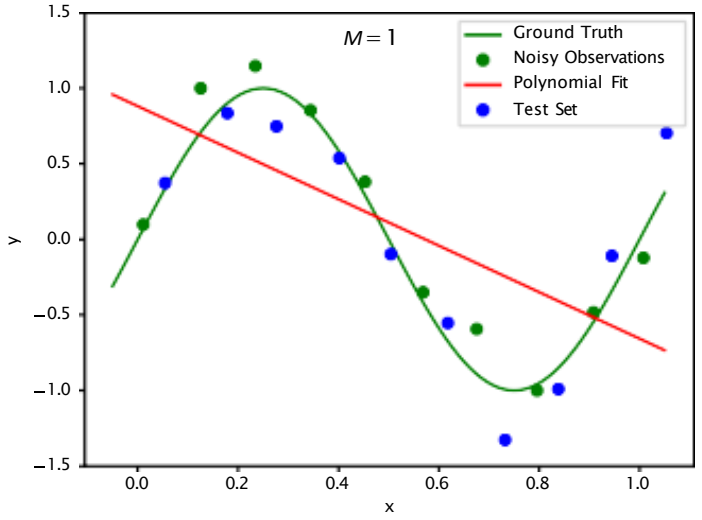
Deep Learning
CS-878

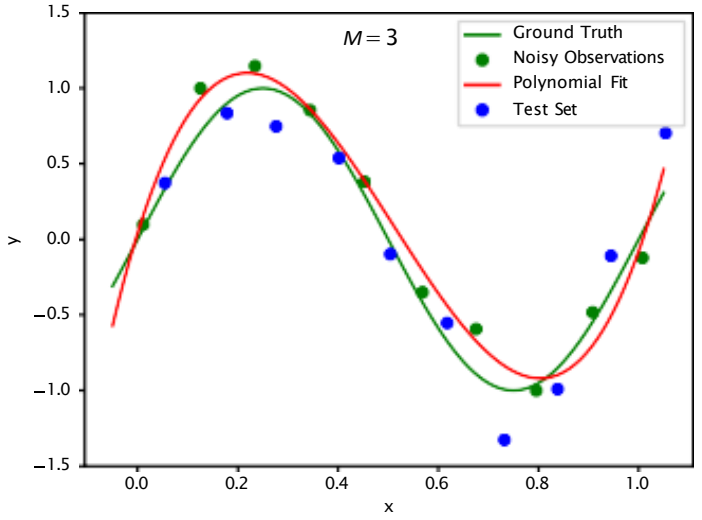
Week-04

Regularizations

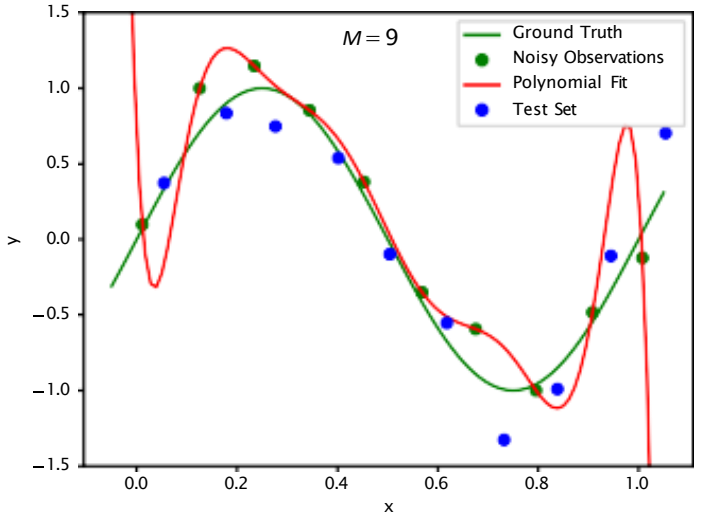
Capacity, Overfitting and Underfitting



Capacity too low



Capacity about right



Capacity too high

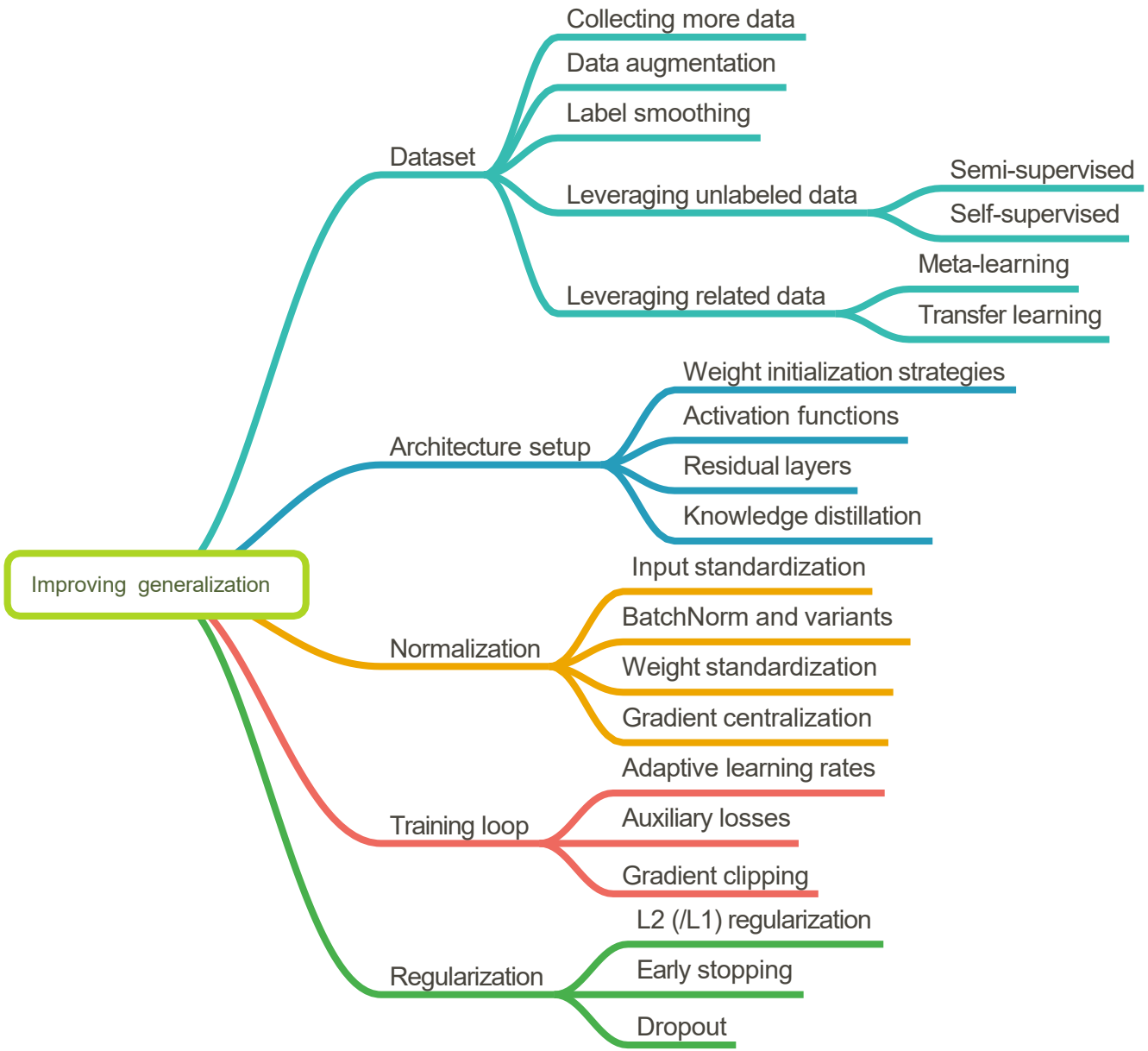
- **Underfitting:** Model too simple, does not achieve low error on training set
- **Overfitting:** Training error small, but test error (= generalization error) large
- **Regularization:** Take model from third regime (right) to second regime (middle)

Goal: Reduce Overfitting

usually achieved by reducing model capacity and/or reduction of the variance of the predictions

Available Options

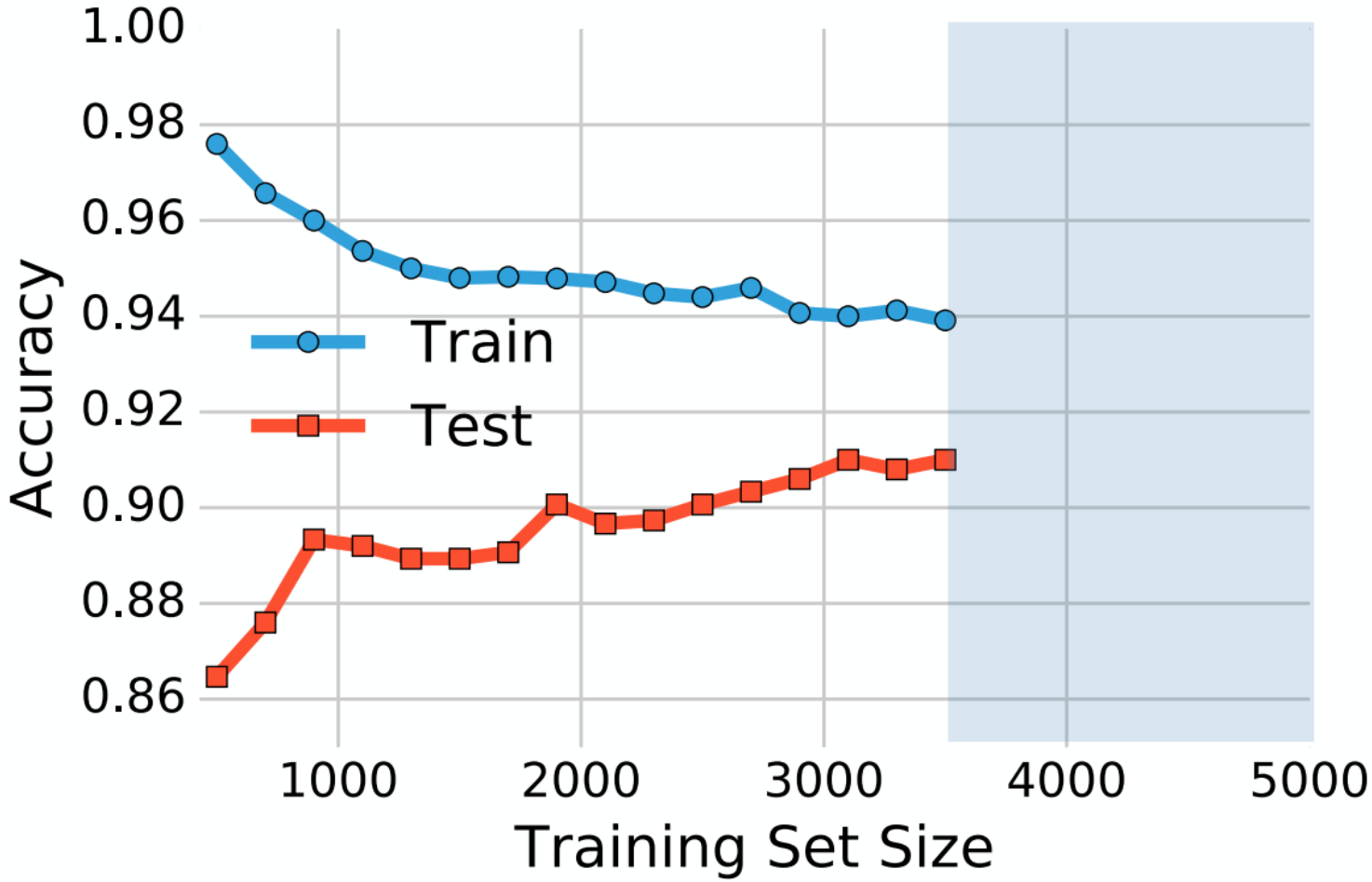
1. **Improving generalization performance**
2. Avoiding overfitting with (1) more data and (2) data augmentation
3. Reducing network capacity & early stopping
4. Adding norm penalties to the loss: L1 & L2 regularization
5. Dropout


Improving generalization
Dataset
Collecting more data
Data augmentation
Label smoothing
Leveraging unlabeled data
Semi-supervised
Self-supervised
Leveraging related data
Meta-learning
Transfer learning
Architecture setup
Weight initialization strategies
Activation functions
Residual layers
Knowledge distillation
Normalization
Input standardization
BatchNorm and variants
Weight standardization
Gradient centralization
Training loop
Adaptive learning rates
Auxiliary losses
Gradient clipping
Regularization
L2 (/L1) regularization
Early stopping
Dropout

First step to improve performance: Focusing on the dataset itself

1. Improving generalization performance
2. **Avoiding overfitting with (1) more data and (2) data augmentation**
3. Reducing network capacity & early stopping
4. Adding norm penalties to the loss: L1 & L2 regularization
5. Dropout

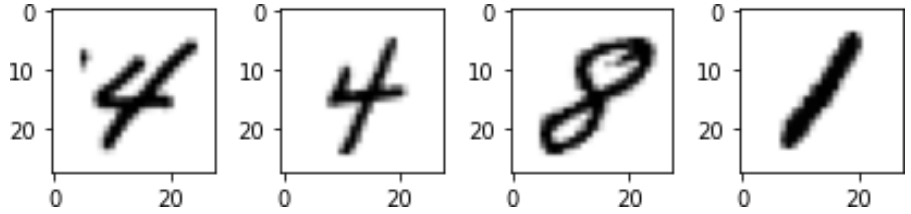
Often, the Best Way to Reduce Overfitting is to collect more data



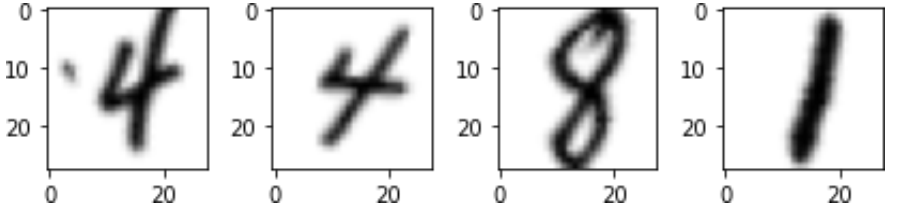
Softmax on MNIST subset (test set size is kept constant)

Data Augmentation

Original

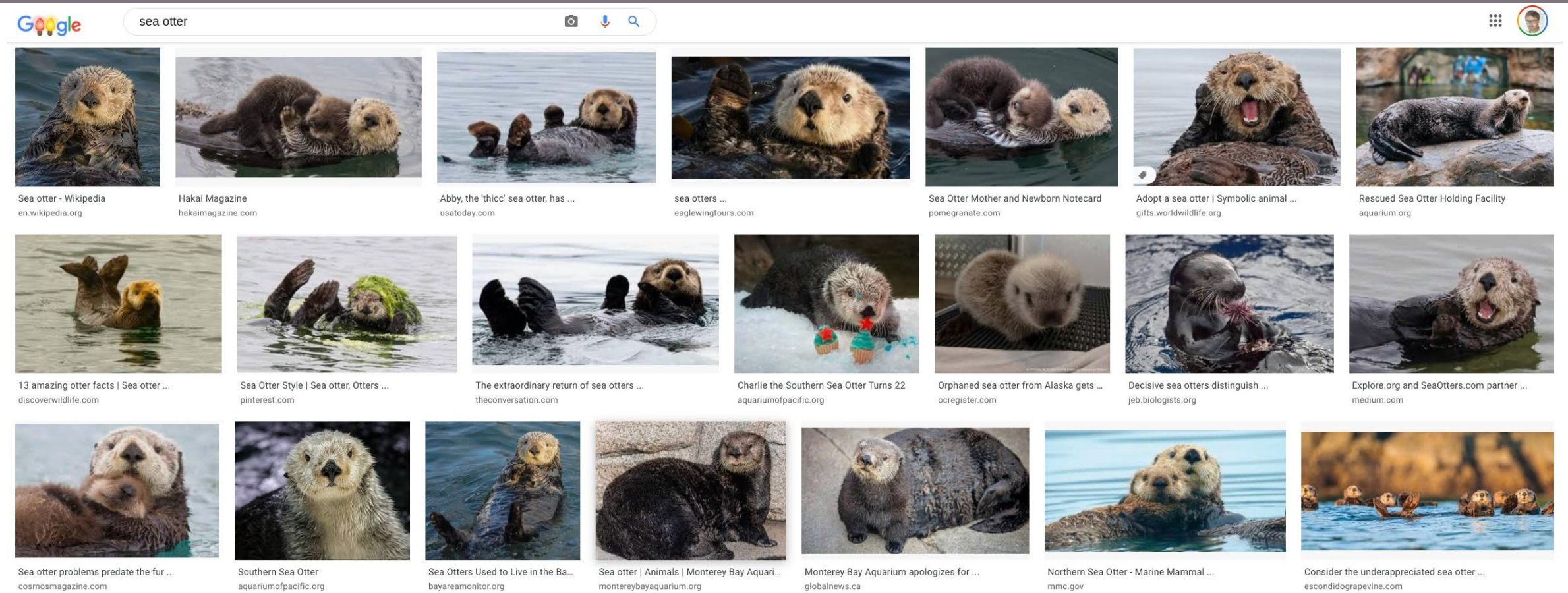


Randomly Augmented



https://github.com/rasbt/stat453-deep-learning-ss21/blob/master/L10/code/data-augmentation.ipynb

Data Augmentation

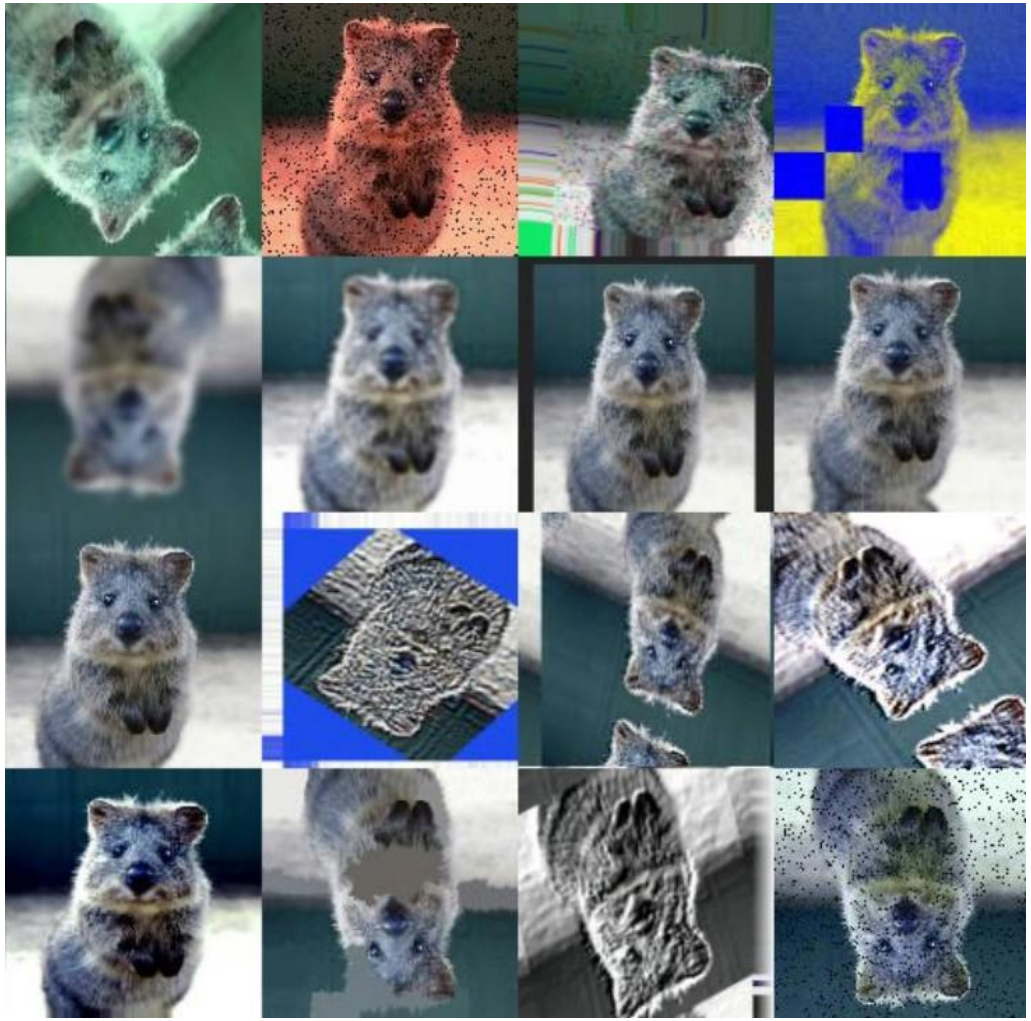


Motivation:

- Deep neural networks must be **invariant** to a wide variety of input variations
- Often **large intra-class variation** in terms of pose, appearance, lighting, etc.

Data Augmentation

- Best way towards better generalization is to **train on more data**
- However, data in practice often limited
- Goal of data augmentation: create **"fake" data** from the existing data (on the fly) and add it to the training set
- New data must **preserve semantics**
- Even **simple operations** like translation or adding per-pixel noise often already greatly improve generalization
- https://github.com/aleju/imgaug



Other Ways for Dealing with Overfitting if Collecting More Data is not Feasible

=> Reducing Network's Capacity by Other Means

1. Improving generalization performance
2. Avoiding overfitting with (1) more data and (2) data augmentation
3. **Reducing network capacity & early stopping**
4. Adding norm penalties to the loss: L1 & L2 regularization
5. Dropout

Early Stopping

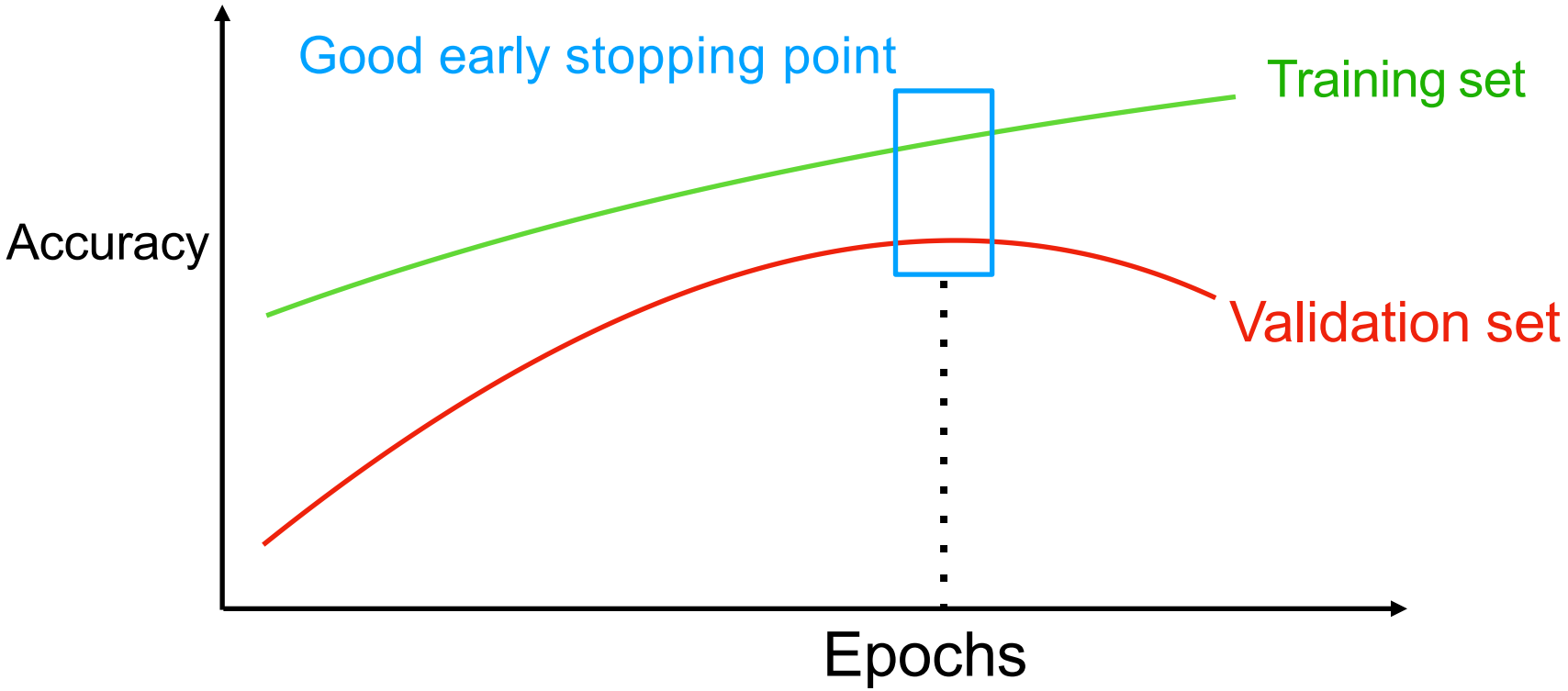
Step 1: Split your dataset into 3 parts

- use test set only once at the end (for unbiased estimate of generalization performance)
- use validation accuracy for tuning

Early Stopping

Step 2: Early stopping

Reduce overfitting by observing the training/validation accuracy gap during training and then stop at the "right" point



Adding a Penalty Against Complexity

1. Improving generalization performance
2. Avoiding overfitting with (1) more data and (2) data augmentation
3. Reducing network capacity & early stopping
4. **Adding norm penalties to the loss: L1 & L2 regularization**
5. Dropout

L_1/L_2 Regularization

- L_1-regularization => LASSO regression
- L_2-regularization => Ridge regression (Thikonov regularization)

Basically, a "weight shrinkage" or a "penalty against complexity"

L_2 Regularization for Linear Models

$$\text{Cost}_{\mathbf{w},\mathbf{b}} = \frac{1}{n}\sum_{i=1}^{n}\mathcal{L}(y^{[i]}, \hat{y}^{[i]})$$

$$\text{L2-Regularized-Cost}_{\mathbf{w},\mathbf{b}} = \frac{1}{n}\sum_{i=1}^{n}\mathcal{L}(y^{[i]}, \hat{y}^{[i]}) + \frac{\lambda}{n}\sum_{j} w_j^2$$

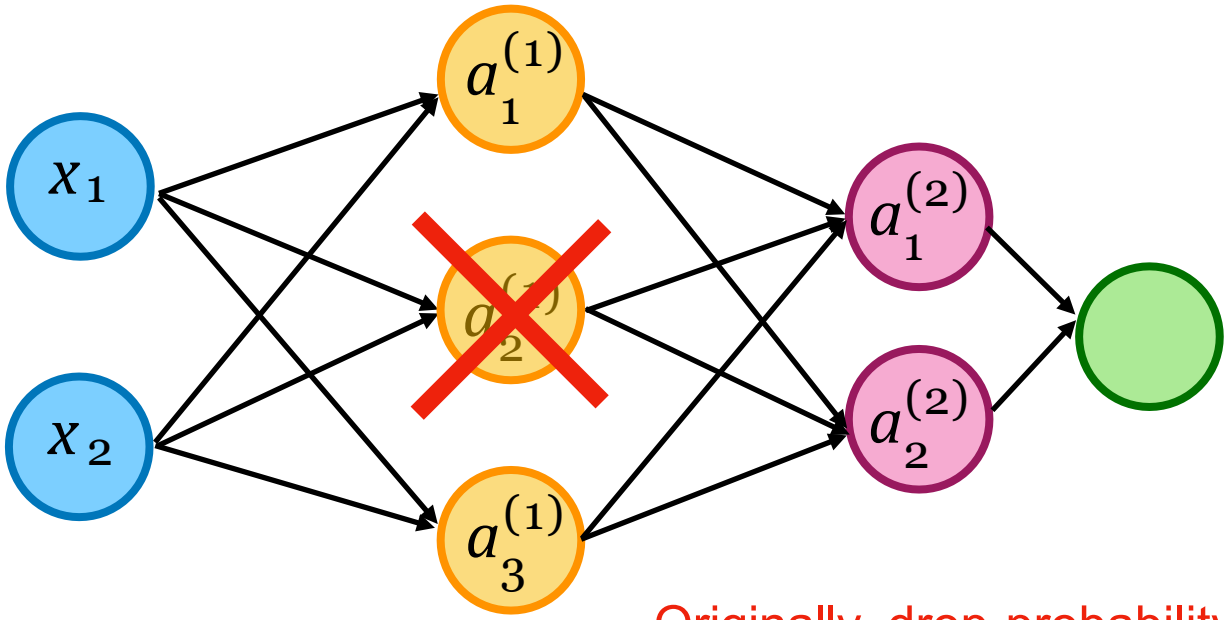
where: $\sum_{j} w_j^2 = ||\mathbf{w}||_2^2$

and λ is a hyperparameter

Dropout

1. Improving generalization performance
2. Avoiding overfitting with (1) more data and (2) data augmentation
3. Reducing network capacity & early stopping
4. Adding norm penalties to the loss: L1 & L2 regularization
5. **Dropout**

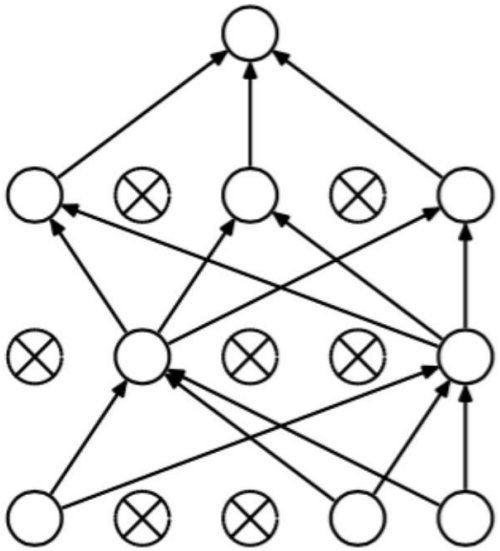
Dropout in a Nutshell: Dropping Nodes

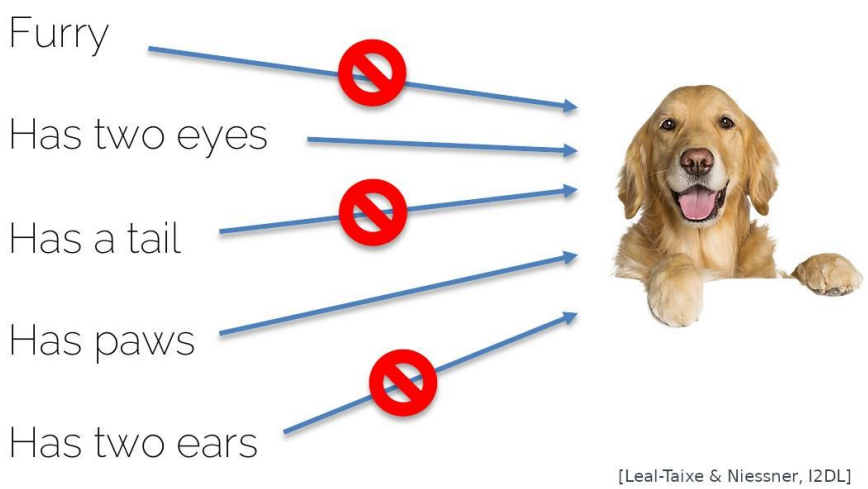


Originally, drop probability 0.5

(but 0.2-0.8 also common now)

Dropout



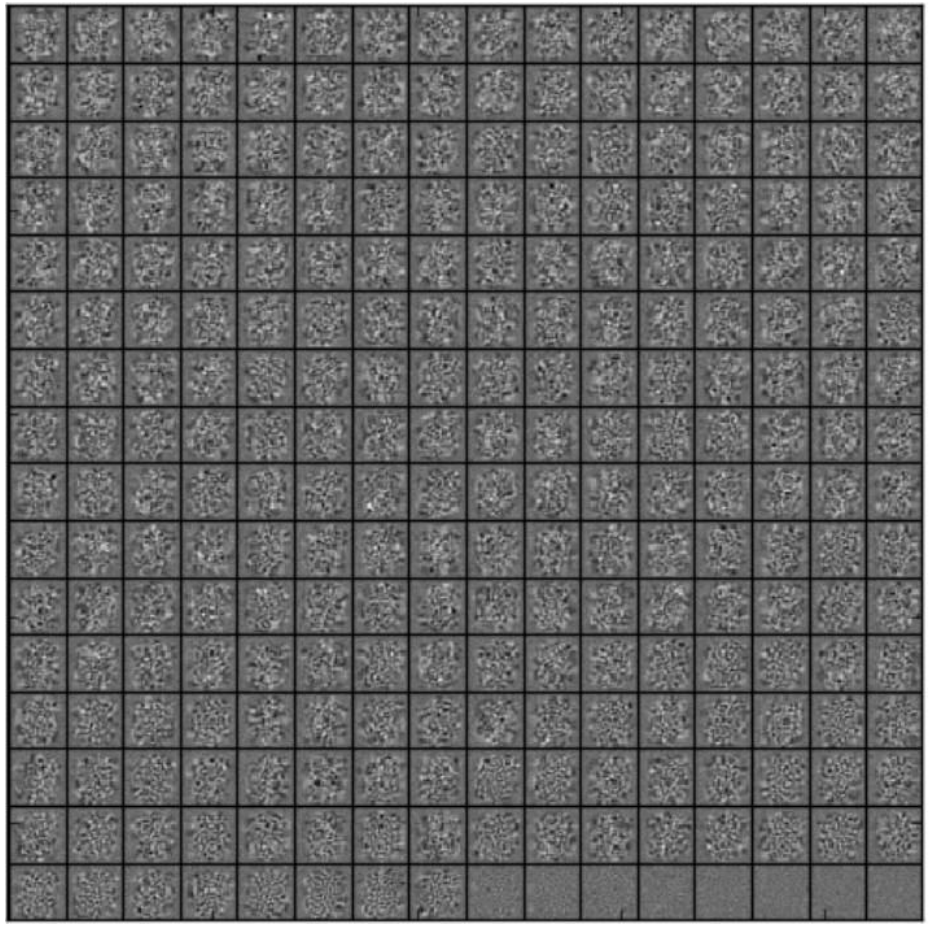


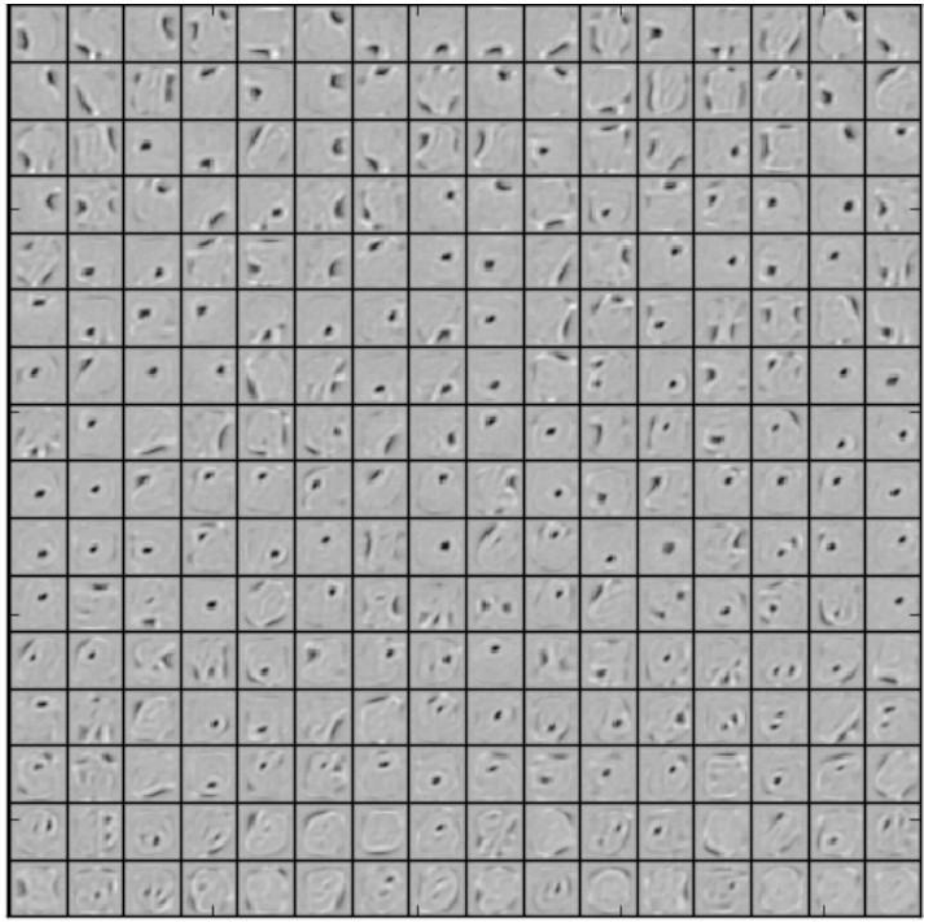
Why is this a good idea?

- Forces the network to learn a **redundant representation** ⇒ regularization
- Reduces effective **model capacity** ⇒ larger models, longer training
- Prevents **co-adaptation** of features (units can't learn to undo output of others)
- Requires only **one forward pass at inference** time

Srivastava, Hinton, Krizhevsky, Sutskever and Salakhutdinov: Dropout: a simple way to prevent neural networks from overfitting. JMLR, 2014.

Dropout



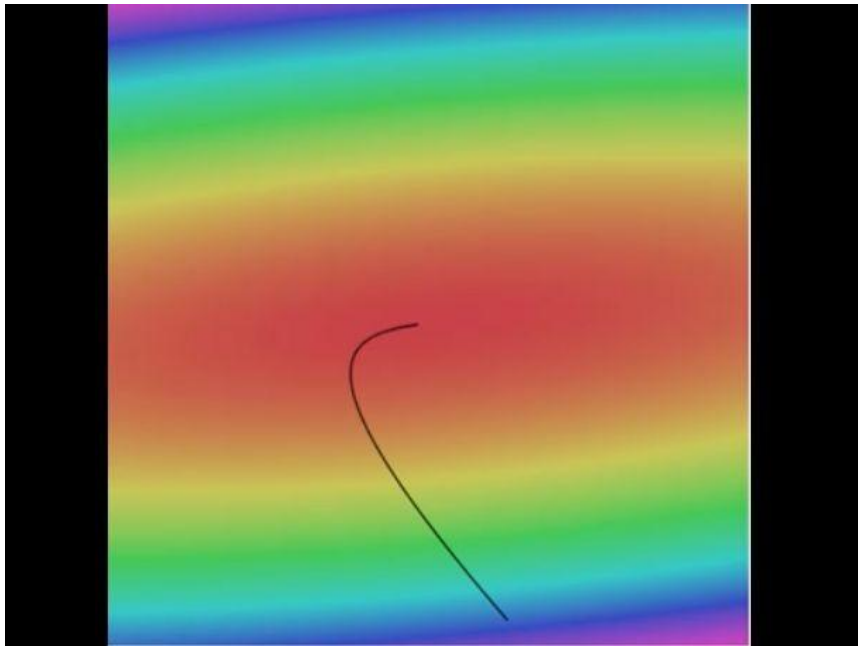


- Features of an Autoencoder on MNIST with a single hidden layer of 256 ReLUs
- Right: With dropout ⇒ less co-adaptation and thus better generalization

Srivastava, Hinton, Krizhevsky, Sutskever and Salakhutdinov: Dropout: a simple way to prevent neural networks from overfitting. JMLR, 2014.

Optimization

Finding the best W: Optimize with Gradient Descent




```
# Vanilla Gradient Descent

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

Follow the slope

In 1-dimension, the derivative of a function:

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

In multiple dimensions, the **gradient** is the vector of (partial derivatives) along each dimension

The slope in any direction is the **dot product** of the direction with the gradient
The direction of steepest descent is the **negative gradient**

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]

loss 1.25347

gradient dW:

[?,
?,
?,
?,
?,
?,
?,
?,
?,…]

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]

loss 1.25347

W + h (first dim):

[0.34 + **0.0001**,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]

loss 1.25322

gradient dW:

[?,
?,
?,
?,
?,
?,
?,
?,
?,
?,…]

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]

loss 1.25347

W + h (first dim):

[0.34 + **0.0001**,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]

loss 1.25322

gradient dW:

[**-2.5**,
?,
?,

(1.25322 - 1.25347)/0.0001
= -2.5

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

?,
?,…]

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]

loss 1.25347

W + h (second dim):

[0.34,
-1.11 + **0.0001**,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]

loss 1.25353

gradient dW:

[-2.5,
?,
?,
?,
?,
?,
?,
?,
?,…]

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]

loss 1.25347

W + h (second dim):

[0.34,
-1.11 + **0.0001**,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]

loss 1.25353

gradient dW:

[-2.5,
0.6,
?,
?,

(1.25353 - 1.25347)/0.0001
= 0.6

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

?,…]

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]

loss 1.25347

W + h (third dim)**:**

[0.34,
-1.11,
0.78 + **0.0001**,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]

loss 1.25347

gradient dW:

[-2.5,
0.6,
?,
?,
?,
?,
?,
?,
?,…]

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]

loss 1.25347

W + h (third dim):

[0.34,
-1.11,
0.78 + **0.0001**,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]

loss 1.25347

gradient dW:

[-2.5,
0.6,
0,
?,
?,…]

(1.25347 - 1.25347)/0.0001
= 0

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]

loss 1.25347

W + h (third dim):

[0.34,
-1.11,
0.78 + **0.0001**,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]

loss 1.25347

gradient dW:

[-2.5,
0.6,
0,
?,
?,
?,…]

Numeric Gradient

- Slow! Need to loop over all dimensions
- Approximate

The loss is just a function of W:

$$L = \frac{1}{N}\sum_{i=1}^{N} L_i + \sum_k W_k^2$$

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

$$s = f(x; W) = Wx$$

want $\nabla_W L$

The loss is just a function of W:

$$L = \frac{1}{N}\sum_{i=1}^{N} L_i + \sum_k W_k^2$$

$$L_i = -\log\left(\frac{e^{sy_i}}{\sum_j e^{s_j}}\right)$$

$$s = f(x; W) = Wx$$

want $\nabla_W L$

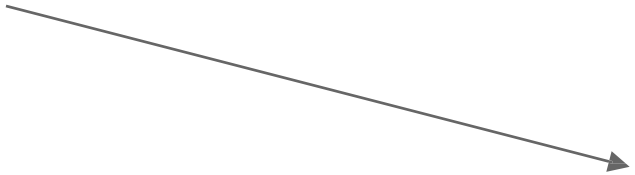
Use calculus to compute an
analytic gradient

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]

loss 1.25347

dW = ...
(some function
data and W)



gradient dW:

[-2.5,
0.6,
0,
0.2,
0.7,
-0.5,
1.1,
1.3,
-2.1,…]

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)

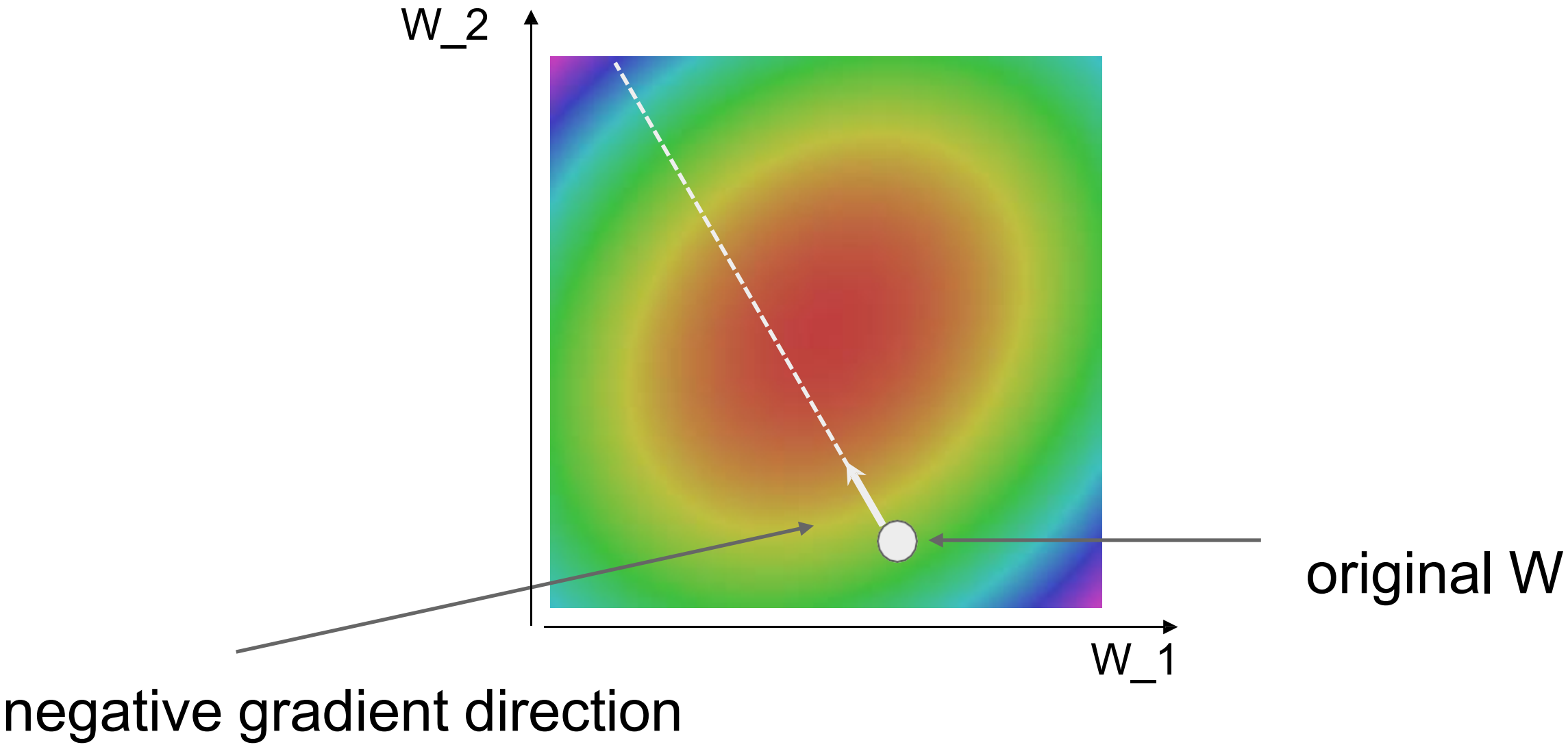
$$L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i(x_i, y_i, W) + \lambda R(W)$$

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

Full sum expensive when N is large!

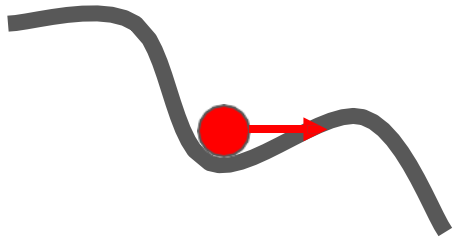
Approximate sum using a **minibatch** of examples
32 / 64 / 128 common

```
# 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
```

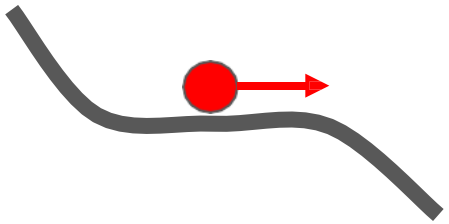

W_2
W_1
original W
negative gradient direction

SGD + Momentum

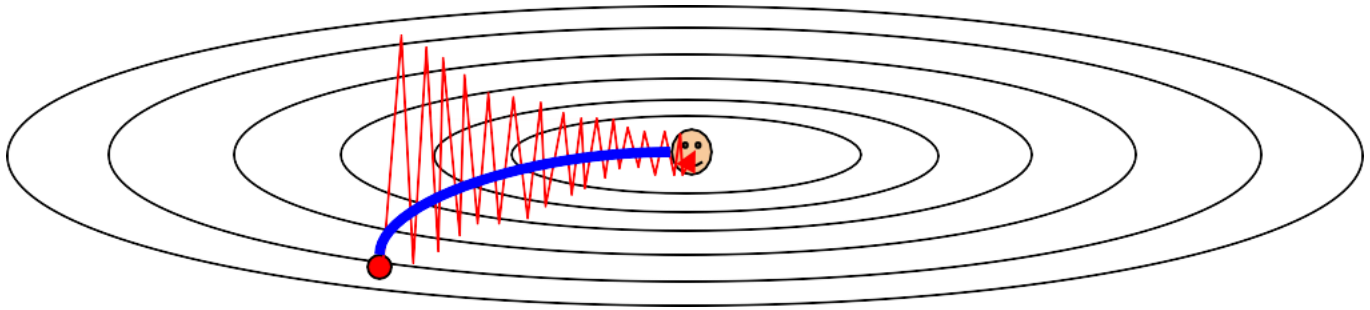
Local Minima



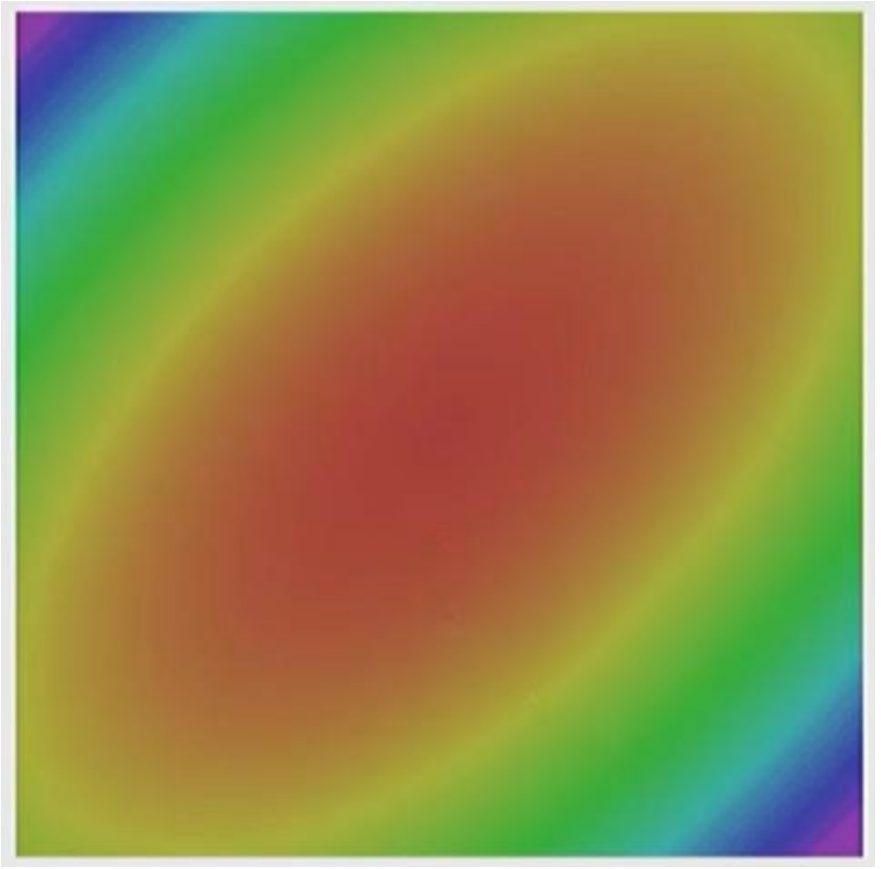
Saddle points



Poor Conditioning



Gradient Noise



SGD SGD+Momentum

SGD + Momentum:

continue moving in the general direction as the previous iterations

SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```
while True:
  dx = compute_gradient(x)
  x -= learning_rate * dx
```

SGD+Momentum

$$v_{t+1} = \rho v_t + \nabla f(x_t)$$

$$x_{t+1} = x_t - \alpha v_{t+1}$$

```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx + dx
  x -= learning_rate * vx
```

- Build up “velocity” as a running mean of gradients
- Rho gives “momentum”; typically rho=0.9 or 0.99

Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013

More Complex Optimizers: RMSProp

SGD + Momentum

```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx + dx
  x -= learning_rate * vx
```

Adds element-wise scaling of the gradient based on the historical sum of squares in each dimension (with decay)

RMSProp

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

Tieleman and Hinton, 2012

More Complex Optimizers: RMSProp

SGD + Momentum

```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx + dx
  x -= learning_rate * vx
```

"Per-parameter learning rates" or "adaptive learning rates"

RMSProp

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

Tieleman and Hinton, 2012

Optimizers: Adam (almost)

```
first_moment = 0
second_moment = 0
while True:
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  x -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7))
```

Kingma and Ba, “Adam: A method for stochastic optimization”, ICLR 2015

Adam (full form)

```
first_moment = 0
second_moment = 0
for t in range(1, num_iterations):
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  first_unbias = first_moment / (1 - beta1 ** t)
  second_unbias = second_moment / (1 - beta2 ** t)
  x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

Momentum

Bias correction

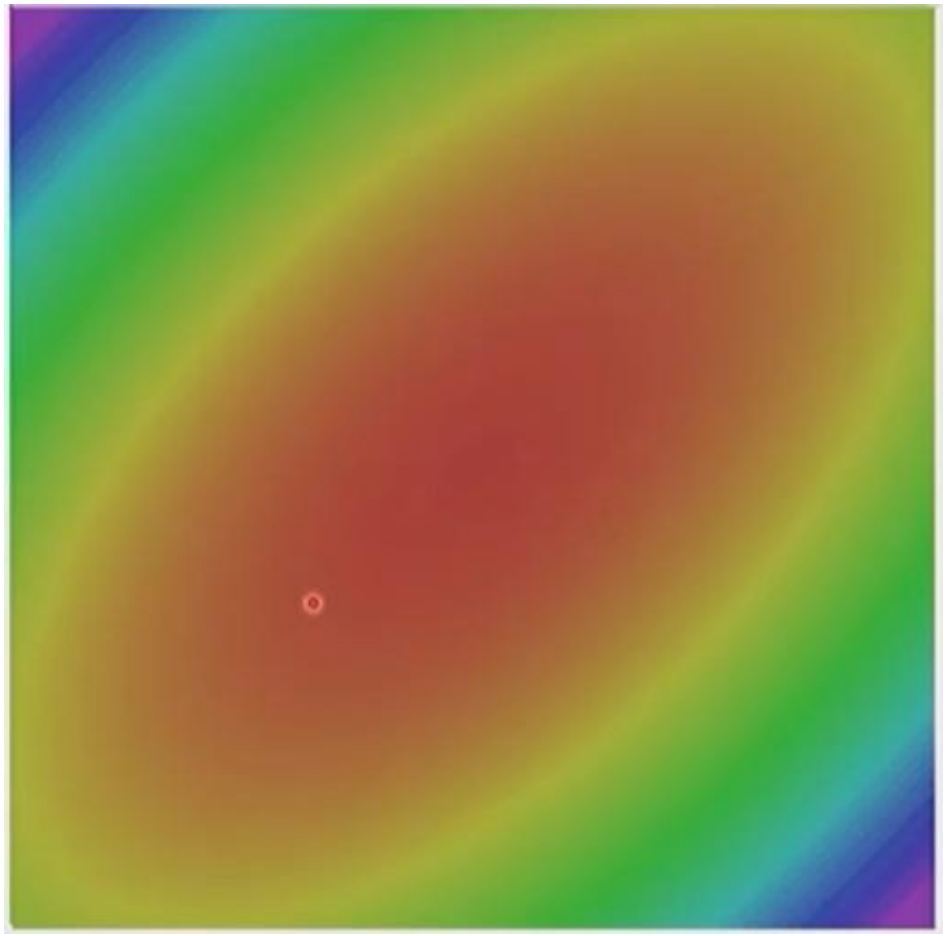
AdaGrad / RMSProp

Bias correction for the fact that first and second moment estimates start at zero

Adam with beta1 = 0.9, beta2 = 0.999, and learning_rate = 1e-3 or 5e-4 is a great starting point for many models!

Kingma and Ba, “Adam: A method for stochastic optimization”, ICLR 2015

Adam



SGD

SGD+Momentum

RMSProp

Adam

AdamW: Adam Variant with Weight Decay

Q: How does regularization interact with the optimizer? (e.g., L2)

```
first_moment = 0
second_moment = 0
for t in range(1, num_iterations):
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  first_unbias = first_moment / (1 - beta1 ** t)
  second_unbias = second_moment / (1 - beta2 ** t)
  x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

AdamW (Weight Decay) adds term here

Computed after the moments!

