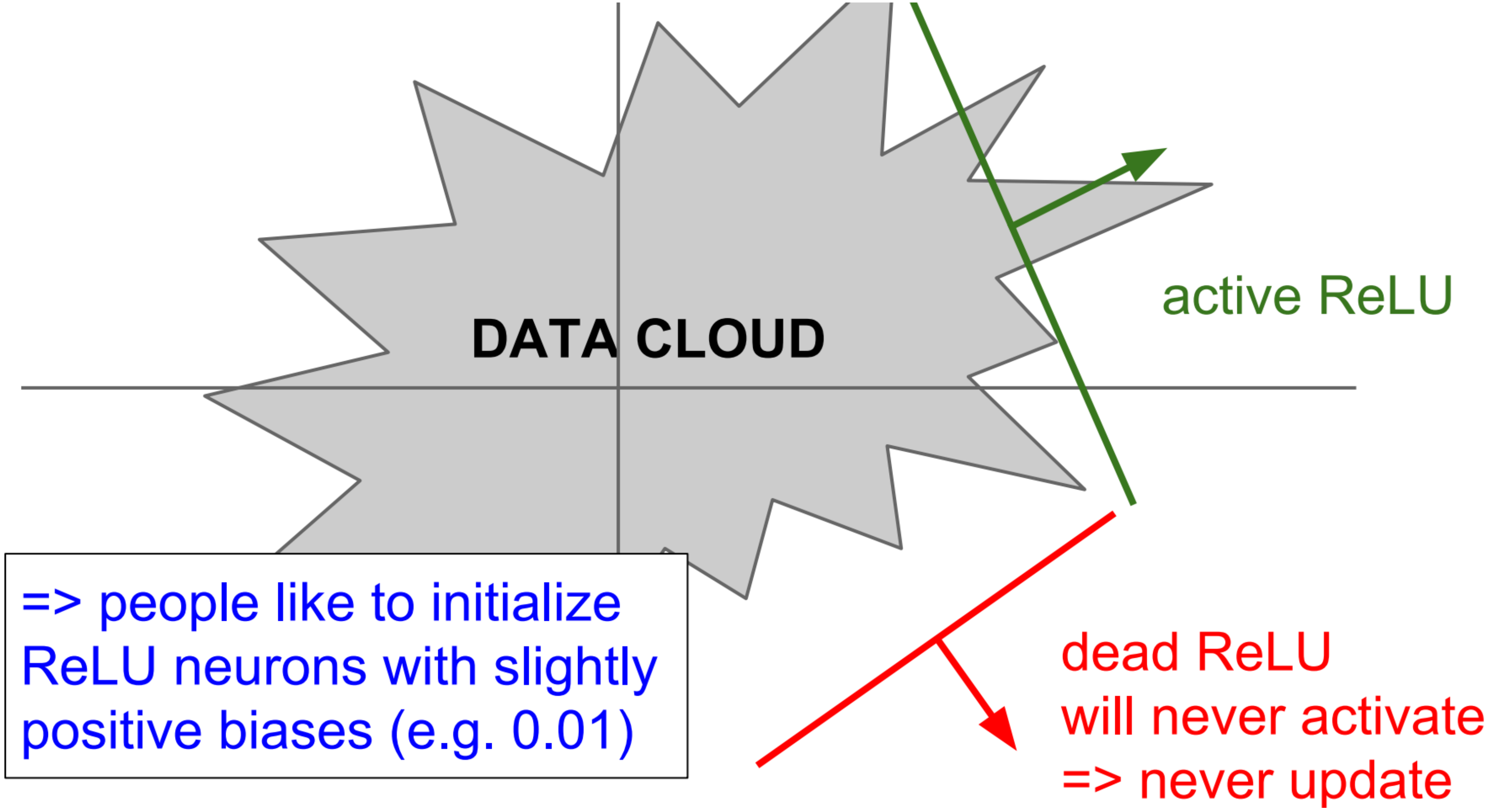
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)
- Actually more biologically plausible than sigmoid

- Not zero-centered output
- An annoyance:

hint: what is the gradient when $x < 0$?

