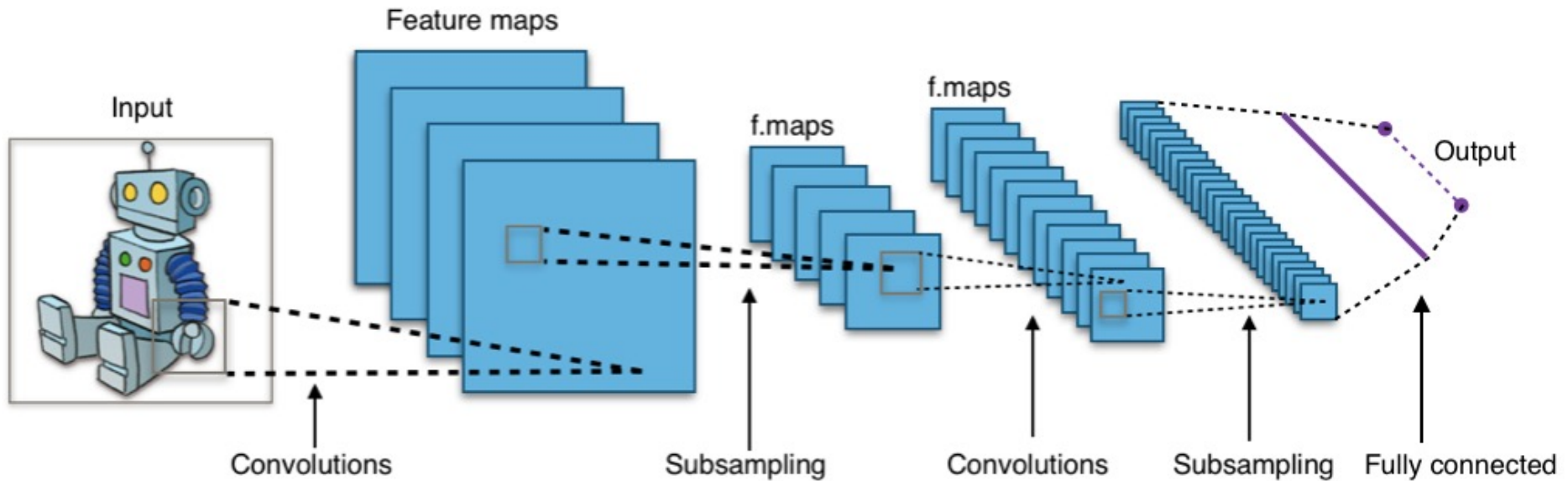


Convolutional neural networks



Overview of today's lecture

- Convolutional neural networks.
- Training ConvNets.

Slide credits

Most of these slides were adapted from:

- Noah Snavely (Cornell University).
- Fei-Fei Li (Stanford University).
- Andrej Karpathy (Stanford University).

Convolutional Neural Networks

Aside: “CNN” vs “ConvNet”

Note:

- There are many papers that use either phrase, but
- “ConvNet” is the preferred term, since “CNN” clashes with other things called CNN



Yann LeCun

Motivation



HOME ▾

MENU ▾

CONNECT

THE LATEST

POPULAR

MOST SHARED



MIT
Technology
Review

10 BREAKTHROUGH TECHNOLOGIES 2013

[Introduction](#)

[The 10 Technologies](#)

[Past Years](#)

Deep Learning

With massive amounts of computational power, machines can now recognize objects and translate speech in real time. Artificial intelligence is finally getting smart. →

Temporary Social Media

Messages that quickly self-destruct could enhance the privacy of online communications and make people freer to be spontaneous. →

Prenatal DNA Sequencing

Reading the DNA of fetuses will be the next frontier of the genomic revolution. But do you really want to know about the genetic problems or musical aptitude of your unborn child? →

Additive Manufacturing

Skeptical about 3-D printing? GE, the world's largest manufacturer, is on the verge of using the technology to make jet parts. →