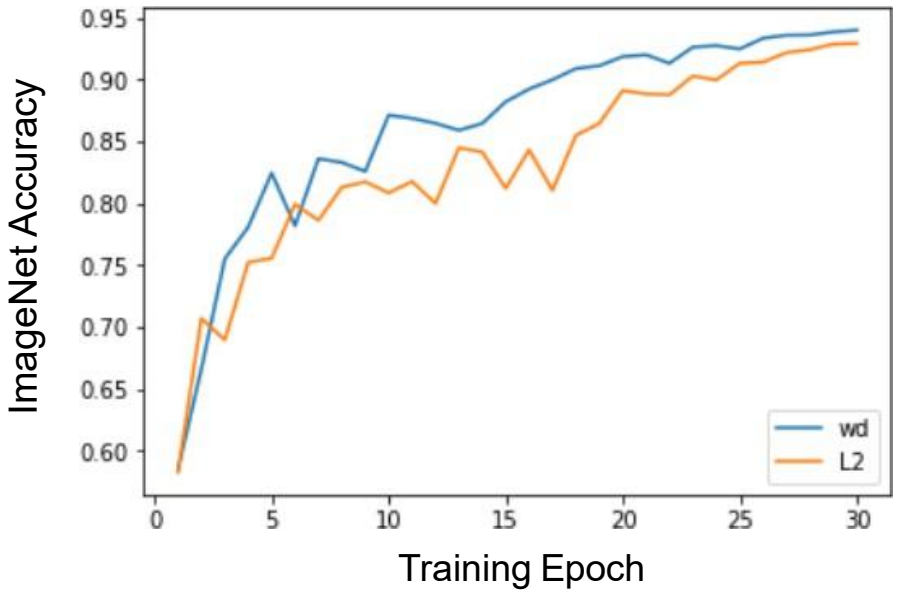
AdamW: Adam Variant with Weight Decay

Q: How does regularization interact with the optimizer? (e.g., L2)

Standard Adam computes L2 here

```
first_moment = 0
second_moment = 0
for t in range(1, num_iterations):
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  first_unbias = first_moment / (1 - beta1 ** t)
  second_unbias = second_moment / (1 - beta2 ** t)
  x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

AdamW (Weight Decay) adds term here



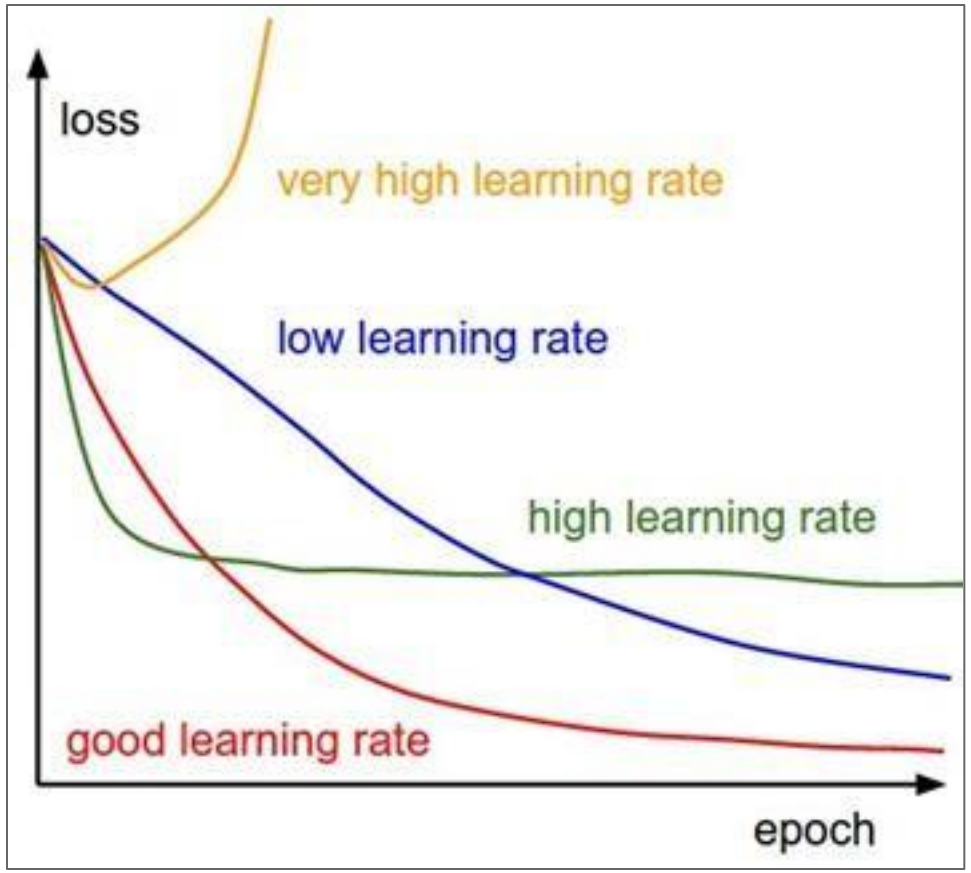
Source: https://www.fast.ai/posts/2018-07-02-adam-weight-decay.html

Learning rate schedules

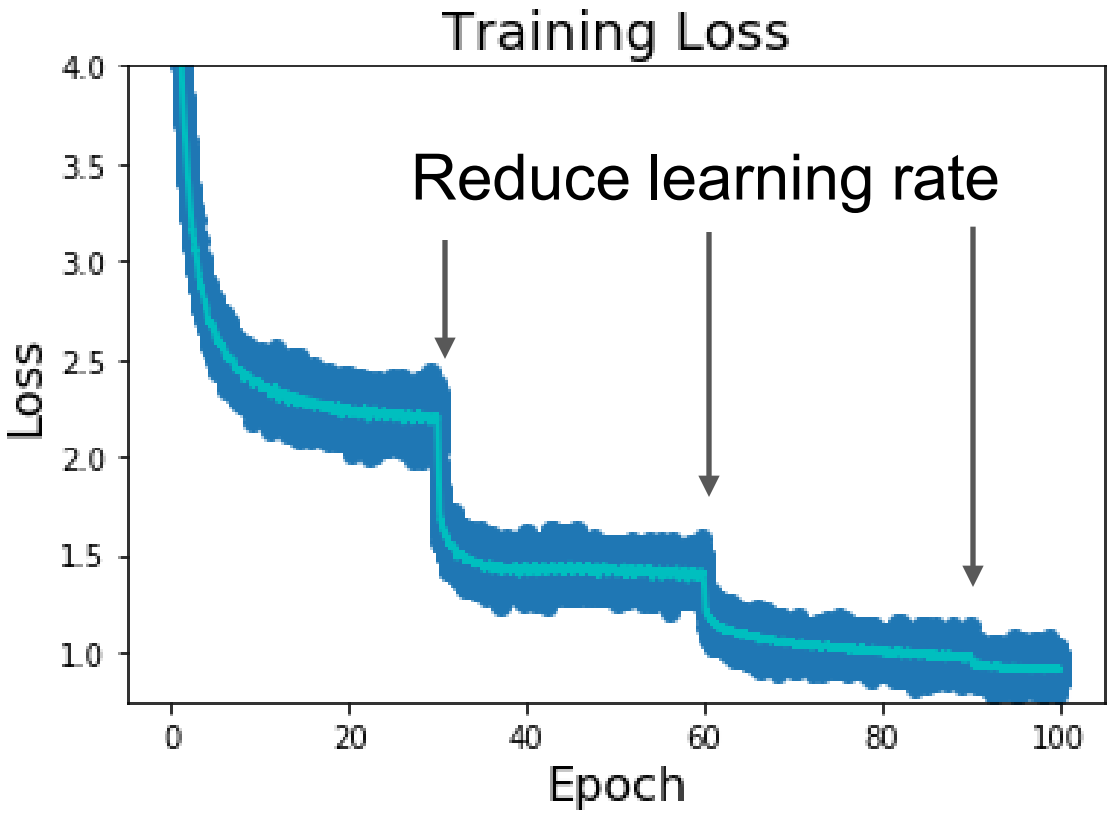
```
# Vanilla Gradient Descent

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

Learning rate

SGD, SGD+Momentum, RMSProp, Adam, AdamW all have **learning rate** as a hyperparameter.

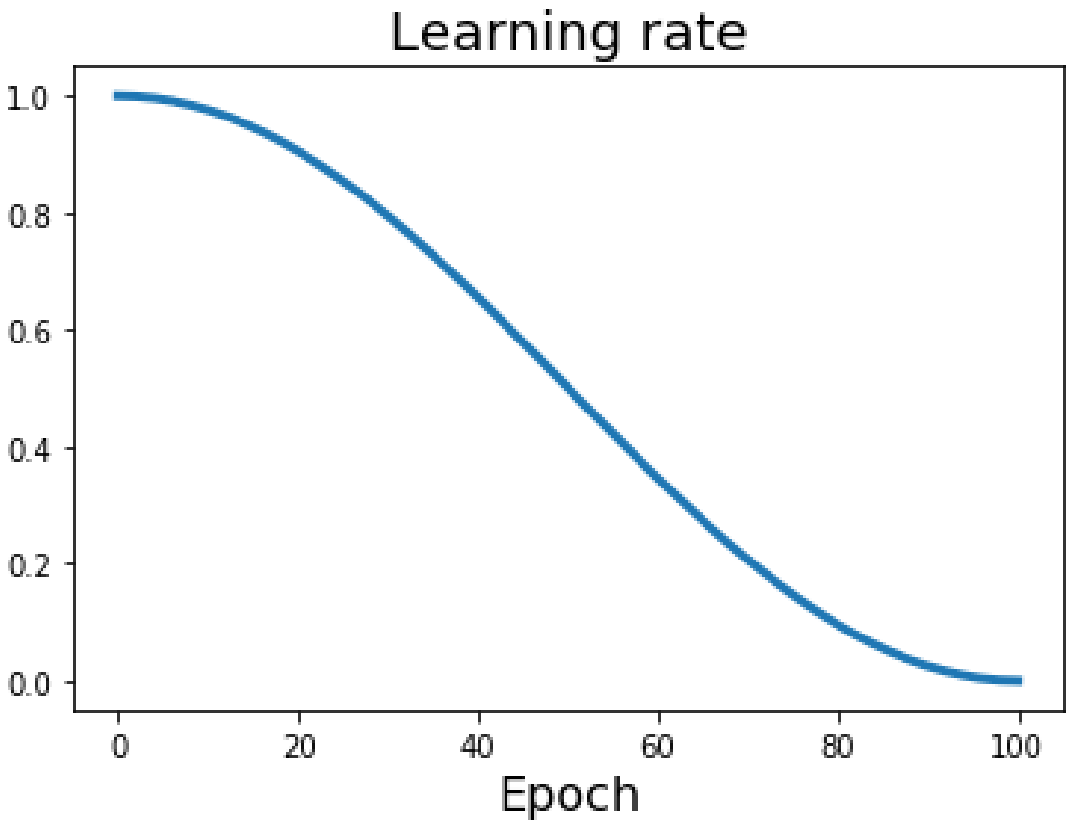


Learning rate



Step: Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

Learning rate



Step: Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

Cosine:

$$\alpha_t = \frac{1}{2}\alpha_0 \left(1 + \cos(t\pi/T)\right)$$

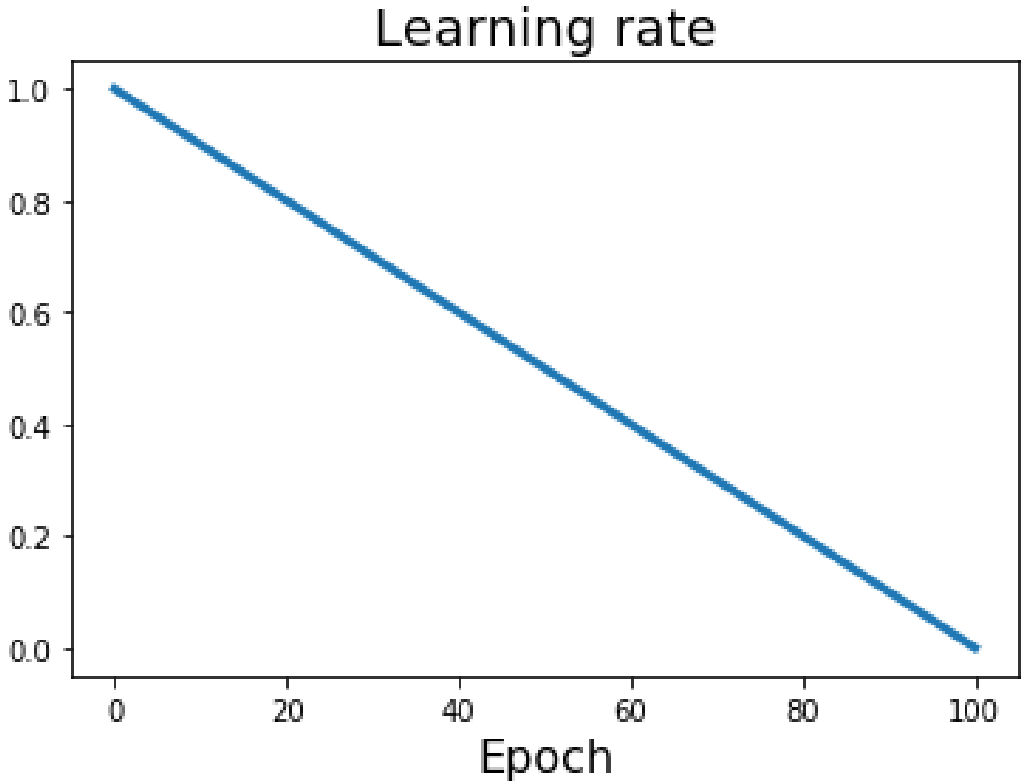
α_0 : Initial learning rate

α_t : Learning rate at epoch t

T : Total number of epochs

Loshchilov and Hutter, “SGDR: Stochastic Gradient Descent with Warm Restarts”, ICLR 2017
Radford et al, “Improving Language Understanding by Generative Pre-Training”, 2018
Feichtenhofer et al, “SlowFast Networks for Video Recognition”, arXiv 2018
Child at al, “Generating Long Sequences with Sparse Transformers”, arXiv 2019

Learning rate



Step: Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

Cosine:

Linear:

$$\alpha_t = \frac{1}{2}\alpha_0 \left(1 + \cos(t\pi/T)\right)$$

$$\alpha_t = \alpha_0(1 - t/T)$$

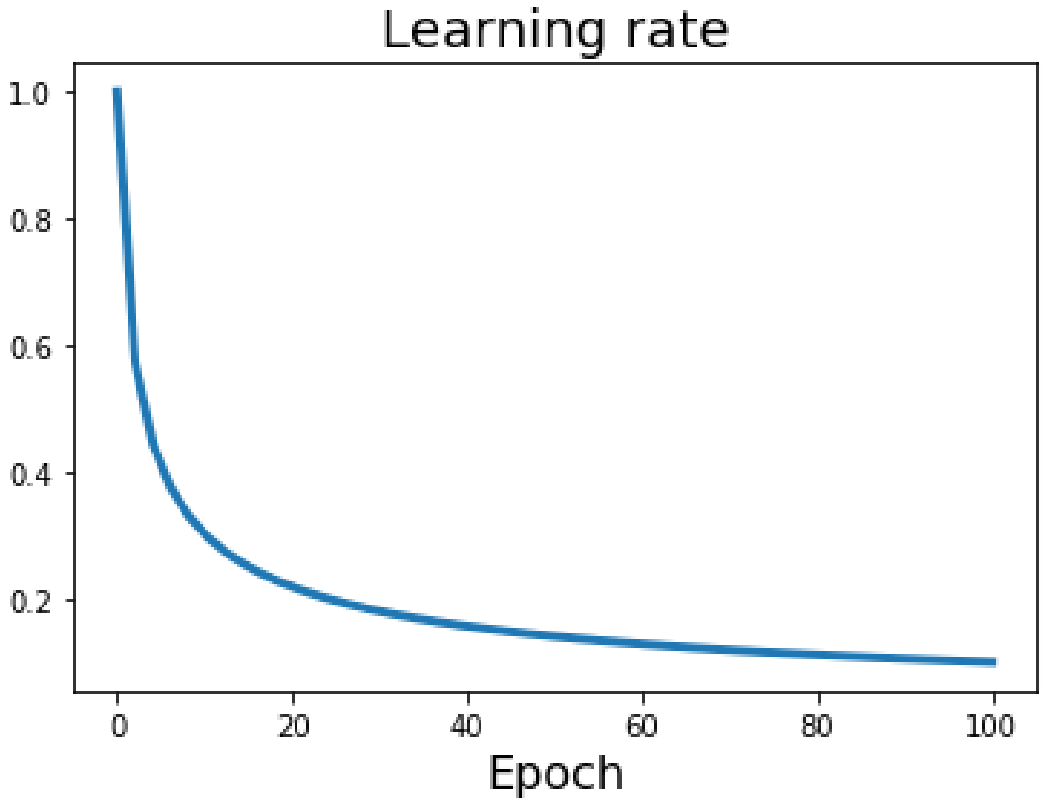
α_0 : Initial learning rate

α_t : Learning rate at epoch t

T : Total number of epochs

Devlin et al, “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”, 2018

Learning rate



Step: Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

Cosine: **Linear:**

$$\alpha_t = \frac{1}{2}\alpha_0 \left(1 + \cos(t\pi/T)\right)$$

$$\alpha_t = \alpha_0(1 - t/T)$$

Inverse sqrt:

$$\alpha_t = \alpha_0/\sqrt{t}$$

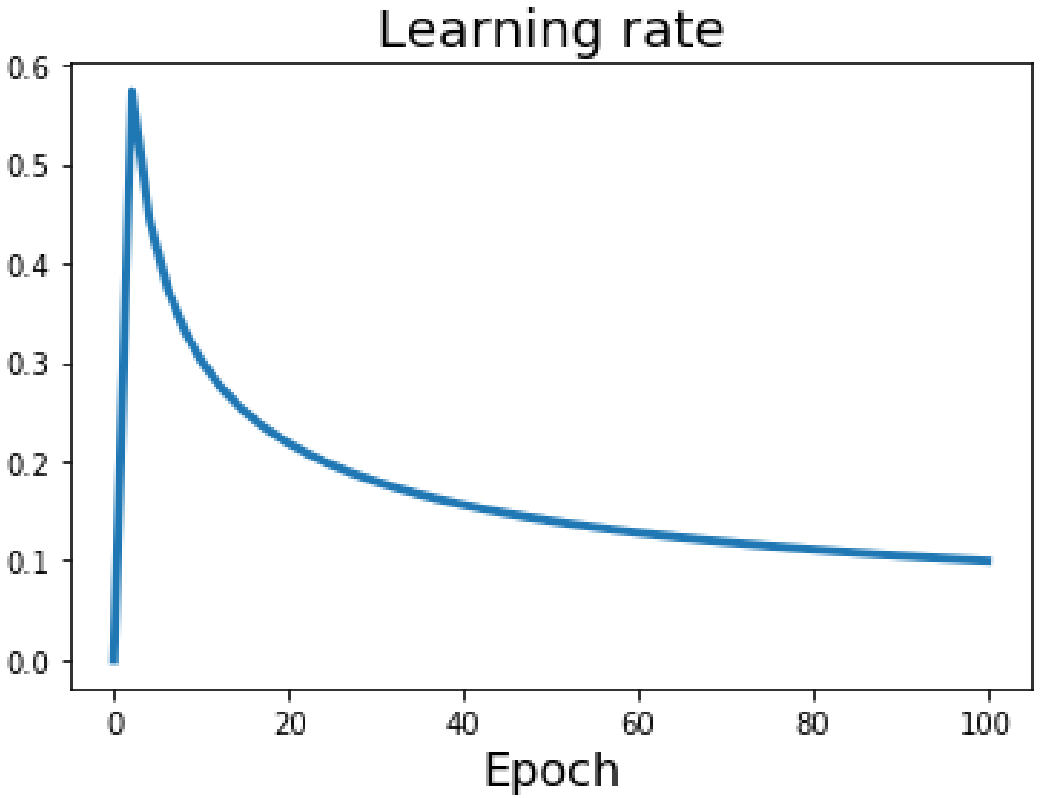
α_0 : Initial learning rate

α_t : Learning rate at epoch t

T : Total number of epochs

Vaswani et al, “Attention is all you need”, NIPS 2017

Learning Rate : Linear Warmup



High initial learning rates can make loss explode; linearly increasing learning rate from 0 over the first ~5,000 iterations can prevent this.

Goyal et al, “Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour”, arXiv 2017

Improving Deep Neural Networks

https://www.youtube.com/watch?v=V0MAIvkokW4

Image Classification

Image Classification: A core task in Computer Vision

(assume given a set of labels)
{dog, cat, truck, plane, ...}

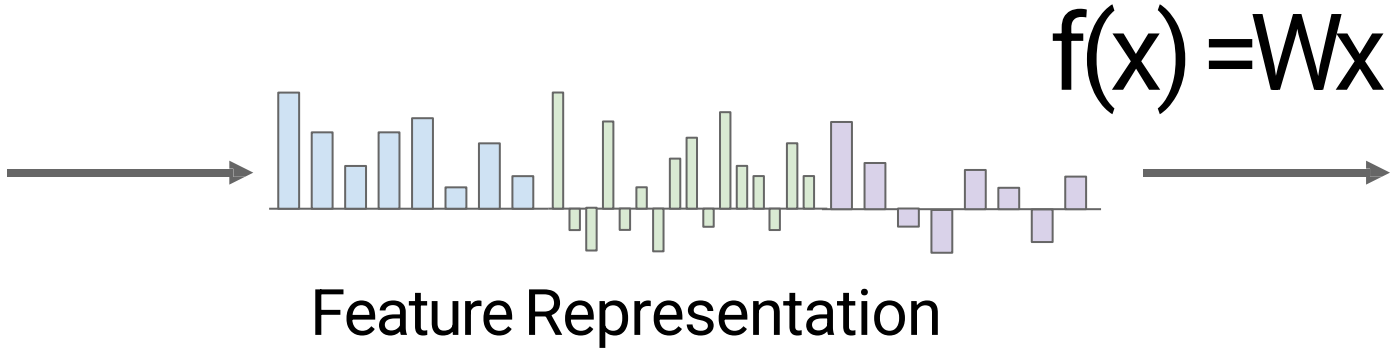
cat
dog
bird
deer
truck

Pixel space

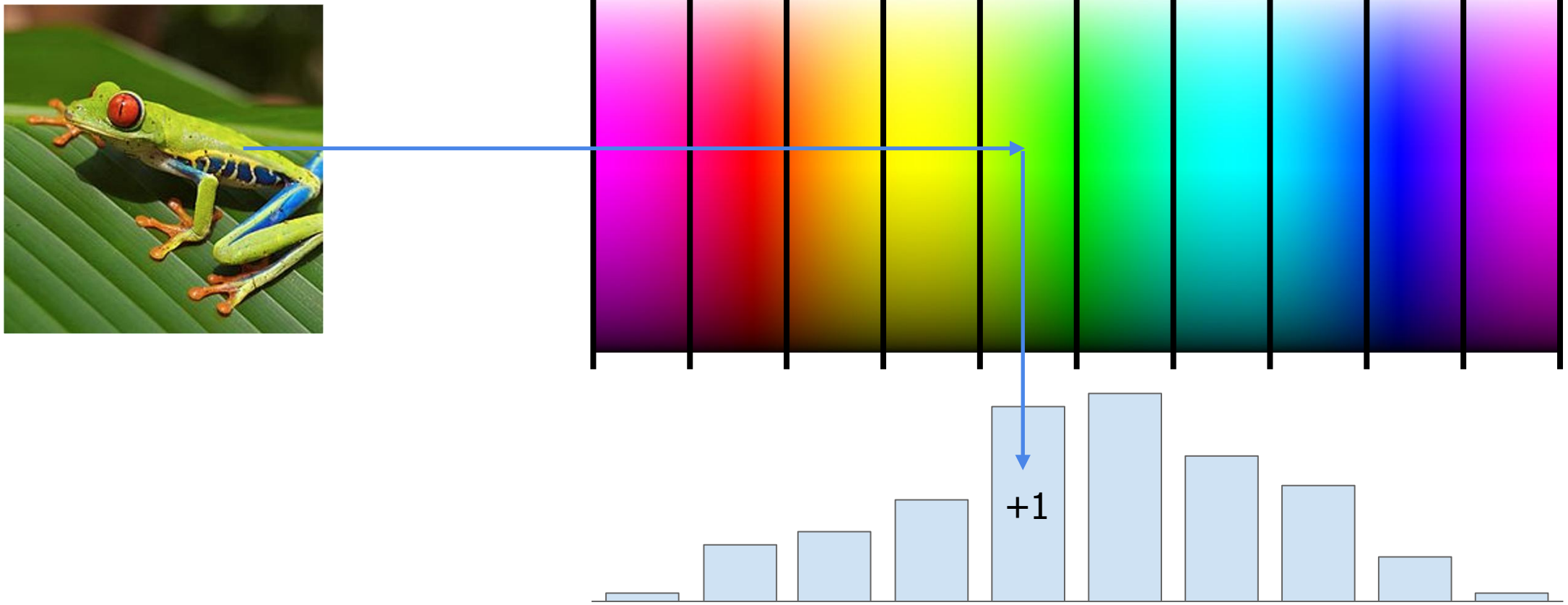
$$f(x) = Wx$$

Class scores

Image features

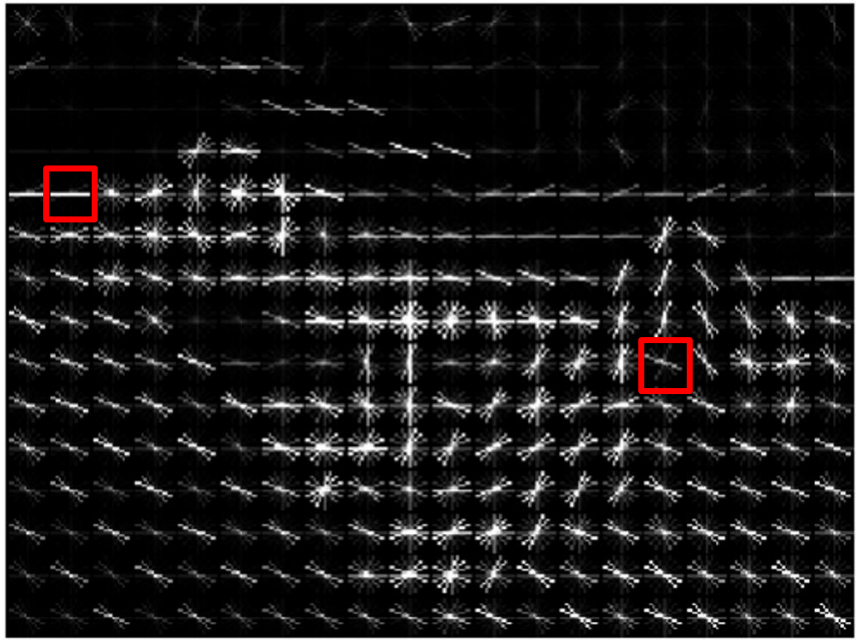


Example: Color Histogram



Example: Histogram of Oriented Gradients (HoG)

Divide image into 8x8 pixel regions Within each region quantize edge direction into 9 bins



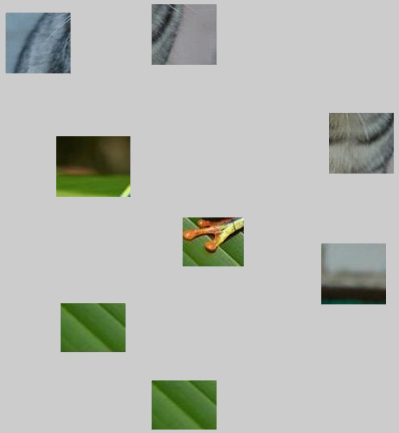
Example: 320x240 image gets divided into 40x30 bins; in each bin there are 9 numbers so feature vector has 30*40*9 = 10,800 numbers

Lowe, "Object recognition from local scale-invariant features", ICCV 1999
Dalal and Triggs, "Histograms of oriented gradients for human detection," CVPR 2005

Example: Bag of Words

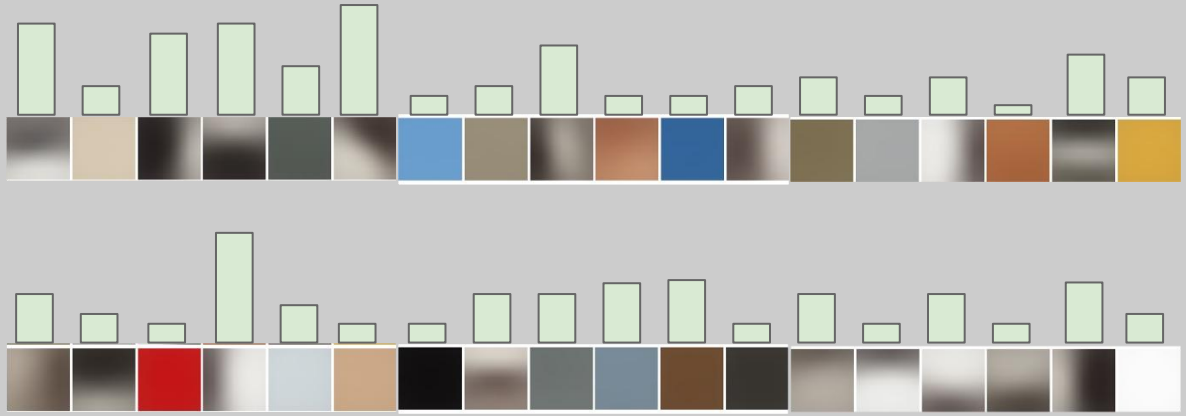
Step 1: Build codebook

Extract random patches



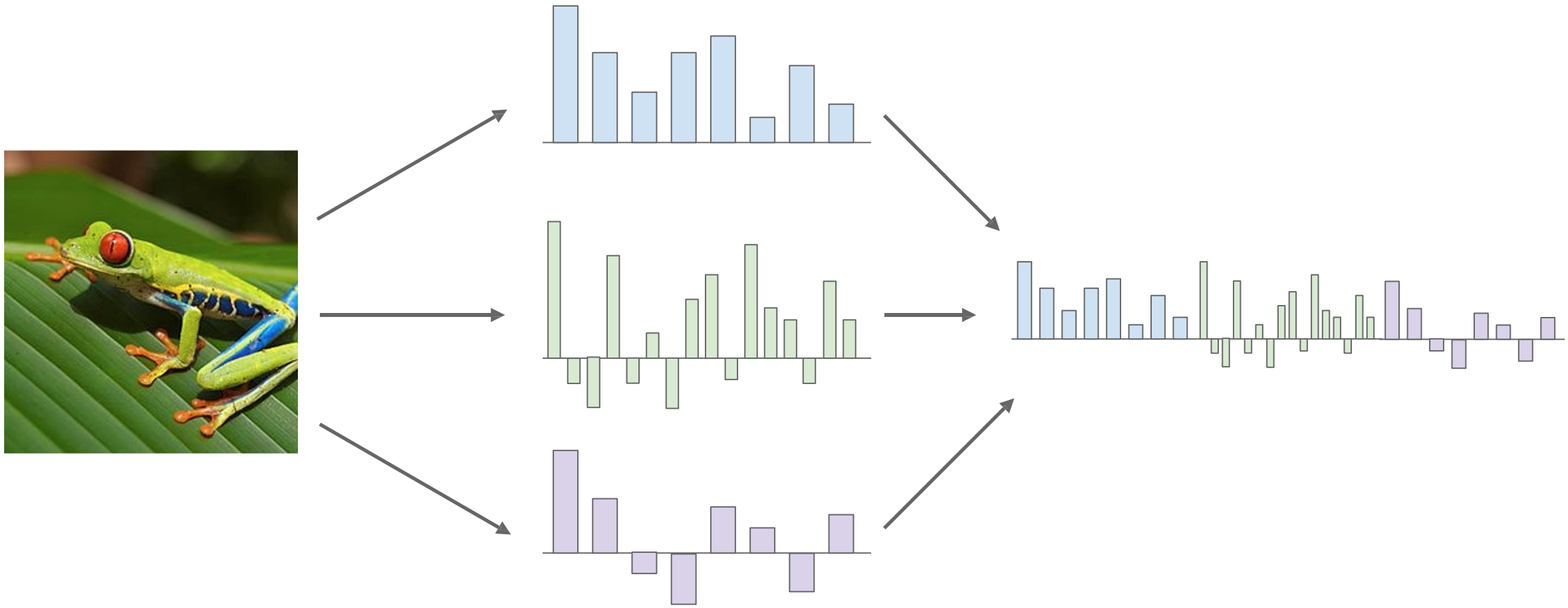
Cluster patches to form "codebook" of "visual words"

Step 2: Encode images



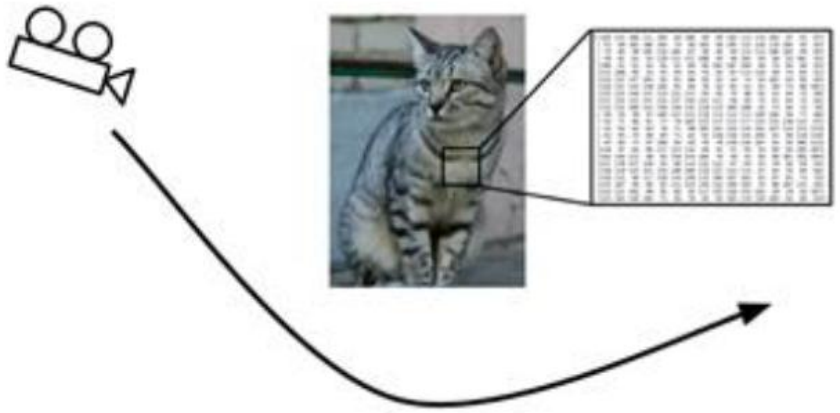
Fei-Fei and Perona, "A bayesian hierarchical model for learning natural scene categories", CVPR 2005

Image features



Challenges

Viewpoint



Illumination

Deformation

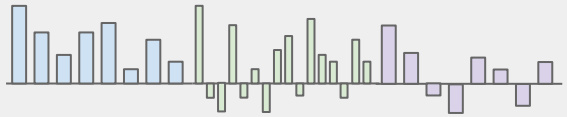
Occlusion

Clutter

Intraclass Variation

Image features vs. ConvNets

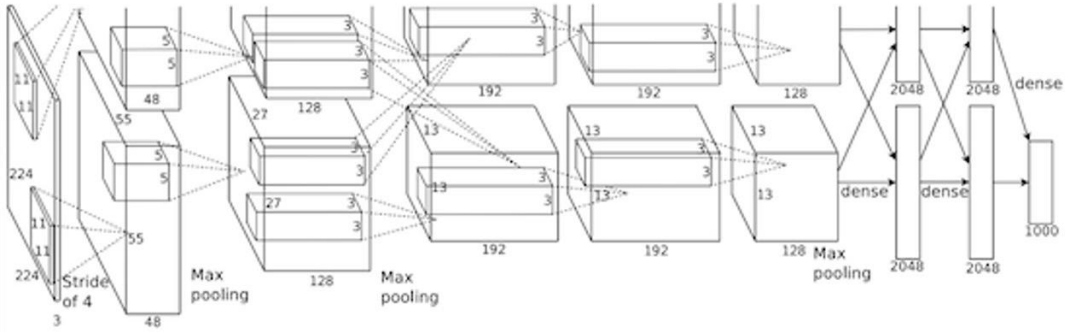
Feature Extraction



f

10 numbers giving scores for classes

training



Krizhevsky, Sutskever, and Hinton, "Imagenet classification with deep convolutional neural networks", NIPS 2012. Figure copyright Krizhevsky, Sutskever, and Hinton, 2012. Reproduced with permission.