From Stanford CS231

Non-linear “activation” functions



=> people like to initialize ReLU neurons with slightly positive biases (e.g. 0.01)

From Stanford CS231

Important components of a CNN

deep imaging

CNN Architecture

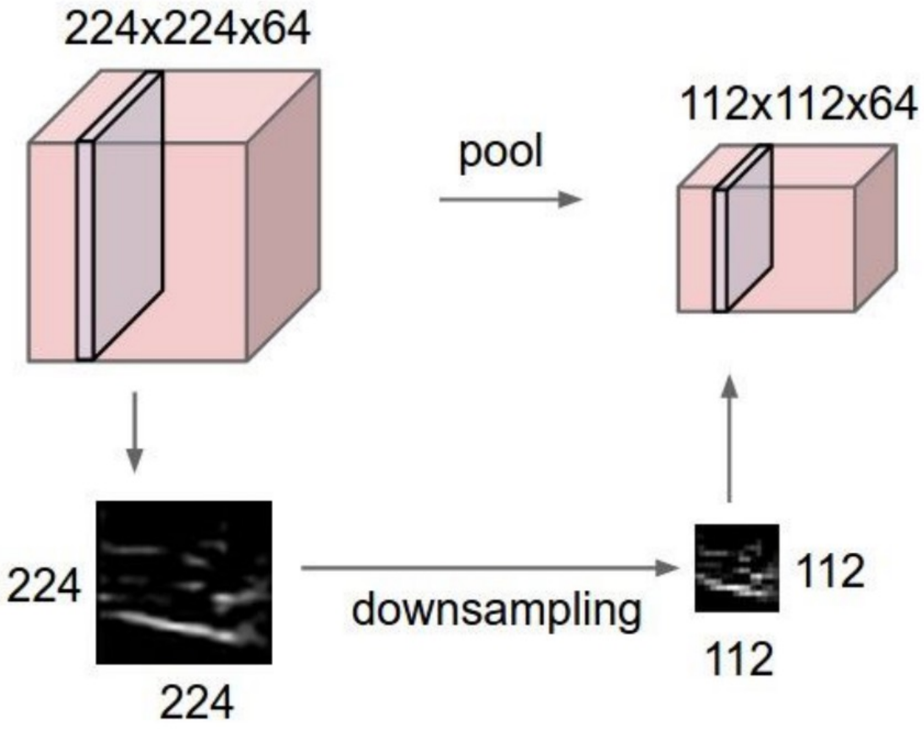
- CONV size, stride, pad, depth
- ReLU & other nonlinearities
- POOL methods
- # of layers, dimensions per layer
- Fully connected layers

Loss function & optimization

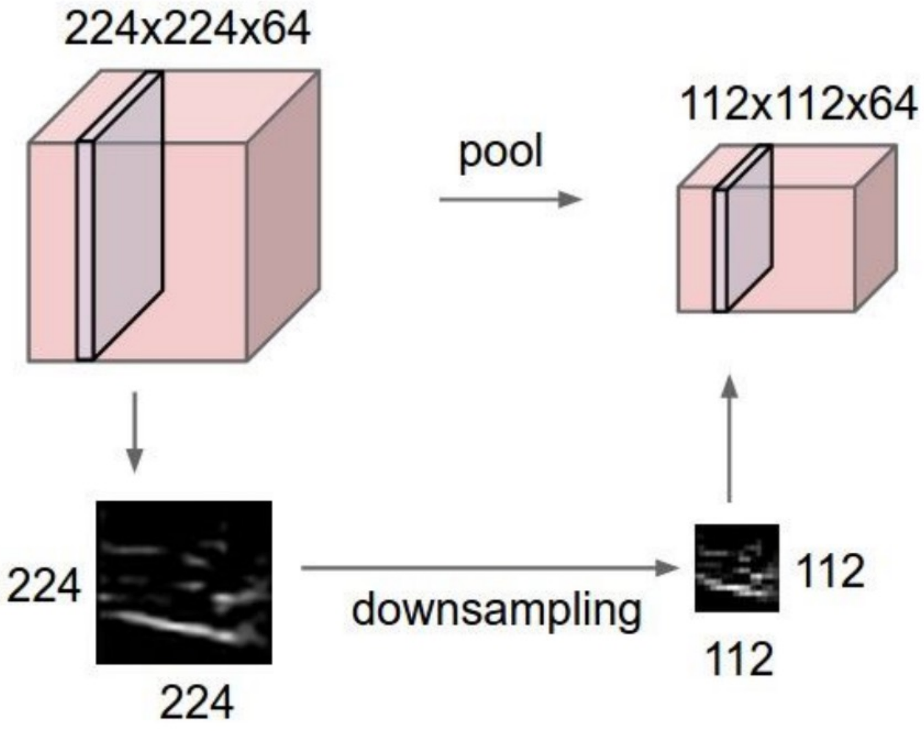
- Type of loss function
- Regularization
- Gradient descent method
- SGD batch and step size