Baxter: The Blue-Collar Robot

Rodney Brooks's newest creation is easy to interact with, but the complex innovations behind the robot show just how hard it is to get along with people. →

Memory Implants

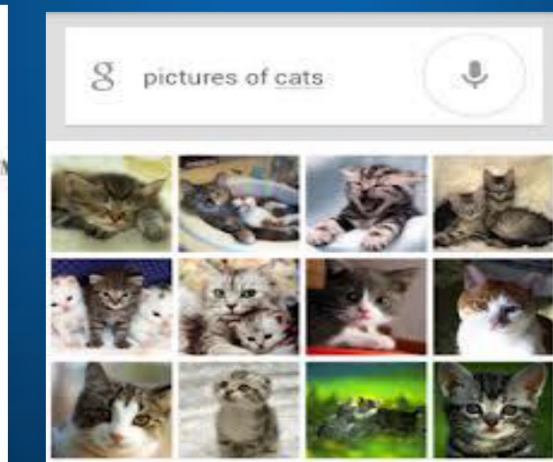
Smart Watches

Ultra-Efficient Solar

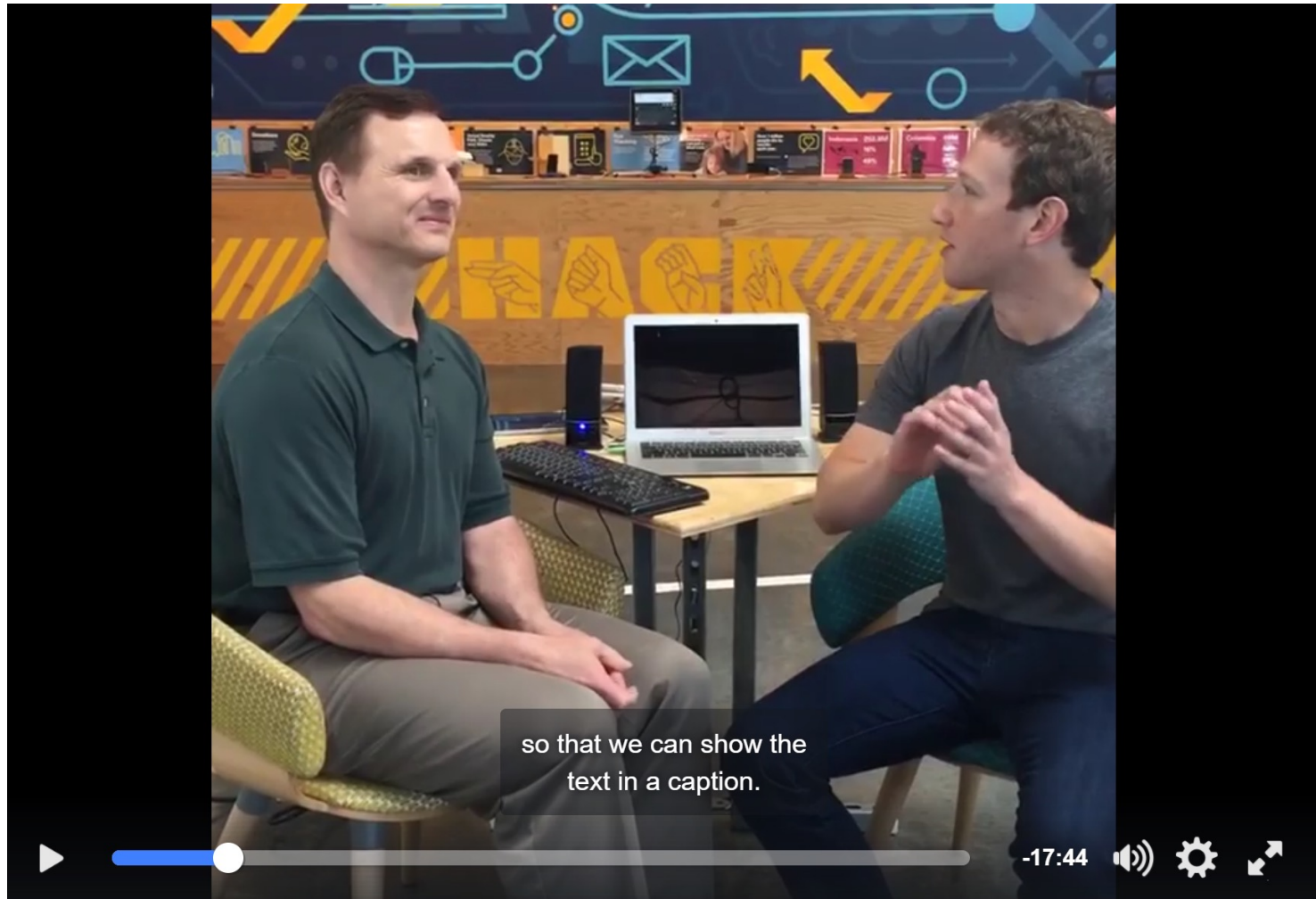
Big Data from