10 numbers giving scores for classes

training

Convolutional Neural Networks

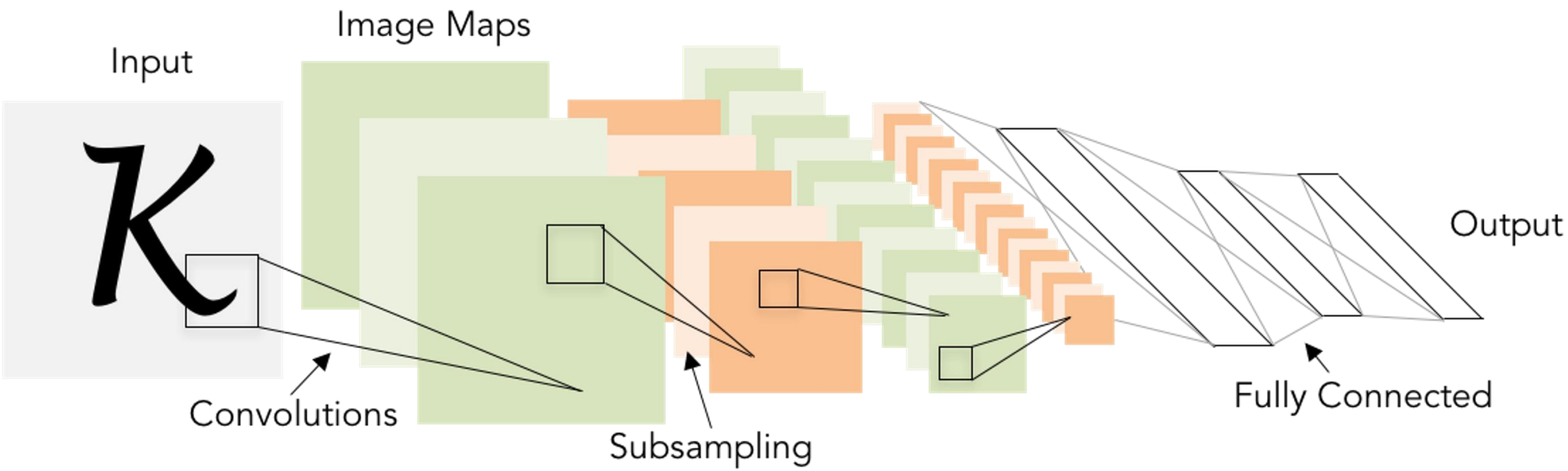


Illustration of LeCun et al. 1998 from CS231n 2017 Lecture 1

Convolutional Neural Networks

Convolution and **pooling** operators extract features while respecting 2D image structure

Fully-Connected layers form an MLP at the end to predict scores

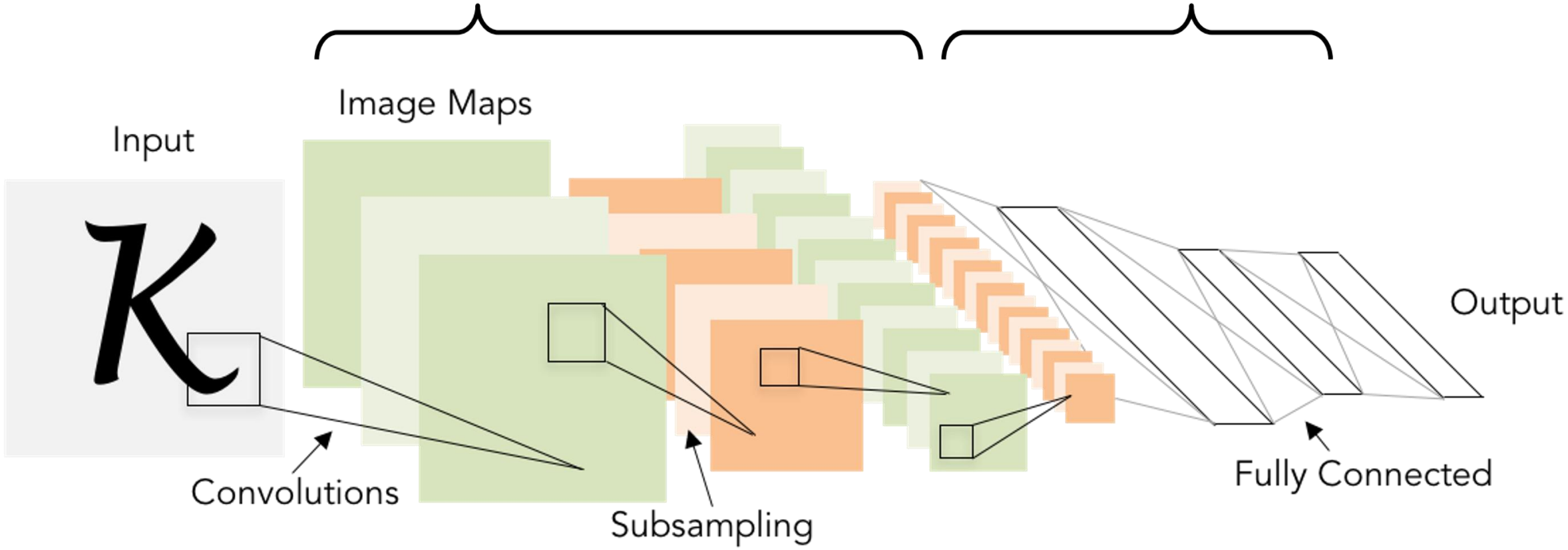


Illustration of LeCun et al. 1998 from CS231n 2017 Lecture 1

Convolutional Neural Networks

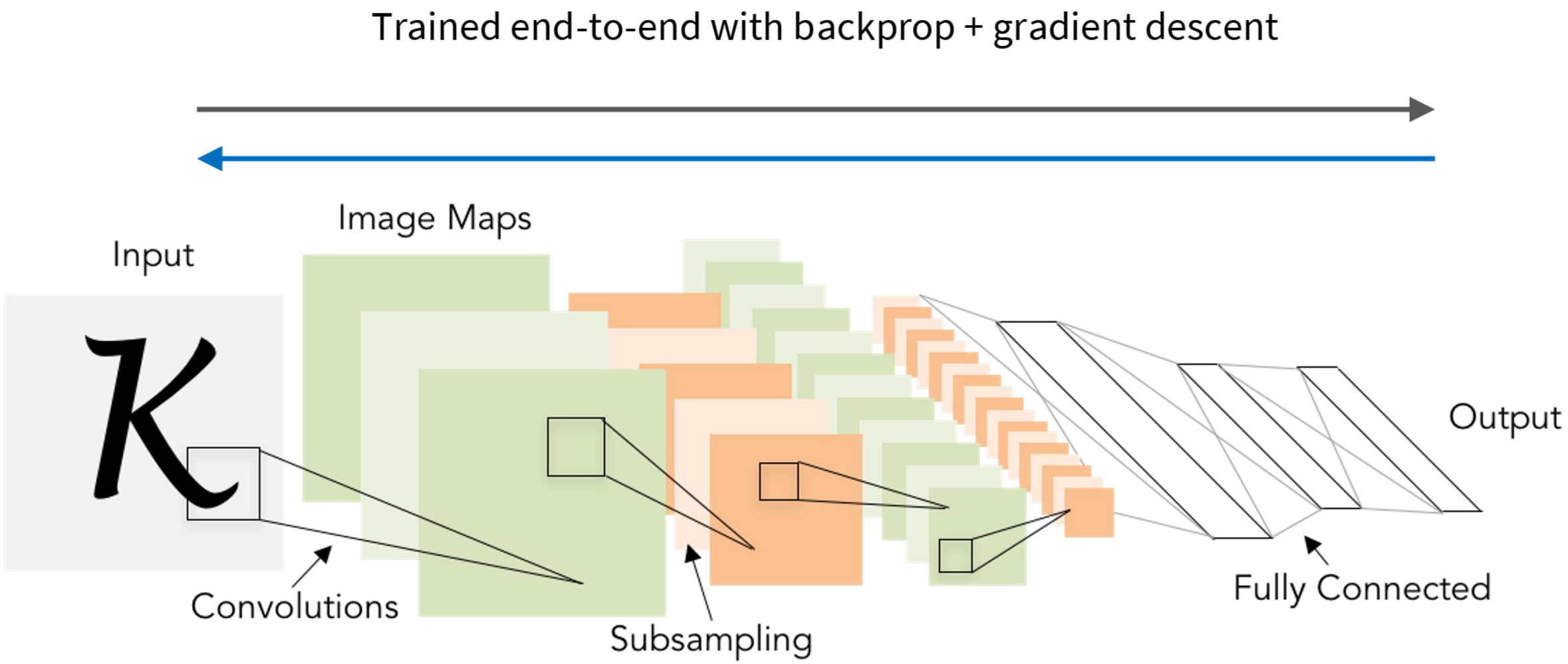
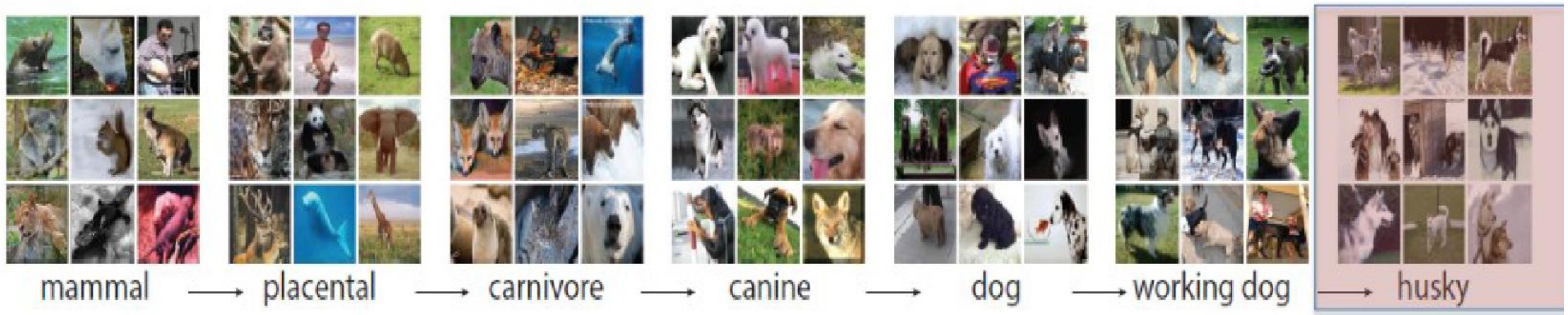


Illustration of LeCun et al. 1998 from CS231n 2017 Lecture 1

IMAGENET

Examples of hammer:

Dataset: ImageNet 2012

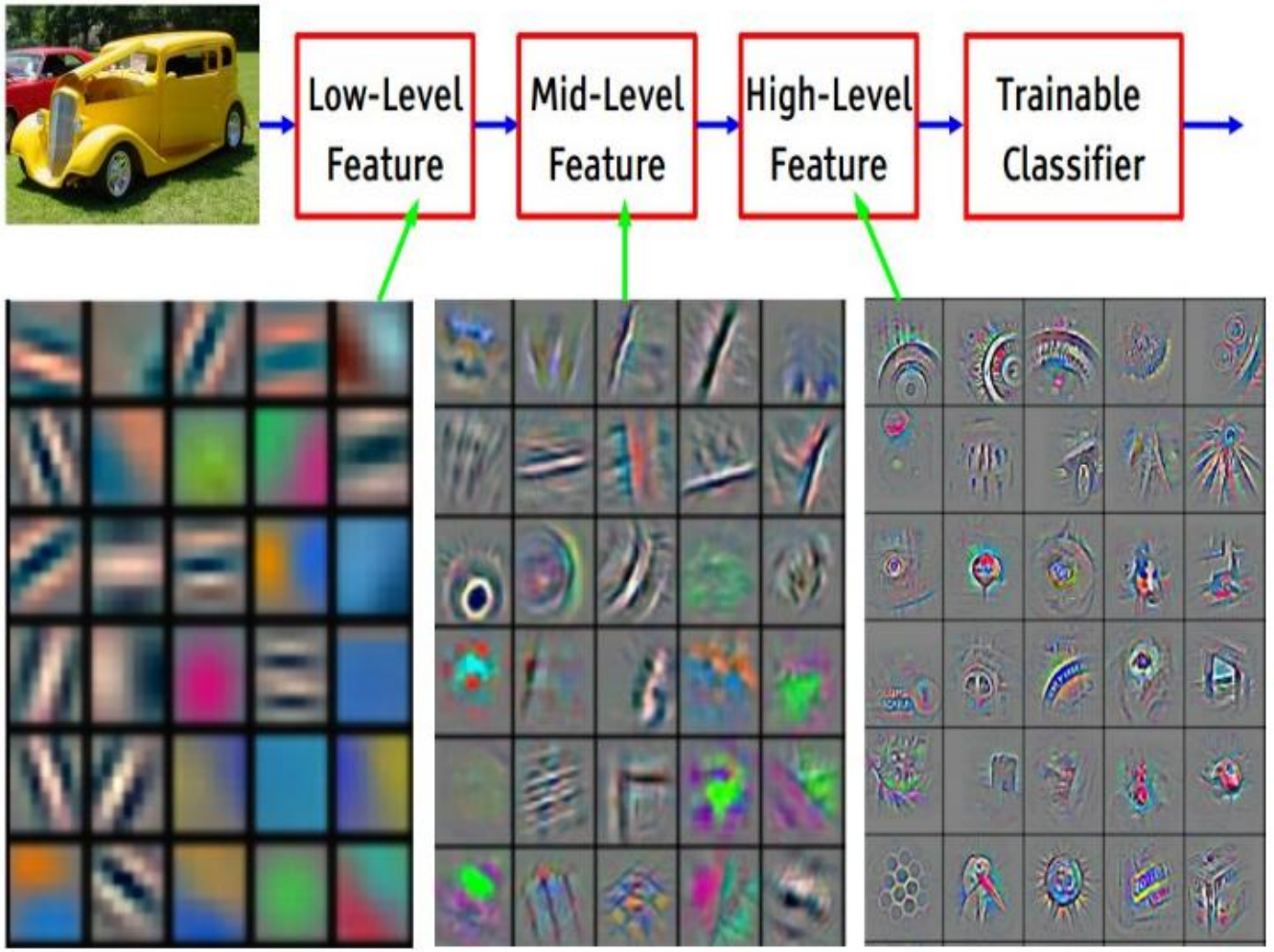


- S: (n) Eskimo dog, husky (breed of heavy-coated Arctic sled dog)
  - *direct hypernym / **inherited hypernym** / sister term*
    - S: (n) working dog (any of several breeds of usually large powerful dogs bred to work as draft animals and guard and guide dogs)
      - S: (n) dog, domestic dog, Canis familiaris (a member of the genus Canis (probably descended from the common wolf) that has been domesticated by man since prehistoric times; occurs in many breeds) *"the dog barked all night"*
        - S: (n) canine, canid (any of various fissiped mammals with nonretractile claws and typically long muzzles)
          - S: (n) carnivore (a terrestrial or aquatic flesh-eating mammal) *"terrestrial carnivores have four or five clawed digits on each limb"*
            - S: (n) placental, placental mammal, eutherian, eutherian mammal (mammals having a placenta; all mammals except monotremes and marsupials)
              - S: (n) mammal, mammalian (any warm-blooded vertebrate having the skin more or less covered with hair; young are born alive except for the small subclass of monotremes and nourished with milk)
                - S: (n) vertebrate, craniate (animals having a bony or cartilaginous skeleton with a segmented spinal column and a large brain enclosed in a skull or cranium)
                  - S: (n) chordate (any animal of the phylum Chordata having a notochord or spinal column)
                    - S: (n) animal, animate being, beast, brute, creature, fauna (a living organism characterized by voluntary movement)
                      - S: (n) organism, being (a living thing that has (or can develop) the ability to act or function independently)
                        - S: (n) living thing, animate thing (a living (or once living) entity)
                          - S: (n) whole, unit (an assemblage of parts that is regarded as a single entity) *"how big is that part compared to the whole?"; "the team is a unit"*
                            - S: (n) object, physical object (a tangible and visible entity; an entity that can cast a shadow) *"it was full of rackets, balls and other objects"*
                              - S: (n) physical entity (an entity that has physical existence)
                                - S: (n) entity (that which is perceived or known or inferred to have its own distinct existence (living or nonliving))

Deng et al. “Imagenet: a large scale hierarchical image database” CVPR 2009

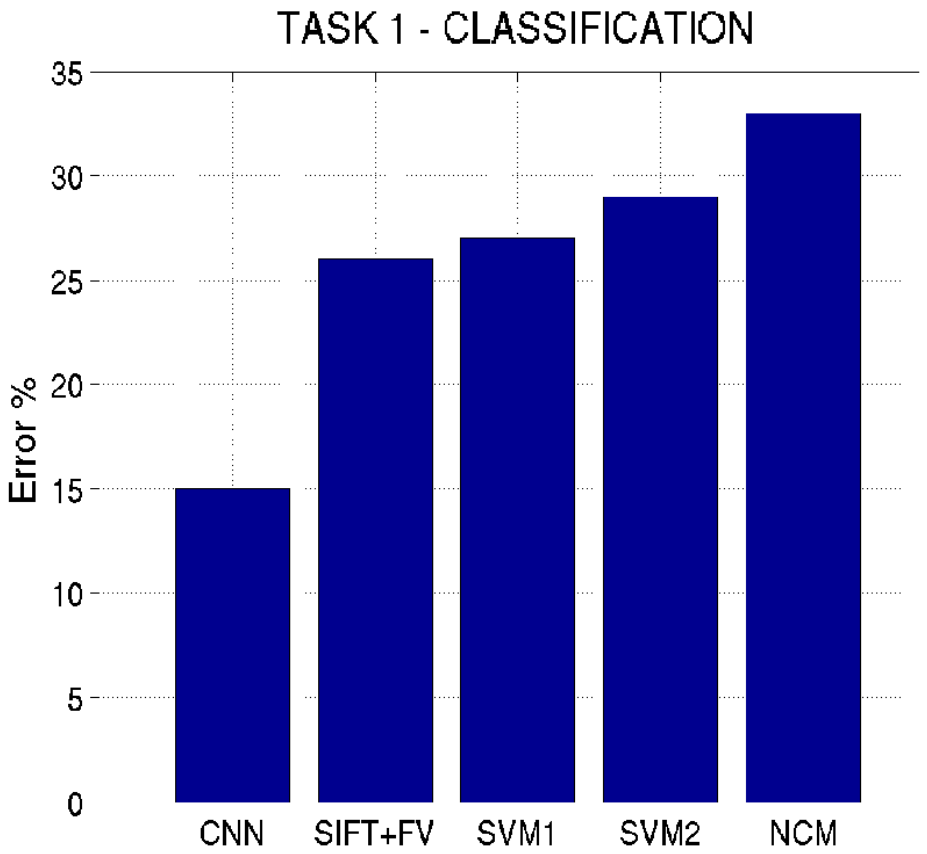
Preview

[From recent Yann LeCun slides]



Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]

ImageNet Large Scale Visual Recognition Challenge, 2012

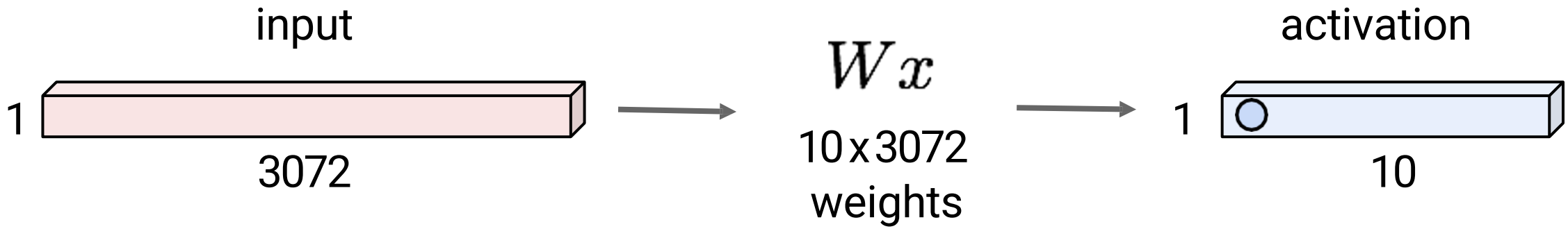


Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." Advances in Neural Information Processing Systems. 2012

Convolutional Neural Networks

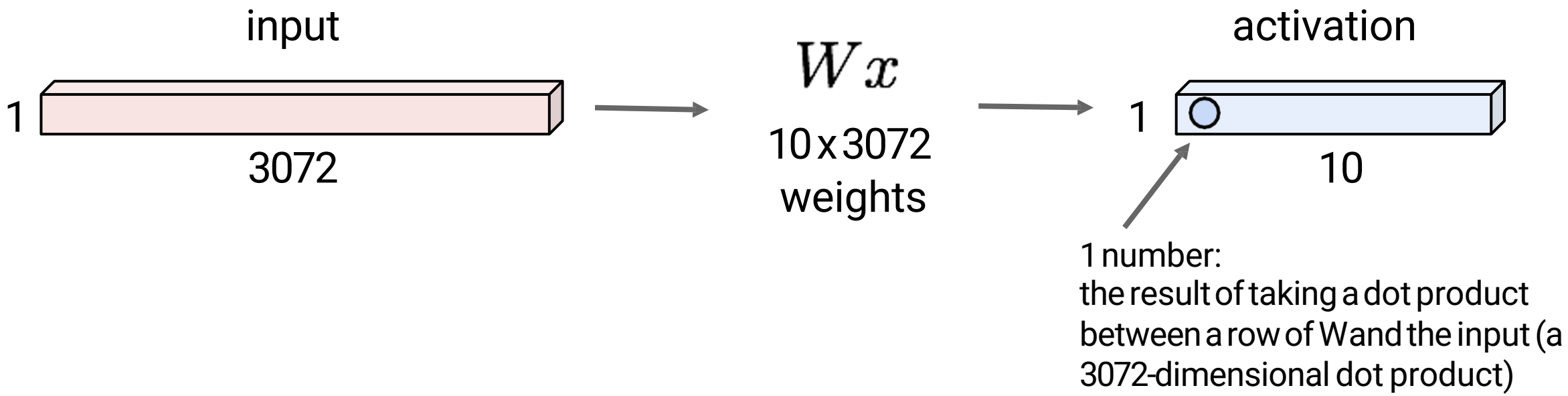
Recap: Fully Connected Layer

32x32x3 image ->stretch to 3072 x 1

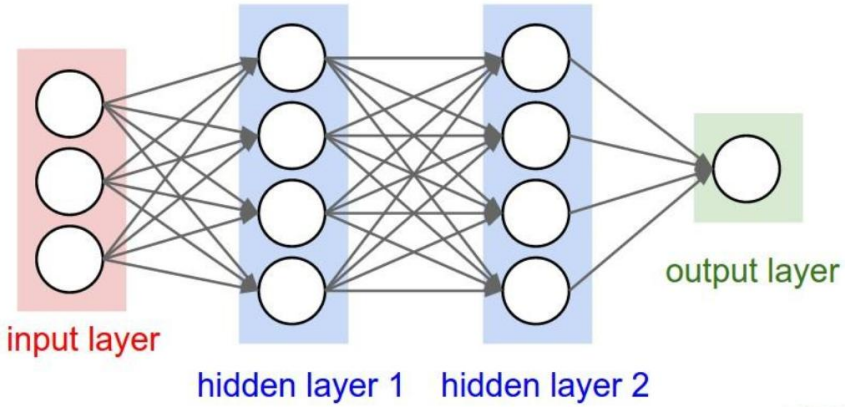


Fully Connected Layer

32x32x3 image ->stretch to 3072 x 1

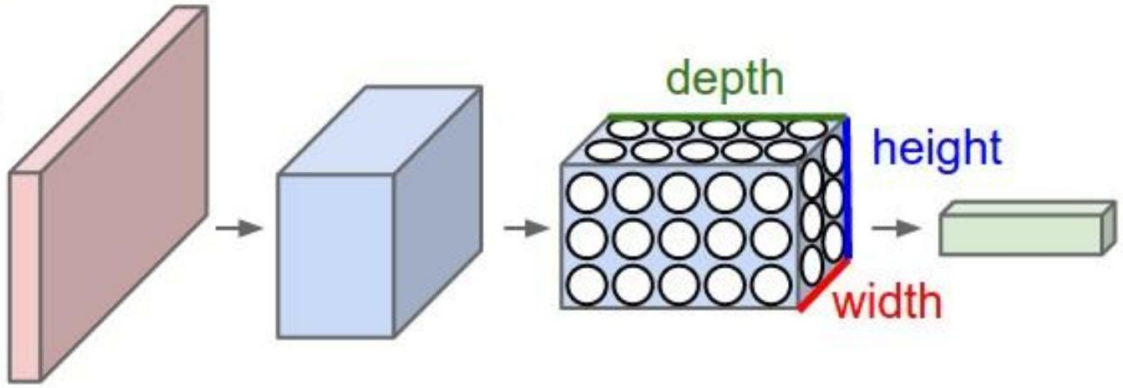


CONVNETS



Layers used to build ConvNets:

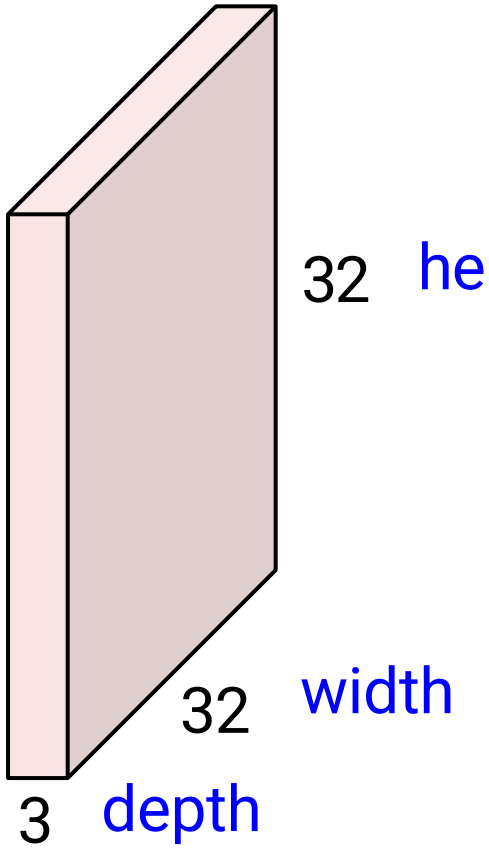
- a stacked sequence of layers. 3 main types
- Convolutional Layer, Pooling Layer, and Fully-Connected Layer



- every layer of a ConvNet transforms one volume of activations to another through a differentiable function.

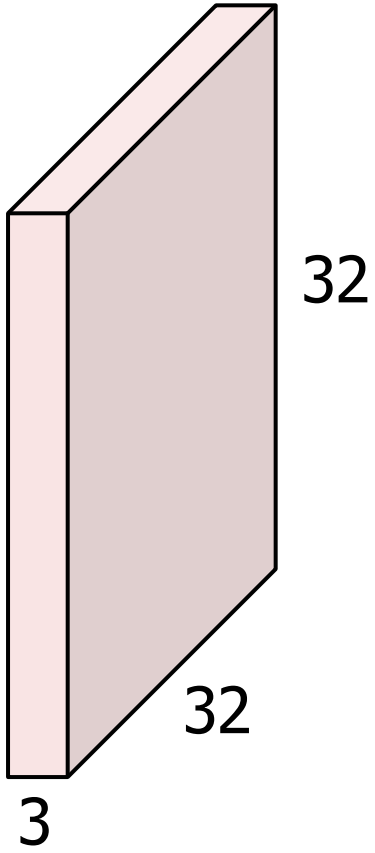
Convolution Layer

32x32x3 image ->preserve spatial structure

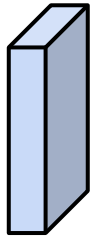


Convolution Layer

32x32x3 image



5x5x3 filter

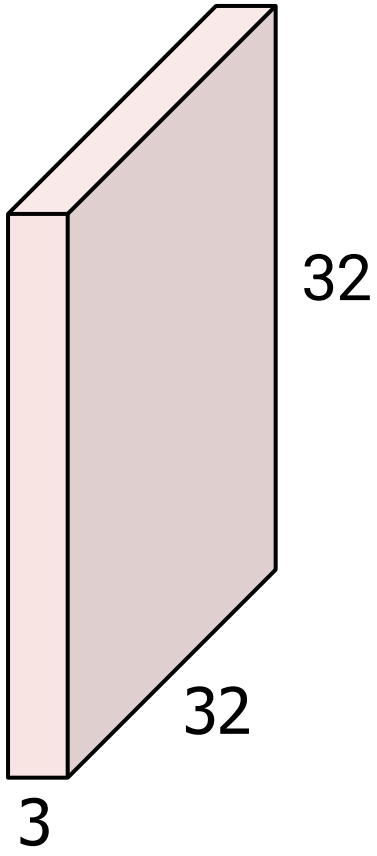


Convolve the filter with the image
i.e. “slide over the image spatially, computing dot products”

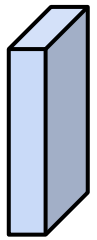
Convolution Layer

Filters always extend the full depth of the input volume

32x32x3 image

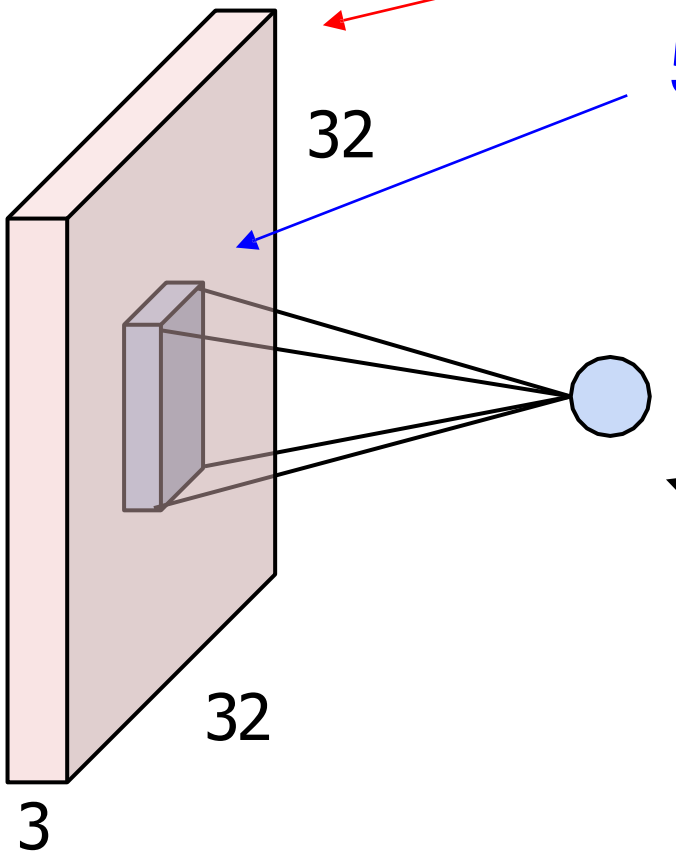


5x5x3 filter



Convolve the filter with the image i.e. “slide over the image spatially, computing dot products”

Convolution Layer



32x32x3 image

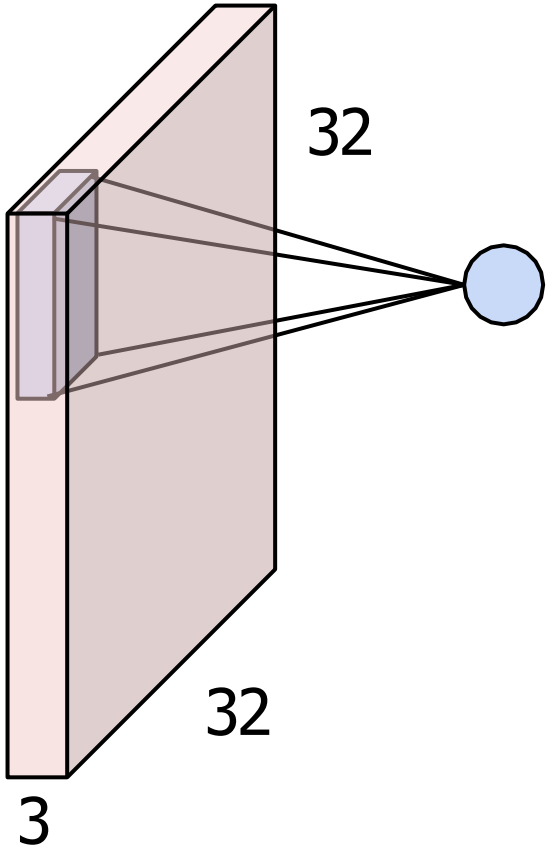
5x5x3 filter w

1 number:

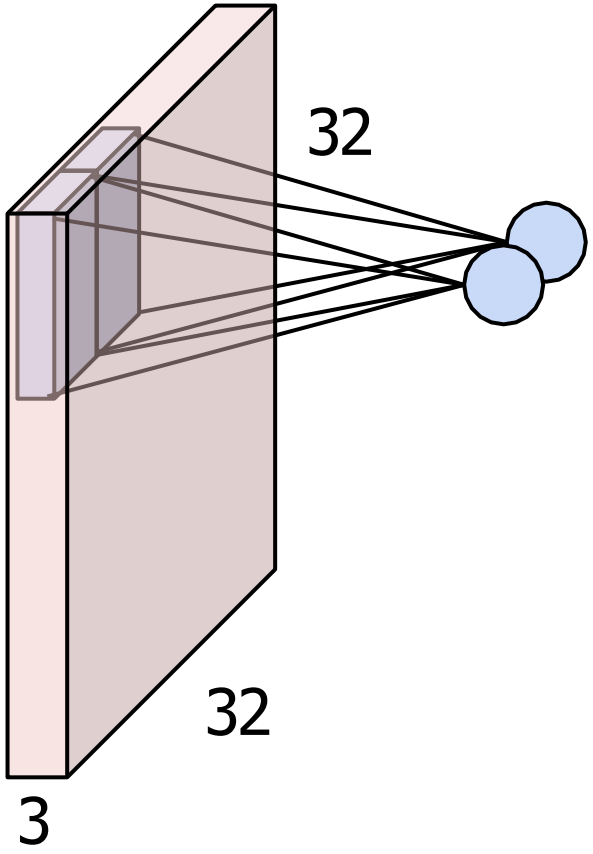
the result of taking a dot product between the filter and a small 5x5x3 chunk of the image (i.e. 5*5*3 =75-dimensional dot product +bias)