Other specifics: Pre-processing**,** initialization**,** dropout, batch normalization, augmentation

Pooling operation – reduce the size of data cubes along space

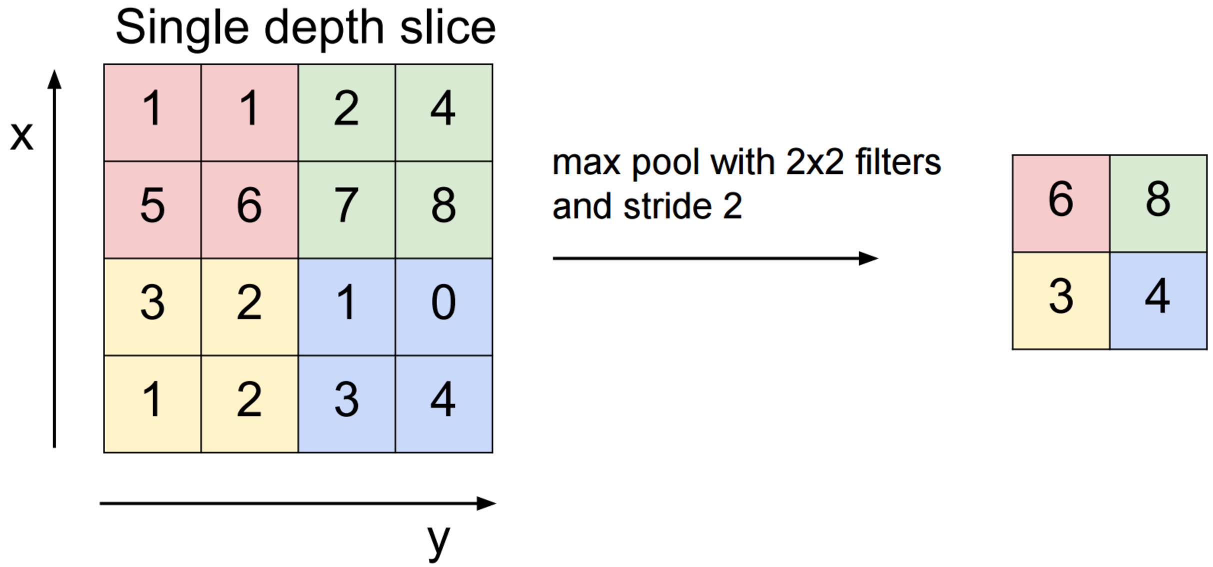


Pooling operation – reduce the size of data cubes along space



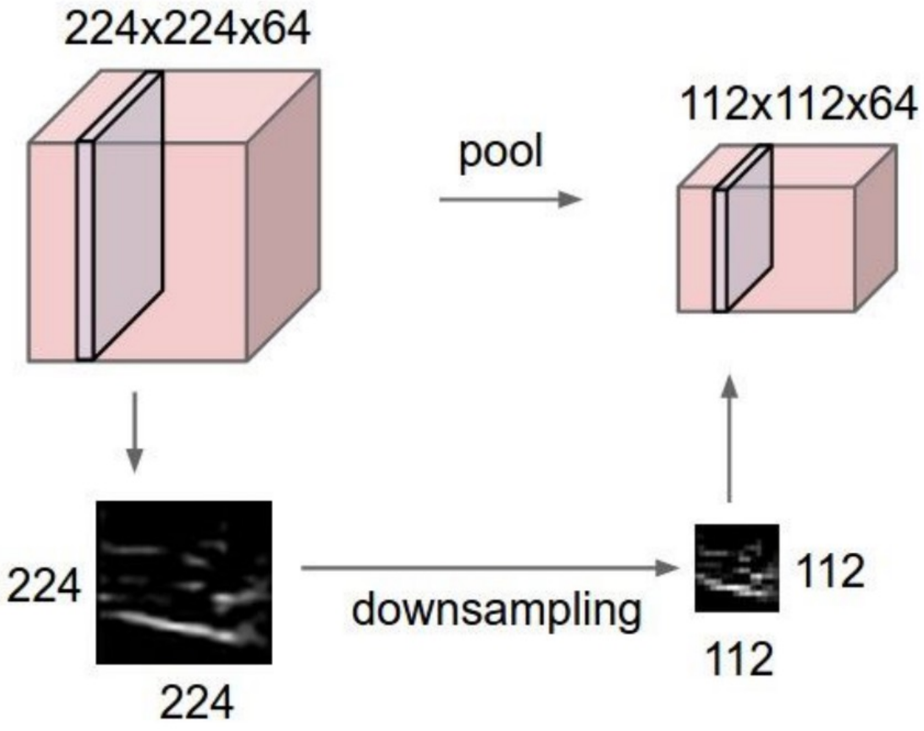
Common option #1:

MAX POOLING



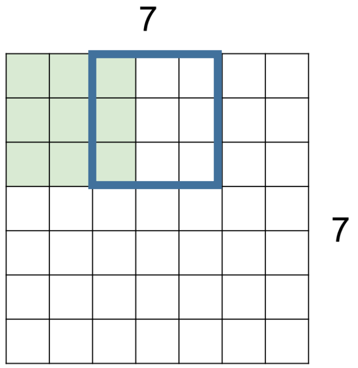
Related options: Sum pooling, mean pooling