Supergrids

Products



Helping the Blind



<https://www.facebook.com/zuck/videos/10102801434799001/>

(Unrelated) Dog vs Food



(Unrelated) Dog vs Food



CNNs in 2012: “SuperVision” (aka “AlexNet”)

“AlexNet” — Won the ILSVRC2012 Challenge

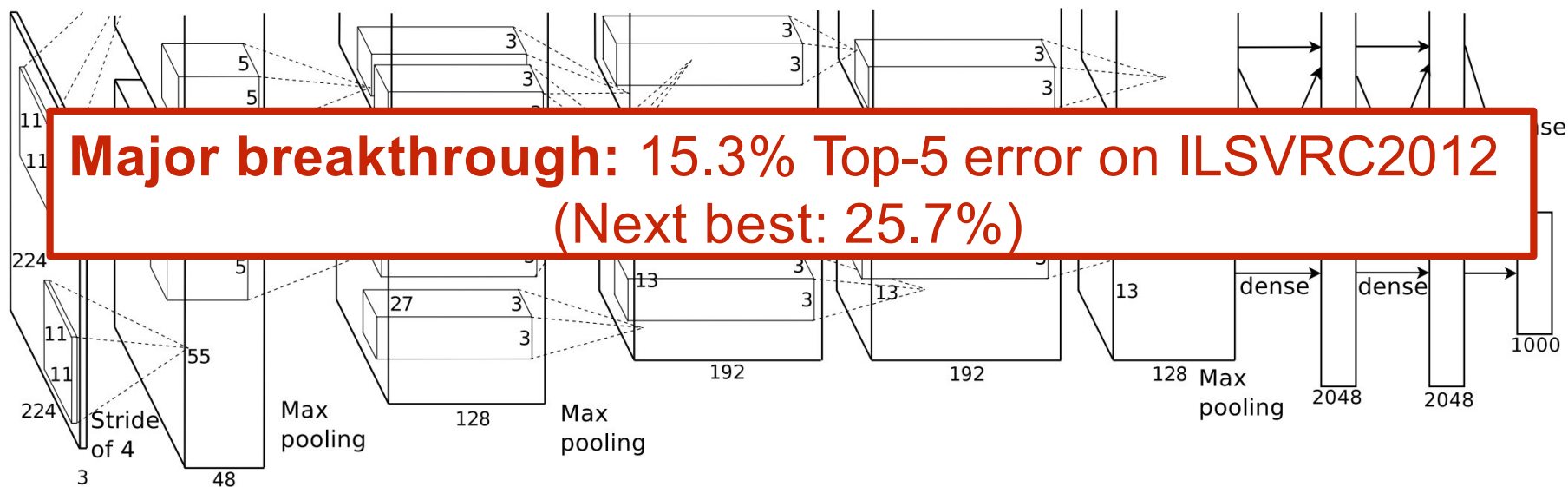
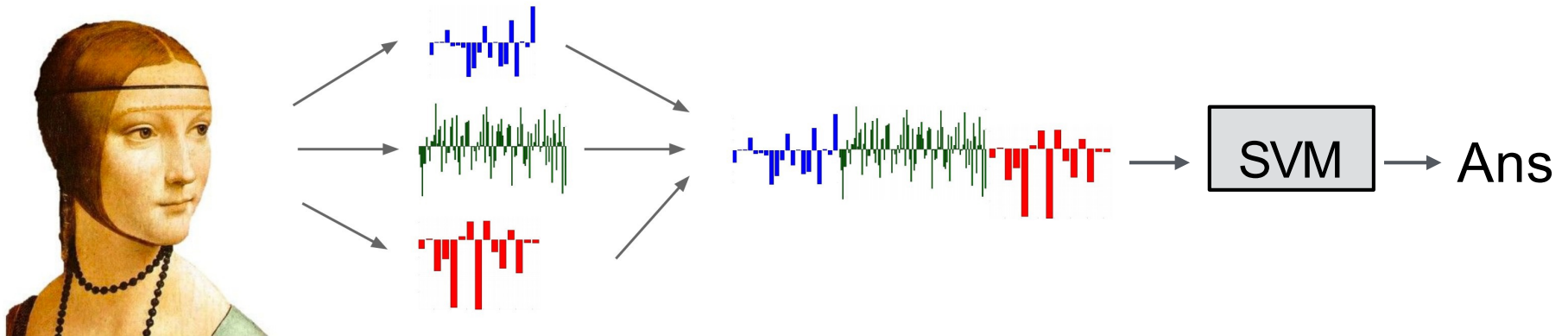