$$w^T x + b$$

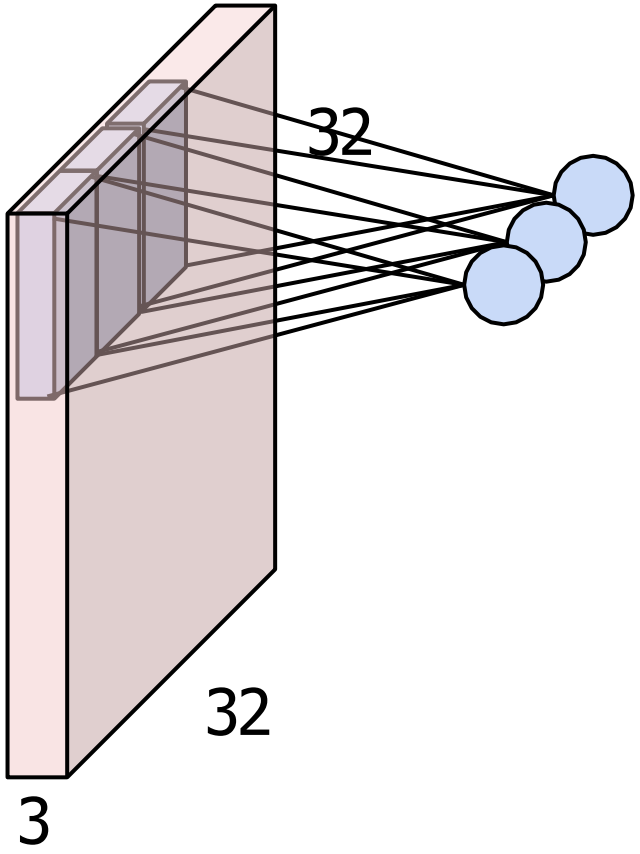
Convolution Layer



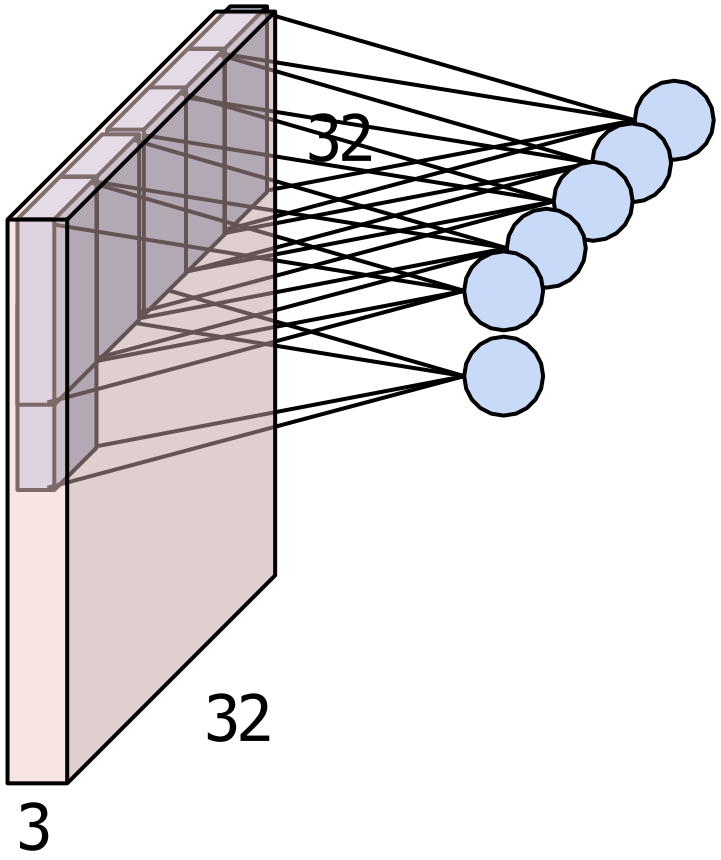
Convolution Layer



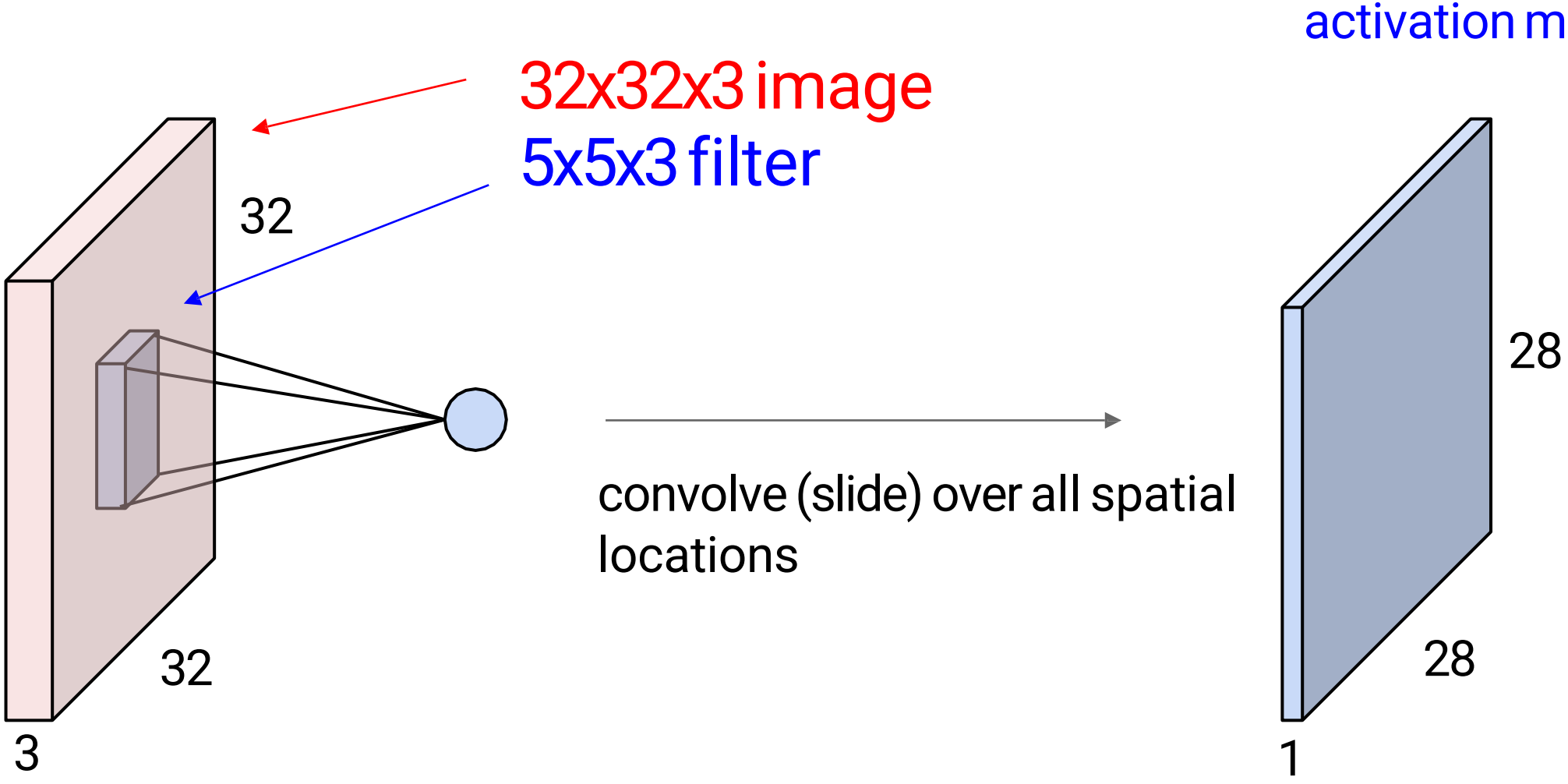
Convolution Layer



Convolution Layer

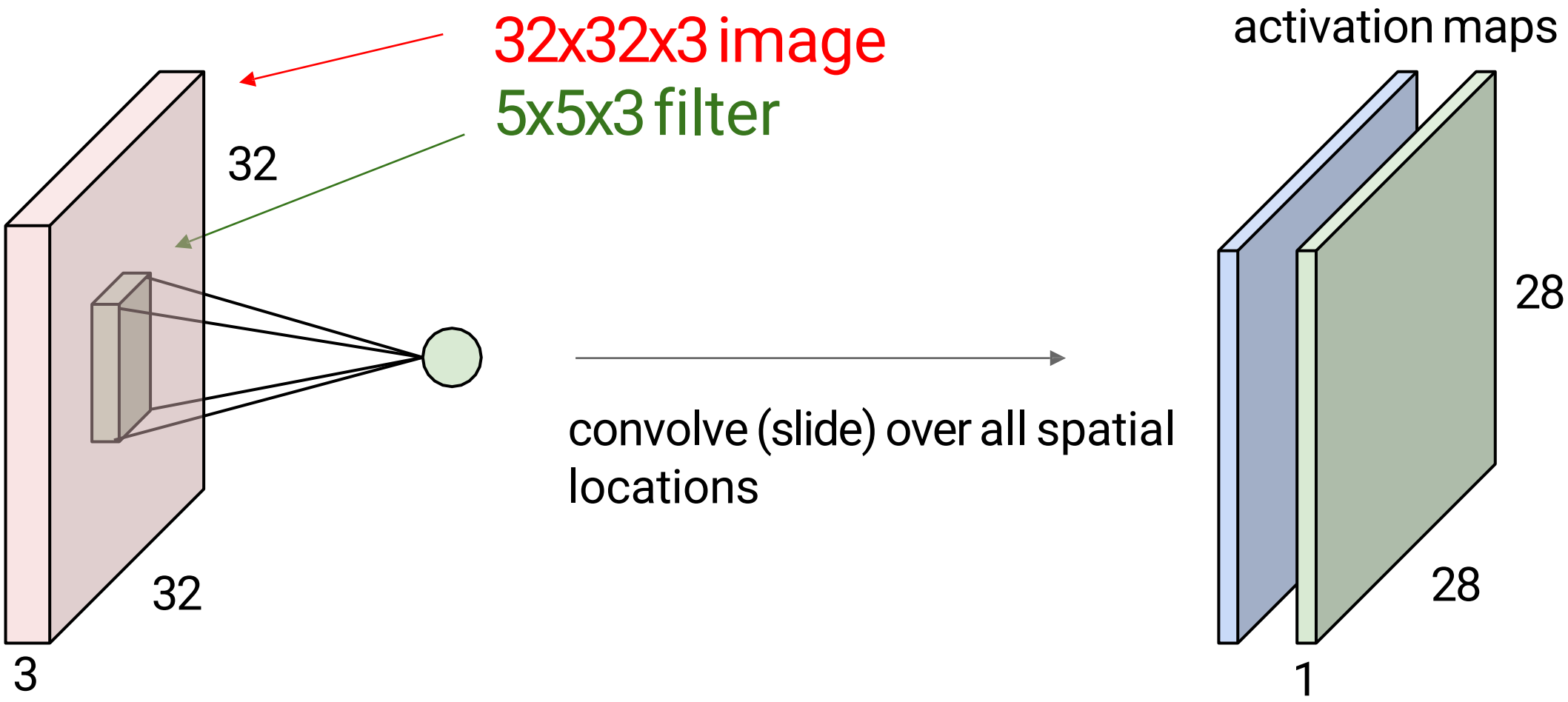


Convolution Layer

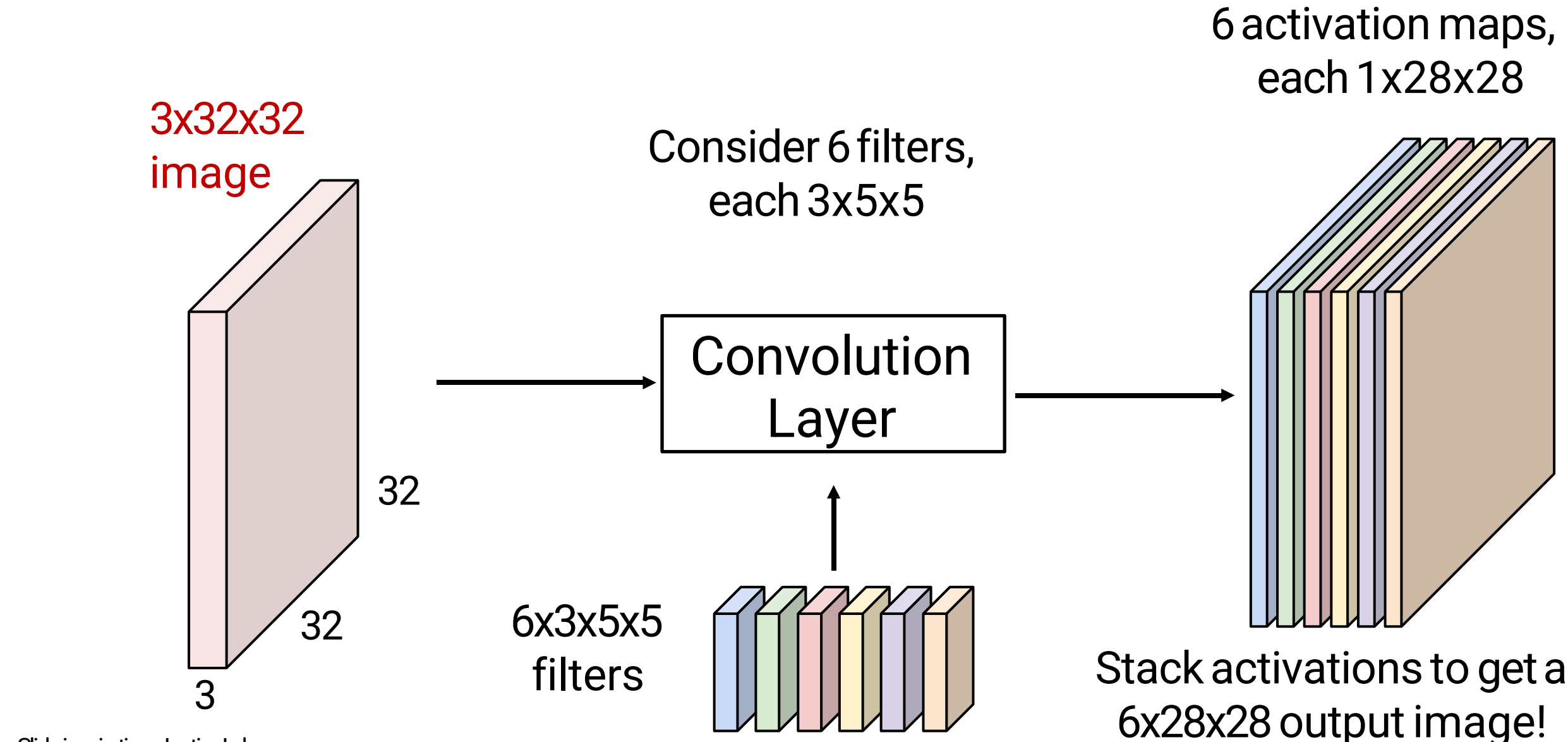


Convolution Layer

consider a second, green filter

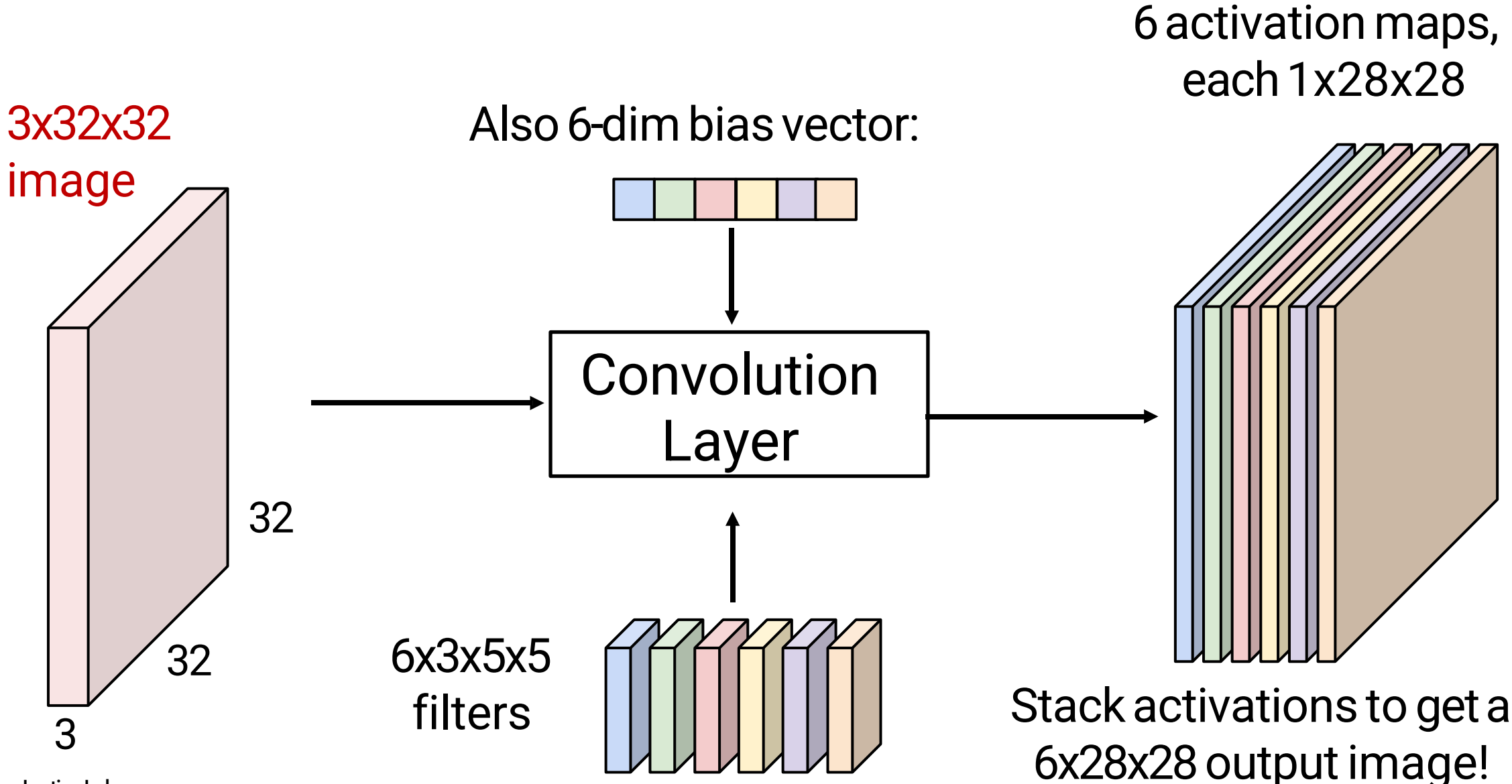


Convolution Layer

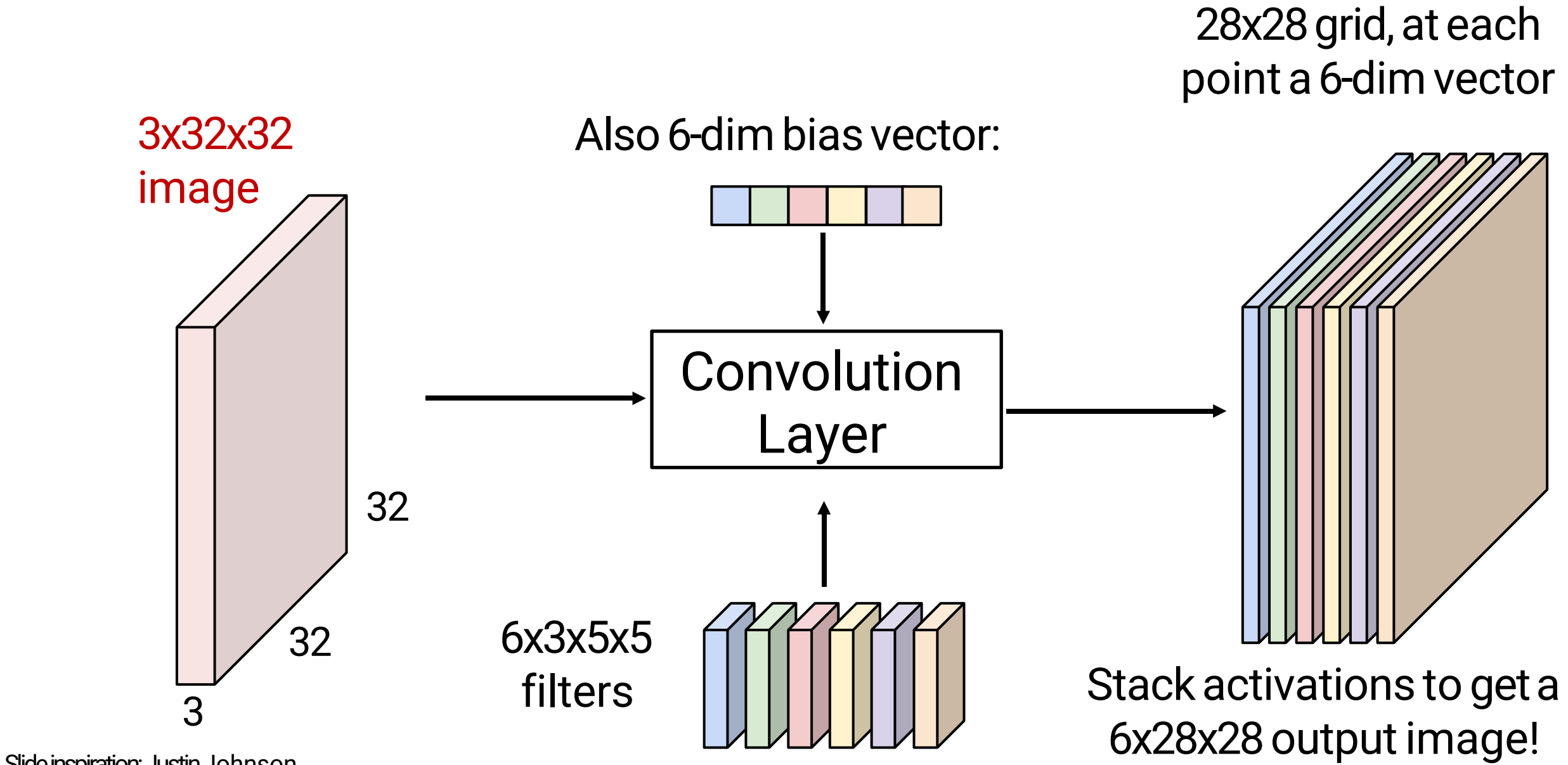


Slide inspiration: Justin Johnson

Convolution Layer

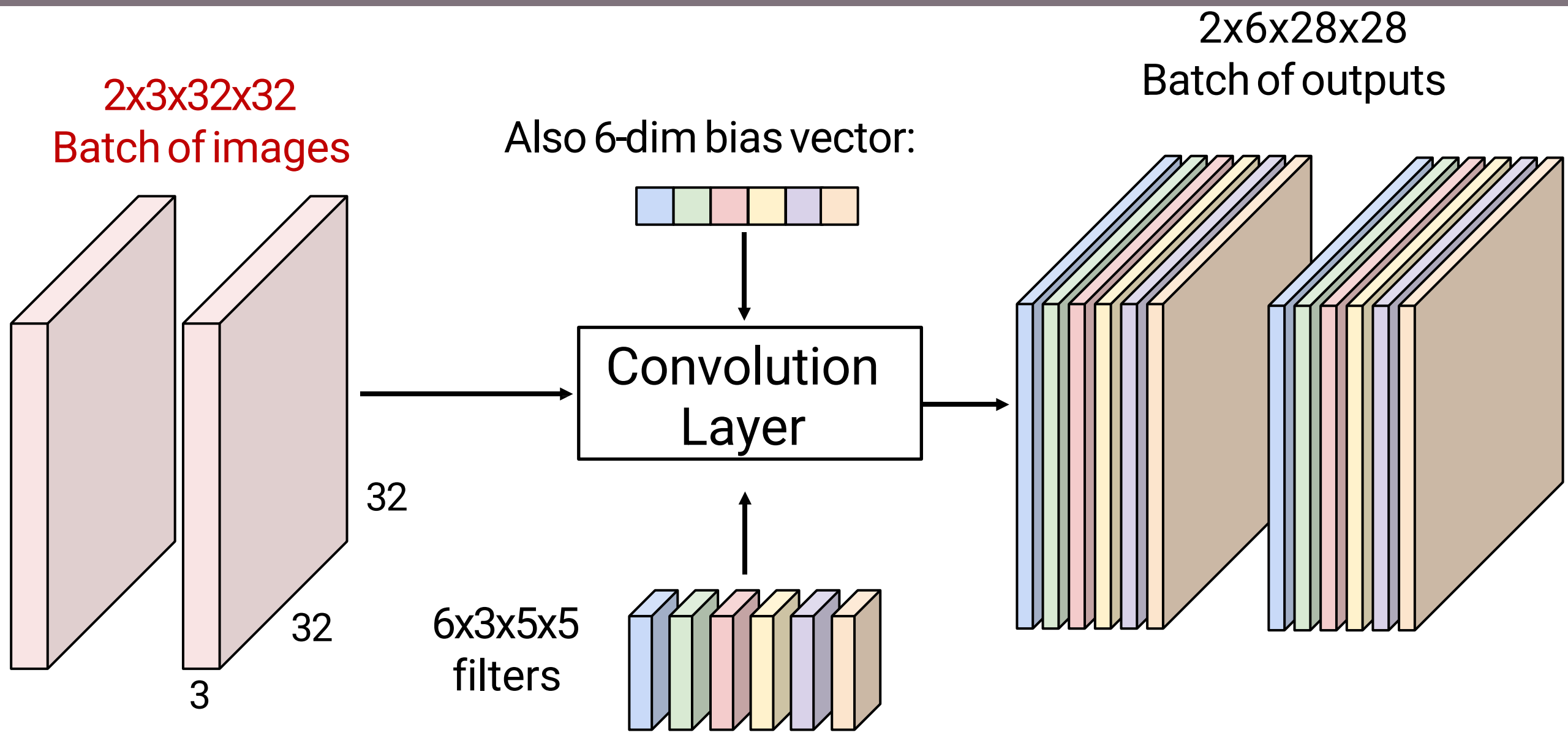


Slide inspiration: Justin Johnson

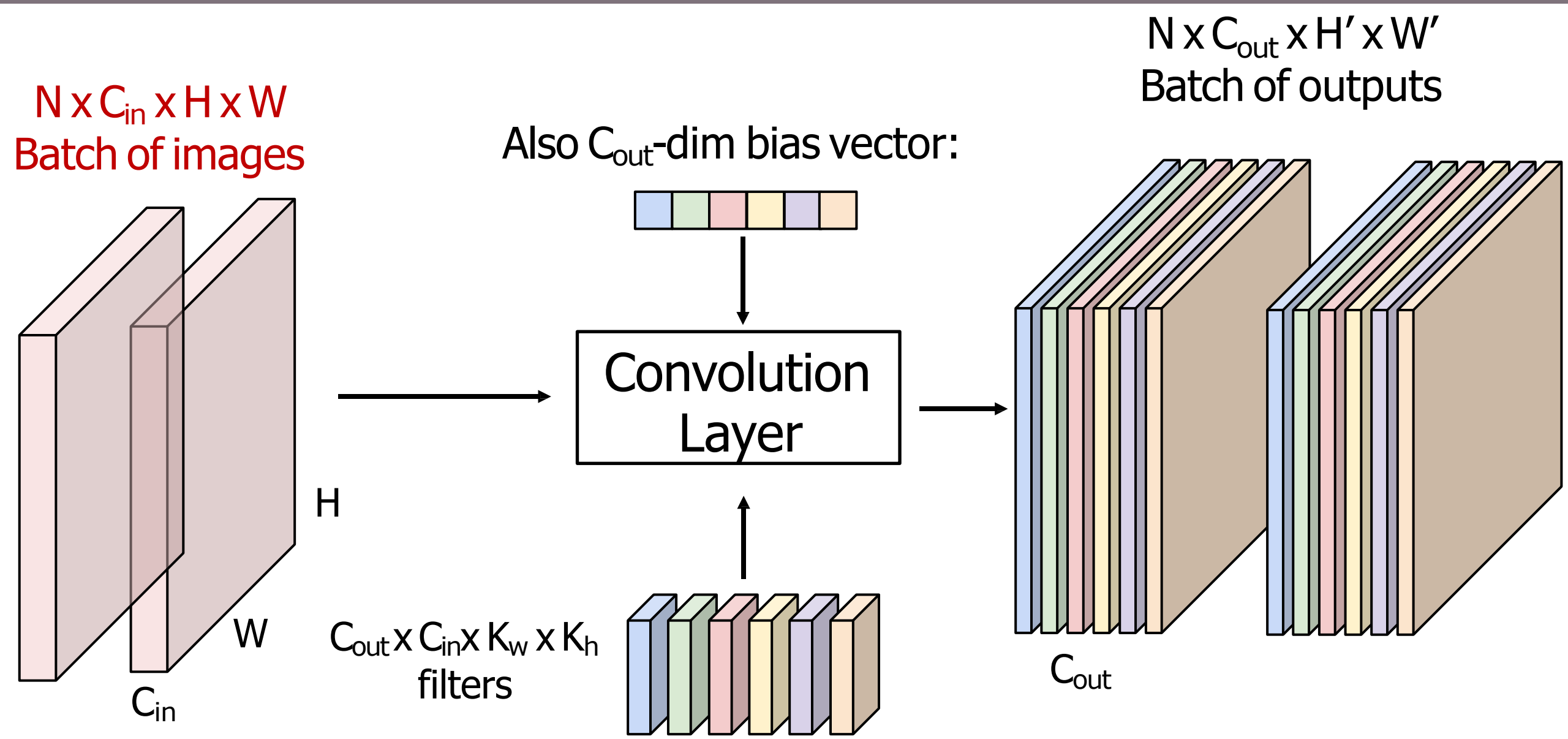


Slide inspiration: Justin Johnson

Convolution Layer

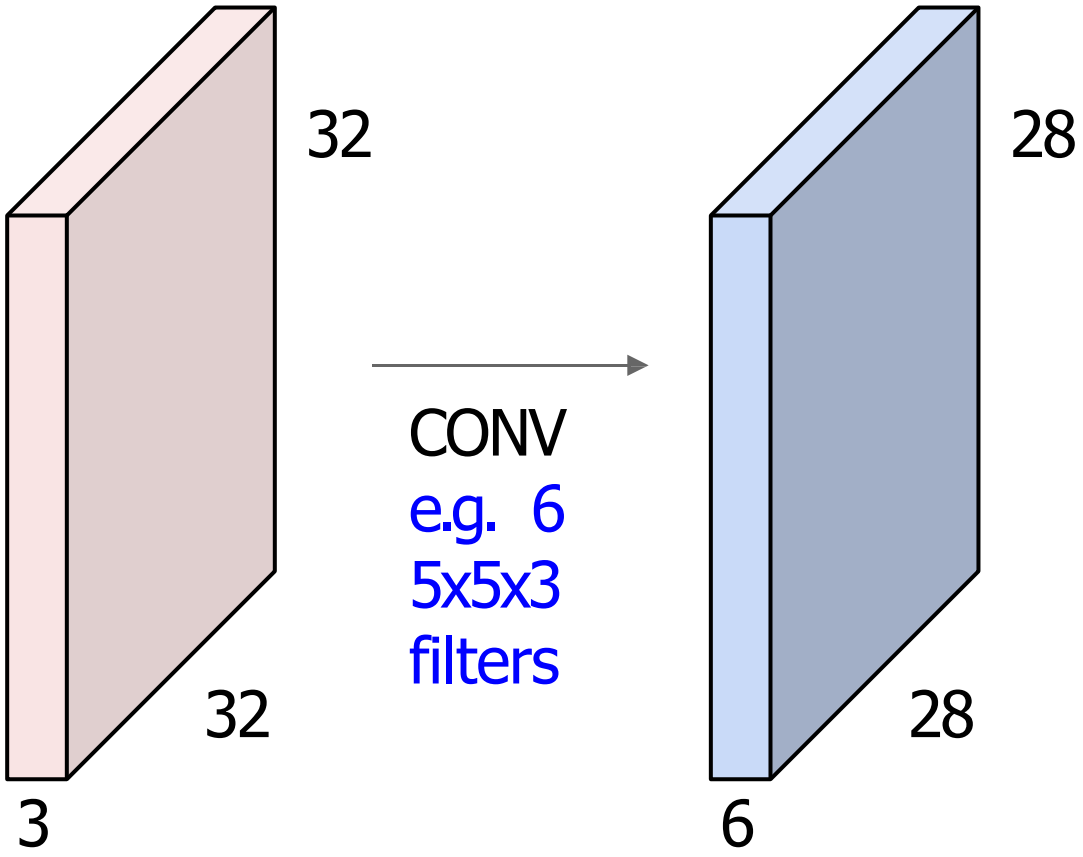


Slide inspiration: Justin Johnson

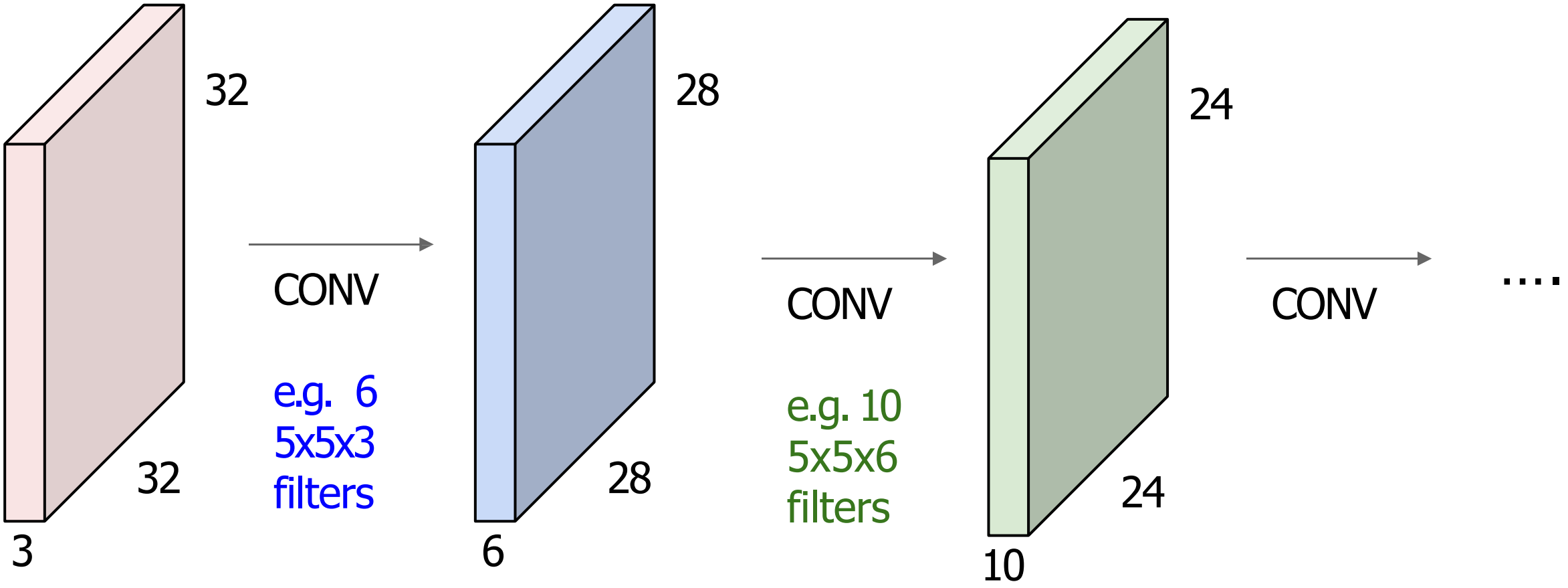


Slide inspiration: Justin Johnson

Preview: ConvNet is a sequence of Convolution Layers

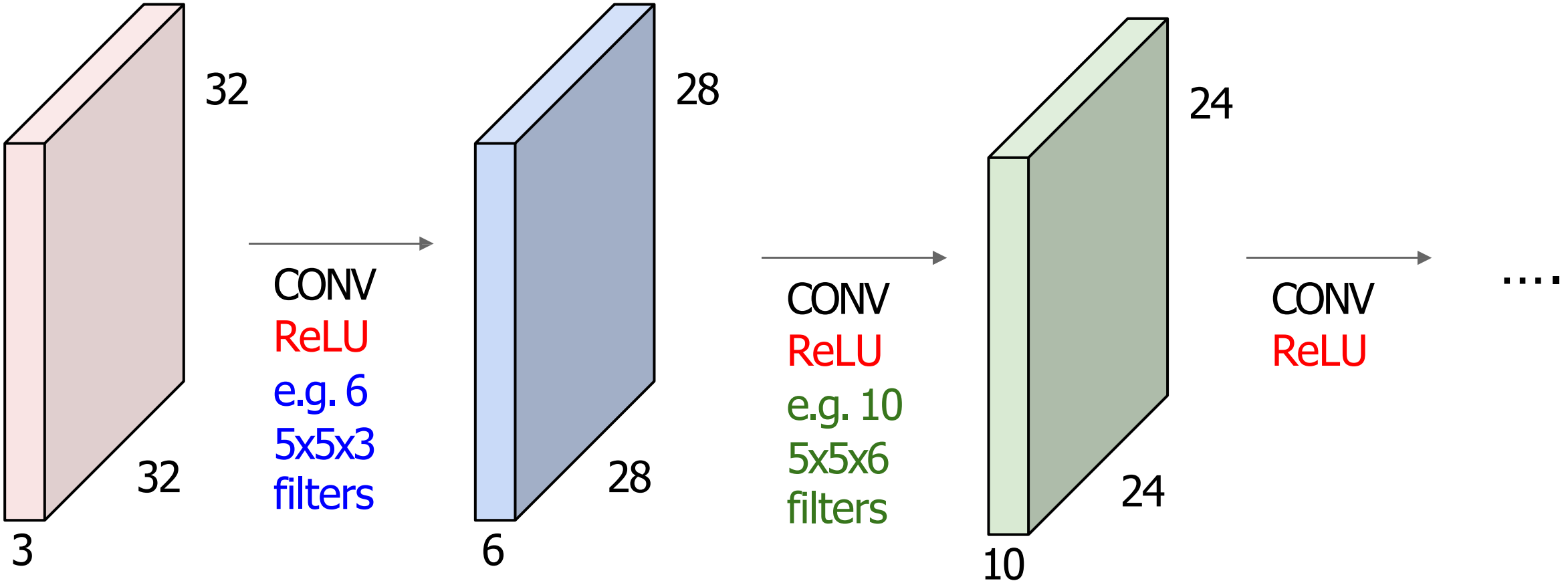


Preview: ConvNet is a sequence of Convolution Layers

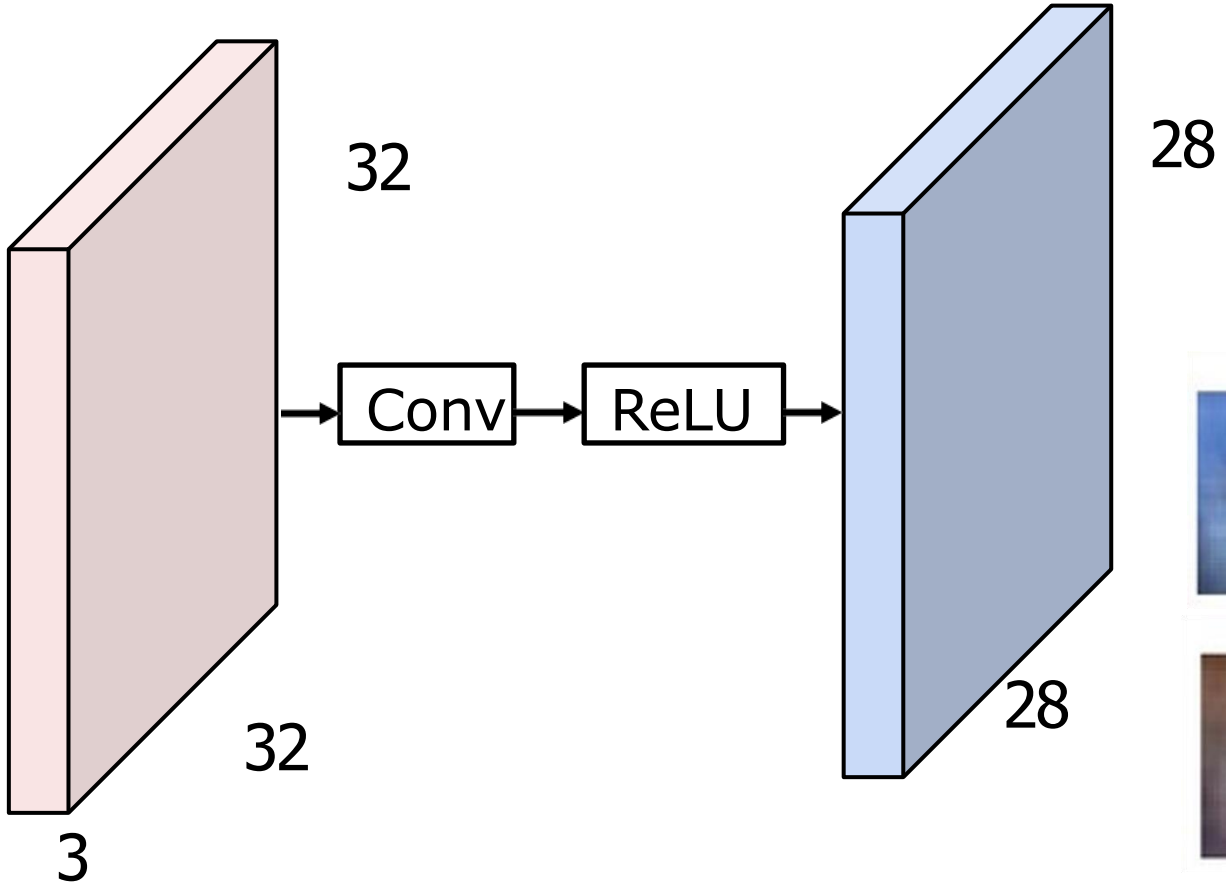


Preview: ConvNet is a sequence of Convolution Layers,

interspersed with activation functions

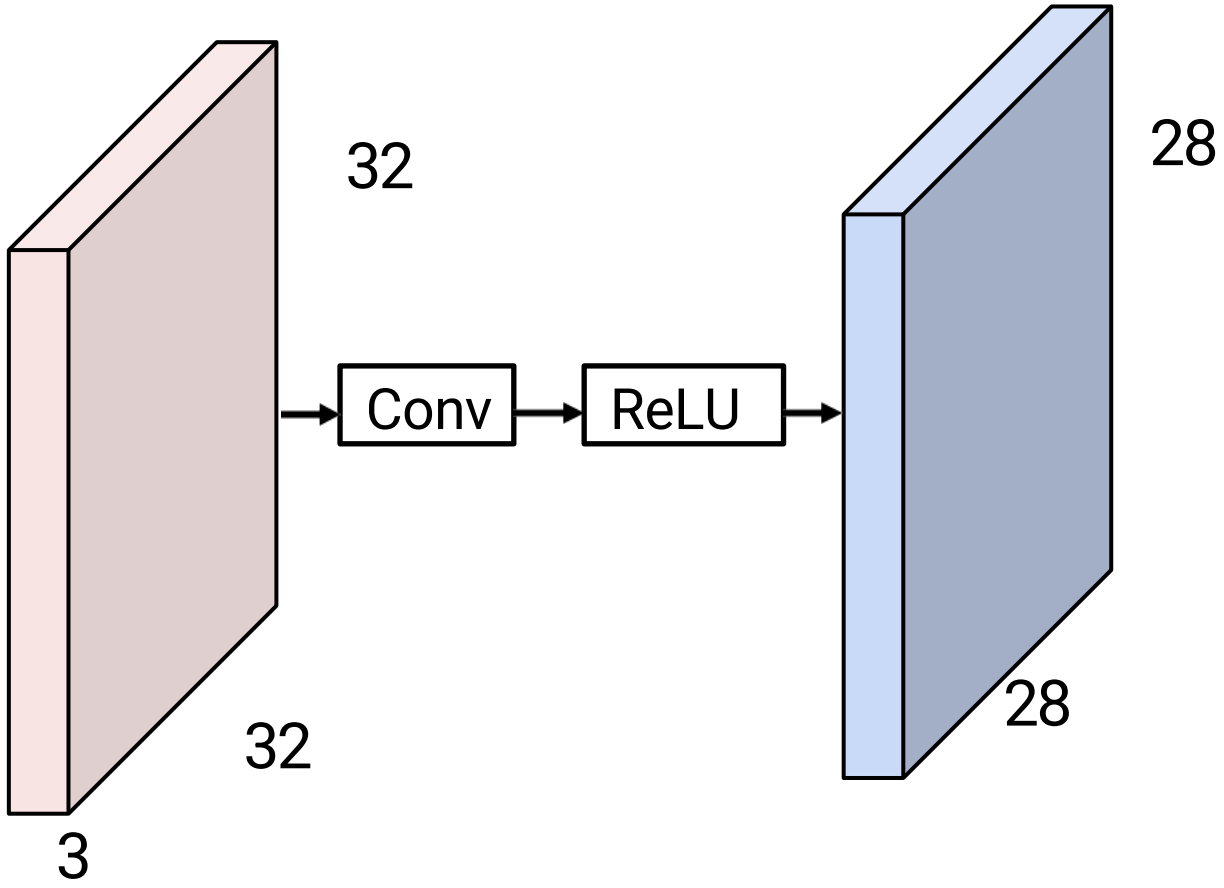


Preview: What do convolutional filters learn?

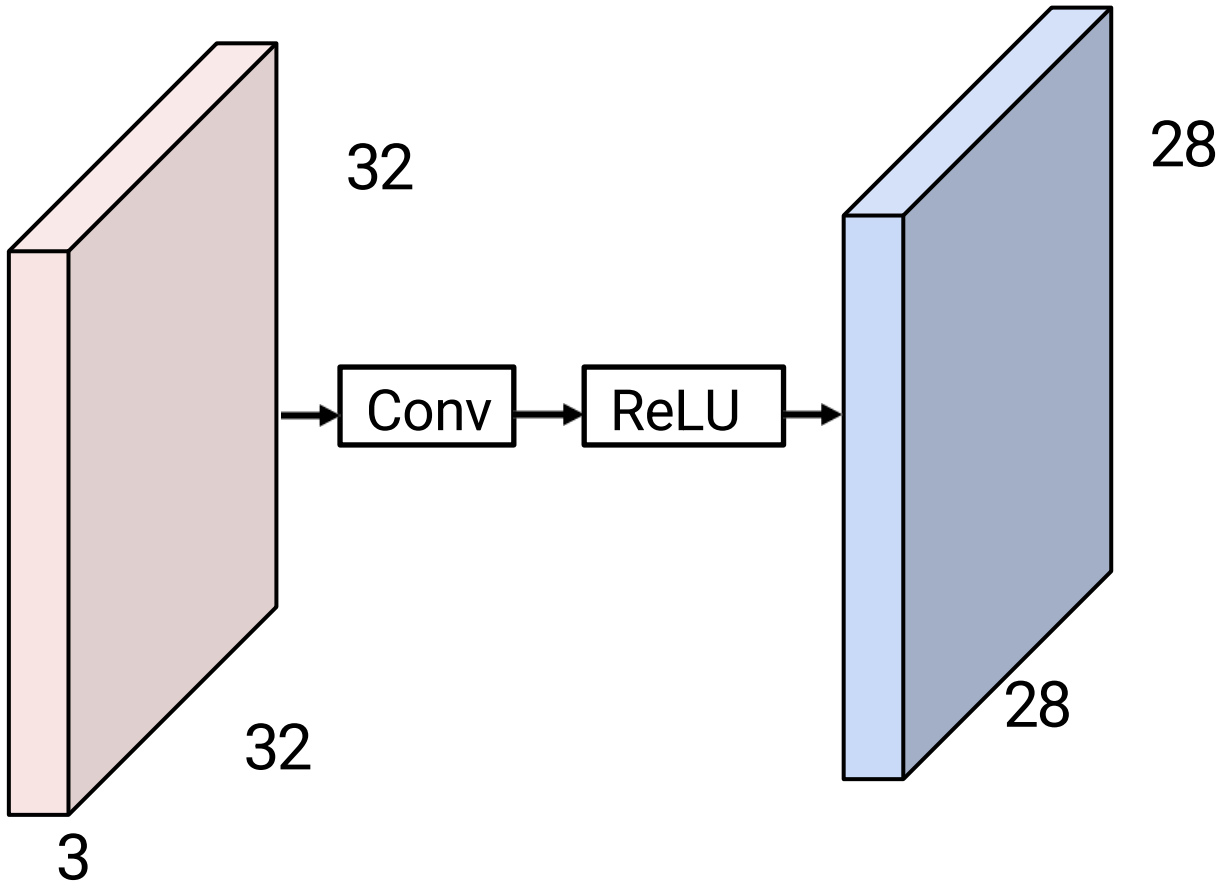


Linear classifier: One template per class

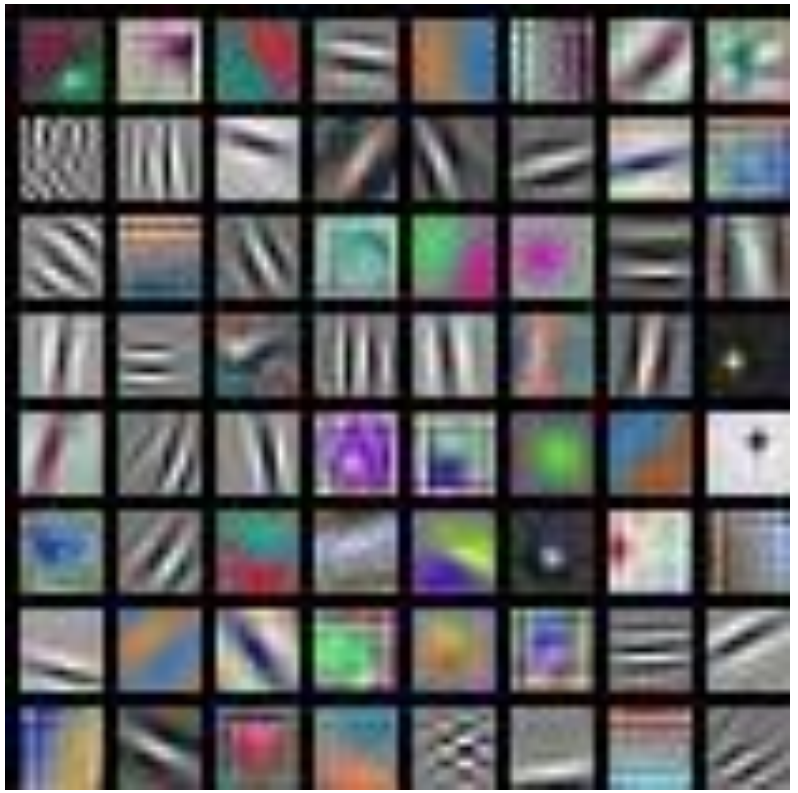
Preview: What do convolutional filters learn?



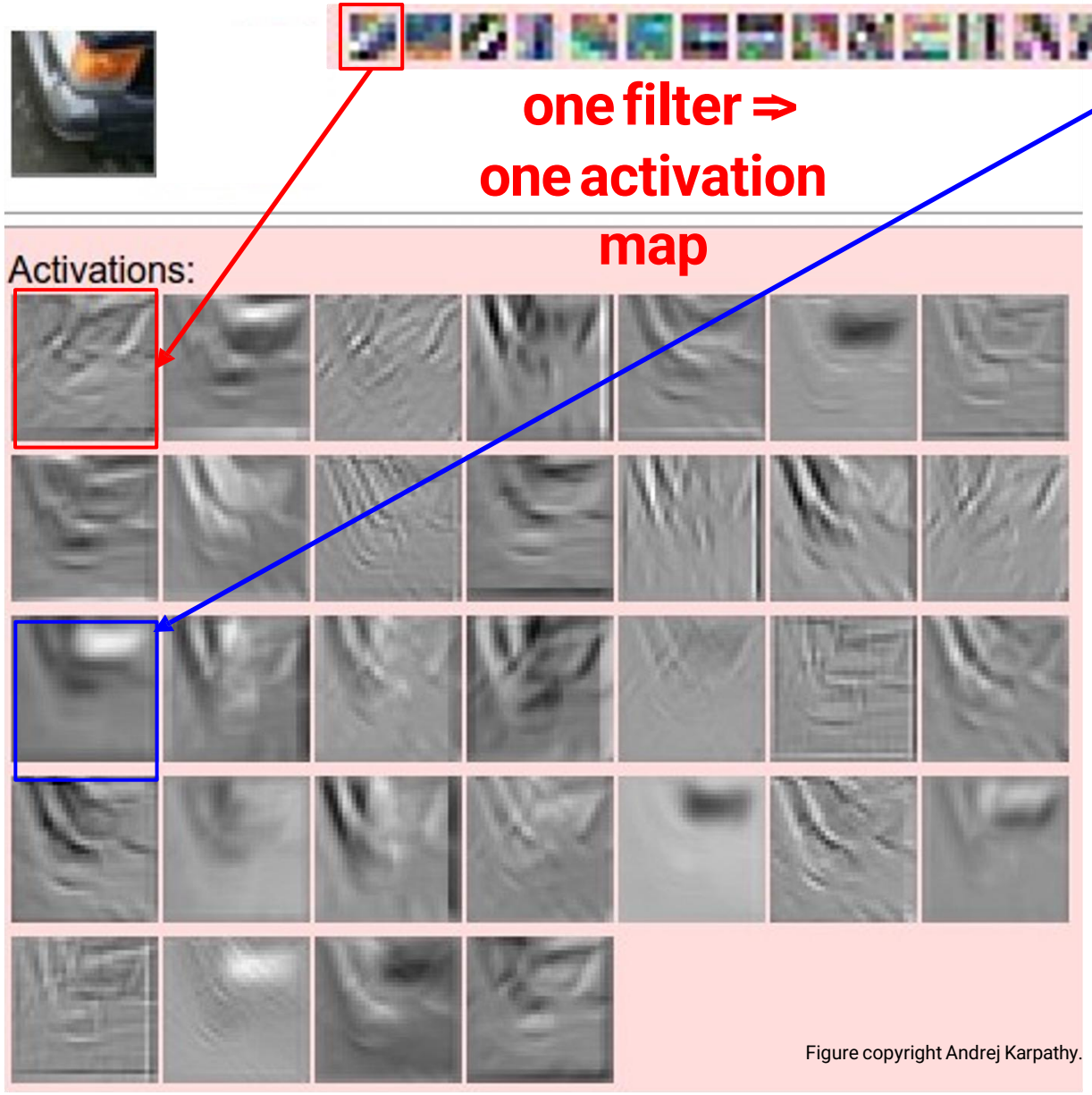
MLP: Bank of whole-image templates



First-layer conv filters: local image templates (Often learns oriented edges, opposing colors)



AlexNet: 64 filters, each 3x11x11



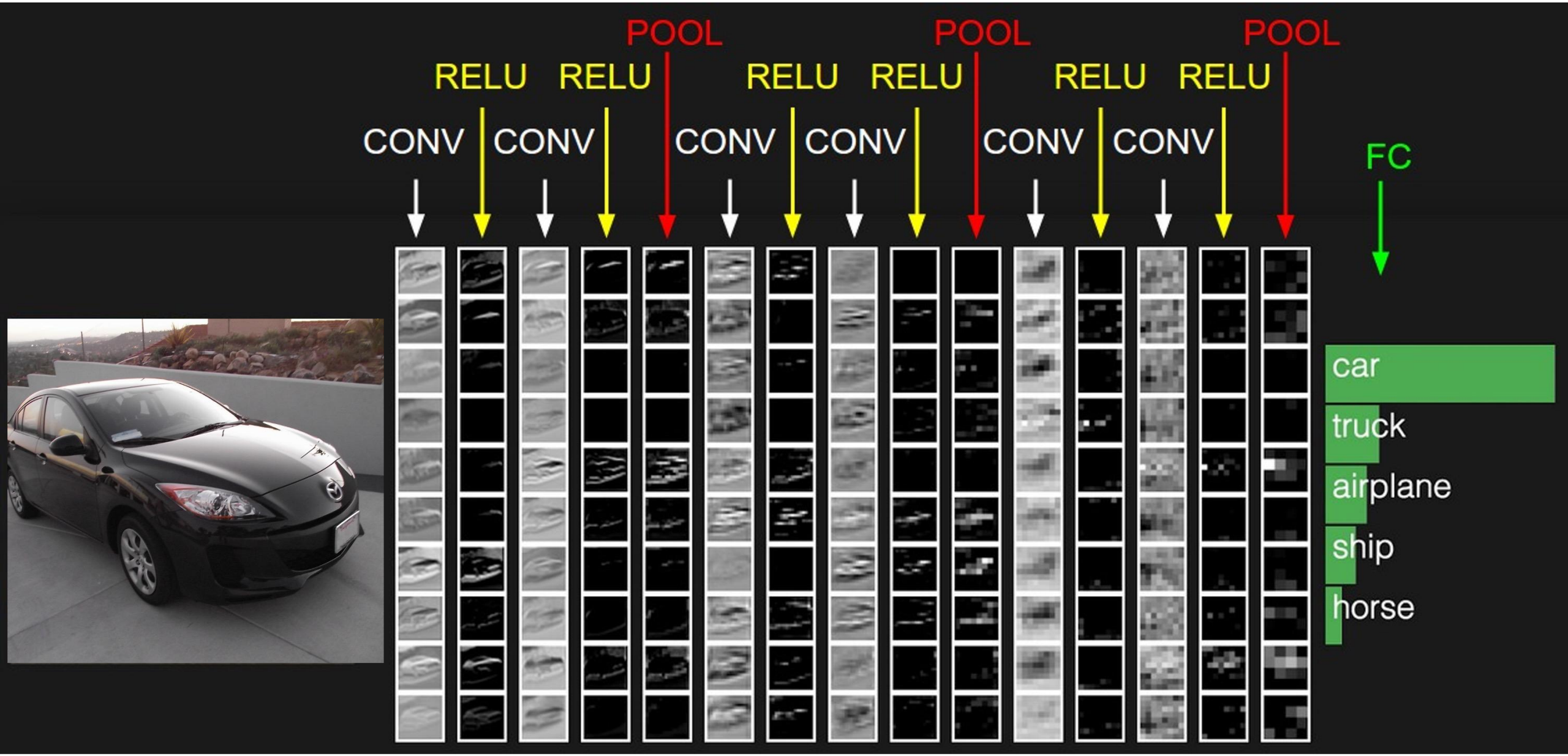
example 5x5 filters
(32 total)

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

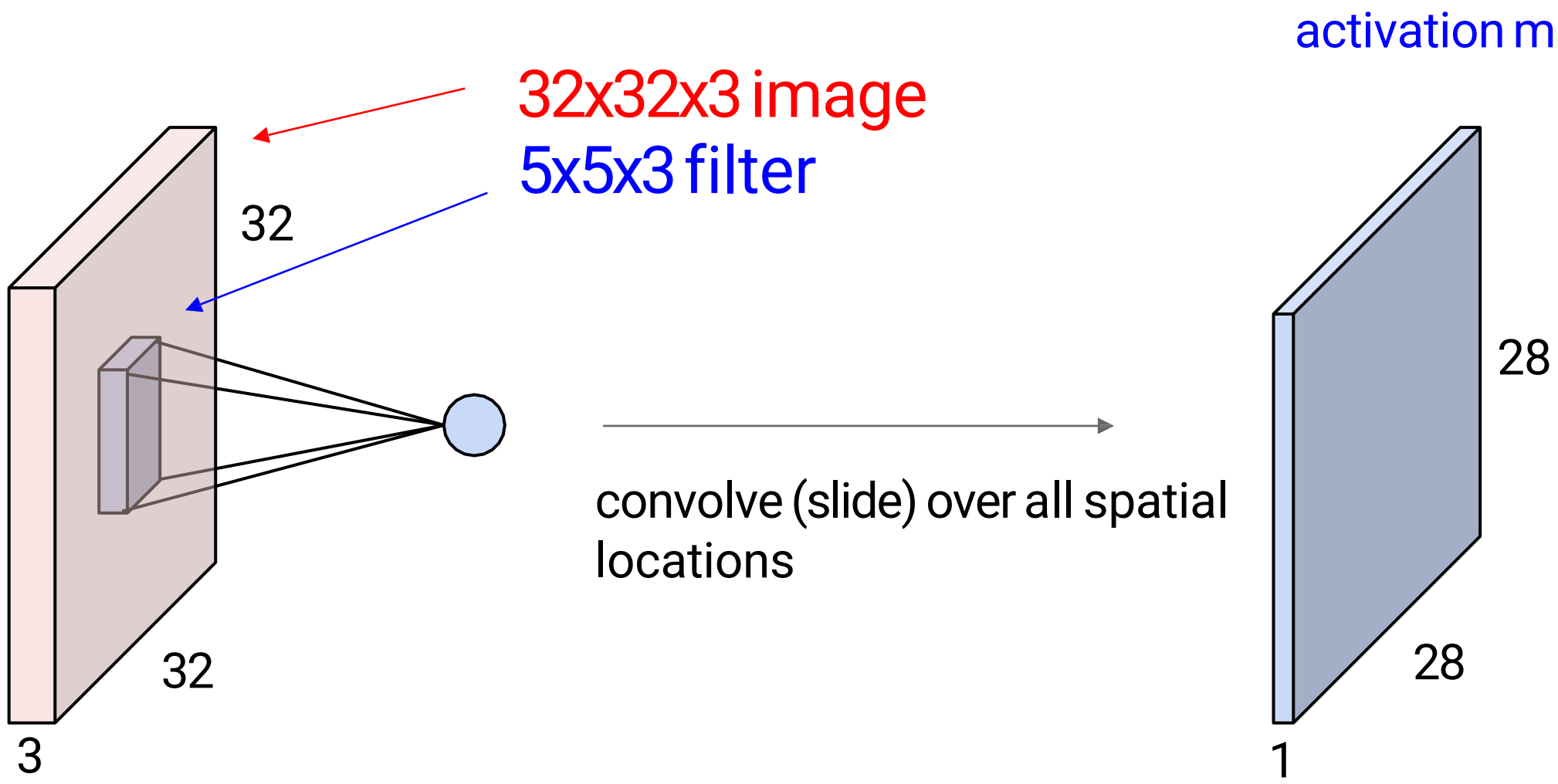
$$f[x,y] * g[x,y] = \sum_{n_1=-\infty}^{\infty} \sum_{n_2=-\infty}^{\infty} f[n_1,n_2] \cdot g[x-n_1, y-n_2]$$

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

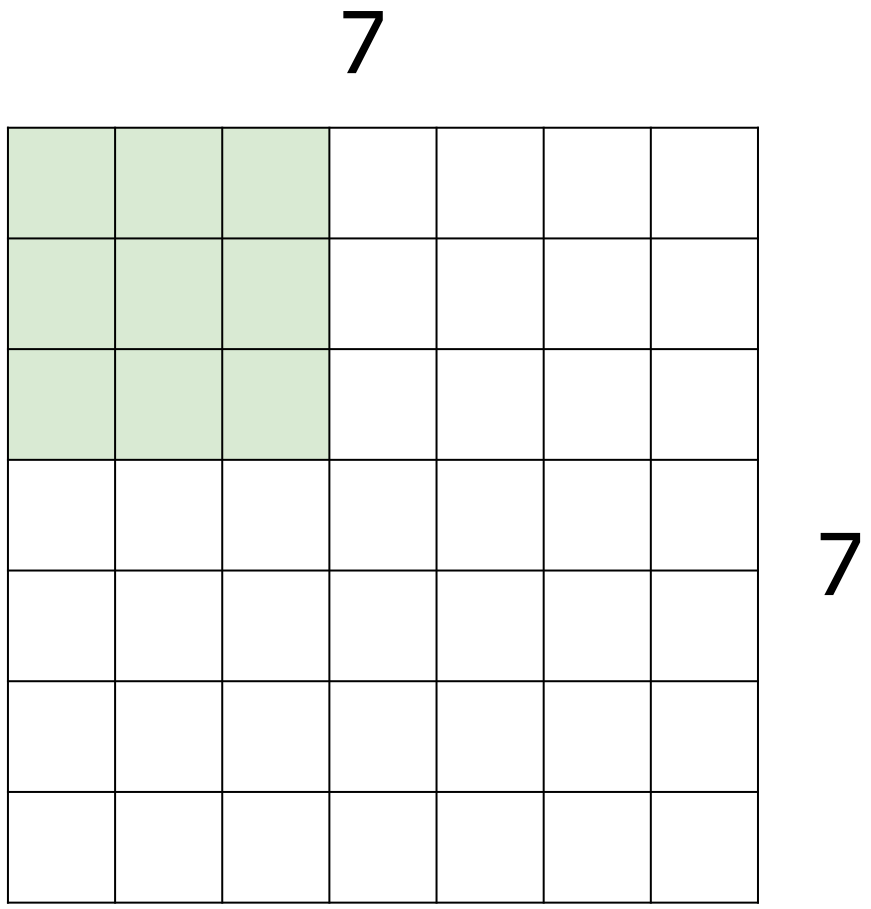
preview:


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:

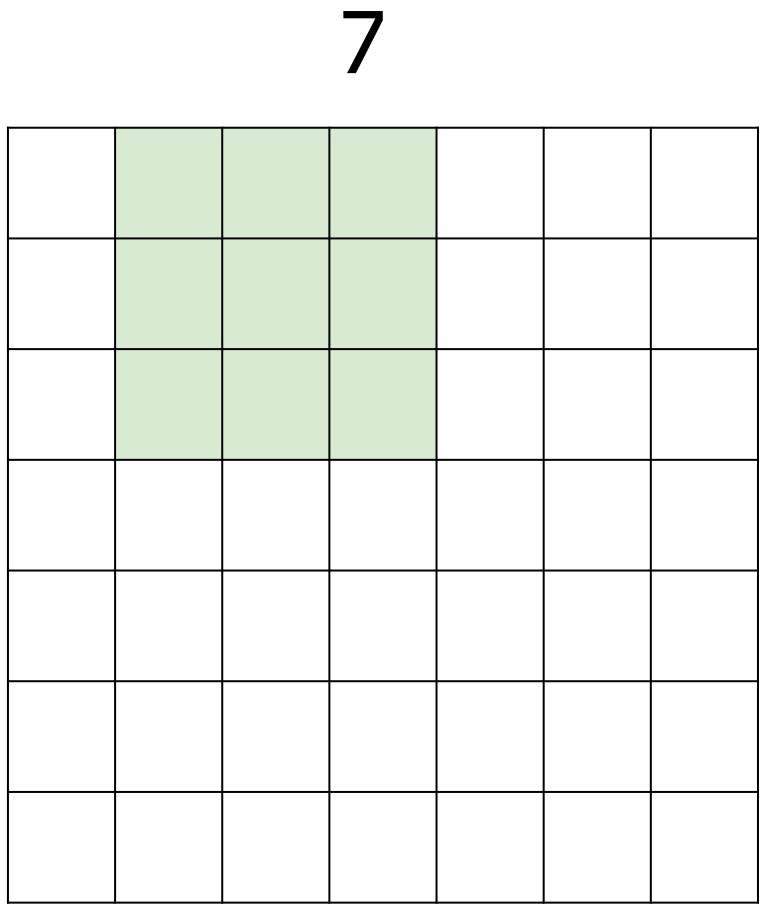


A closer look at spatial dimensions:



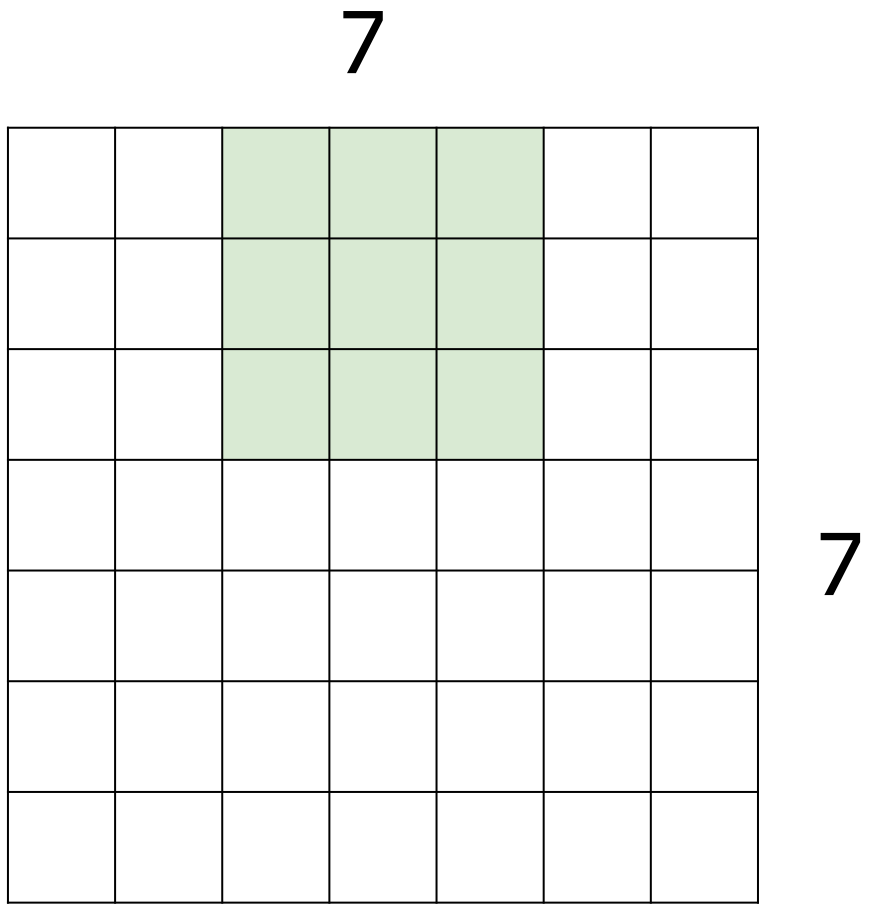
7x7 input (spatially)
assume 3x3 filter

A closer look at spatial dimensions:



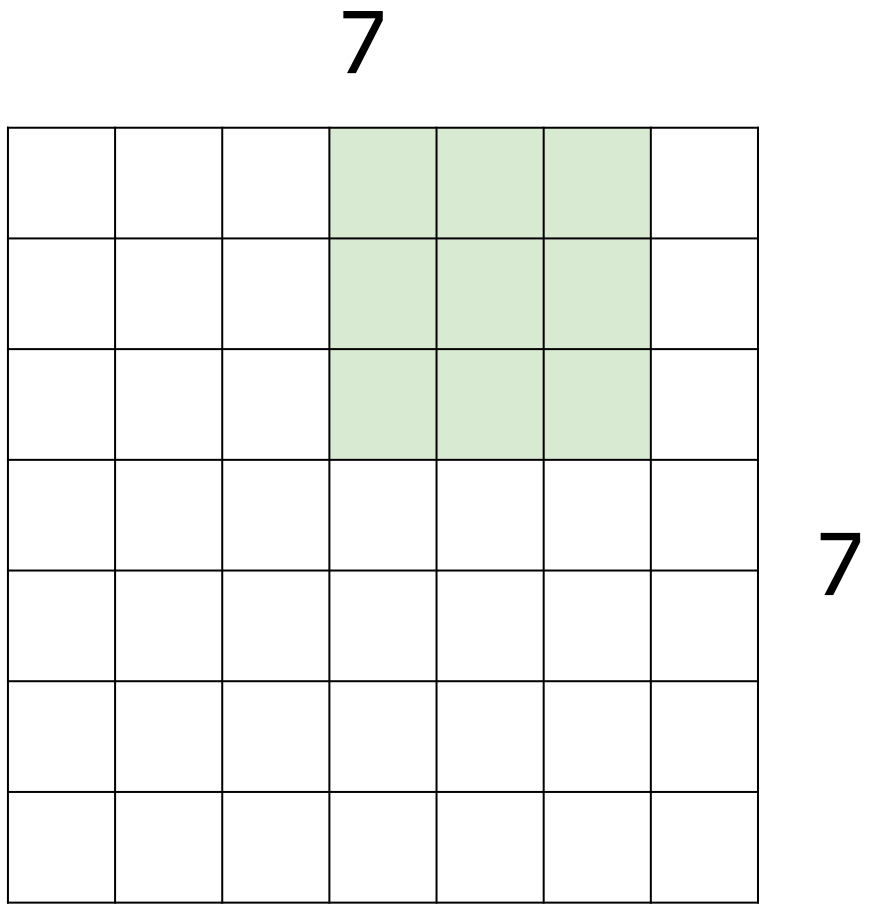
7x7 input (spatially)
assume 3x3 filter

A closer look at spatial dimensions:



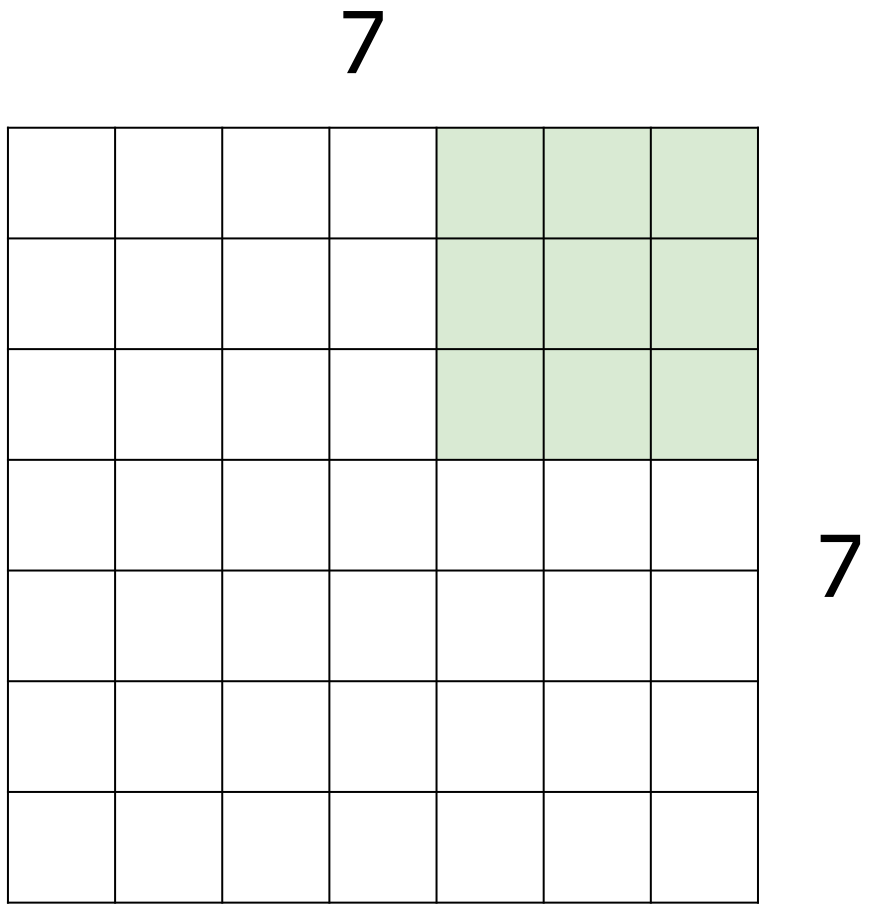
7x7 input (spatially)
assume 3x3 filter

A closer look at spatial dimensions:



7x7 input (spatially)
assume 3x3 filter

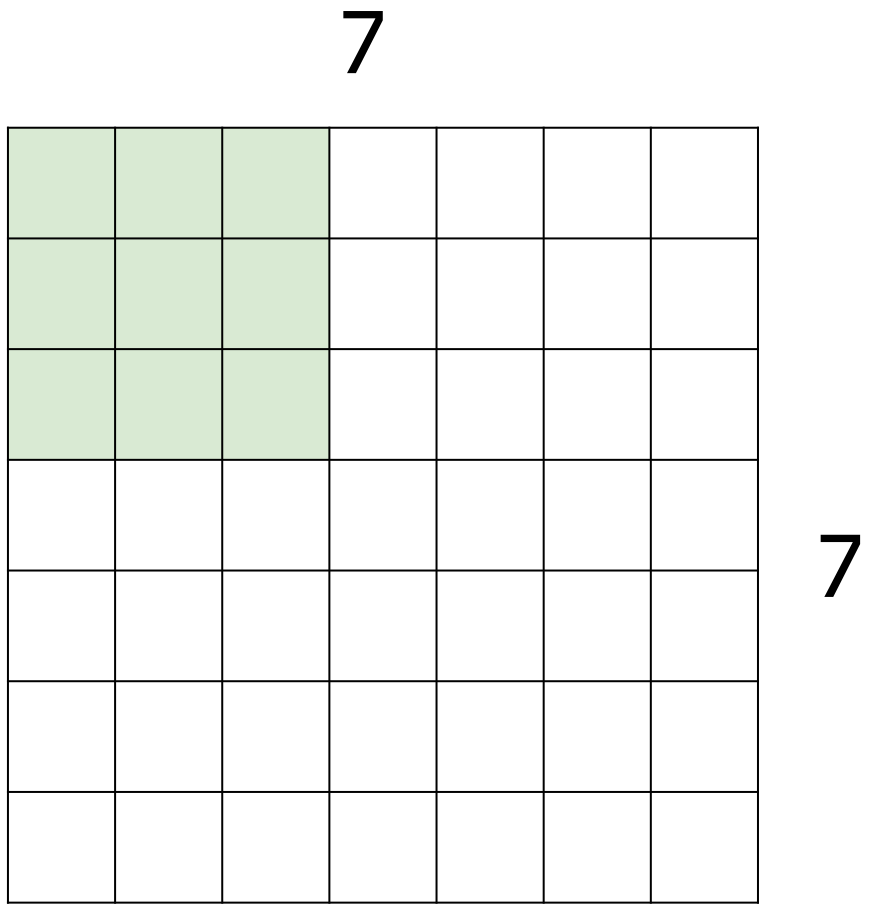
A closer look at spatial dimensions:



7x7 input (spatially)
assume 3x3 filter

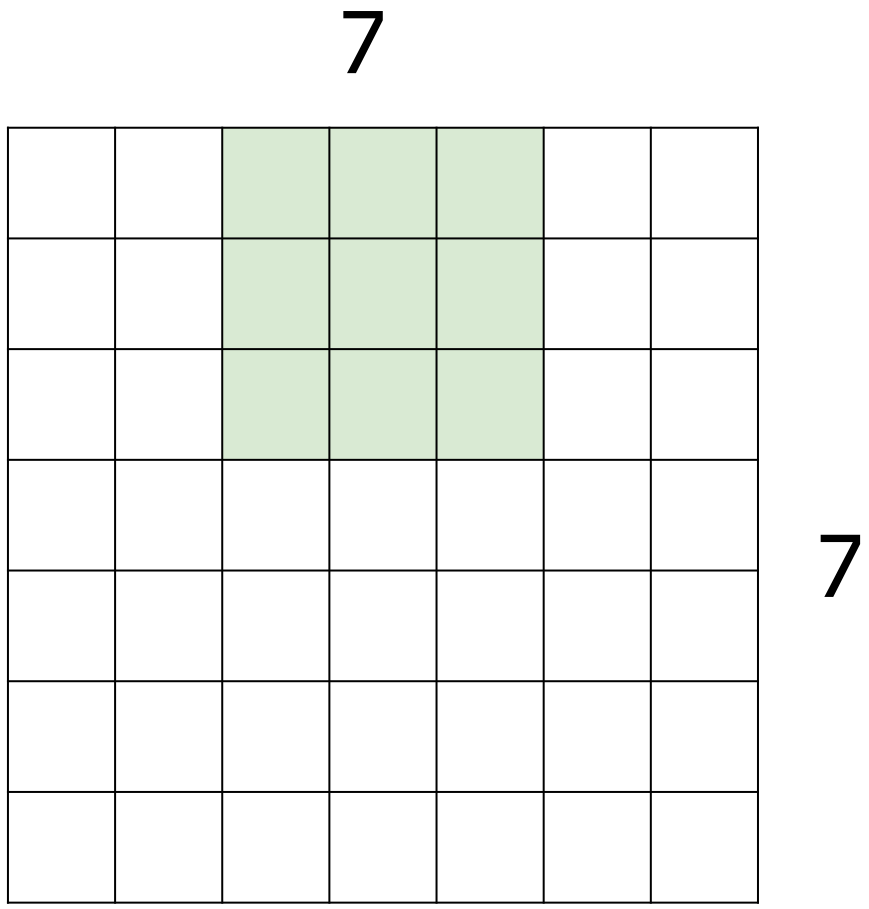
⇒5x5 output

A closer look at spatial dimensions:



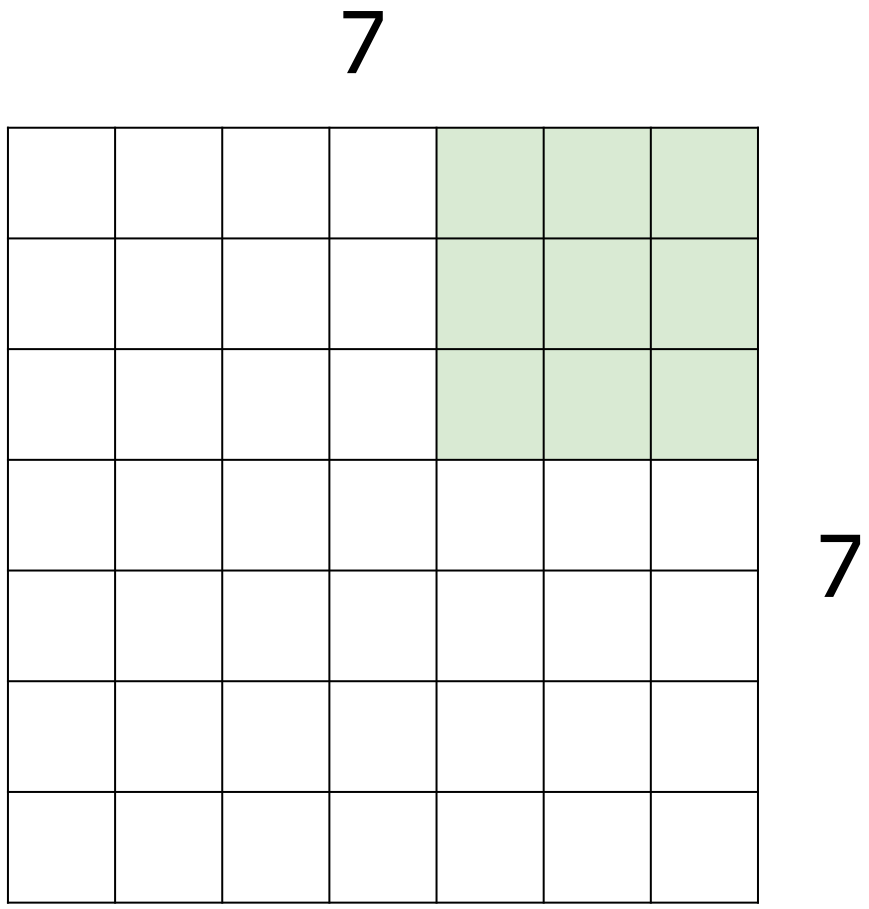
7x7 input (spatially)
assume 3x3 filter
applied with stride 2

A closer look at spatial dimensions:



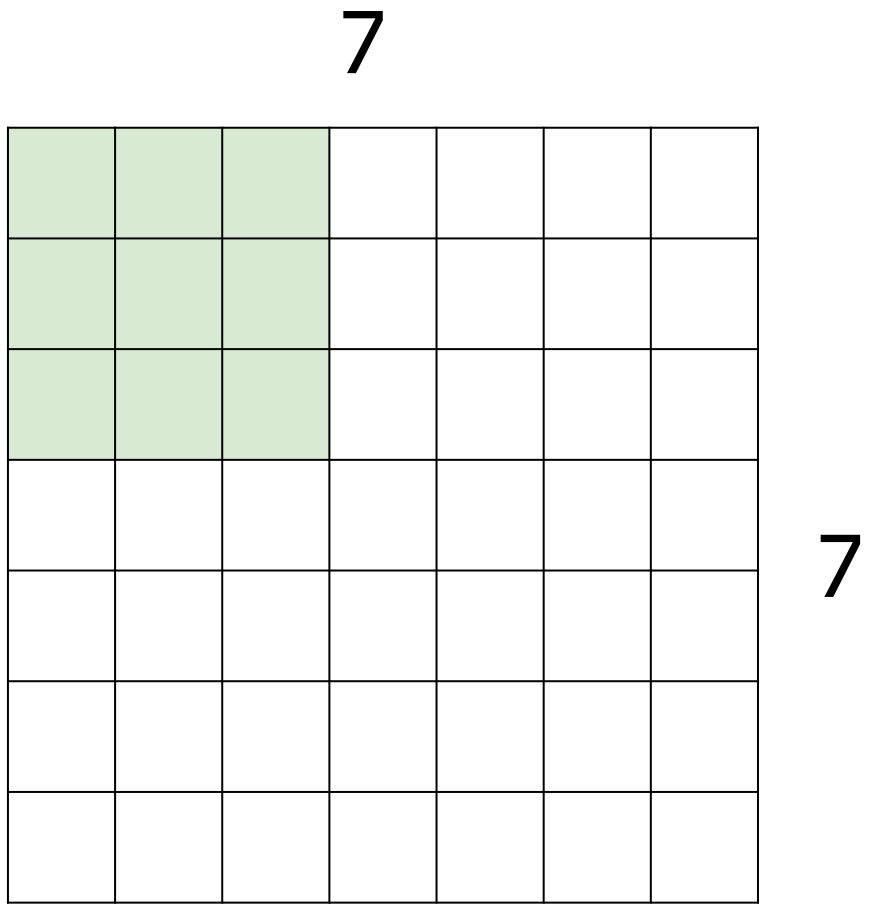
7x7 input (spatially)
assume 3x3 filter
applied with stride 2

A closer look at spatial dimensions:



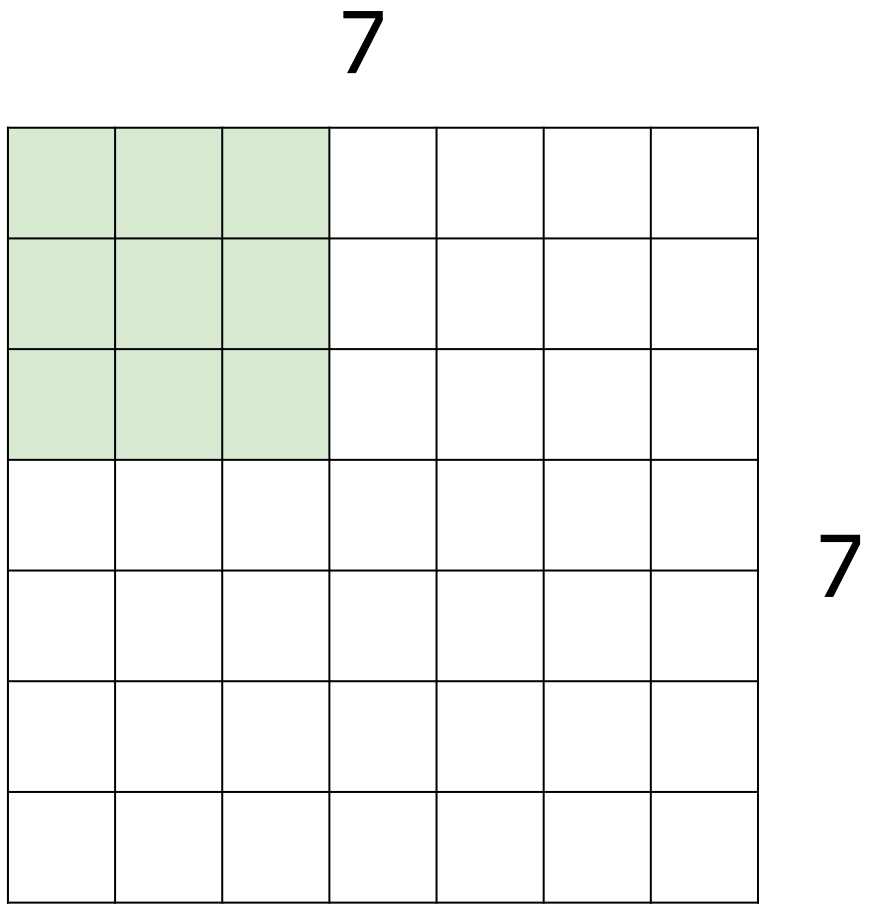
7x7 input (spatially)
assume 3x3 filter
applied with stride 2
⇒3x3 output!

A closer look at spatial dimensions:



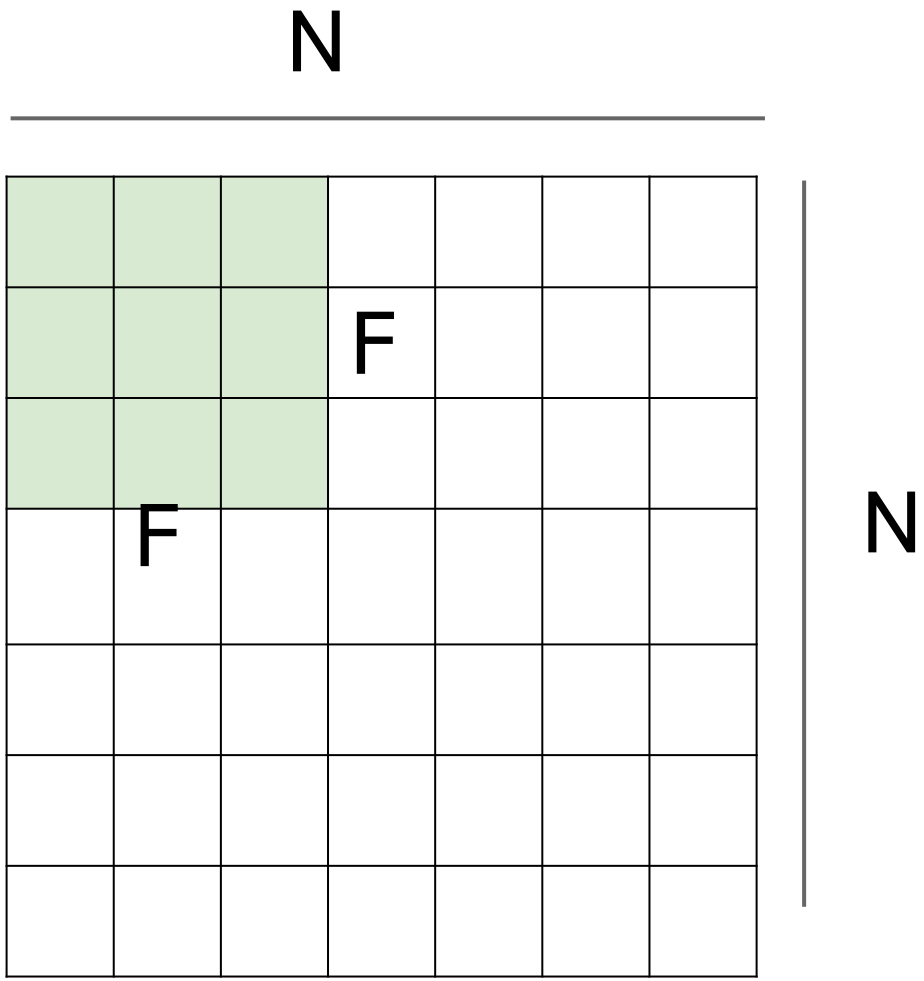
7x7 input (spatially)
assume 3x3 filter
applied with stride 3?

A closer look at spatial dimensions:



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

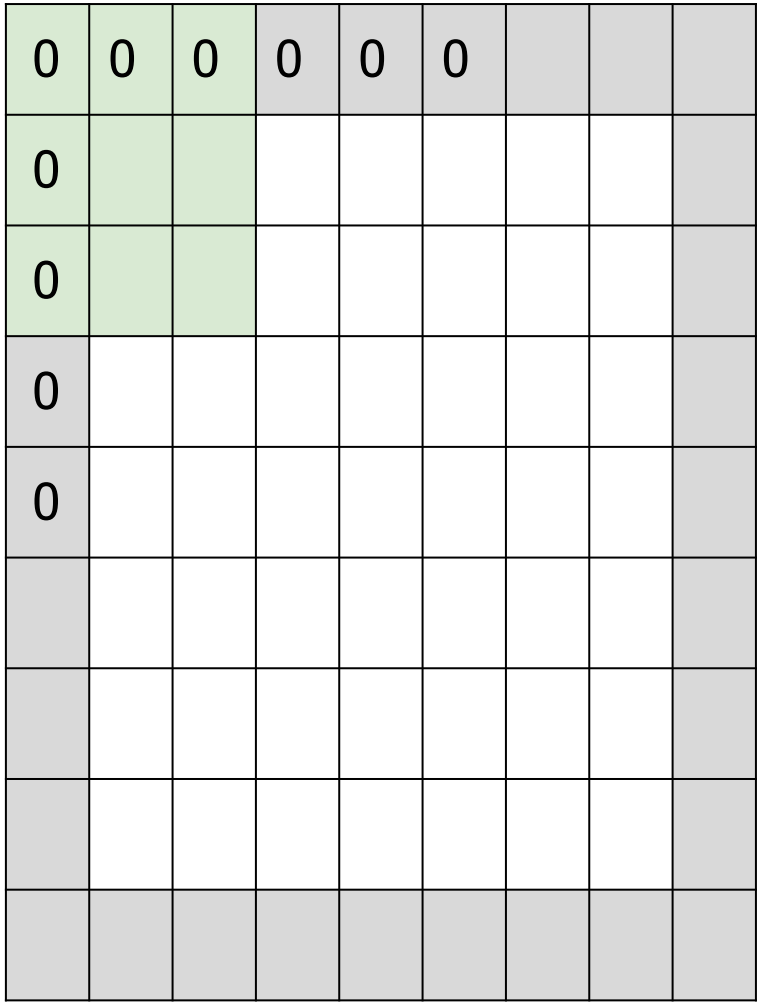
doesn't fit!
cannot apply 3x3 filter on 7x7 input with stride 3.



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 :\

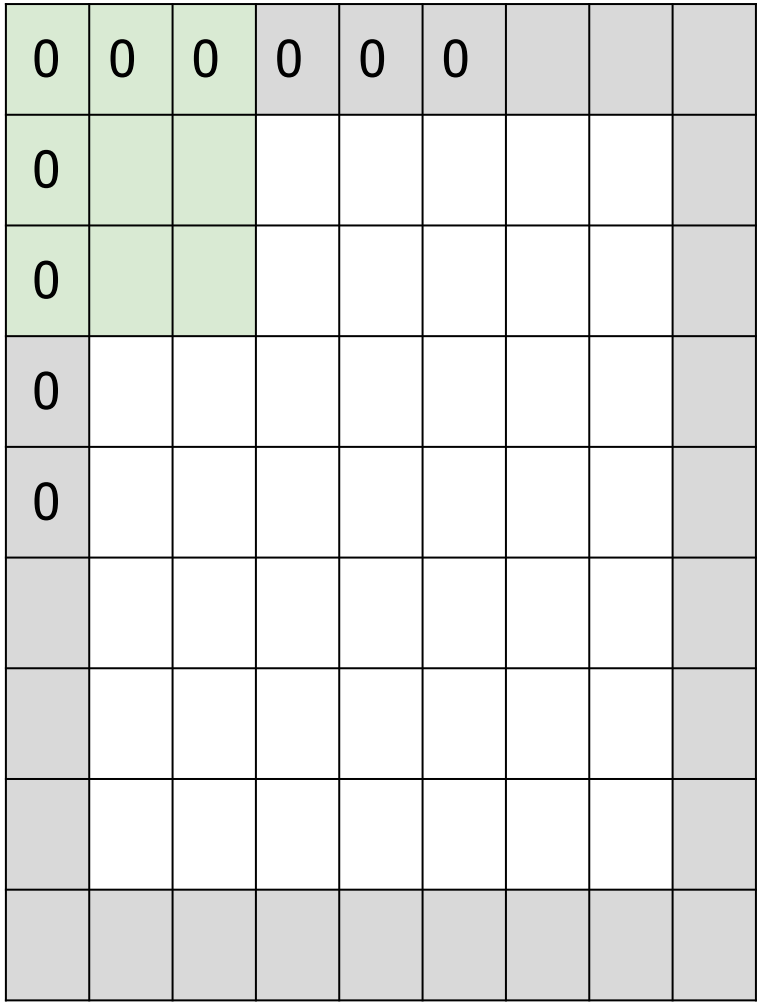
In practice: Common to zero pad the border



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

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

In practice: Common to zero pad the border

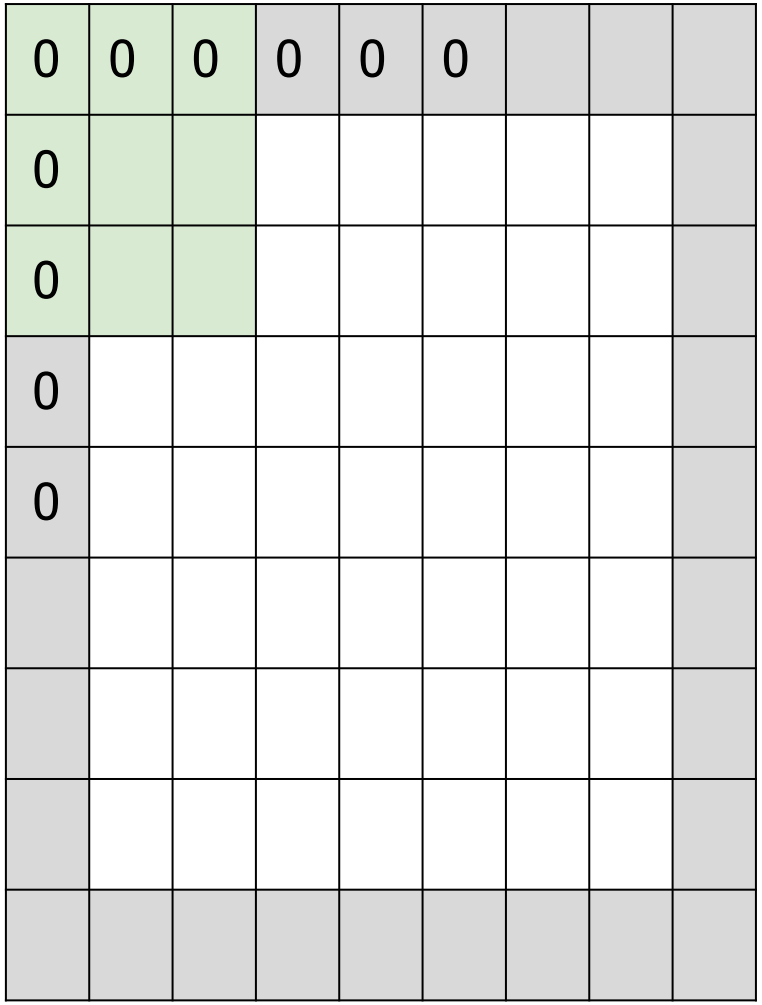


e.g. input 7x7
3x3 filter, applied with stride 1
pad with 1 pixel border ⇒what is the output?

7x7 output!

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

In practice: Common to zero pad the border

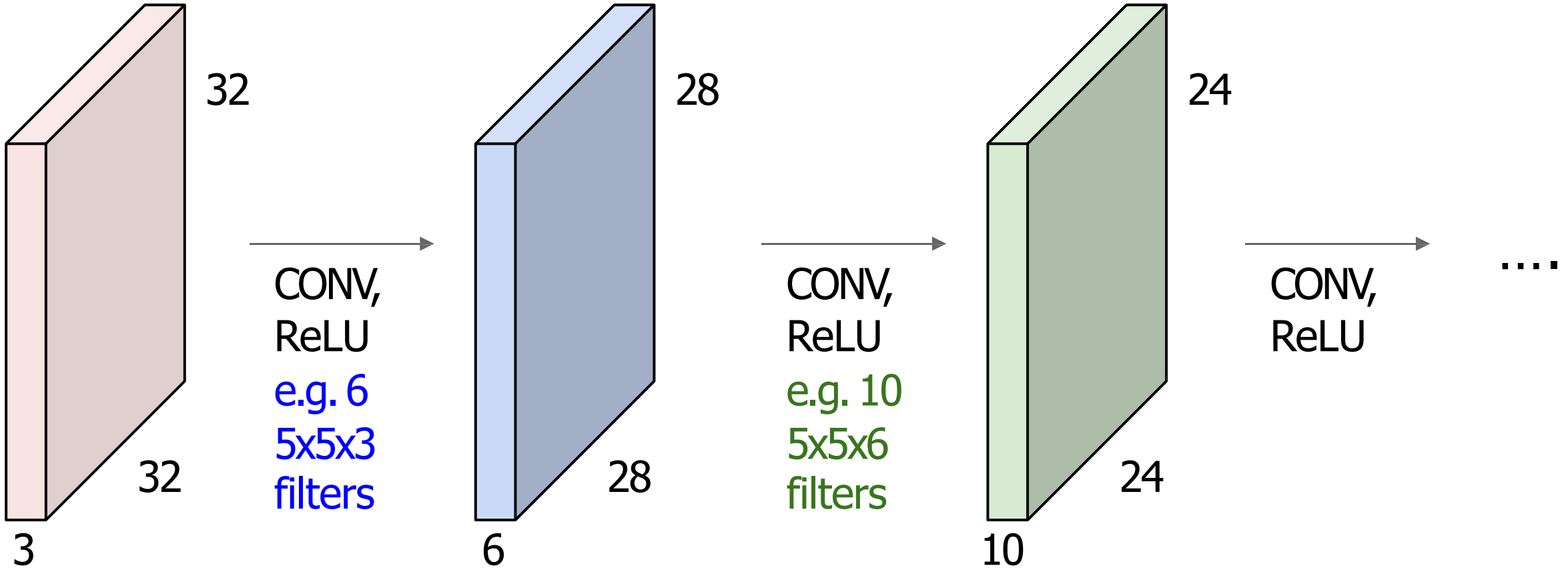


e.g. input 7x7
3x3 filter, applied with stride 1
pad with 1 pixel border ⇒what is the output?

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

Remember back to...

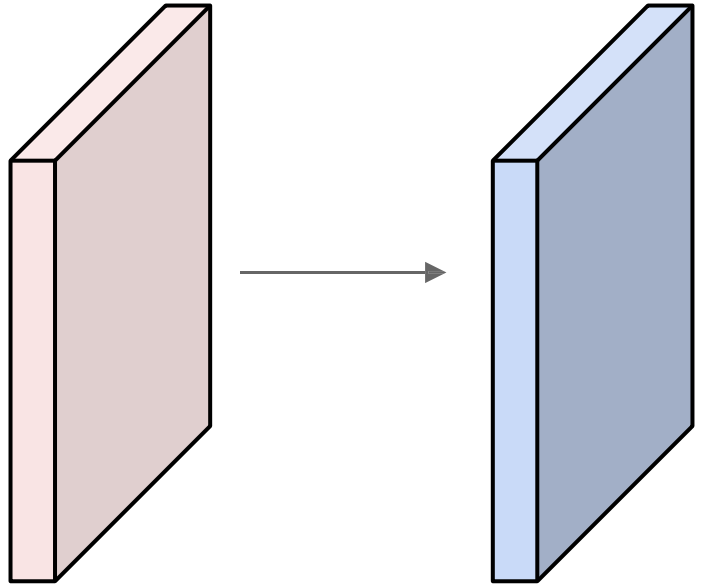
E.g. 32x32 input convolved repeatedly with 5x5 filters shrinks volumes spatially! (32 ->28 ->24 ...). Shrinking too fast is not good, doesn't work well.



Examples time:

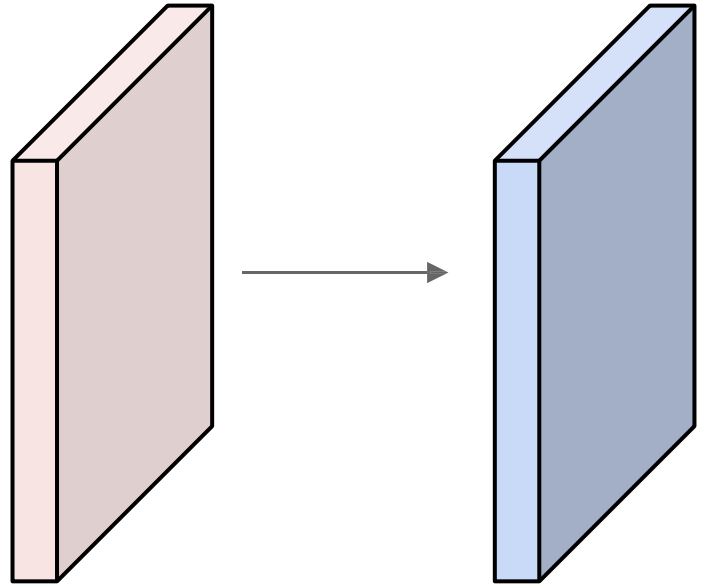
Input volume: 32x32x3
10 5x5 filters with stride 1, pad 2

Output volume size: ?



Input volume: 32x32x3
10 5x5 filters with stride 1, pad 2

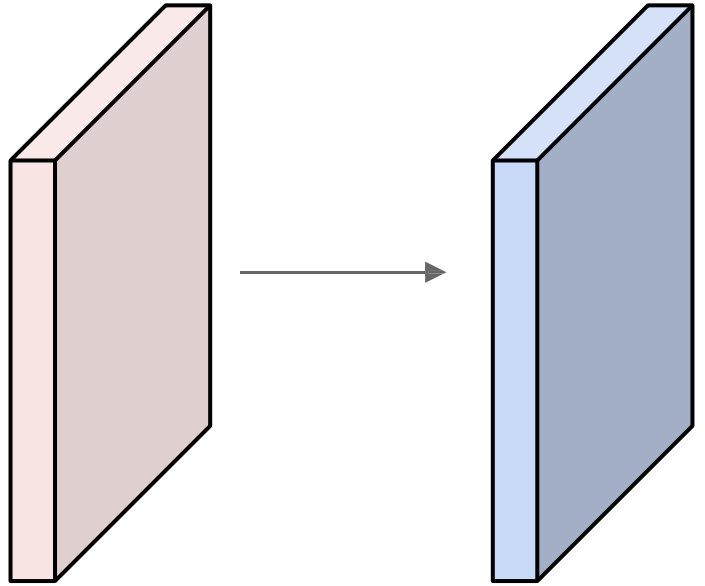
Output volume size:
(32+2*2-5)/1+1 = 32 spatially, so 32x32x10



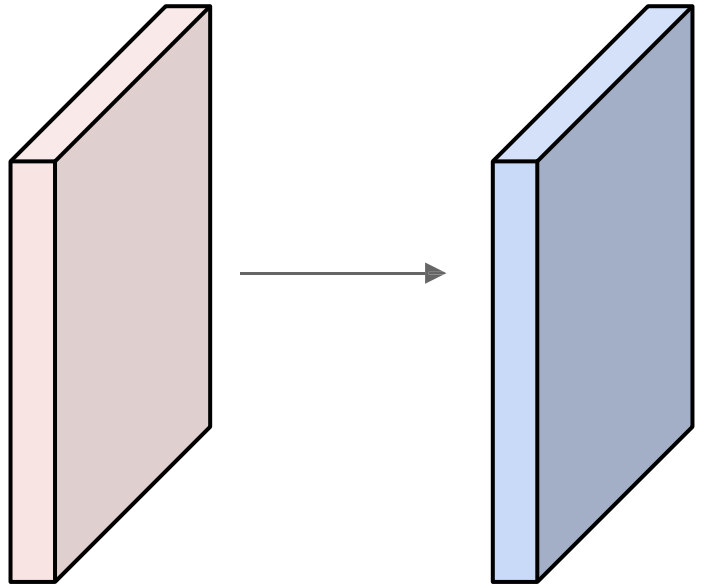
Examples time:

Input volume: 32x32x3
10 5x5 filters with stride 1, pad 2

Number of parameters in this layer?



Examples time:



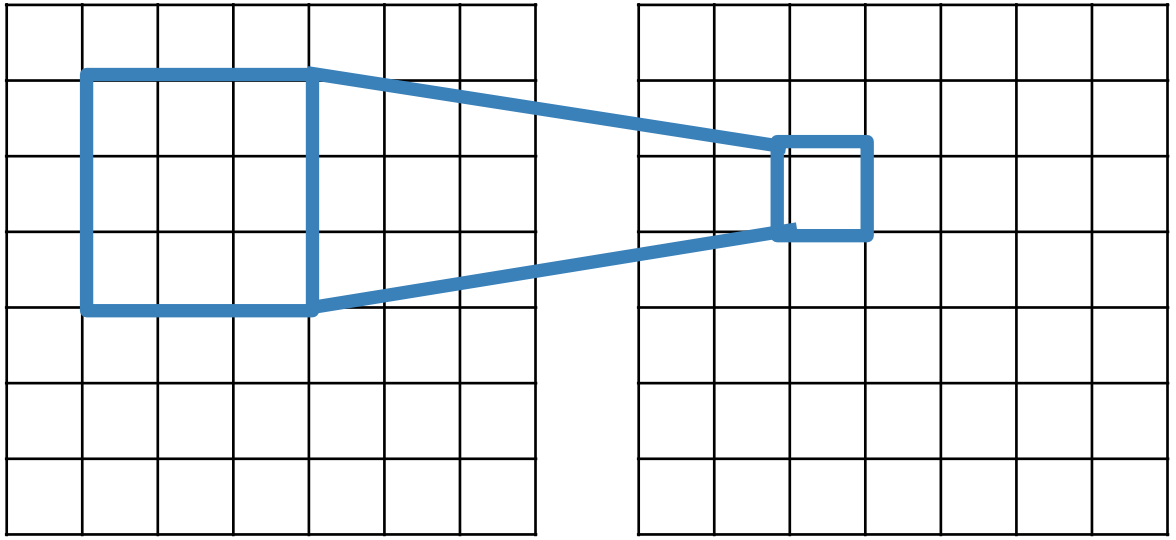
Input volume: 32x32x3
10 5x5 filters with stride 1, pad 2

Number of parameters in this layer? each filter has 5*5*3+1 =76 params
=>76*10=760

(+1 for bias)

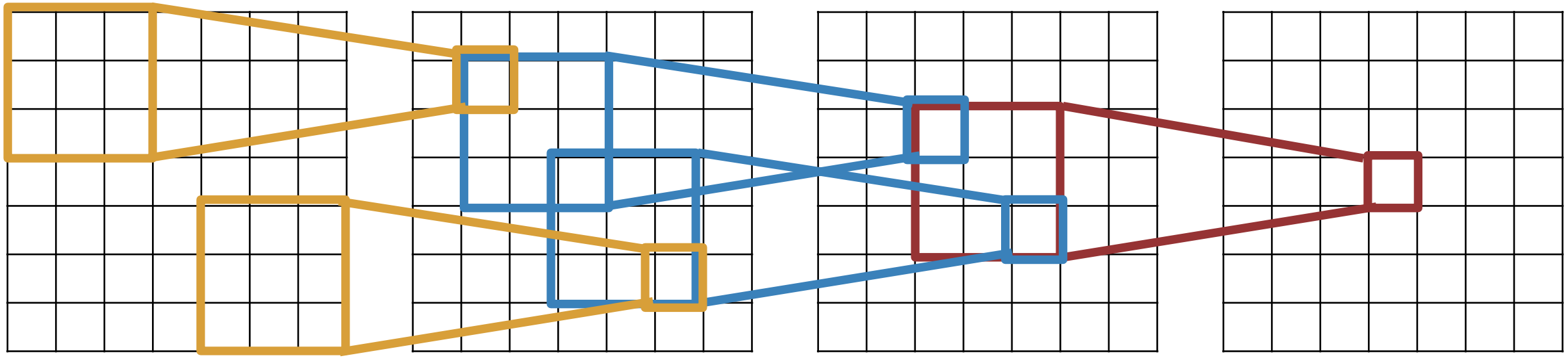
Receptive Fields

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



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)

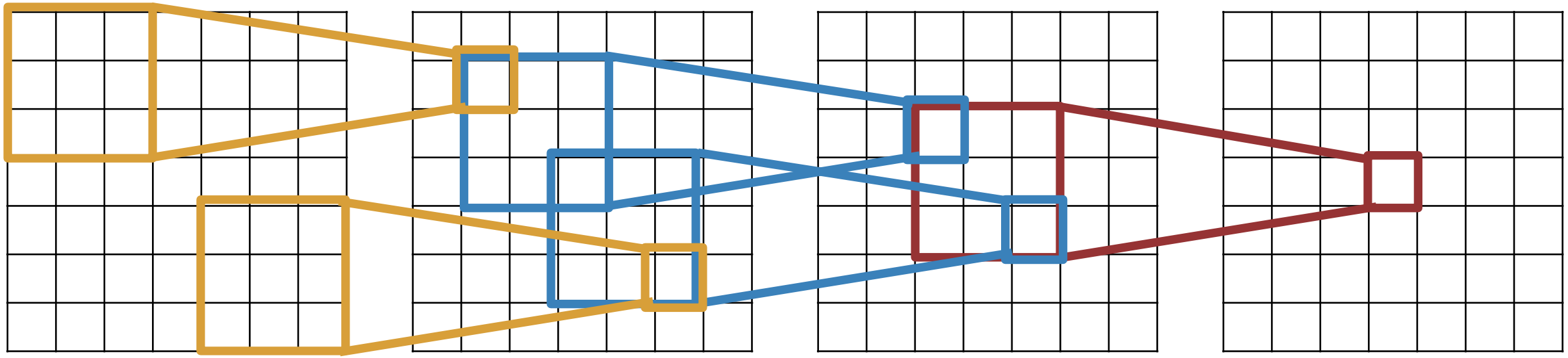


Be careful – "receptive field in the input" vs. "receptive field in the previous layer"

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)



Input

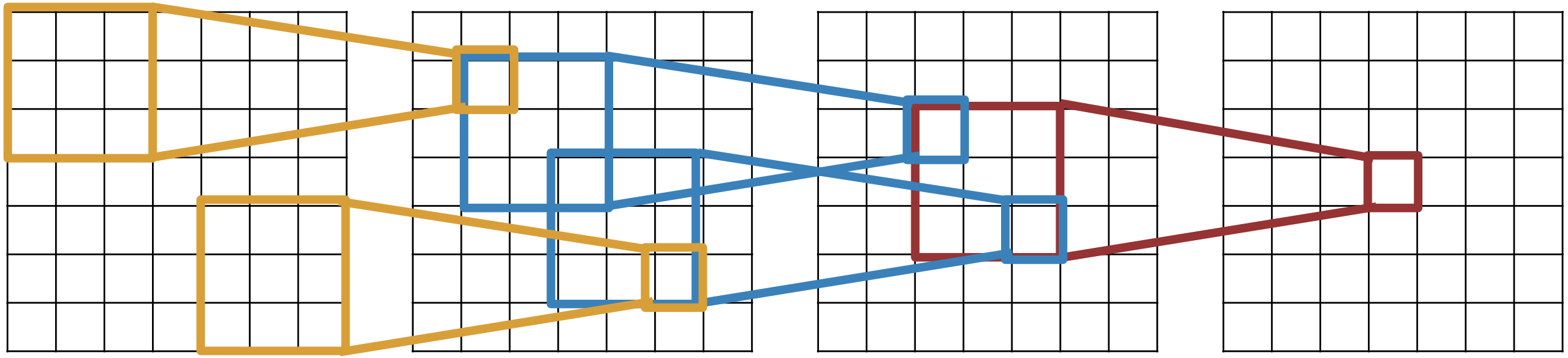
Output

Problem: For large images we need many layers for each output to “see” the whole 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)