Pooling operation – reduce the size of data cubes along space

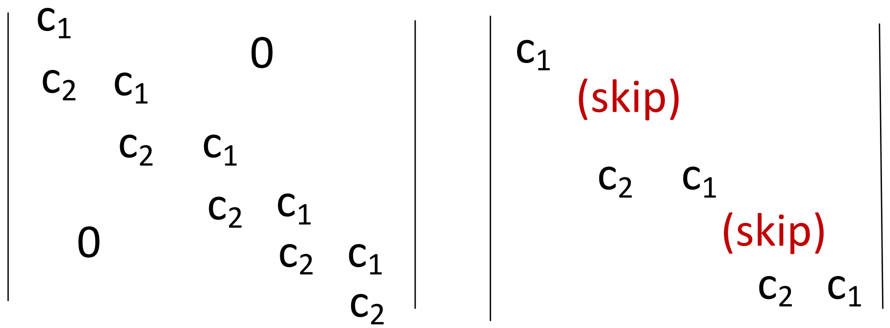


Common option #2: just use bigger strides

STRIDE = 2



7x7 input -> 3x3 output



Important components of a CNN

deep imaging

CNN Architecture

- CONV size, stride, pad, depth
- ReLU & other nonlinearities
- POOL methods
- # of layers, dimensions per layer
- Fully connected layers

Let's view some code!

Loss function & optimization

- Type of loss function
- Regularization
- Gradient descent method
- SGD batch and step size

Other specifics: Pre-processing**,** initialization**,** dropout, batch normalization, augmentation

Important components of a CNN

deep imaging

CNN Architecture

- CONV size, stride, pad, depth
- ReLU & other nonlinearities
- POOL methods
- # of layers, dimensions per layer
- Fully connected layers

Loss function & optimization

- Type of loss function
- Regularization
- Gradient descent method
- SGD batch and step size

Other specifics: Pre-processing**,** initialization**,** dropout, batch normalization, augmentation

Common loss functions used for CNN optimization

deep imaging

- Cross-entropy loss function
  - Softmax cross-entropy – use with single-entry labels
  - Weighted cross-entropy – use to bias towards true pos./false neg.
  - Sigmoid cross-entropy
  - KL Divergence
- Pseudo-Huber loss function
- L1 loss loss function
- MSE (Euclidean error, L2 loss function)
- Mixtures of the above functions

Important components of a CNN

deep imaging

CNN Architecture

- CONV size, stride, pad, depth
- ReLU & other nonlinearities
- POOL methods
- # of layers, dimensions per layer
- Fully connected layers

Loss function & optimization

- Type of loss function
- Regularization
- Gradient descent method
- SGD batch and step size

Other specifics: Pre-processing**,** initialization**,** dropout, batch normalization, augmentation

Regularization – the basics

λ = regularization strength (hyperparameter)

$$L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i) + \lambda R(W)$$

Data loss: Model predictions should match training data

Regularization: Prevent the model from doing *too* well on training data

Simple examples

L2 regularization: $R(W) = \sum_k \sum_l W_{k,l}^2$

L1 regularization: $R(W) = \sum_k \sum_l |W_{k,l}|$

Elastic net (L1 + L2): $R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$

Regularization prefers less complex models & help avoids overfitting

$$x = [1, 1, 1, 1]$$

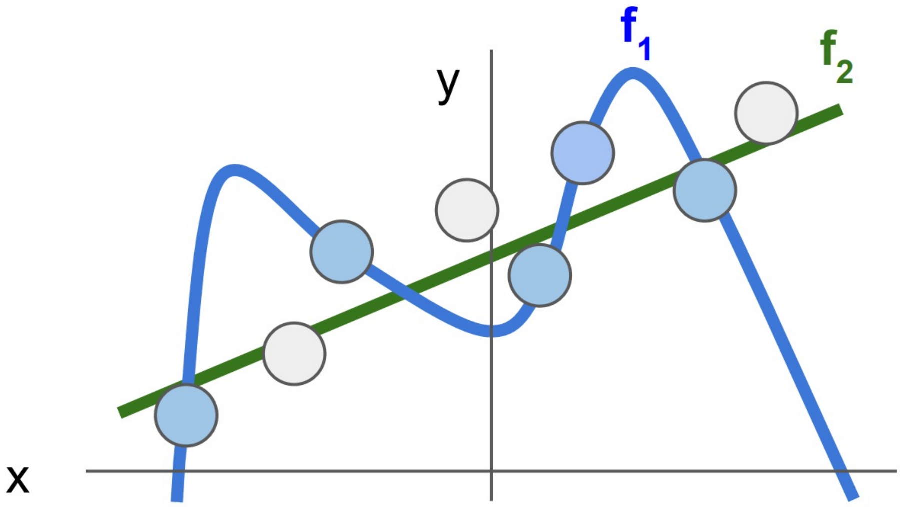
$$w_1 = [1, 0, 0, 0]$$

$$w_2 = [0.25, 0.25, 0.25, 0.25]$$

$$w_1^T x = w_2^T x = 1$$

L2 Regularization

$$R(W) = \sum_k \sum_l W_{k,l}^2$$



Regularization pushes against fitting the data *too* well so we don't fit noise in the data

deep imaging

A two-layer neural network with regularization:

$$L(w) = \frac{1}{N}\sum_{i=1}^{N} ln(1 + e^{-y_i W_2 \max(W_1 x_i, 0)}) + \lambda(||W_1||_2 + ||W_2||_2)$$

Q: How do we determine the best weights W_1 and W_2 to use from this model?