Input

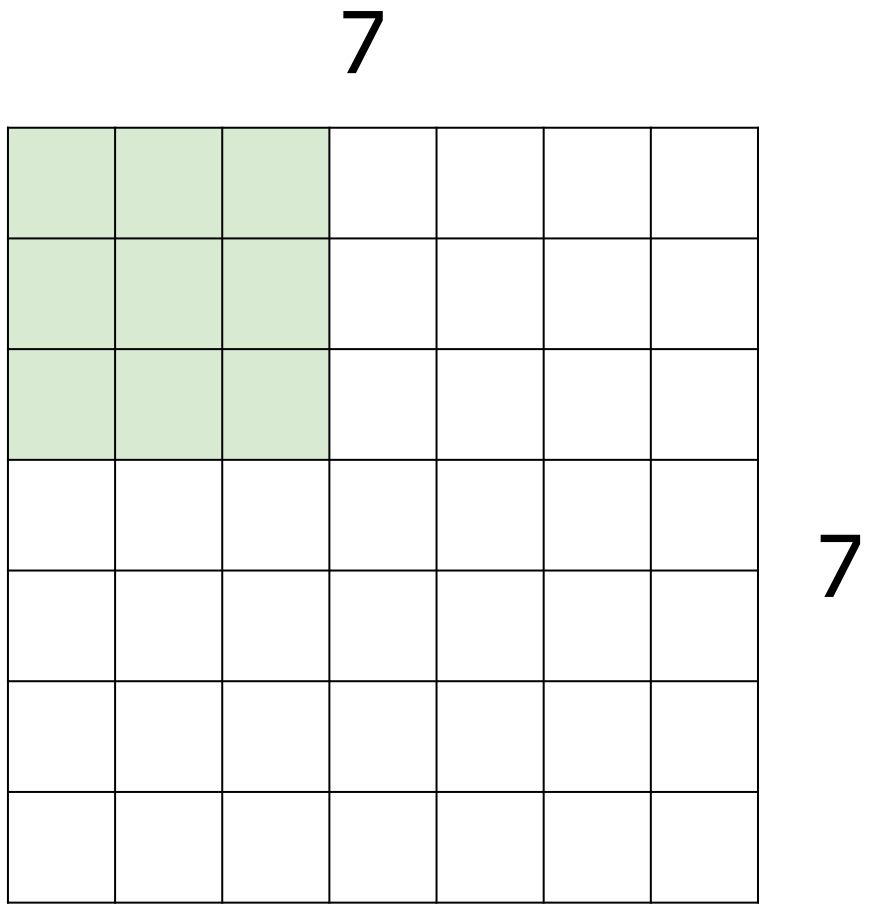
Output

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

Solution: Downsample inside the network

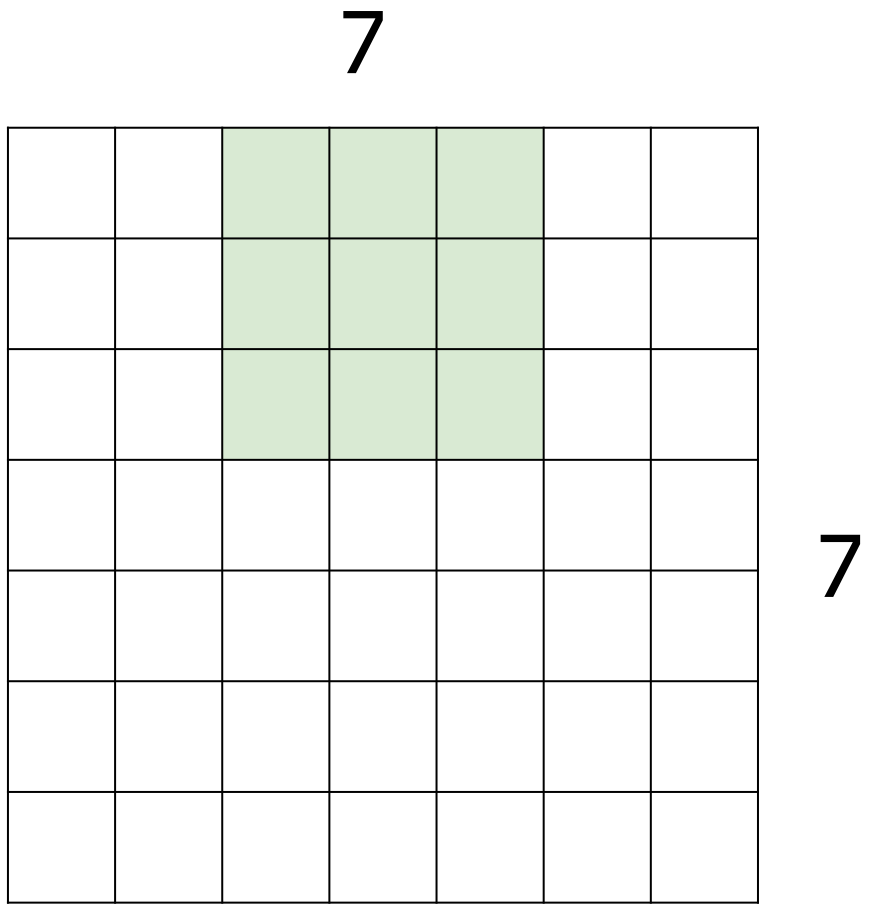
Slide inspiration: Justin Johnson

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

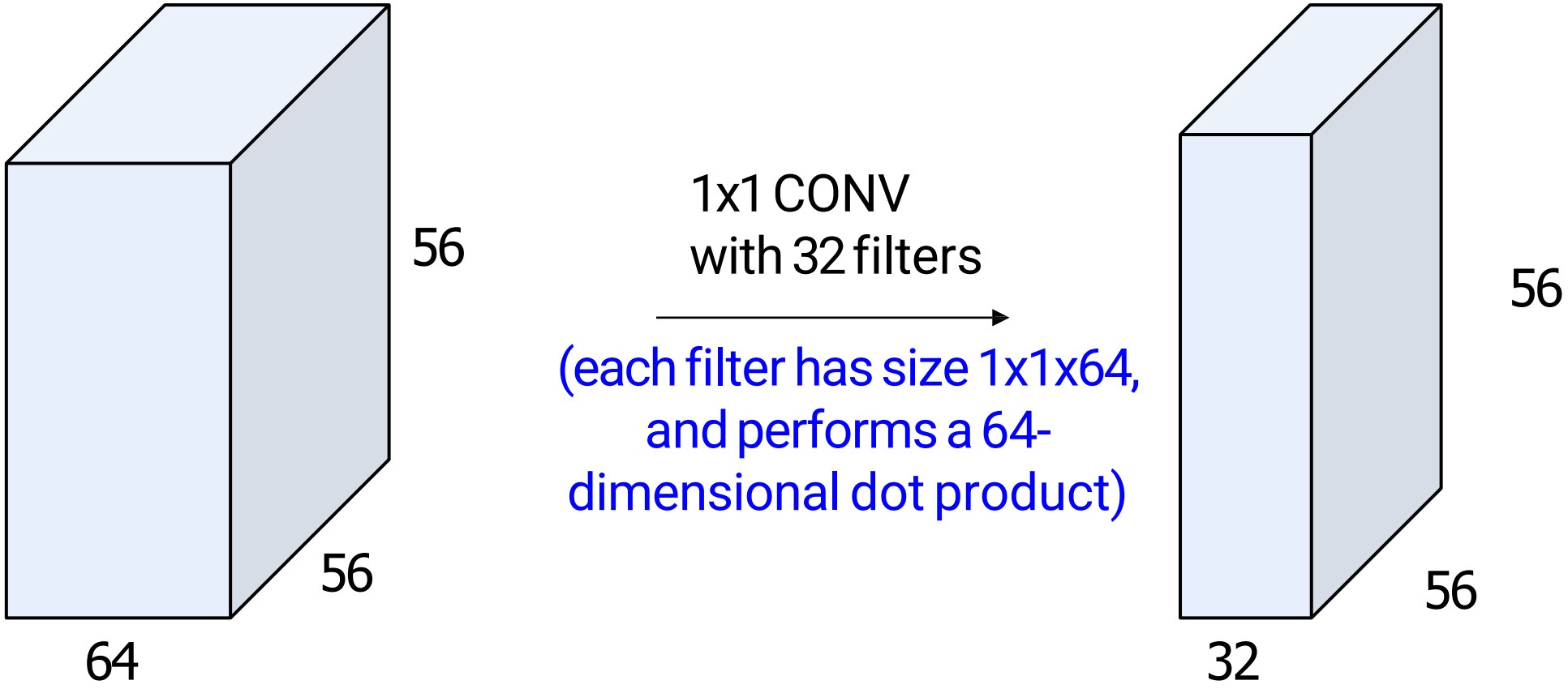
This will produce an output of $W_2 \times H_2 \times K$ where:

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

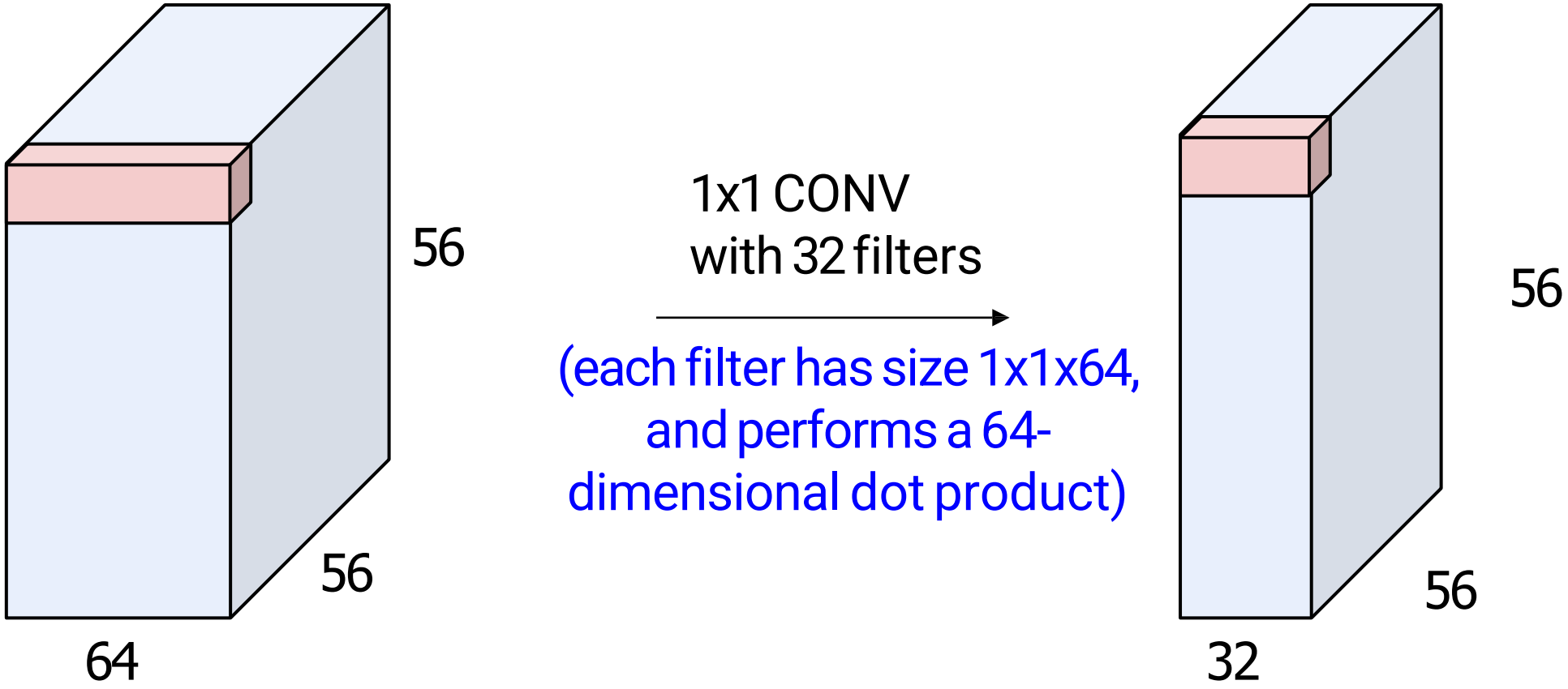
Number of parameters: F^2CK and K biases

Common settings:
$K_H = K_W$ (Small square filters)
P = (K – 1) / 2 ("Same" padding)
C_{in}, C_{out} = 32, 64, 128, 256 (powers of 2)
K = 3, P = 1, S = 1 (3x3 conv)
K = 5, P = 2, S = 1 (5x5 conv)
K = 1, P = 0, S = 1 (1x1 conv)
K = 3, P = 1, S = 2 (Downsample by 2)

(btw, 1x1 convolution layers make perfect sense)



(btw, 1x1 convolution layers make perfect sense)



Example: CONV layer in PyTorch

Conv layer needs 4 hyperparameters:

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

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_{in}, H, W) and output $(N, C_{\text{out}}, H_{\text{out}}, W_{\text{out}})$ can be precisely described as:

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

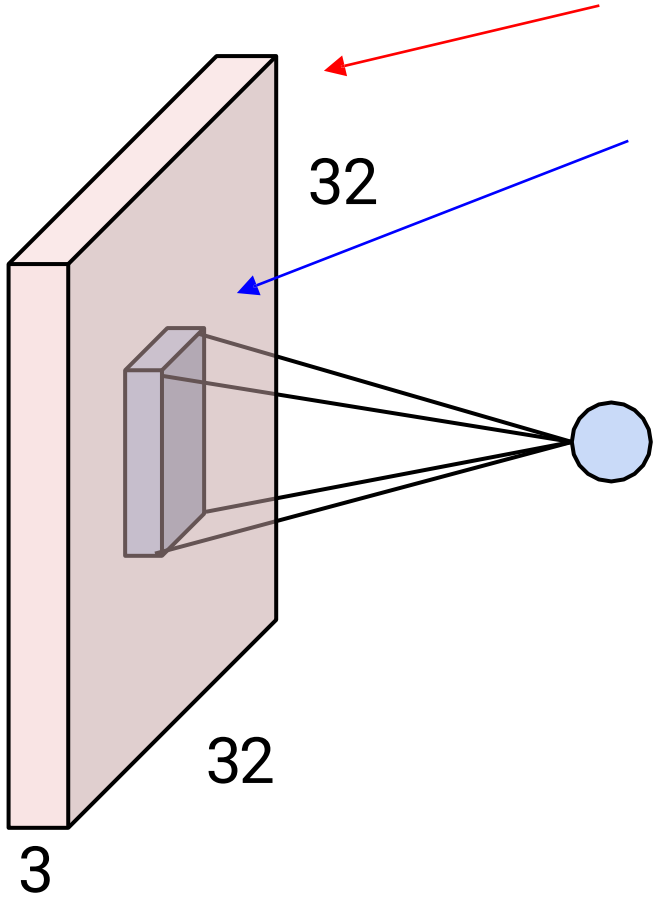
where $\star$ is the valid 2D cross-correlation operator, N is a batch size, C denotes a number of channels, H is a height of input planes in pixels, and W is width in pixels.

- `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:

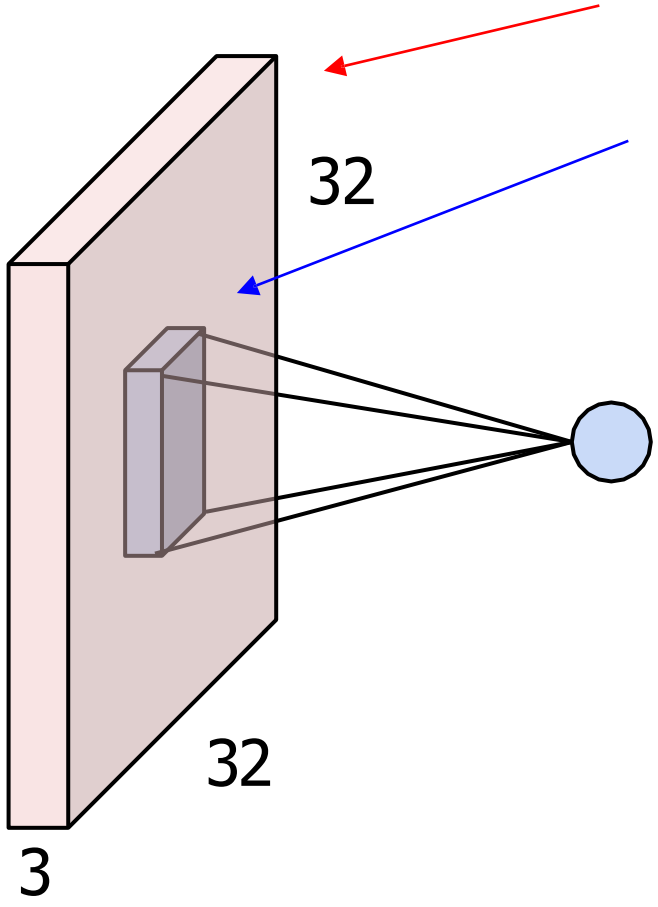
- a single `int` – in which case the same value is used for the height and width dimension
- a `tuple` of two ints – in which case, the first *int* is used for the height dimension, and the second *int* for the width dimension

The brain/neuron view of CONV Layer



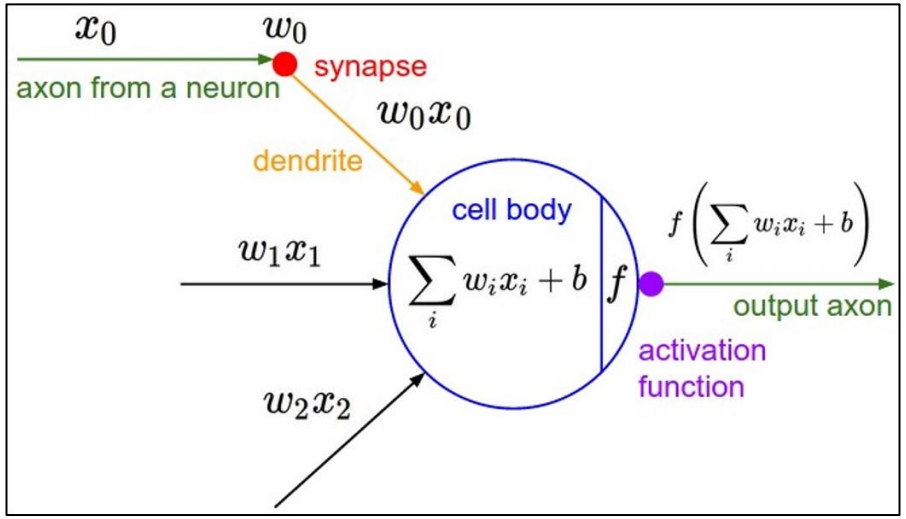
1 number:
the result of taking a dot product between the filter and this part of the image
(i.e. 5*5*3 =75-dimensional dot product)

The brain/neuron view of CONV Layer



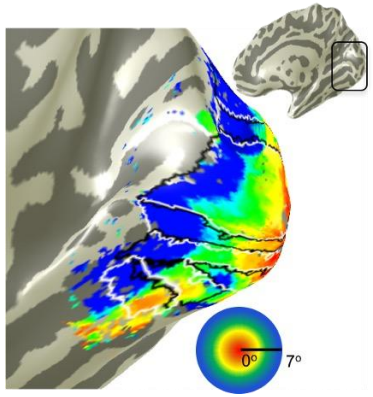
32x32x3 image

5x5x3 filter

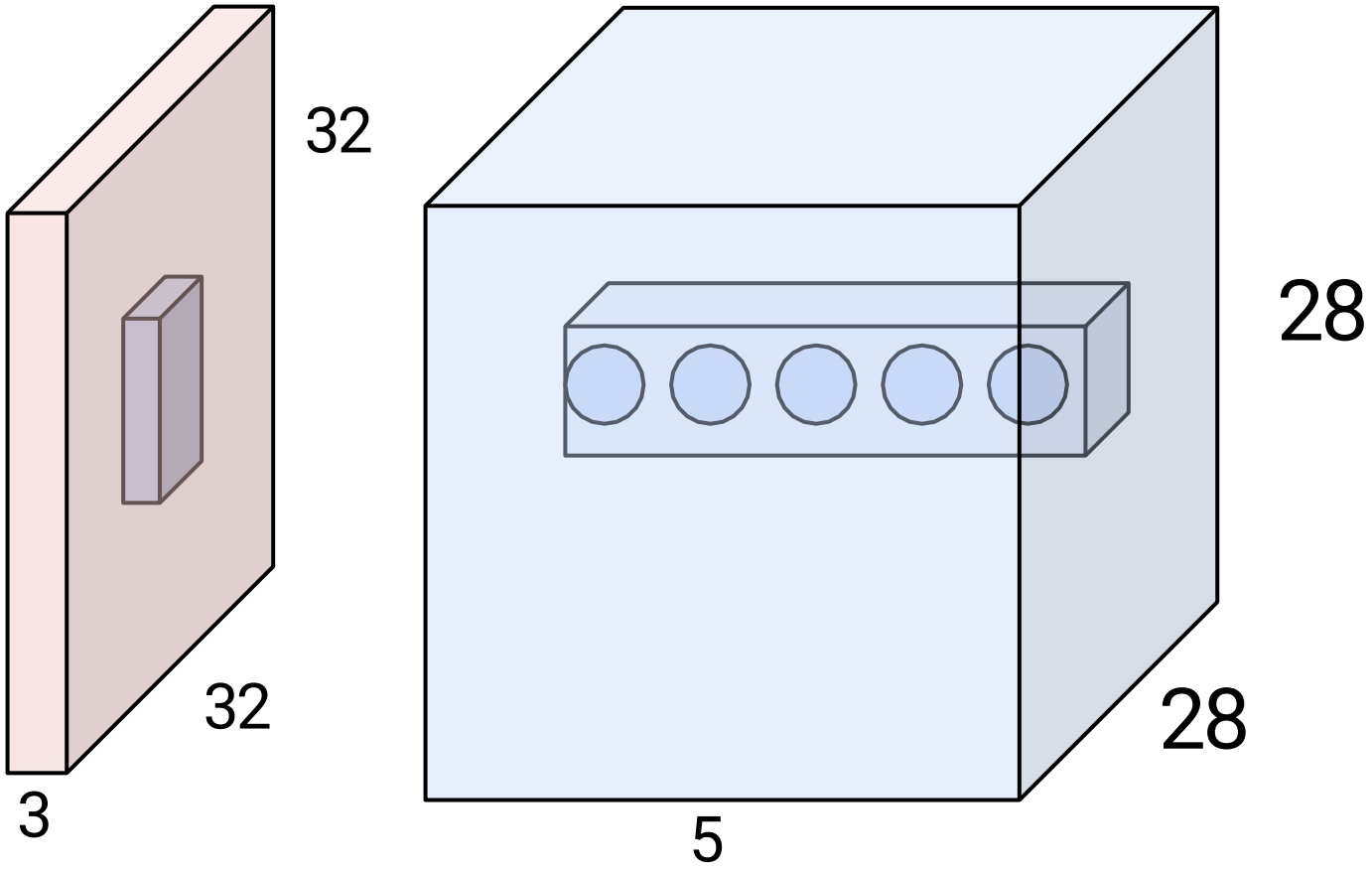


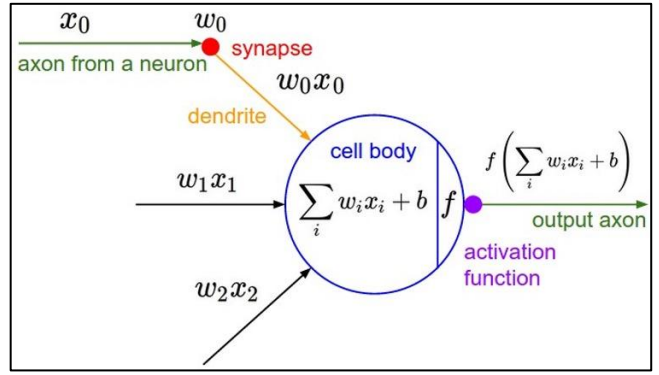
It's just a neuron with local connectivity...

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





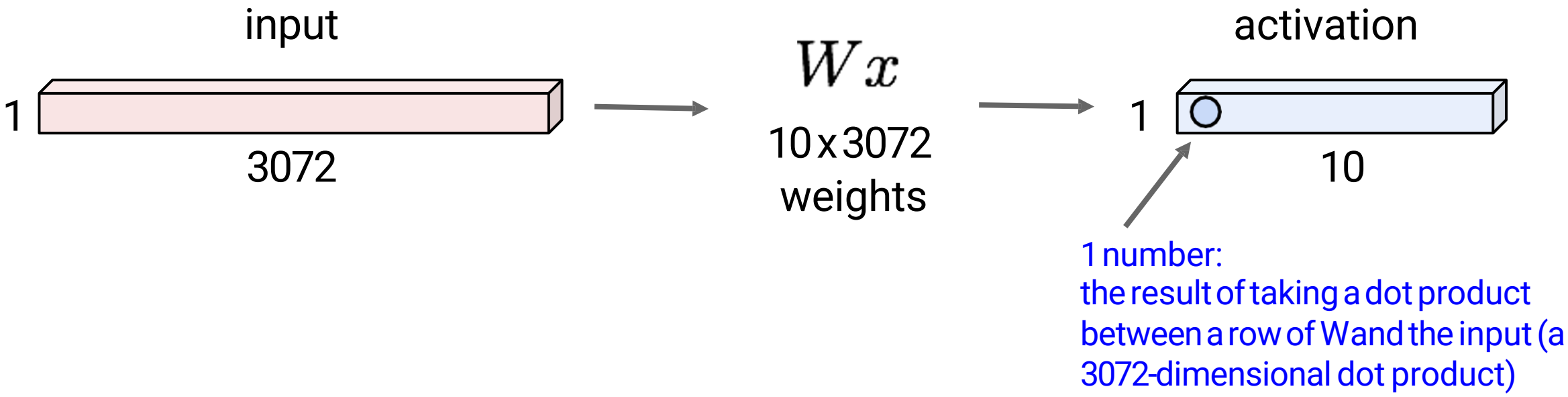
E.g. with 5 filters,
CONV layer consists of neurons arranged in a 3D grid (28x28x5)

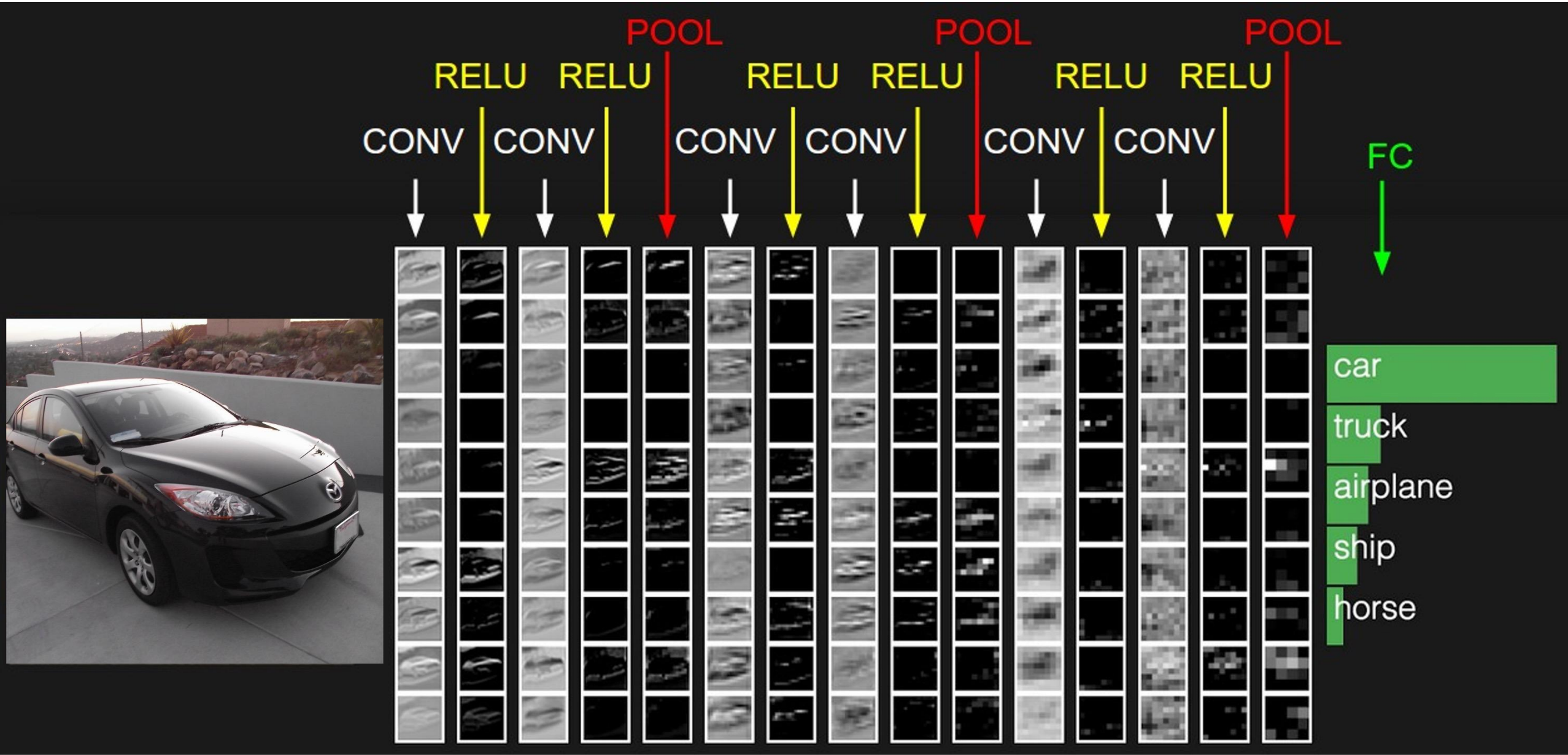
There will be 5 different neurons all looking at the same region in the input volume

Reminder: Fully Connected Layer

32x32x3 image ->stretch to 3072 x 1

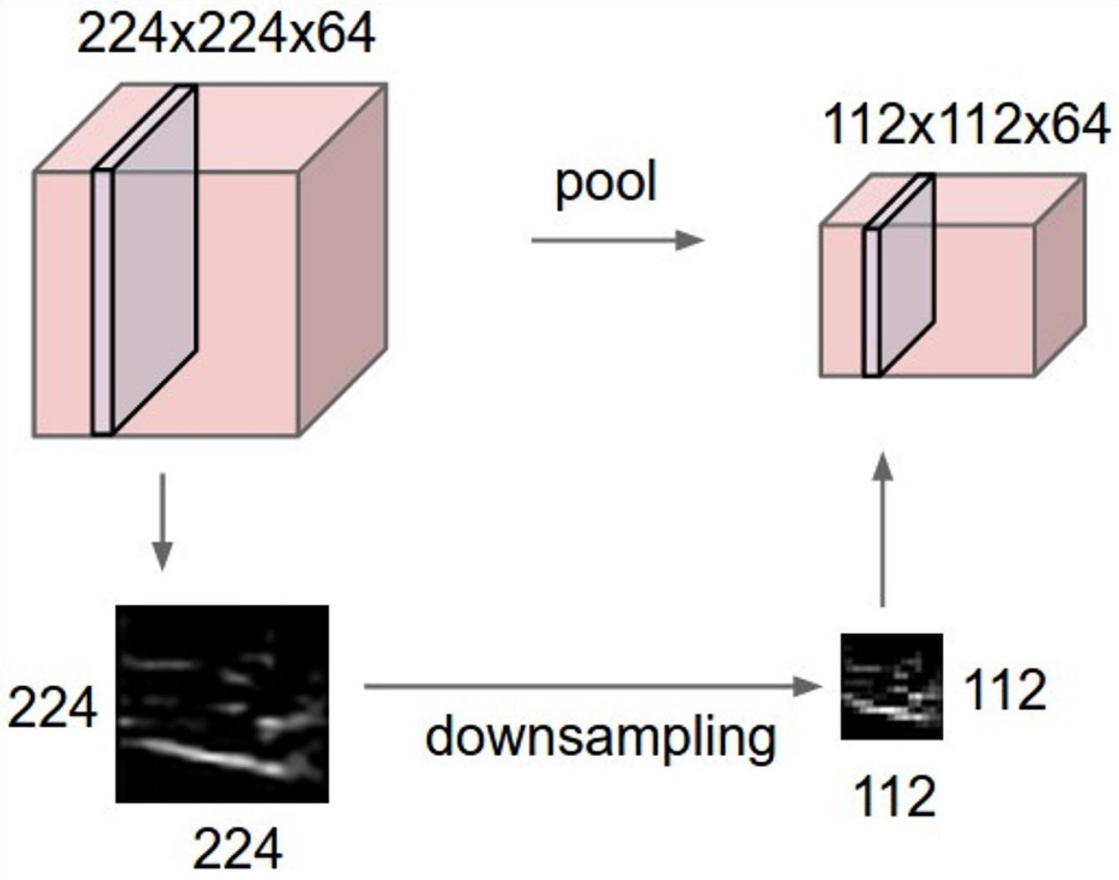
Each neuron looks at the full input volume




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

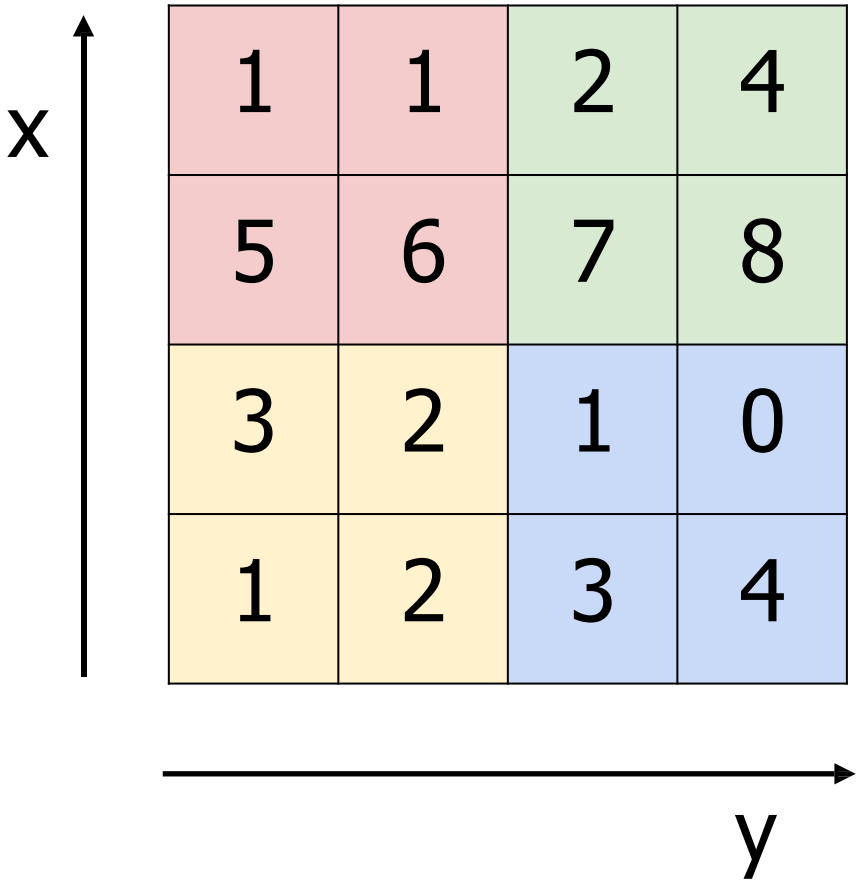
Pooling layer

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

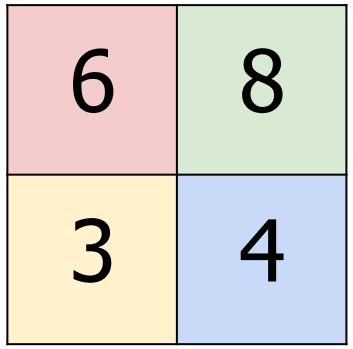


MAXPOOLING

Single depth slice

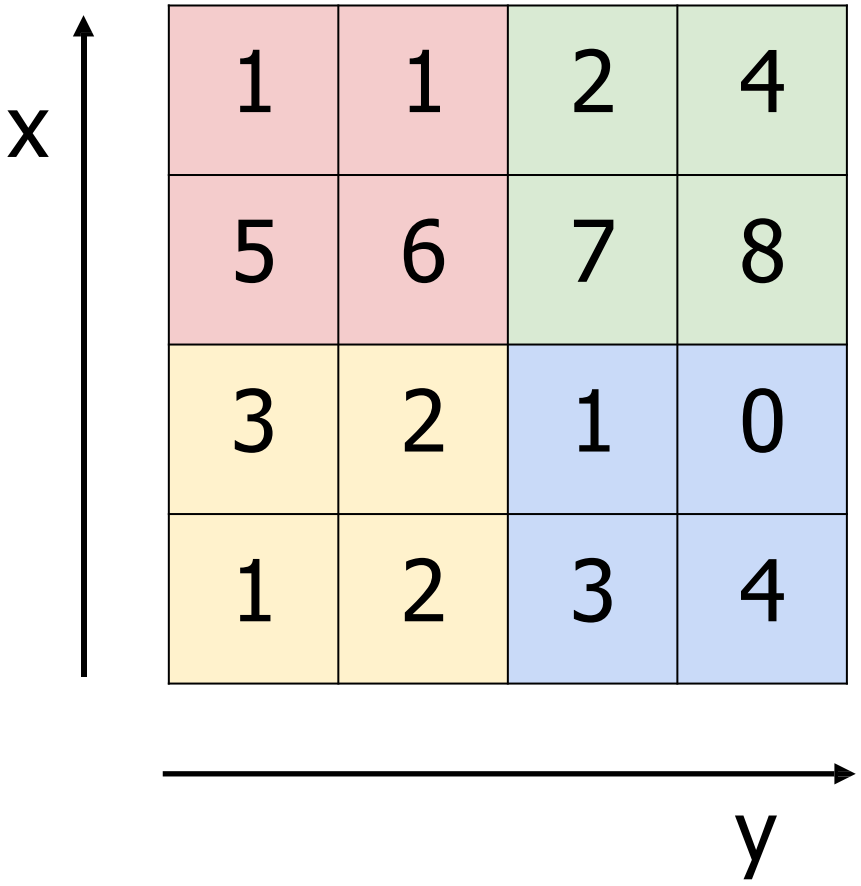


max pool with 2x2 filters
and stride 2

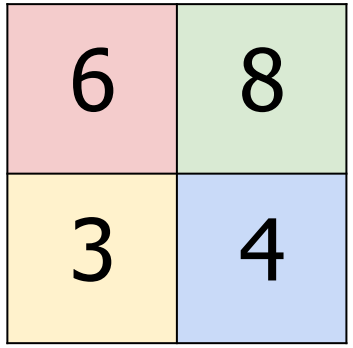


MAXPOOLING

Single depth slice

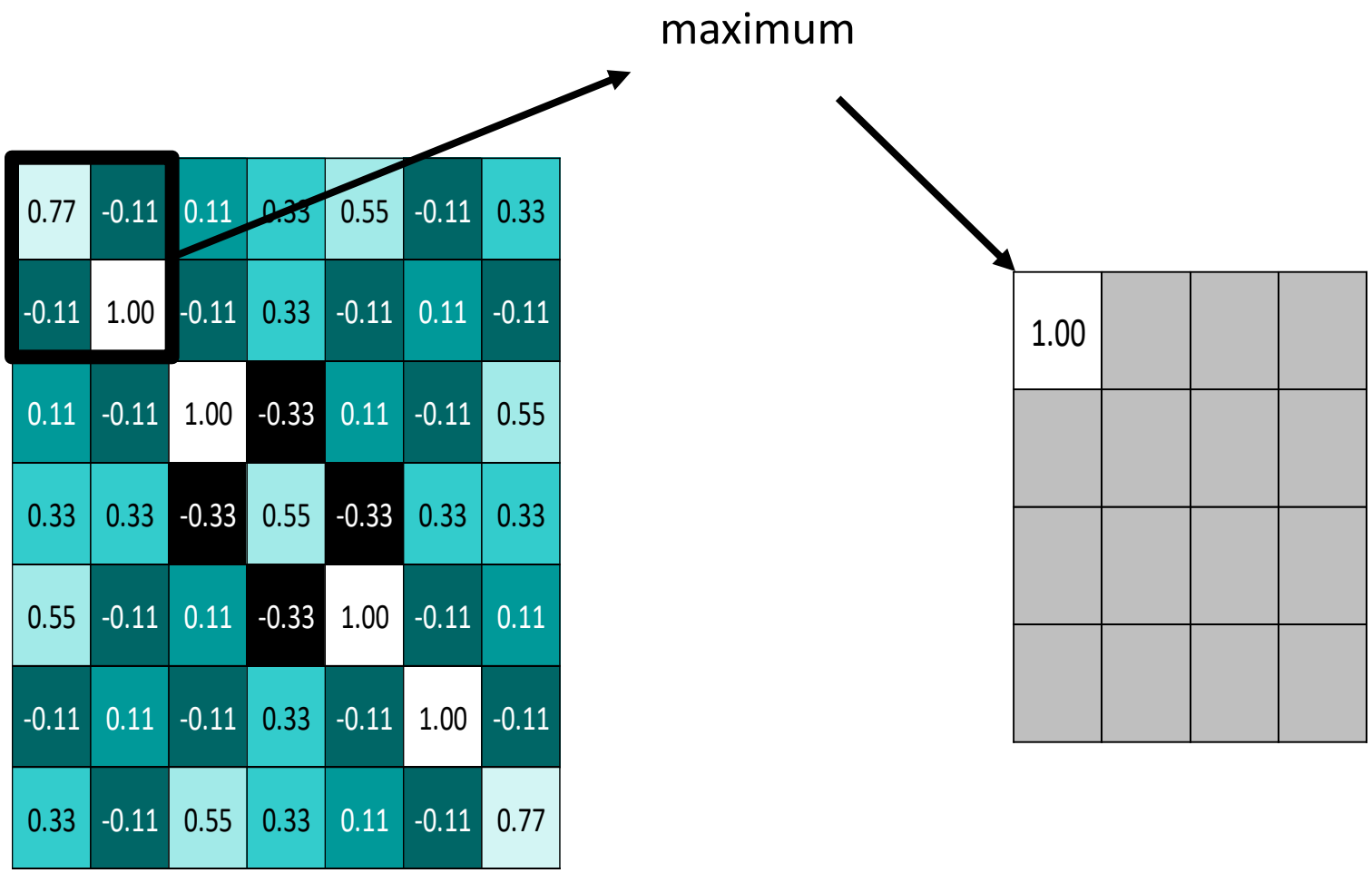


max pool with 2x2 filters
and stride 2



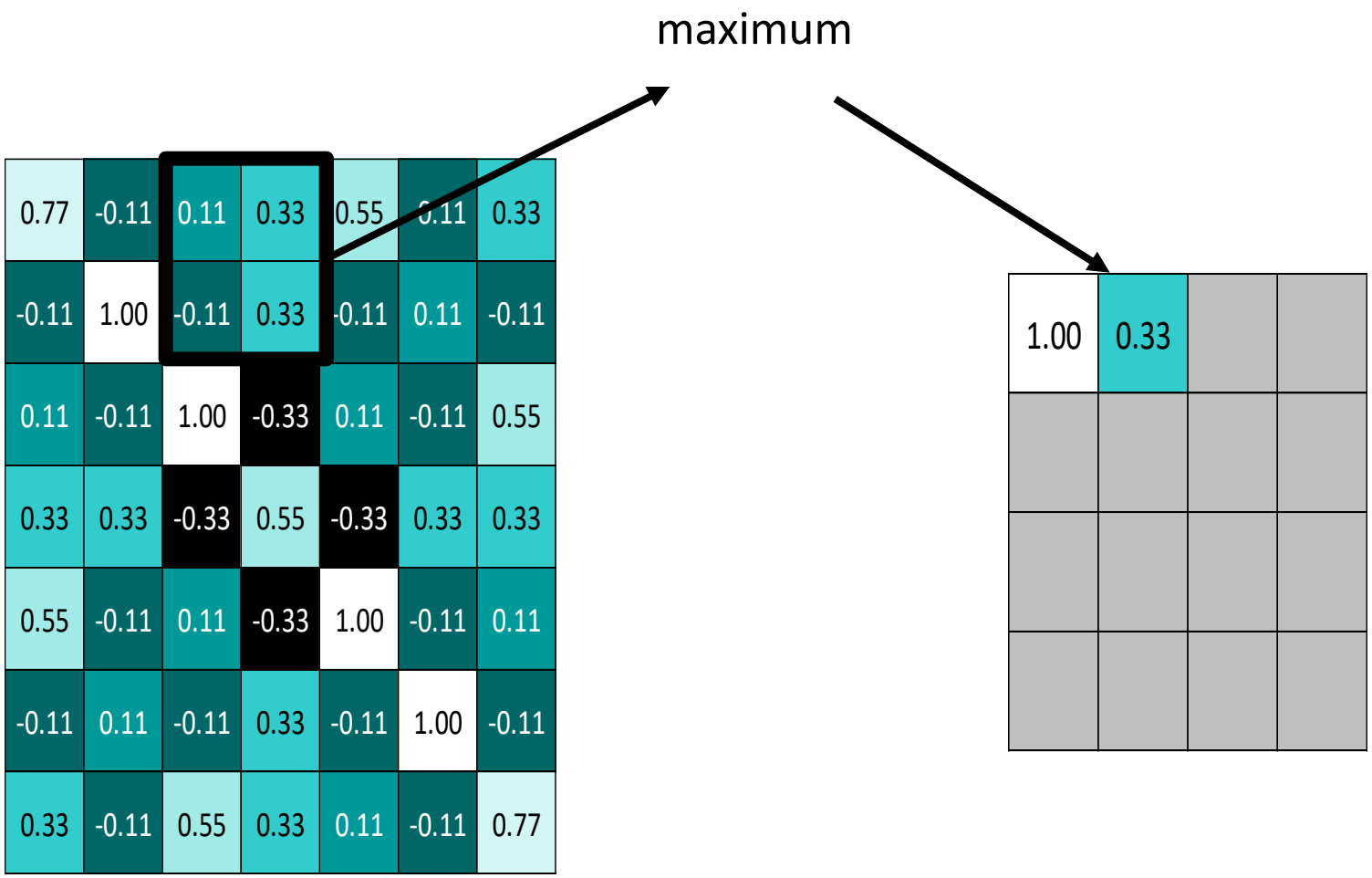
- No learnable parameters
- Introduces spatial invariance

Pooling



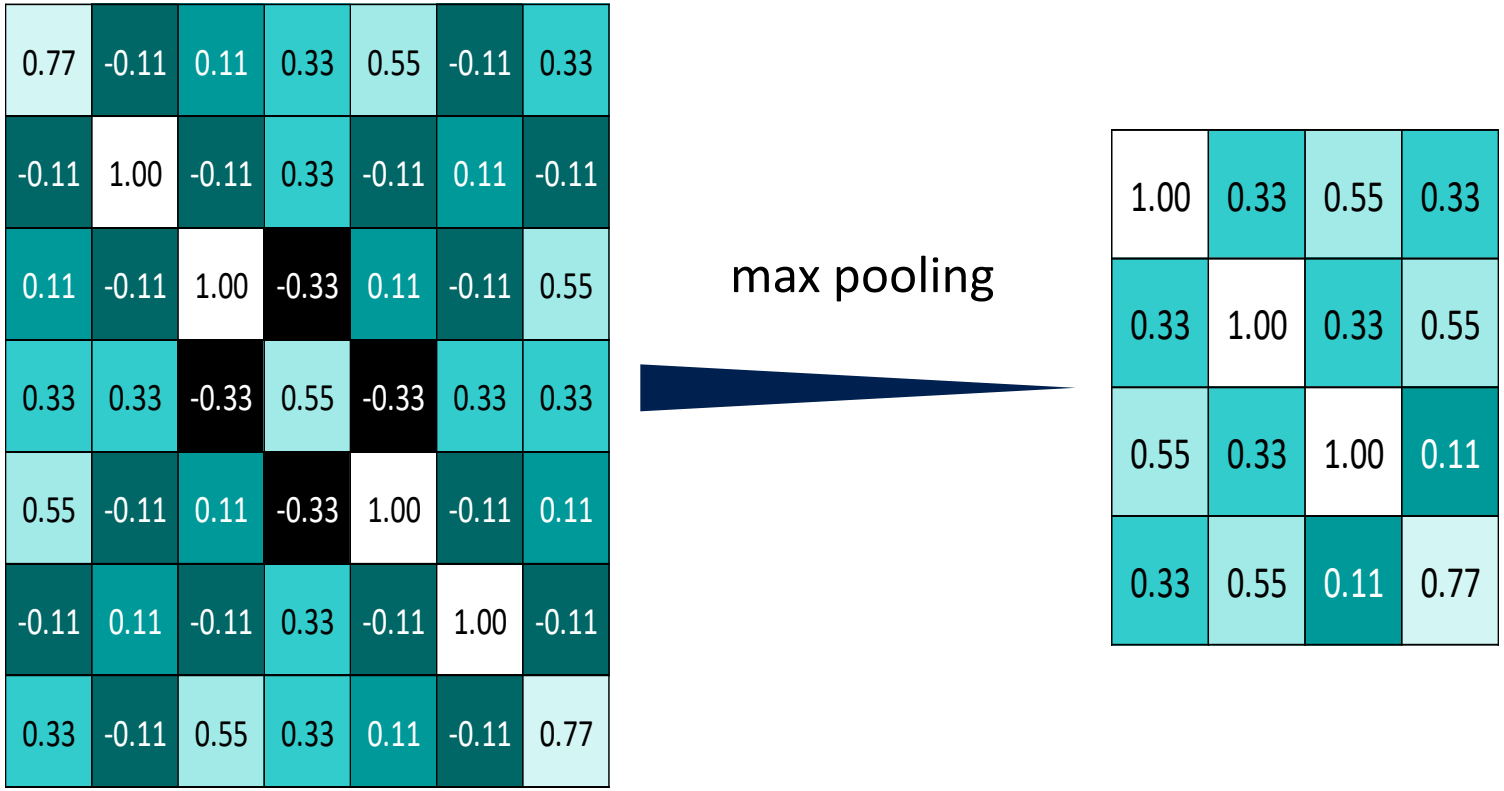
Brandon Rohrer https://brohrer.github.io/how_convolutional_neural_networks_work.html

Pooling



Brandon Rohrer https://brohrer.github.io/how_convolutional_neural_networks_work.html