A two-layer neural network with regularization:

$$L(w) = \frac{1}{N}\sum_{i=1}^{N} ln(1 + e^{-y_i W_2 \max(W_1 x_i, 0)}) + \lambda(||W_1||_2 + ||W_2||_2)$$

Q: How do we determine the best weights W_1 and W_2 to use from this model?

A: Gradient descent!

Q: How does Tensorflow figure out the gradients for dL/dW_1 and dL/dW_2?

A two-layer neural network with regularization:

$$L(w) = \frac{1}{N}\sum_{i=1}^{N} ln(1 + e^{-y_i W_2 \max(W_1 x_i, 0)}) + \lambda(||W_1||_2 + ||W_2||_2)$$

Q: How do we determine the best weights W_1 and W_2 to use from this model?

A: Gradient descent!

Q: How does Tensorflow figure out the gradients for dL/dW_1 and dL/dW_2?

A: Chain rule! (next lecture or two)

A variety of gradient descent solvers available in Tensorflow

deep imaging

- Stochastic Gradient Descent (bread-and-butter, when in doubt…)
- Adam Optimizer (update learning rates with mean and variance)
- Nesterov / Momentum (add a velocity term)
- AdaGrad (Adaptive Subgradients, change learning rates)
- Proximal AdaGrad (Proximal = solve second problem to stay close)
- Ftrl Proximal (Follow-the-regularized-leader)
- AdaDelta (Adaptive learning rate)

Implementation detail #1 – method for gradient descent

```
# Vanilla Gradient Descent

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```

Stochastic Gradient Descent (SGD)

$$L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i(x_i, y_i, W) + \lambda R(W)$$

$$\nabla_W L(W) = \frac{1}{N}\sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W) + \lambda \nabla_W R(W)$$

deep imaging

Implementation detail #1 – method for gradient descent

```
# Vanilla Gradient Descent

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```

Stochastic Gradient Descent (SGD)

$$L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i(x_i, y_i, W) + \lambda R(W)$$

$$\nabla_W L(W) = \frac{1}{N}\sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W) + \lambda \nabla_W R(W)$$

Full sum expensive when N is large!

Approximate sum using a **minibatch** of examples
32 / 64 / 128 common

Implementation detail #1 – method for gradient descent

```
# Vanilla Minibatch Gradient Descent

while True:
  data_batch = sample_training_data(data, 256) # sample 256 examples
  weights_grad = evaluate_gradient(loss_fun, data_batch, weights)
  weights += - step_size * weights_grad # perform parameter update
```

Stochastic Gradient Descent (SGD)

$$L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i(x_i, y_i, W) + \lambda R(W)$$

$$\nabla_W L(W) = \frac{1}{N}\sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W) + \lambda \nabla_W R(W)$$

Full sum expensive when N is large!

Approximate sum using a **minibatch** of examples
32 / 64 / 128 common

Question: Why does gradient descent still work with mini-batches?

Question: Why does gradient descent still work with mini-batches?