Pooling


0.77 -0.11 0.11 0.33 0.55 -0.11 0.33
-0.11 1.00 -0.11 0.33 -0.11 0.11 -0.11
0.11 -0.11 1.00 -0.33 0.11 -0.11 0.55
0.33 0.33 -0.33 0.55 -0.33 0.33 0.33
0.55 -0.11 0.11 -0.33 1.00 -0.11 0.11
-0.11 0.11 -0.11 0.33 -0.11 1.00 -0.11
0.33 -0.11 0.55 0.33 0.11 -0.11 0.77
max pooling
1.00 0.33 0.55 0.33
0.33 1.00 0.33 0.55
0.55 0.33 1.00 0.11
0.33 0.55 0.11 0.77

Pooling layer: summary

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

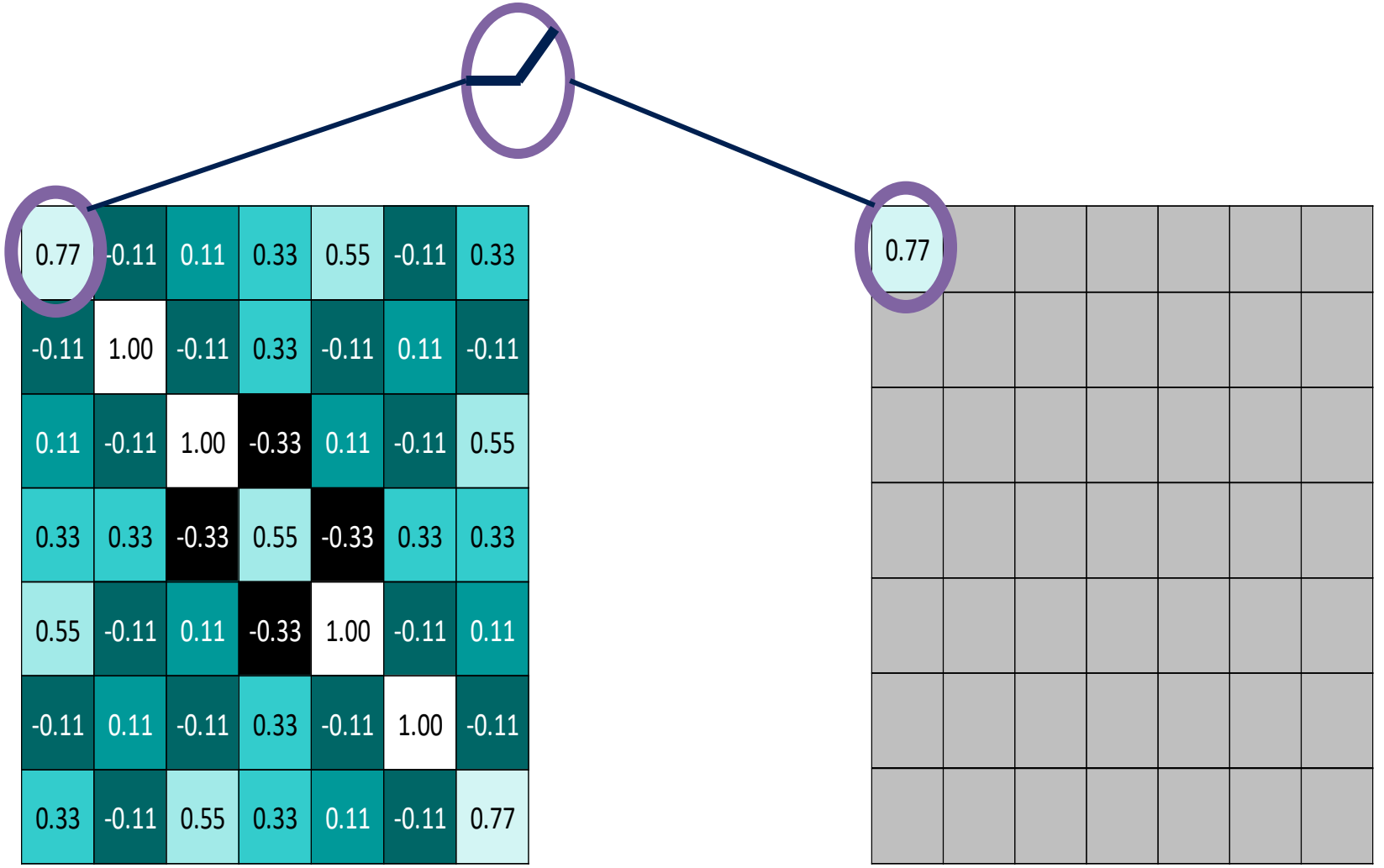
- The spatial extent F
- The stride S

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

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

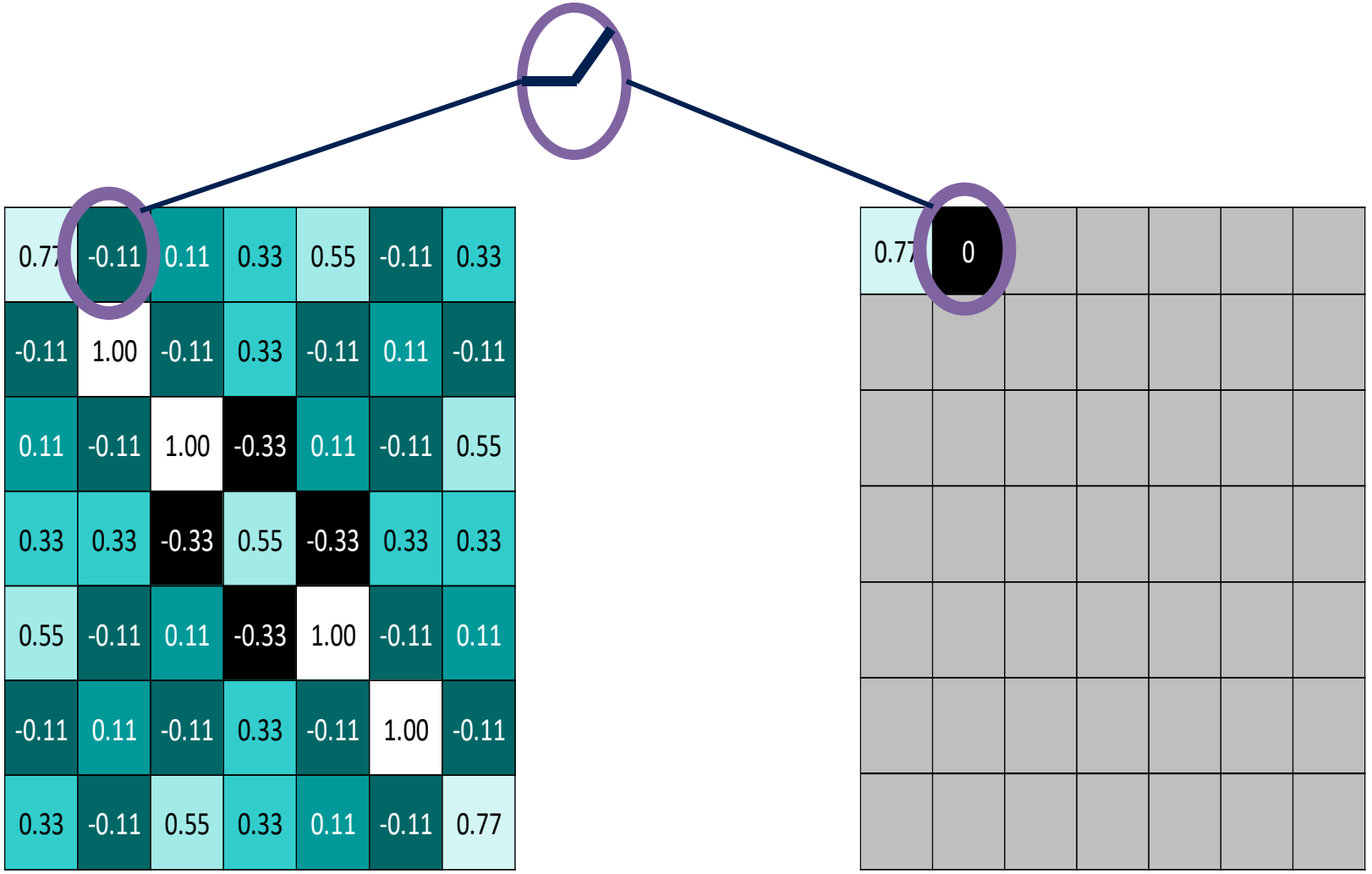
Number of parameters: 0

Rectified Linear Units (ReLUs)



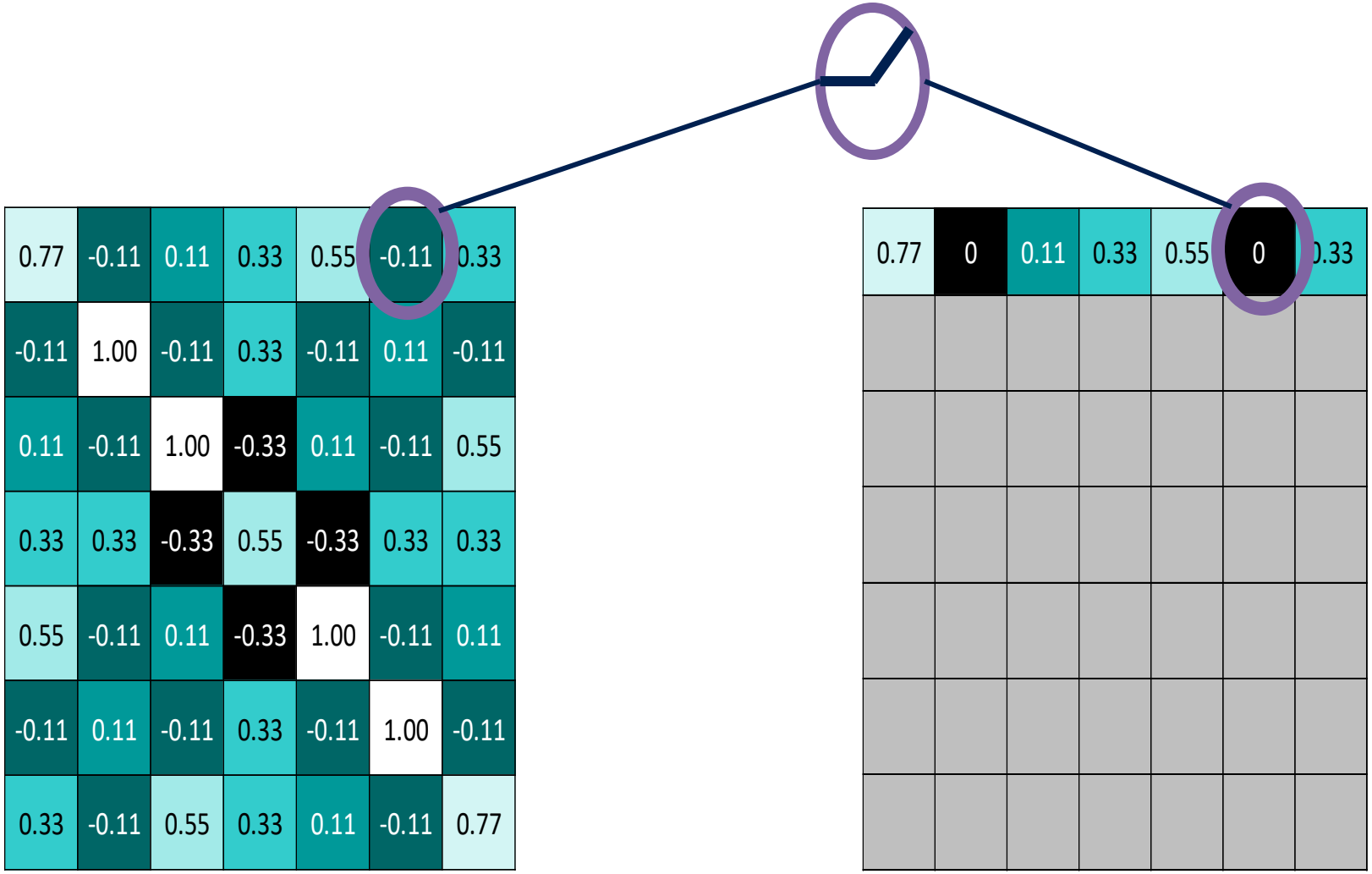
Brandon Rohrer https://brohrer.github.io/how_convolutional_neural_networks_work.html

Rectified Linear Units (ReLUs)



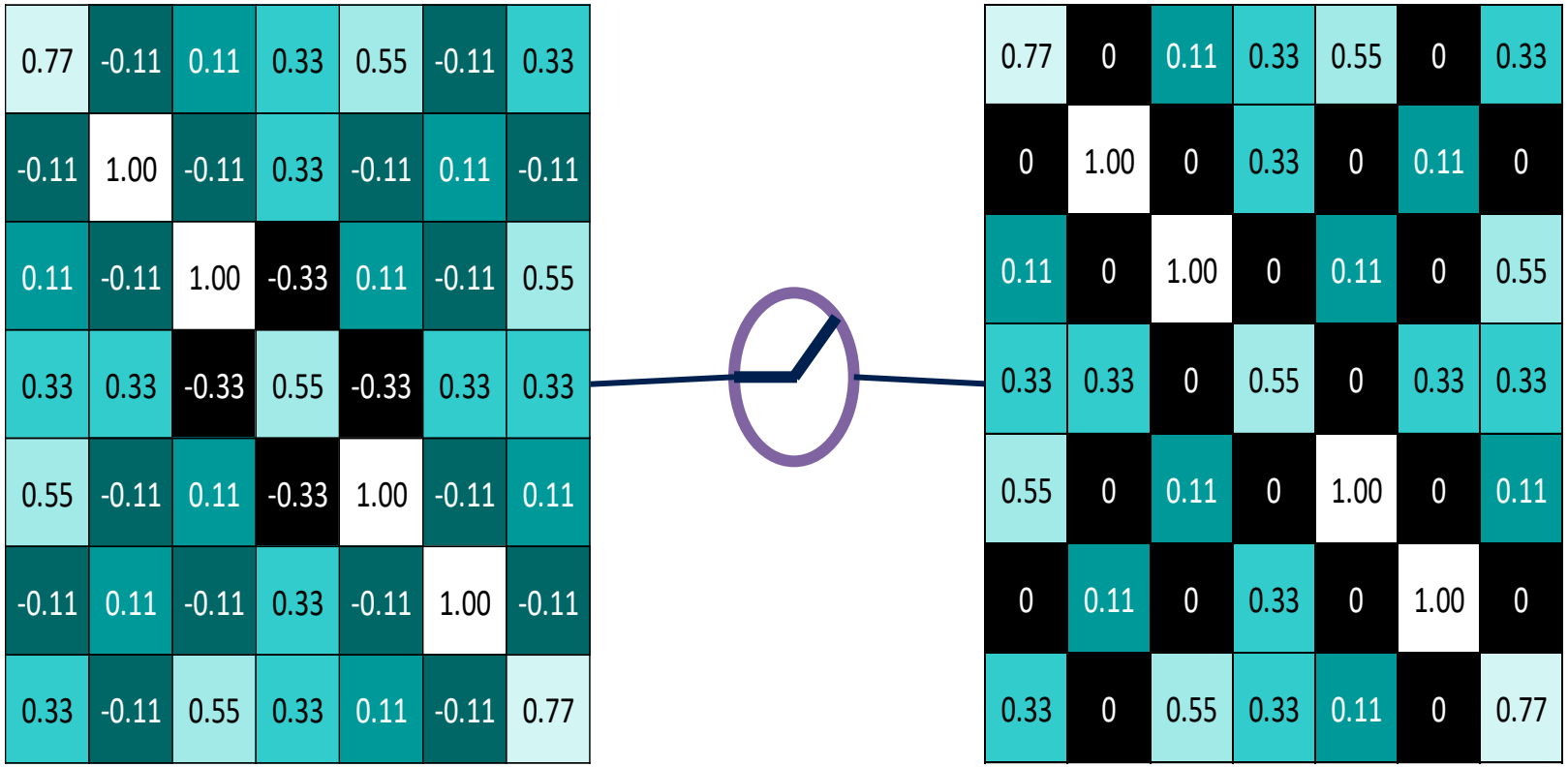
Brandon Rohrer https://brohrer.github.io/how_convolutional_neural_networks_work.html

Rectified Linear Units (ReLUs)



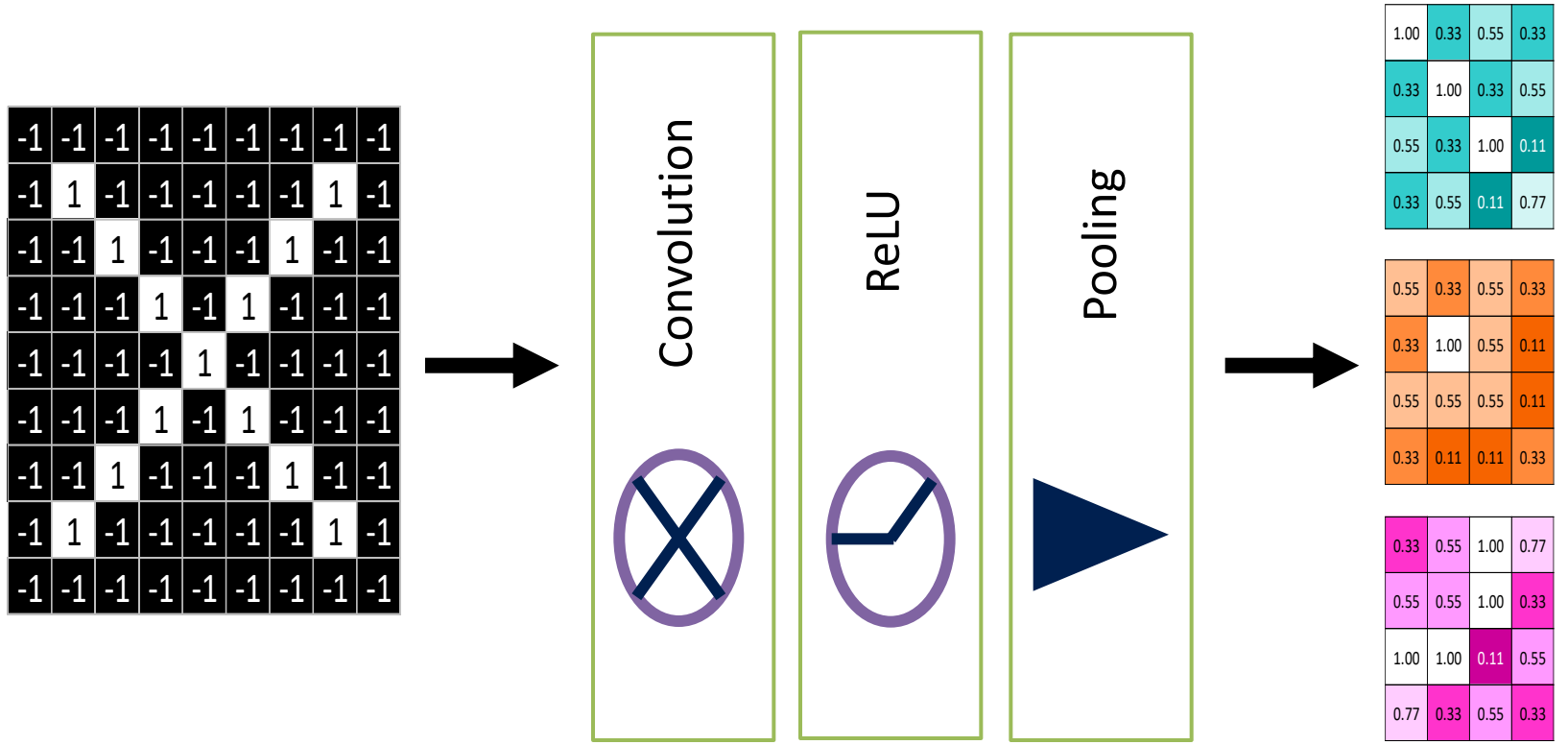
Brandon Rohrer https://brohrer.github.io/how_convolutional_neural_networks_work.html

Rectified Linear Units (ReLUs)



Layers get stacked

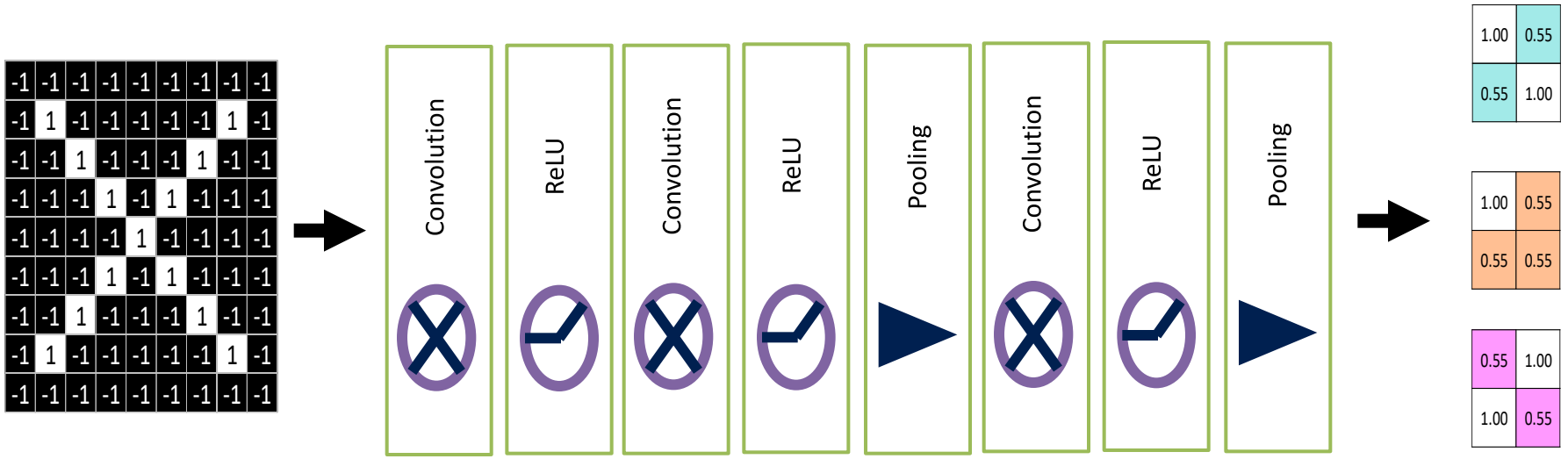
The output of one becomes the input of the next.



Brandon Rohrer https://brohrer.github.io/how_convolutional_neural_networks_work.html

Deep stacking

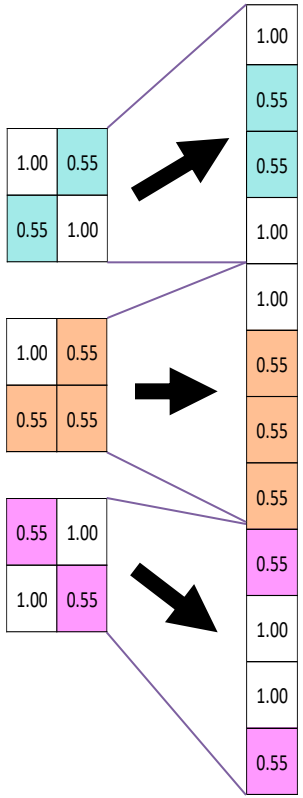
Layers can be repeated several (or many) times.



Brandon Rohrer https://brohrer.github.io/how_convolutional_neural_networks_work.html

Fully connected layer

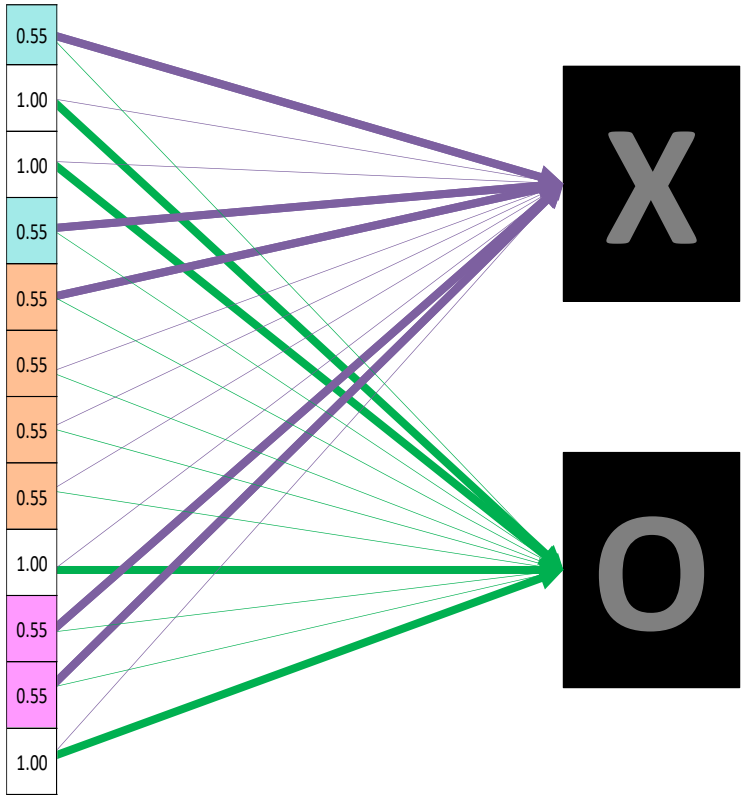
Every value gets a vote



Brandon Rohrer https://brohrer.github.io/how_convolutional_neural_networks_work.html

Fully connected layer

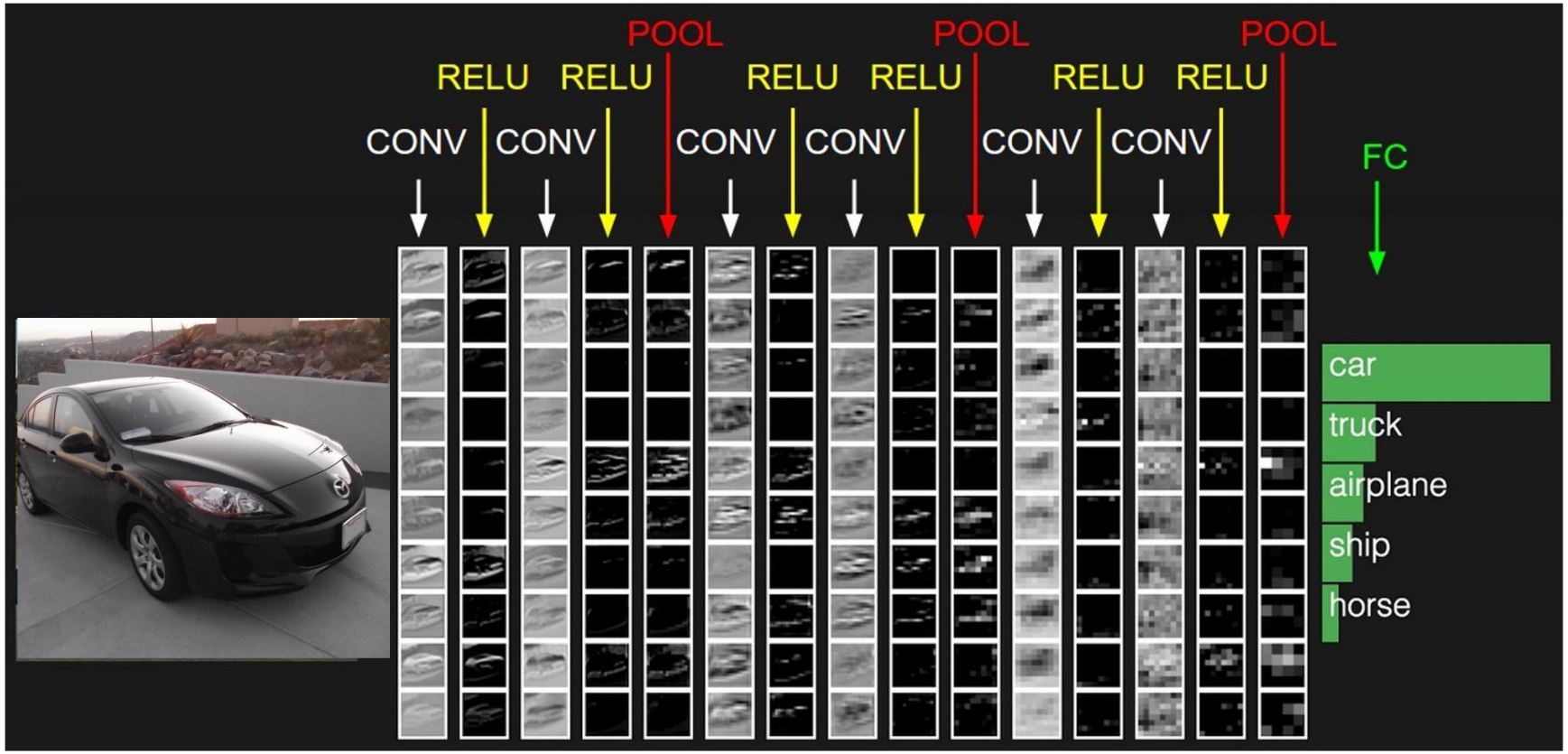
Vote depends on how strongly a value predicts X or O



Brandon Rohrer https://brohrer.github.io/how_convolutional_neural_networks_work.html

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.

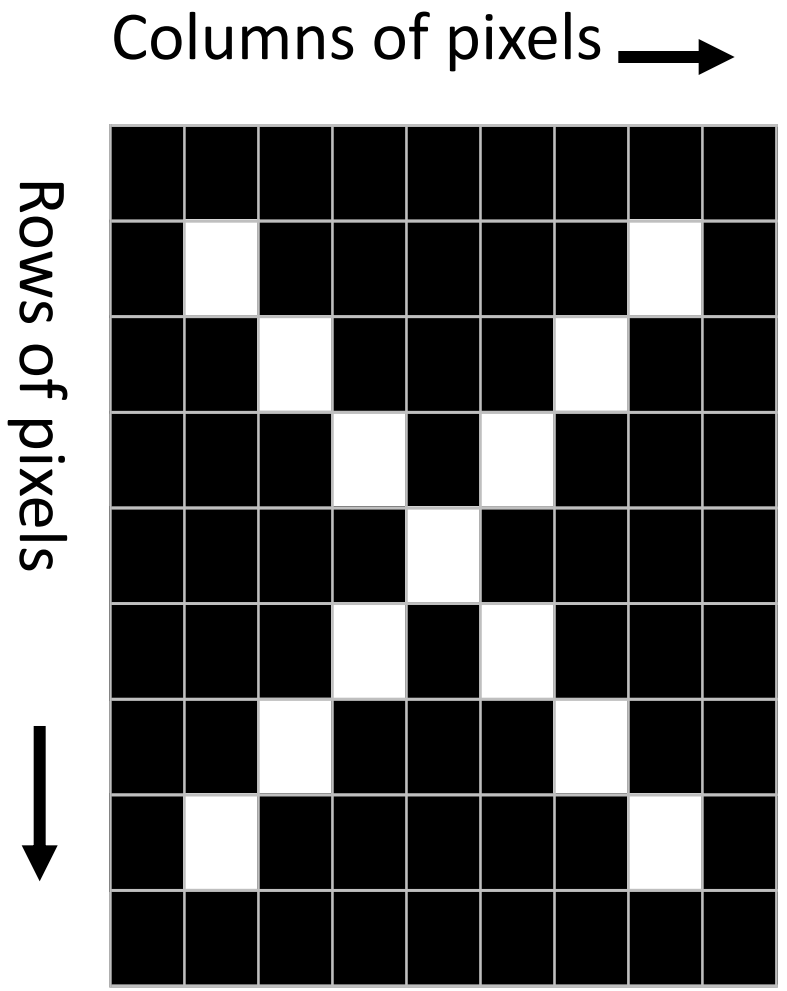
This dataset is more difficult and it takes longer to train a network. Data augmentation includes random flipping and random image shifts by up to 2px horizontally and verically.

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.

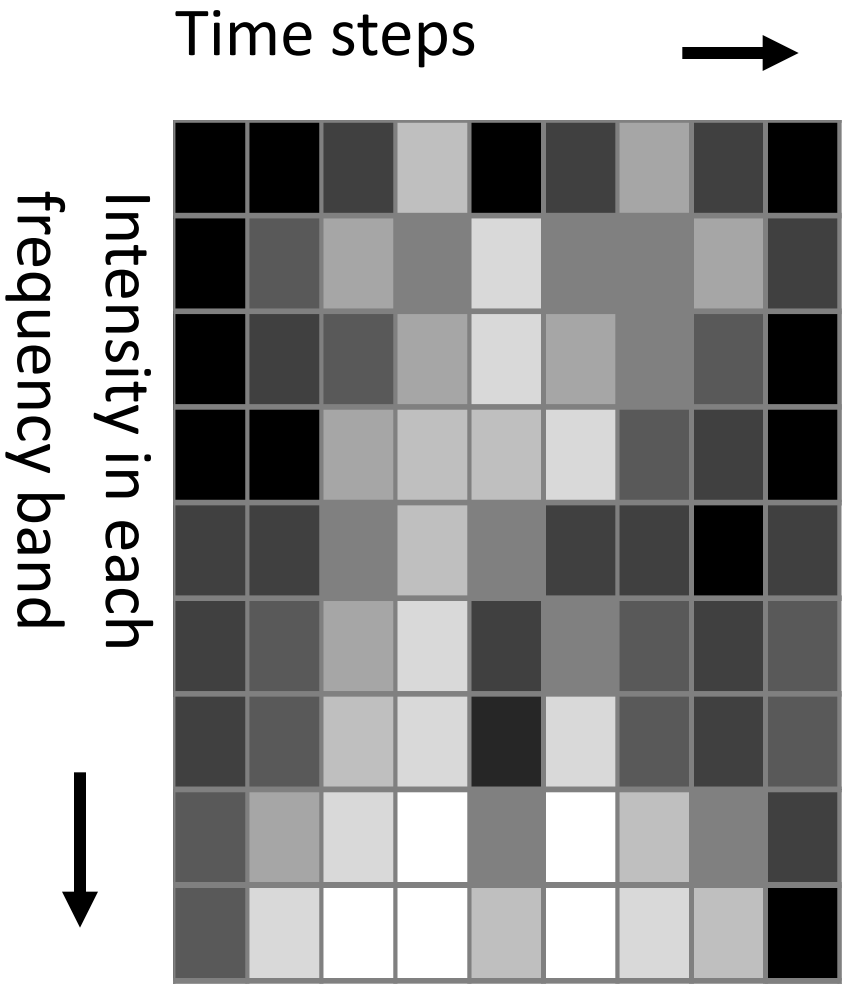
http://cs.stanford.edu/people/karpathy/convnetjs/demo/cifar10.html

Images



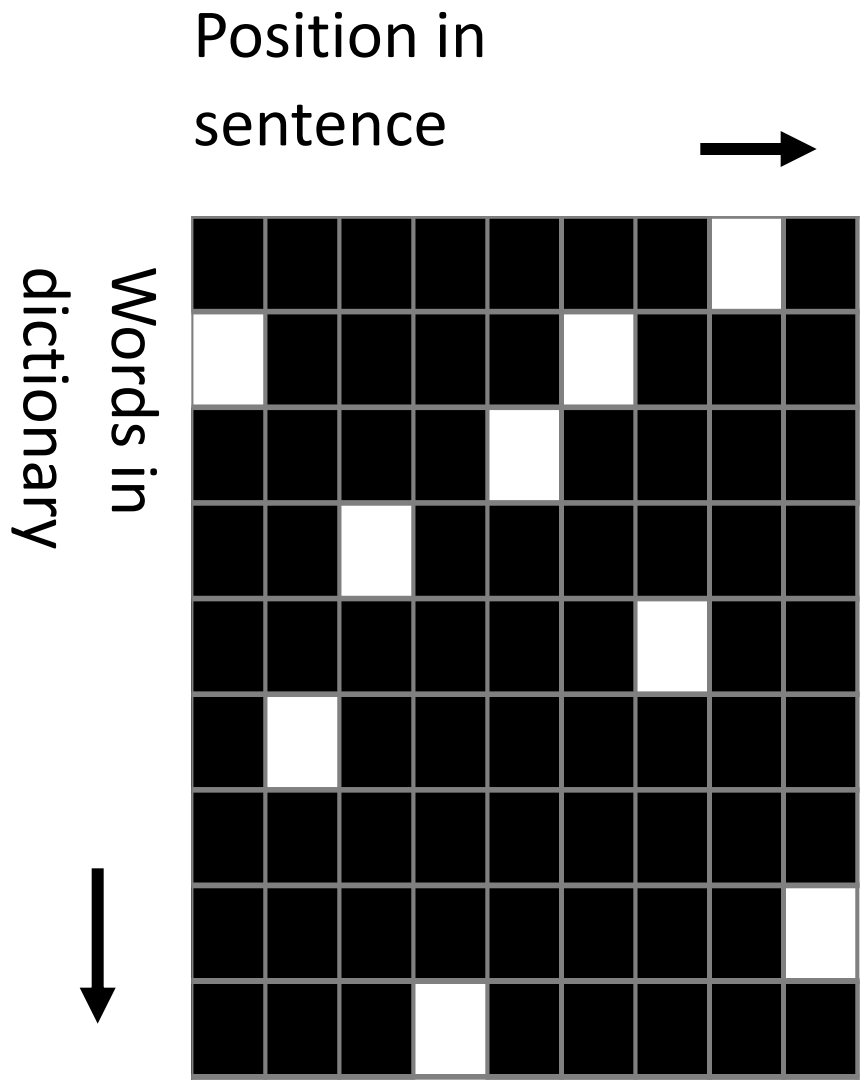
Brandon Rohrer https://brohrer.github.io/how_convolutional_neural_networks_work.html

Sound



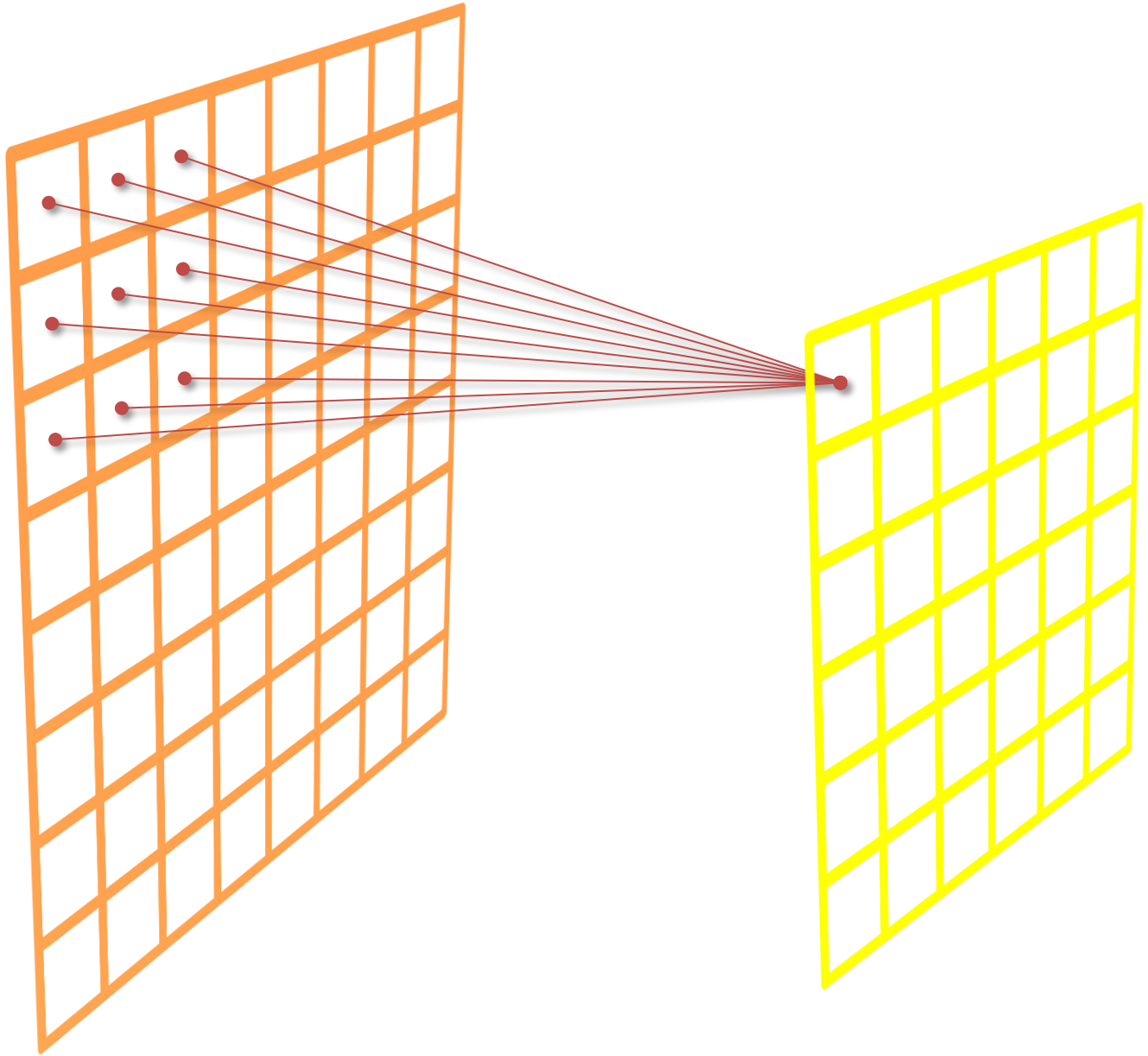
Brandon Rohrer https://brohrer.github.io/how_convolutional_neural_networks_work.html

Text

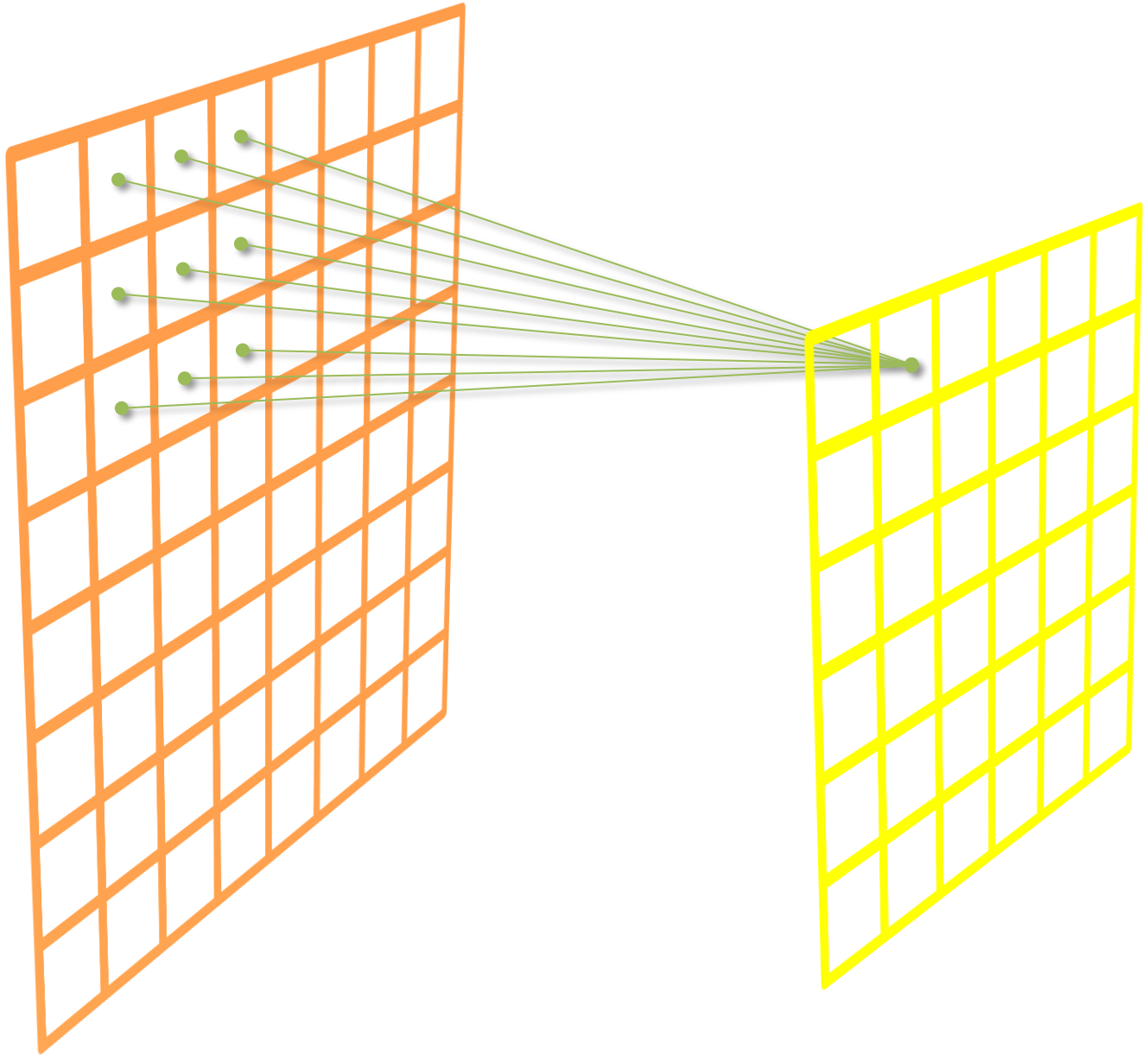


Brandon Rohrer https://brohrer.github.io/how_convolutional_neural_networks_work.html

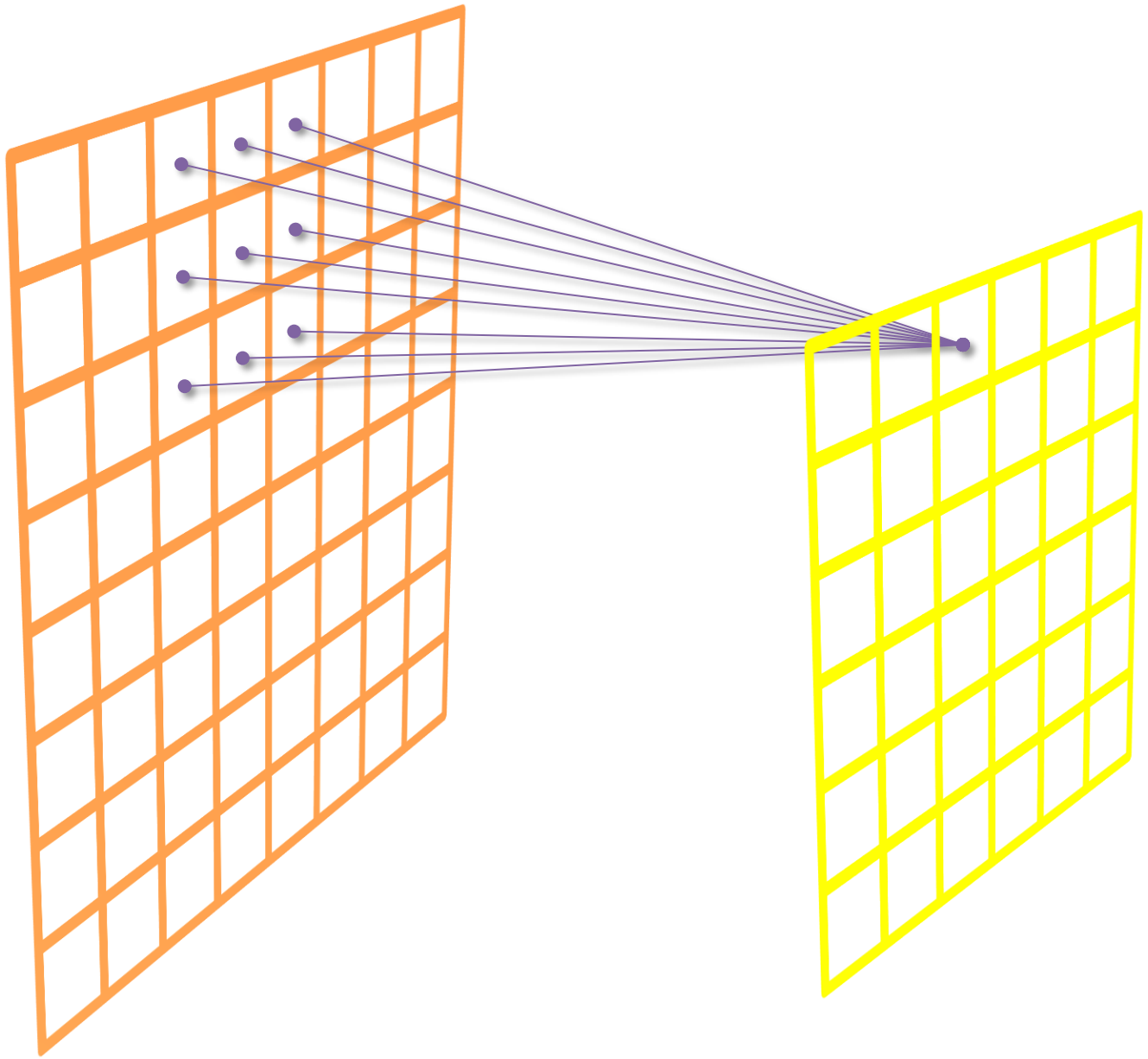
8x8 image, 3x3 filter, Stride 1



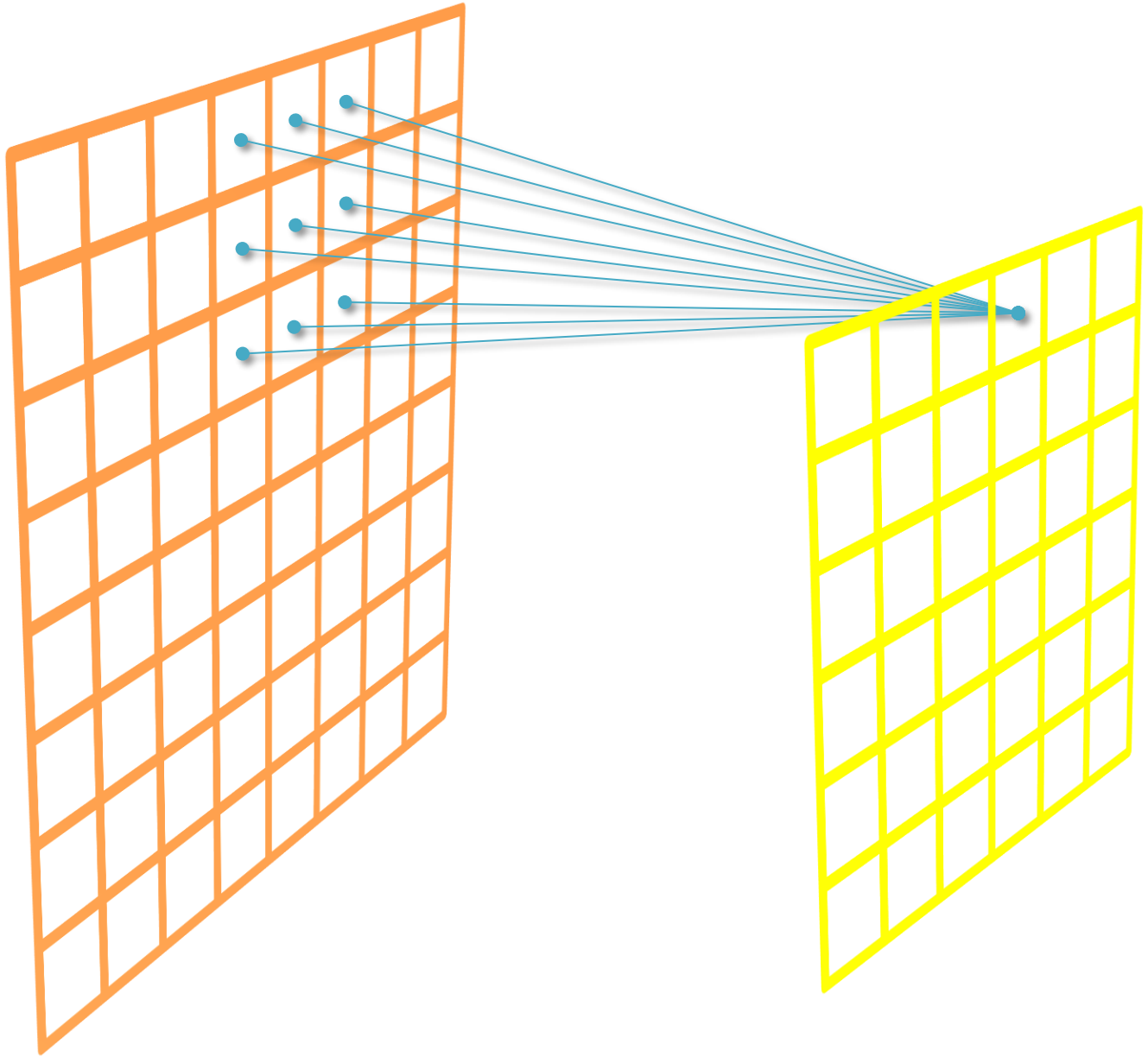
8x8 image, 3x3 filter, Stride 1



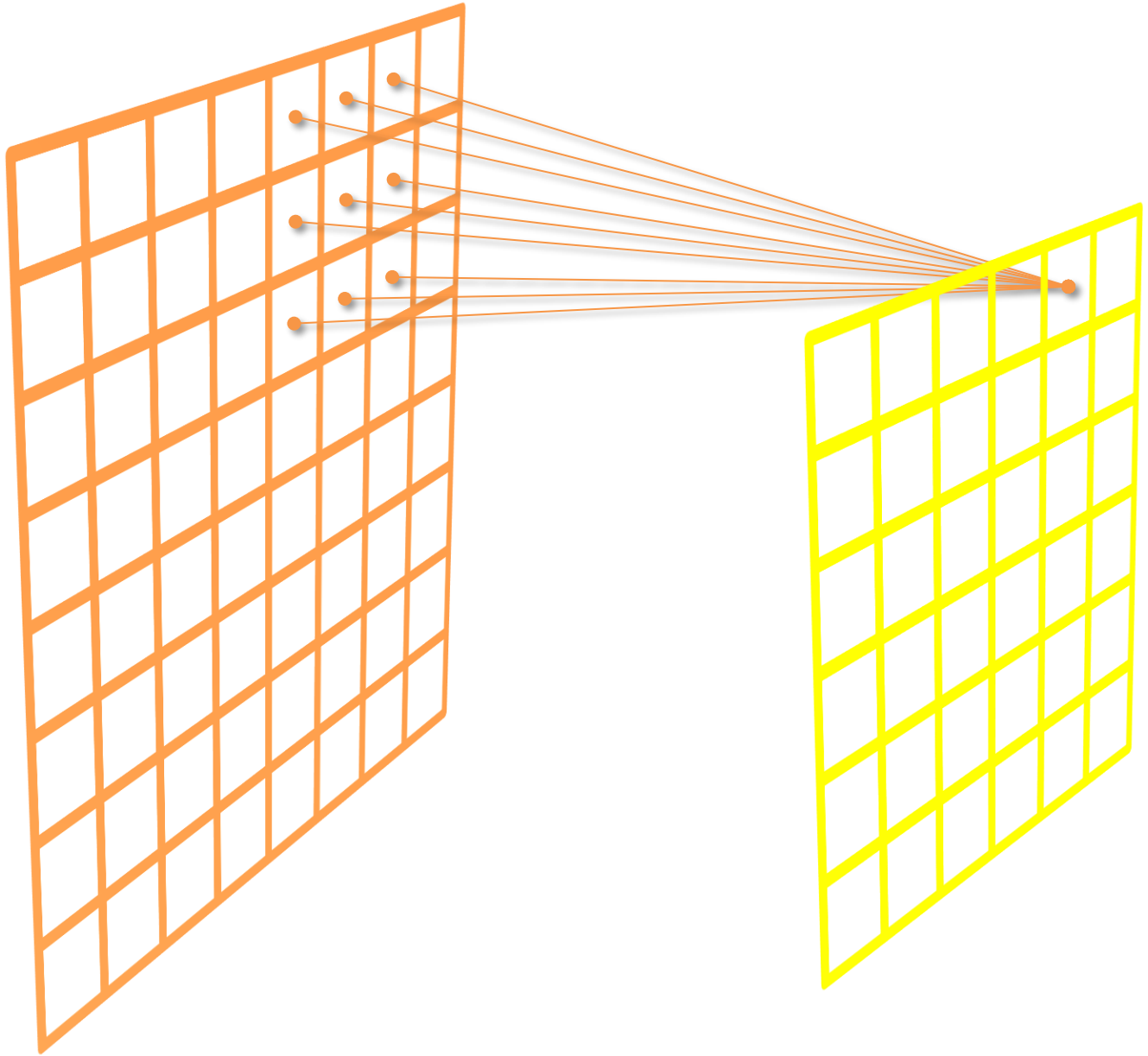
8x8 image, 3x3 filter, Stride 1



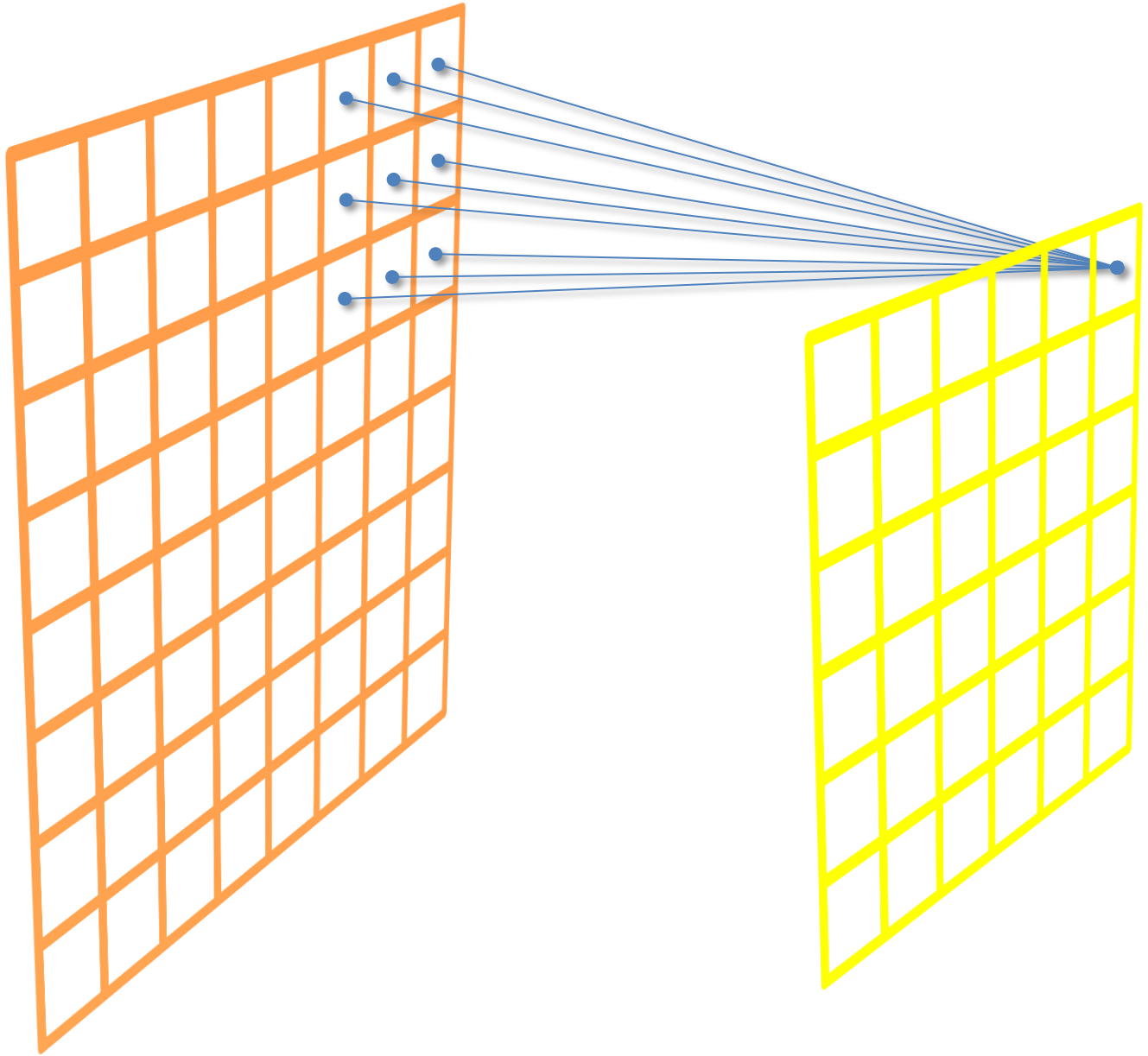
8x8 image, 3x3 filter, Stride 1



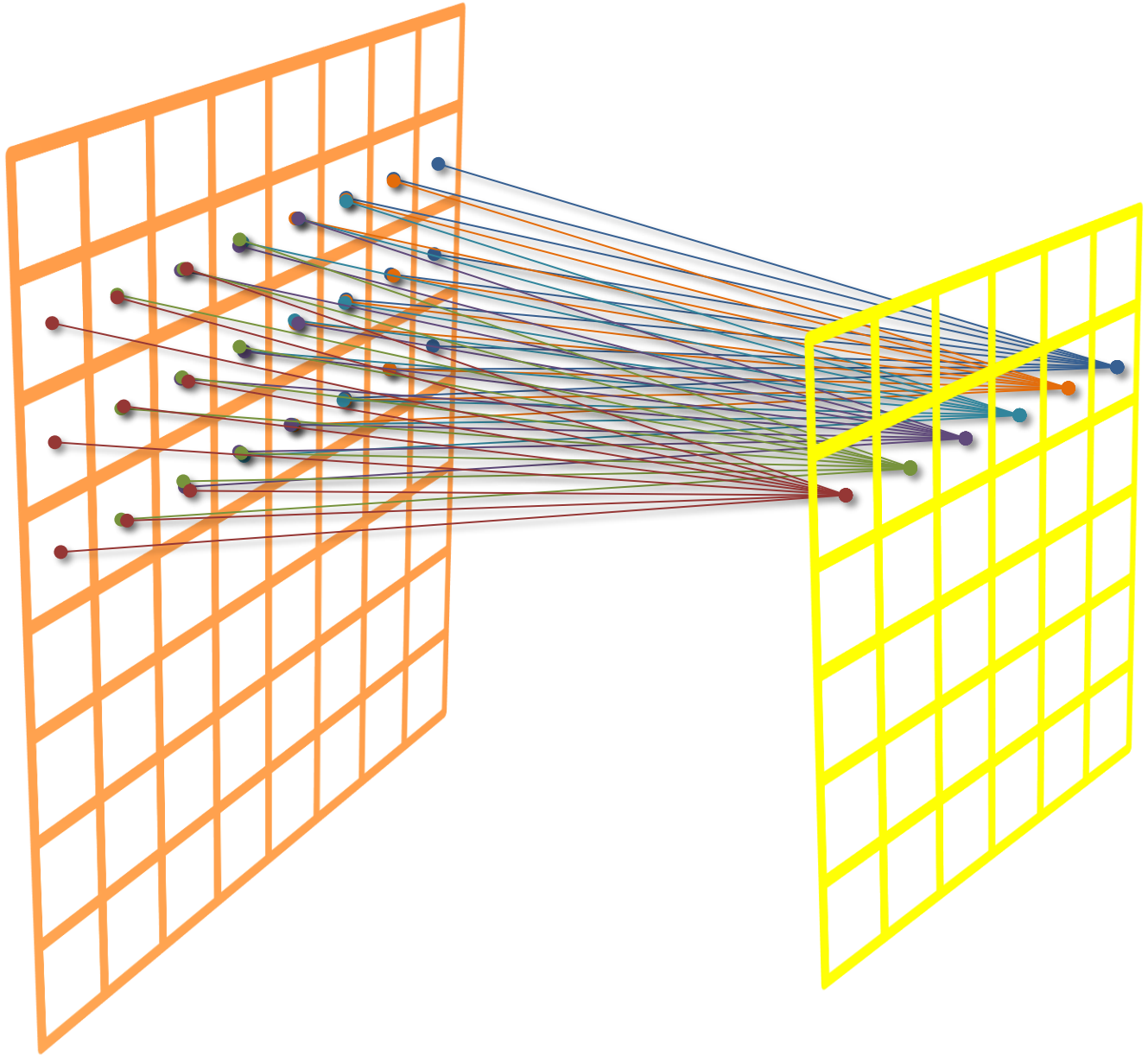
8x8 image, 3x3 filter, Stride 1



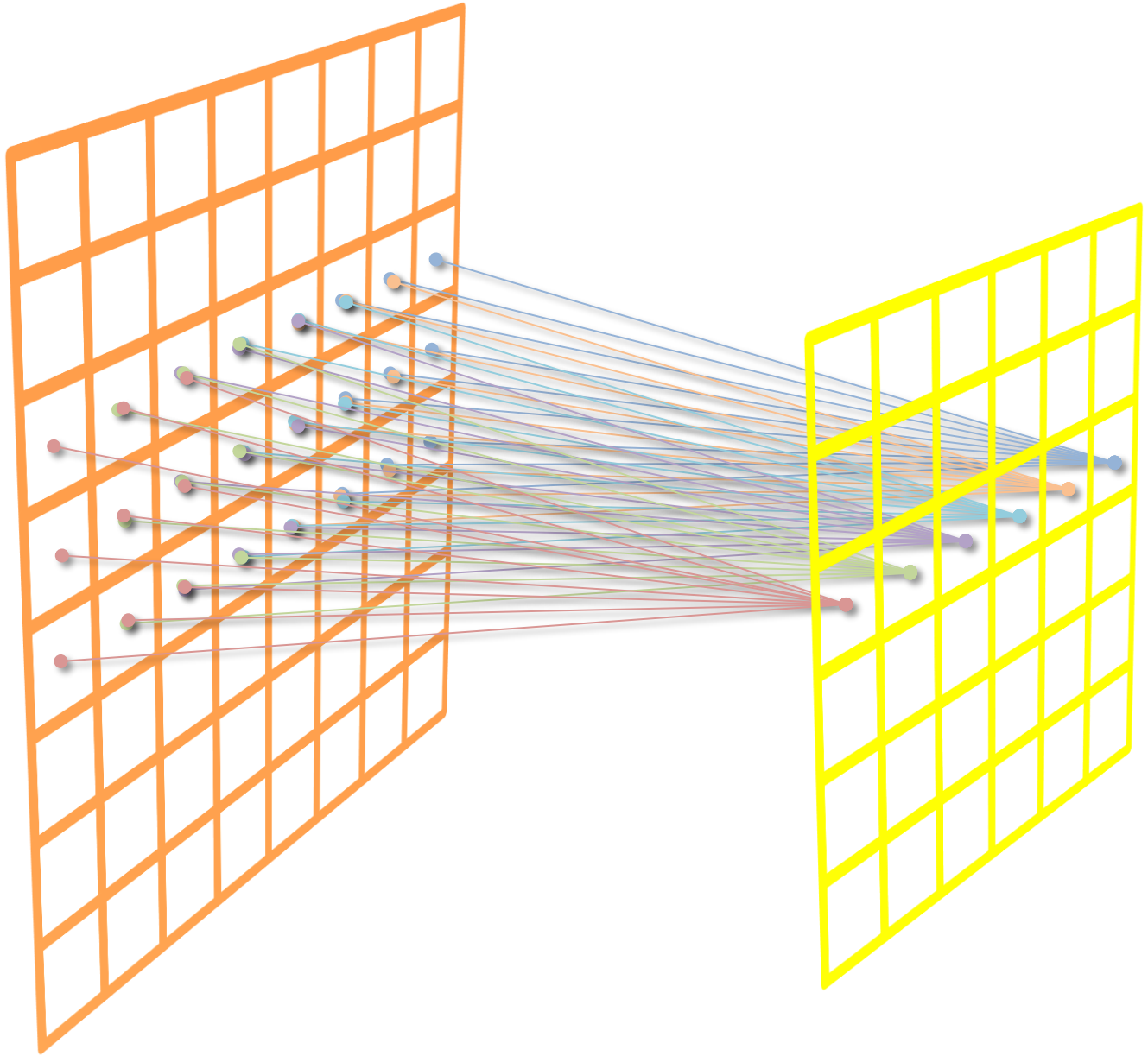
8x8 image, 3x3 filter, Stride 1

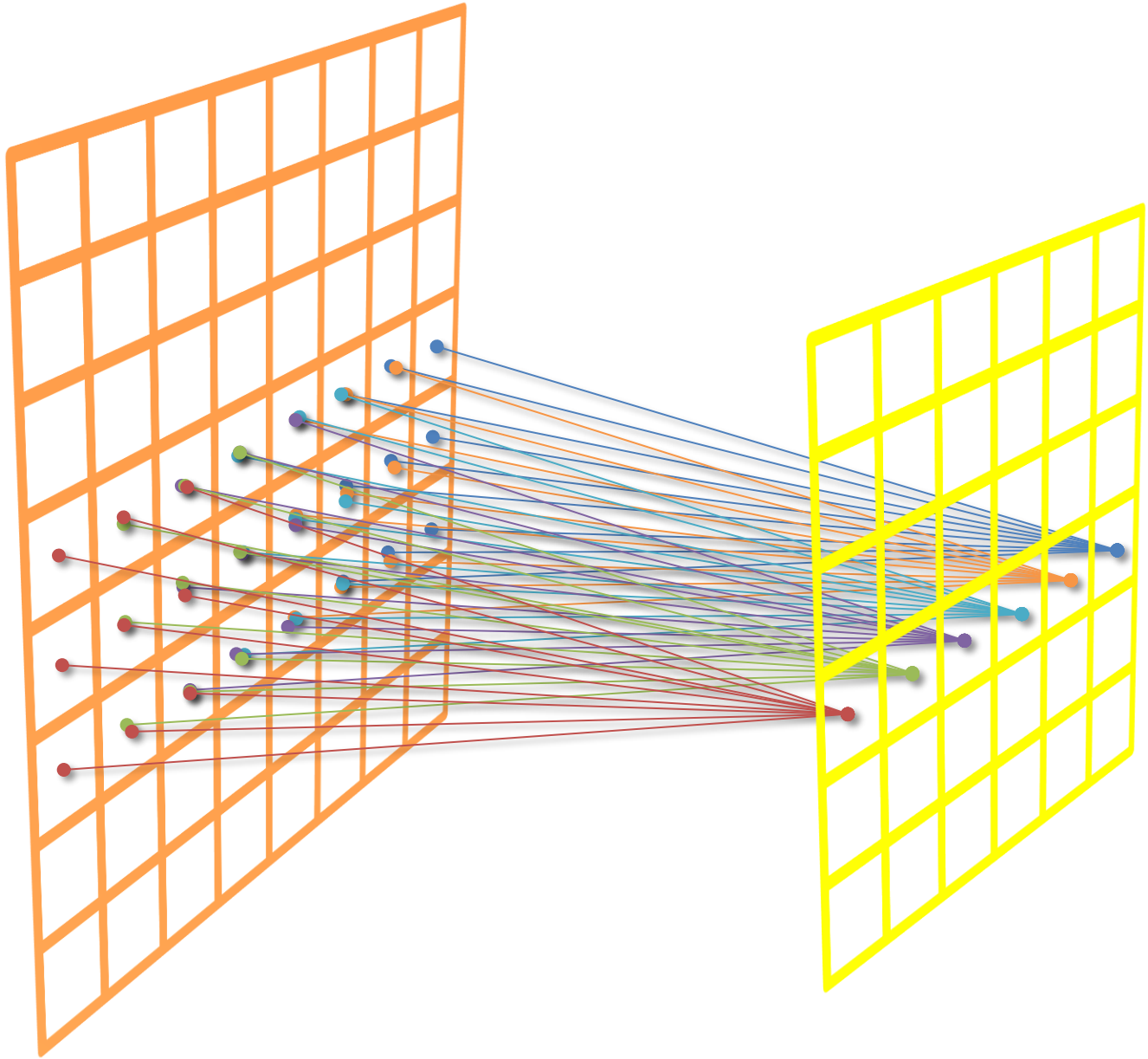


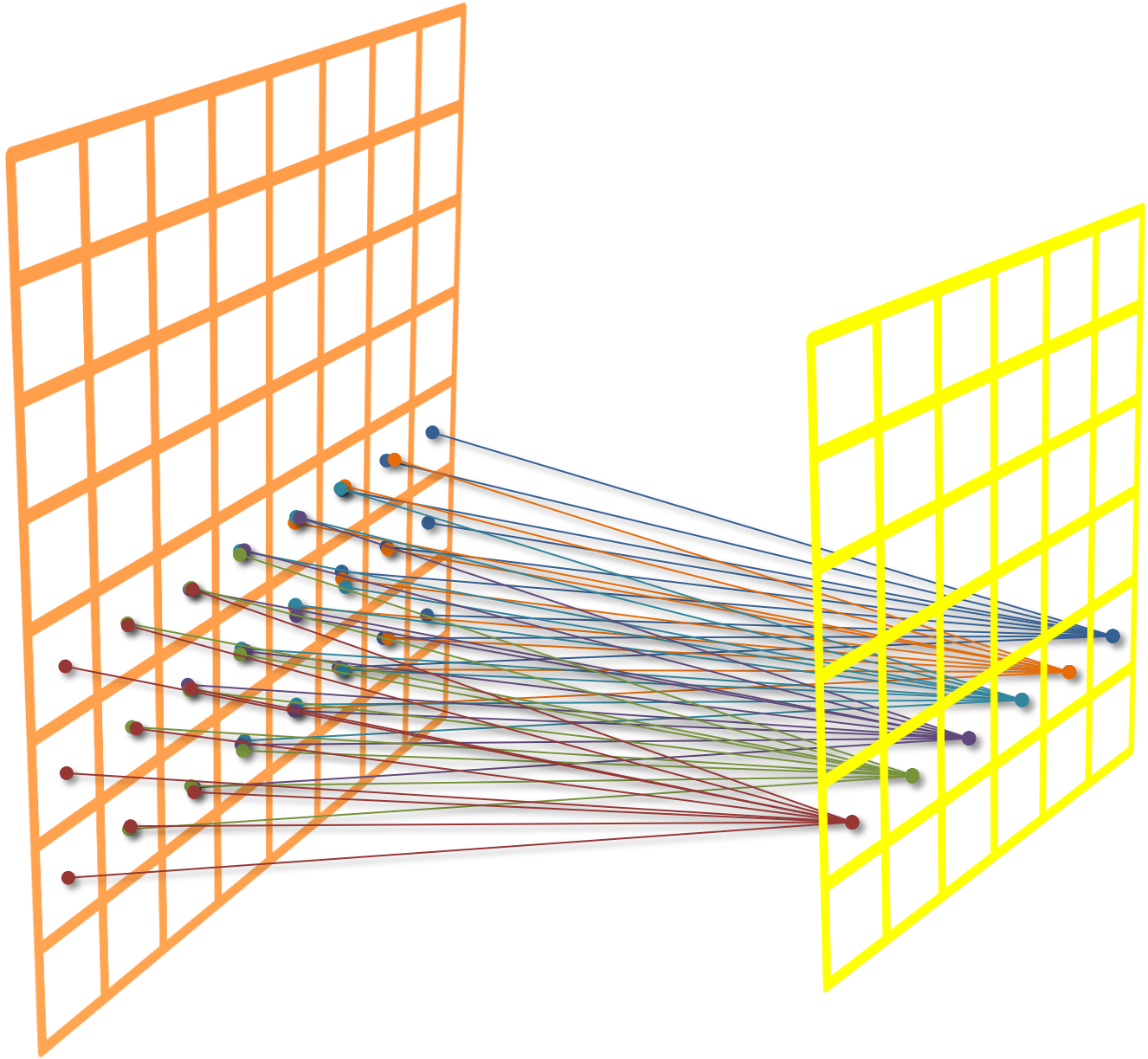
8x8 image, 3x3 filter, Stride 1



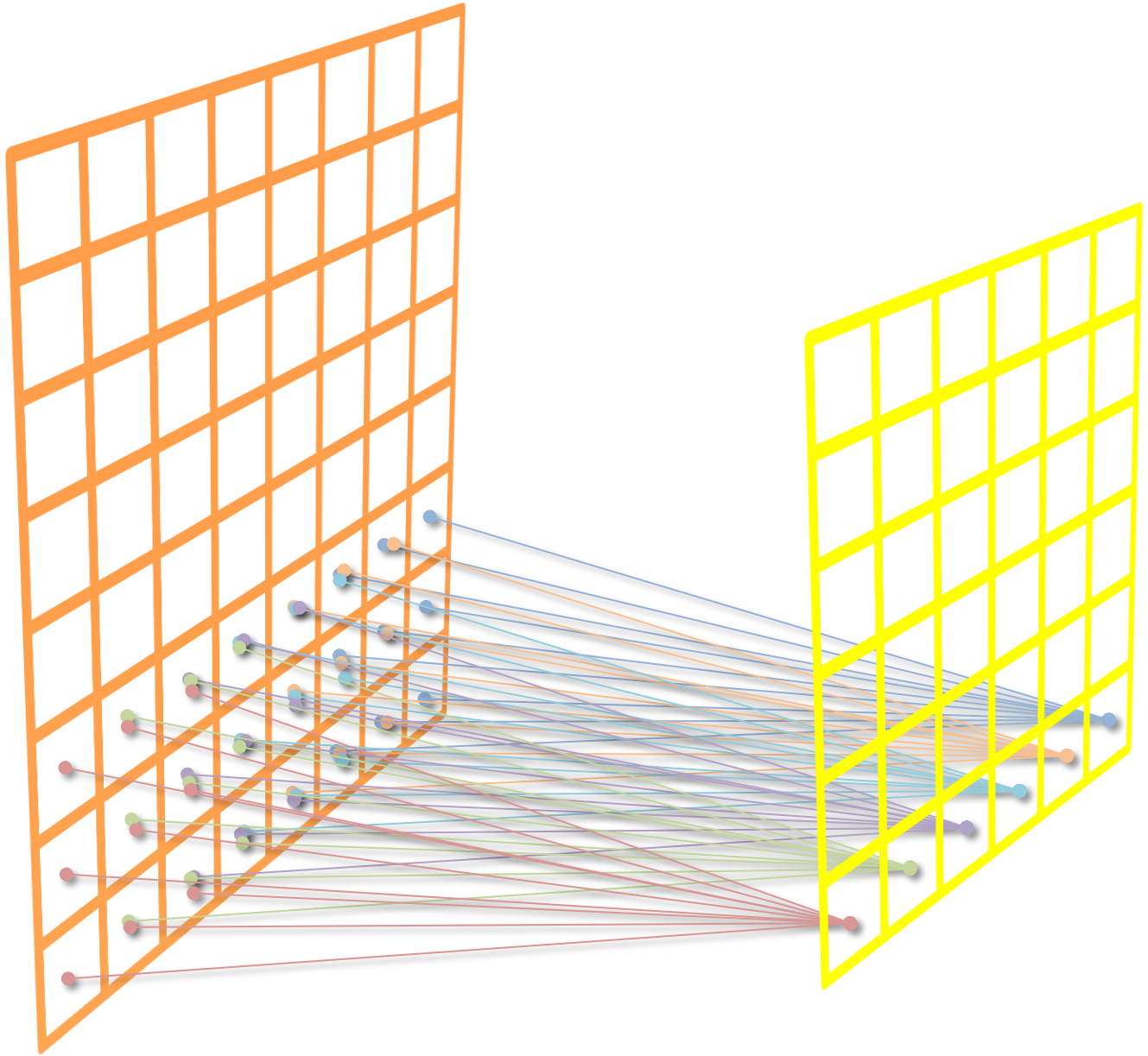
8x8 image, 3x3 filter, Stride 1



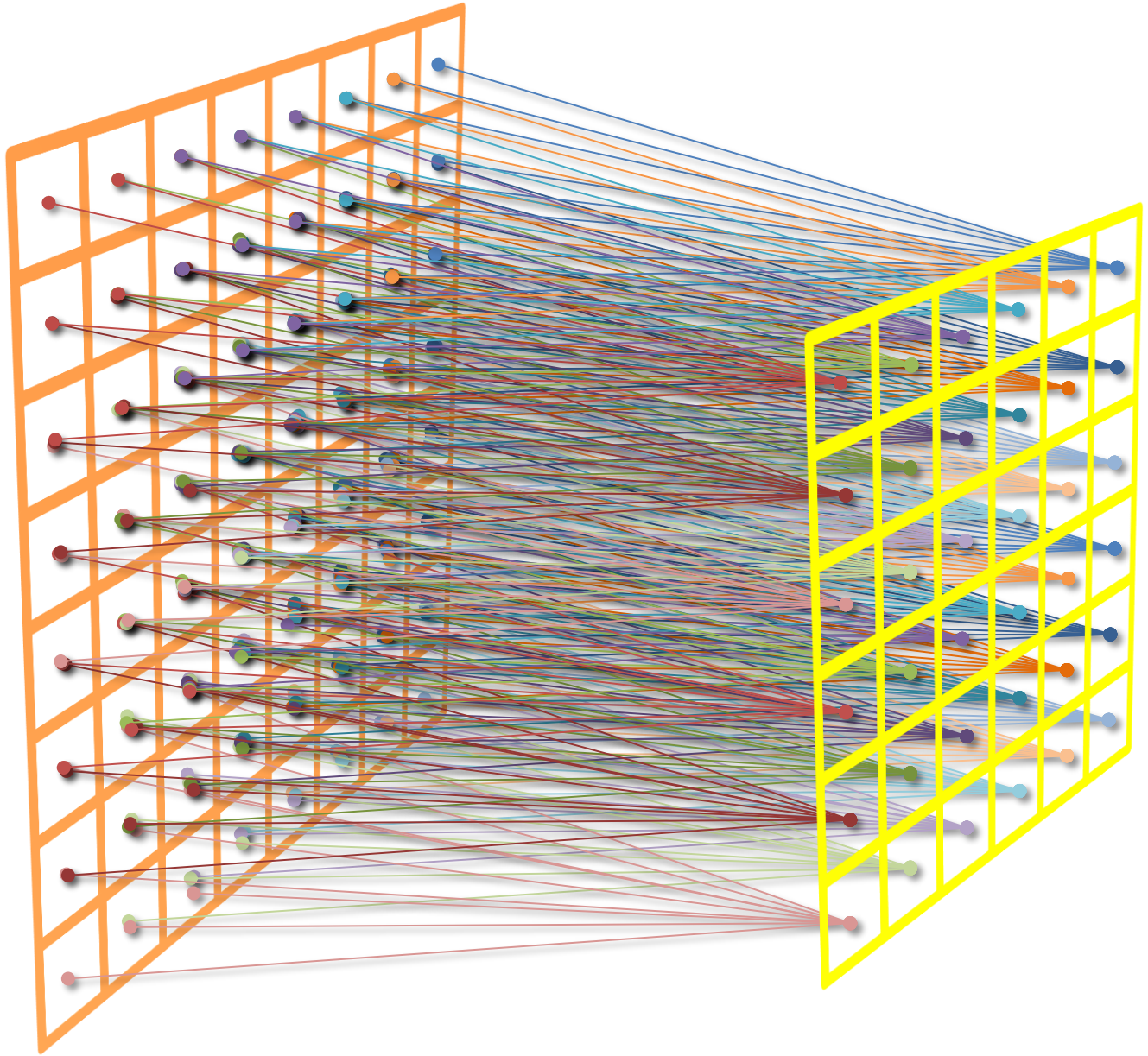




8x8 image, 3x3 filter, Stride 1



8x8 image, 3x3 filter, Stride 1



Total Connections?

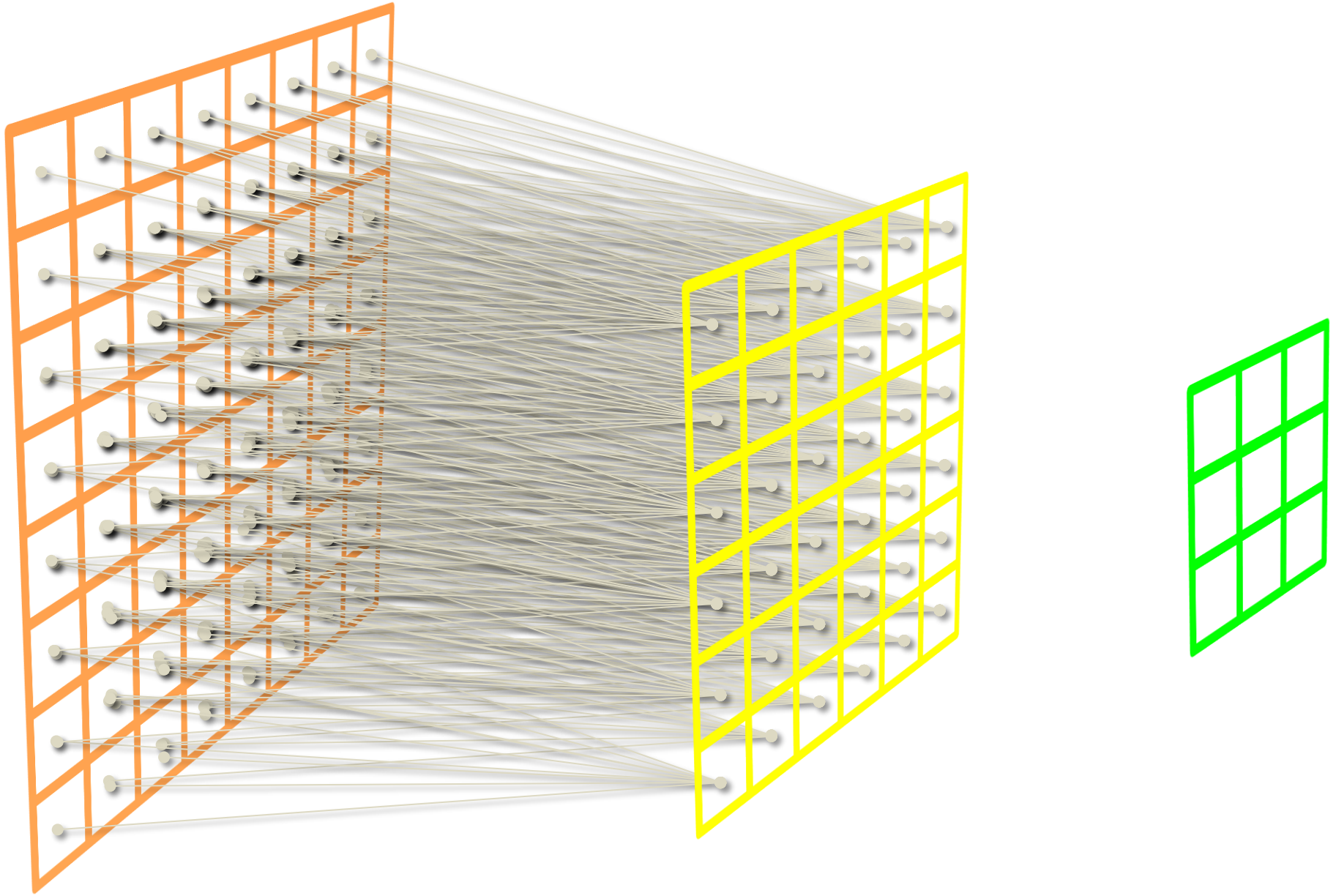
6x6x1 x 3x3x1

Total Unique Parameters?

3x3x1
+ bias

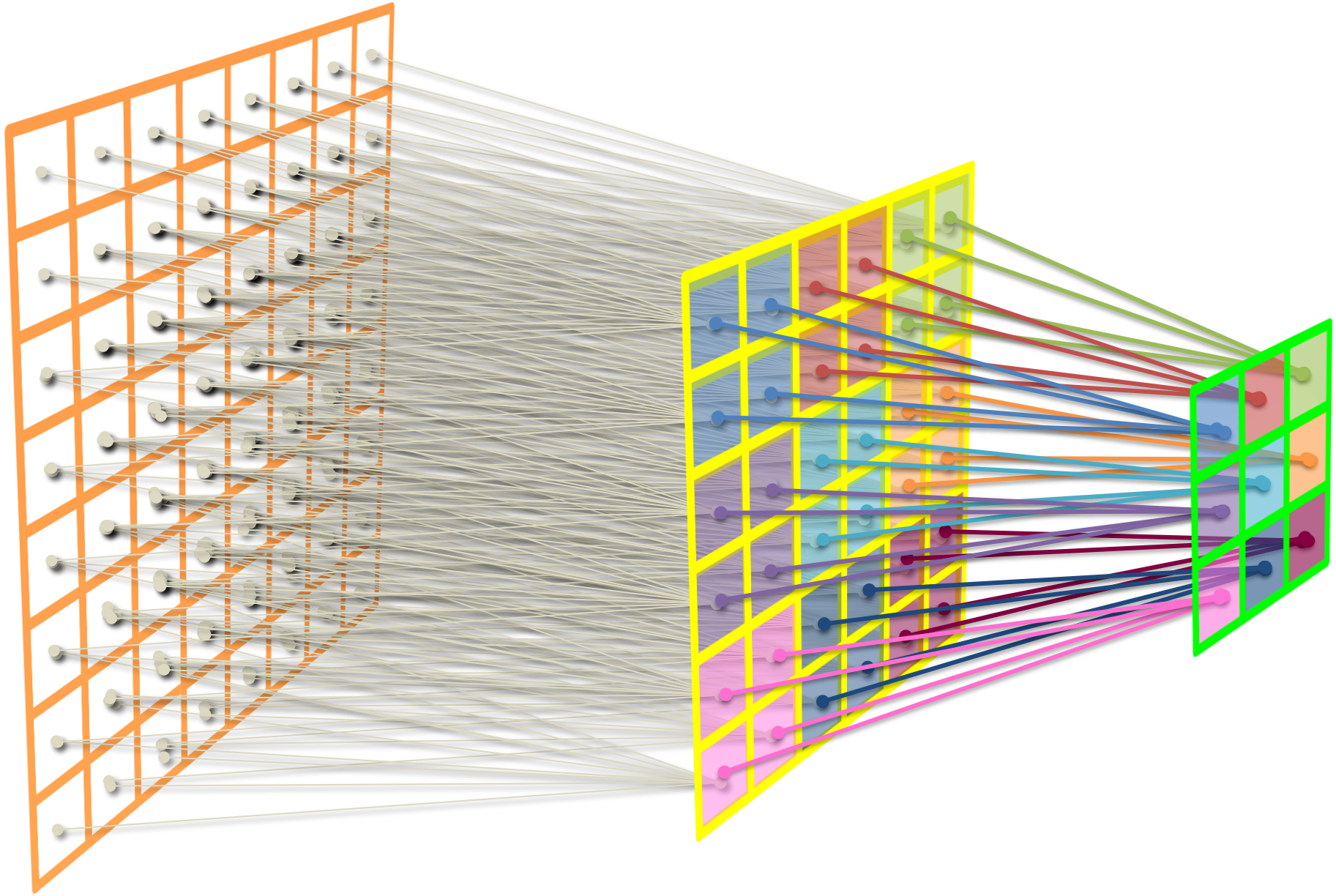
8x8 image, 3x3 filter, Stride 1

2x2 pooling, Stride 2



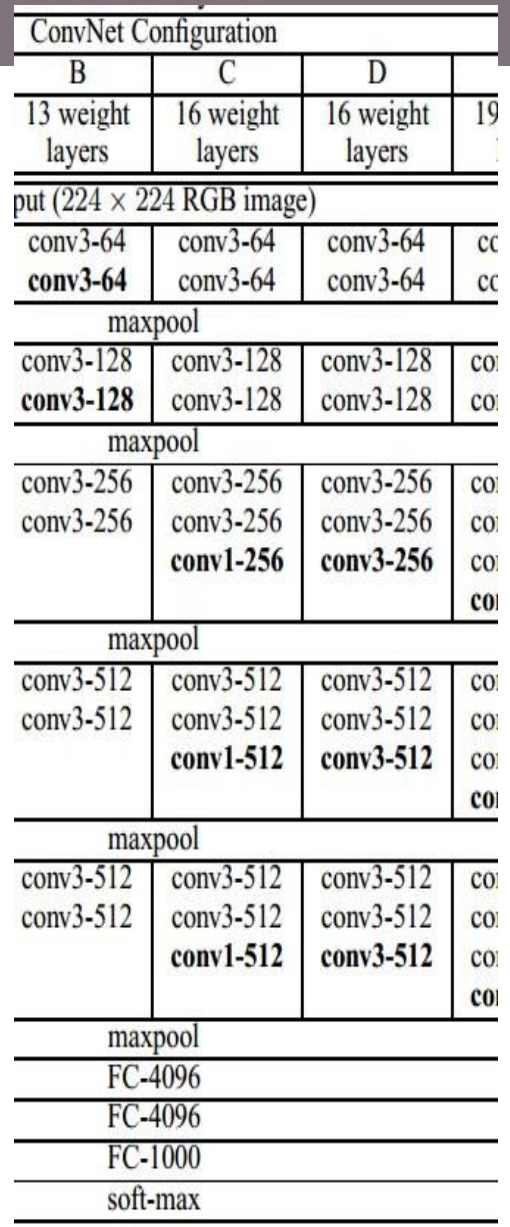
8x8 image, 3x3 filter, Stride 1

2x2 pooling, Stride 2



Model	Memory	Parameter
INPUT: [224x224x3]		
CONV3-64: [224x224x64]		
CONV3-64: [224x224x64]		
POOL2: [112x112x64]		
CONV3-128: [112x112x128]		
CONV3-128: [112x112x128]		
POOL2: [56x56x128]		
CONV3-256: [56x56x256]		
CONV3-256: [56x56x256]		
CONV3-256: [56x56x256]		
POOL2: [28x28x256]		
CONV3-512: [28x28x512]		
CONV3-512: [28x28x512]		
CONV3-512: [28x28x512]		
POOL2: [14x14x512]		
CONV3-512: [14x14x512]		
CONV3-512: [14x14x512]		
CONV3-512: [14x14x512]		
POOL2: [7x7x512]		
FC: [1x1x4096]		
FC: [1x1x4096]		
FC: [1x1x1000]		

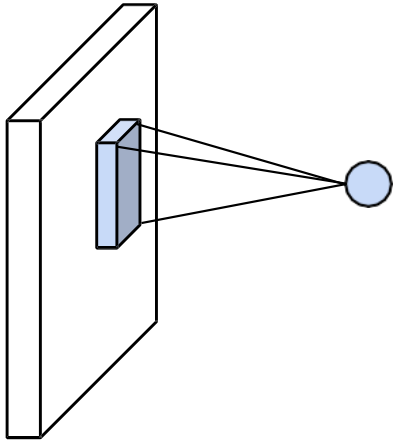
INPUT: [224x224x3] memory: 224*224*3=150K params: 0
CONV3-64: [224x224x64] memory: 224*224*64=3.2M params: (3*3*3)*64 = 1,728
CONV3-64: [224x224x64] memory: 224*224*64=3.2M params: (3*3*64)*64 = 36,864
POOL2: [112x112x64] memory: 112*112*64=800K params: 0
CONV3-128: [112x112x128] memory: 112*112*128=1.6M params: (3*3*64)*128 = 73,728
CONV3-128: [112x112x128] memory: 112*112*128=1.6M params: (3*3*128)*128 = 147,456
POOL2: [56x56x128] memory: 56*56*128=400K params: 0
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*128)*256 = 294,912
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*256)*256 = 589,824
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*256)*256 = 589,824
POOL2: [28x28x256] memory: 28*28*256=200K params: 0
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*256)*512 = 1,179,648
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*512)*512 = 2,359,296
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*512)*512 = 2,359,296
POOL2: [14x14x512] memory: 14*14*512=100K params: 0
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
POOL2: [7x7x512] memory: 7*7*512=25K params: 0
FC: [1x1x4096] memory: 4096 params: 7*7*512*4096 = 102,760,448
FC: [1x1x4096] memory: 4096 params: 4096*4096 = 16,777,216
FC: [1x1x1000] memory: 1000 params: 4096*1000 = 4,096,000



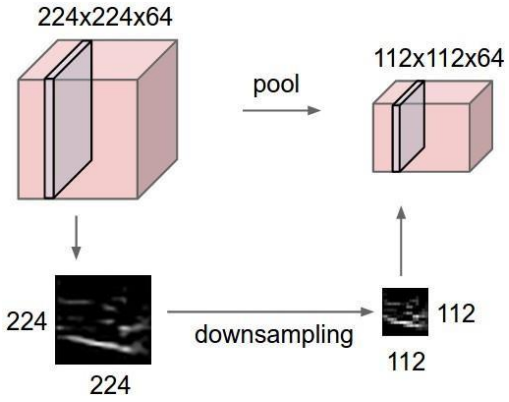
ConvNet Configuration			
B	C	D	
13 weight layers	16 weight layers	16 weight layers	19
put (224 × 224 RGB image)			
conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	cc cc
maxpool			
conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	co co
maxpool			
conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	co co co co
maxpool			
conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	co co co co
maxpool			
conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	co co co co
maxpool			
FC-4096			
FC-4096			
FC-1000			
soft-max			

Components of CNNs

Convolution Layers

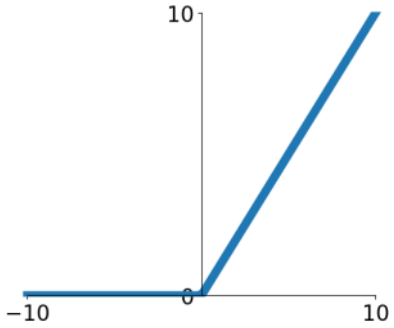


Pooling Layers



Fully-Connected Layers

Activation Function



CNN Architectures

Case Studies

- AlexNet
- VGG
- ResNet

Also....

- GoogLeNet
- ZFNet
- SENet
- Wide ResNet
- ResNeXT
- DenseNet
- MobileNets
- NASNet
- EfficientNet

Next time: CNN Architectures