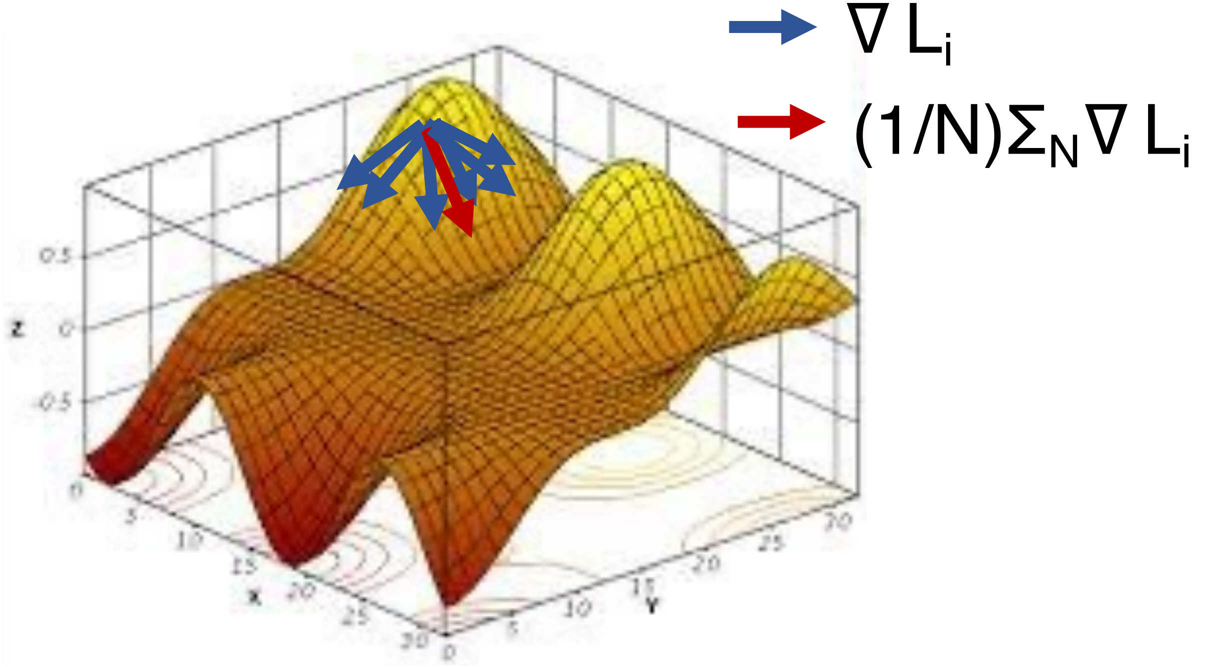
deep imaging

Answer: With stochastic gradient descent, random sub-set averaging of gradients still allows one to find their way down the hill to global minimum, at least with convex and quasi-convex functions [1].

Stochastic Gradient Descent (SGD)

$$L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i(x_i, y_i, W) + \lambda R(W)$$

$$\nabla_W L(W) = \frac{1}{N}\sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W) + \lambda \nabla_W R(W)$$



Question: Why does gradient descent still work with mini-batches?

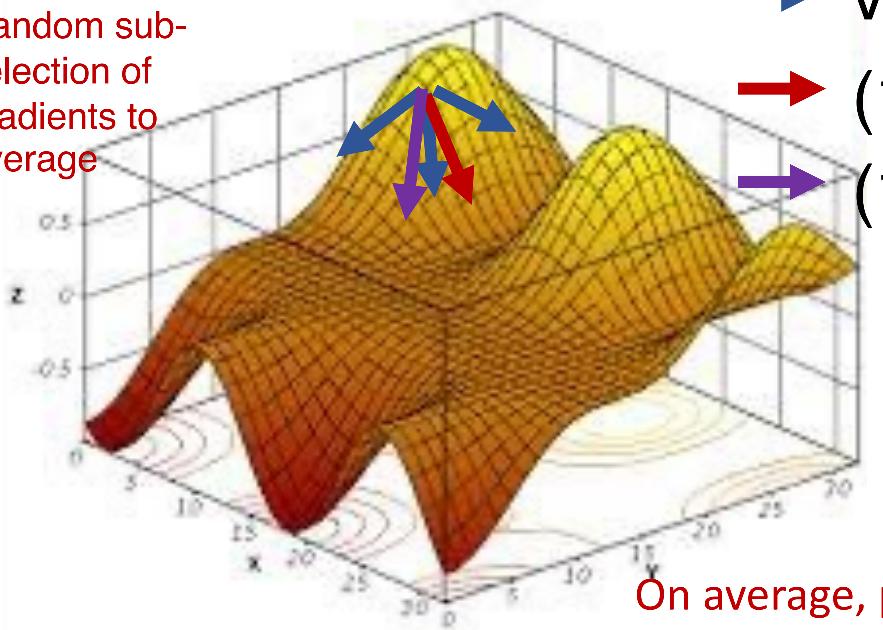
Answer: With stochastic gradient descent, random sub-set averaging of gradients still allows one to find their way down the hill to global minimum, at least with convex and quasi-convex functions [1].

Stochastic Gradient Descent (SGD)

$$L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i(x_i, y_i, W) + \lambda R(W)$$

$$\nabla_W L(W) = \frac{1}{N}\sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W) + \lambda \nabla_W R(W)$$

Random sub-selection of gradients to average



∇L_i

$(1/N)\Sigma_N \nabla L_i$

$(1/M)\Sigma_M \nabla L_i$

On average, purple arrow points in same direction as red arrow

[1] Bottou, Léon (1998). "Online Algorithms and Stochastic Approximations": https://leon.bottou.org/publications/pdf/online-1998.pdf

deep imaging

A variety of gradient descent solvers available in Tensorflow

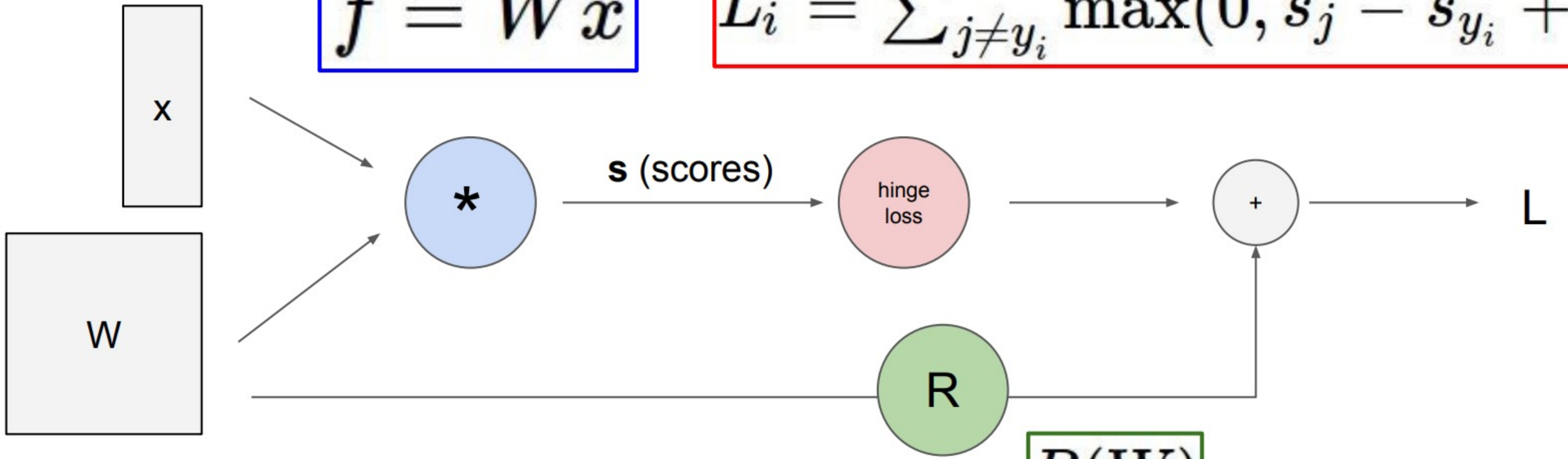
- Stochastic Gradient Descent (bread-and-butter, when in doubt…)
- Adam Optimizer (update learning rates with mean and variance)
- Nesterov / Momentum (add a velocity term)
- AdaGrad (Adaptive Subgradients, change learning rates)
- Proximal AdaGrad (Proximal = solve second problem to stay close)
- Ftrl Proximal (Follow-the-regularized-leader)
- AdaDelta (Adaptive learning rate)

Next lecture: how Tensorflow actually solves gradient descent for you

Computational Graphs and the Chain Rule!

$$f = Wx$$

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$



$$R(W)$$