Figure 2: An illustration of the architecture of our CNN, explicitly showing the delineation of responsibilities between the two GPUs. One GPU runs the layer-parts at the top of the figure while the other runs the layer-parts at the bottom. The GPUs communicate only at certain layers. The network’s input is 150,528-dimensional, and the number of neurons in the network’s remaining layers is given by 253,440–186,624–64,896–64,896–43,264–4096–4096–1000.

[Krizhevsky, Sutskever, Hinton. NIPS 2012]

Recap: Before Deep Learning



*Input
Pixels*

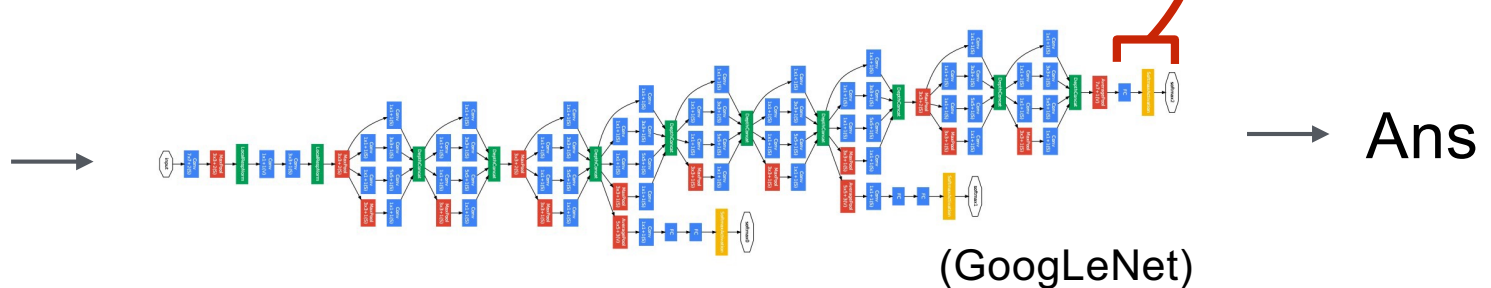
*Extract
Features*

*Concatenate into
a vector x*

*Linear
Classifier*

The last layer of (most) CNNs are linear classifiers

This piece is just a linear classifier



*Input
Pixels*

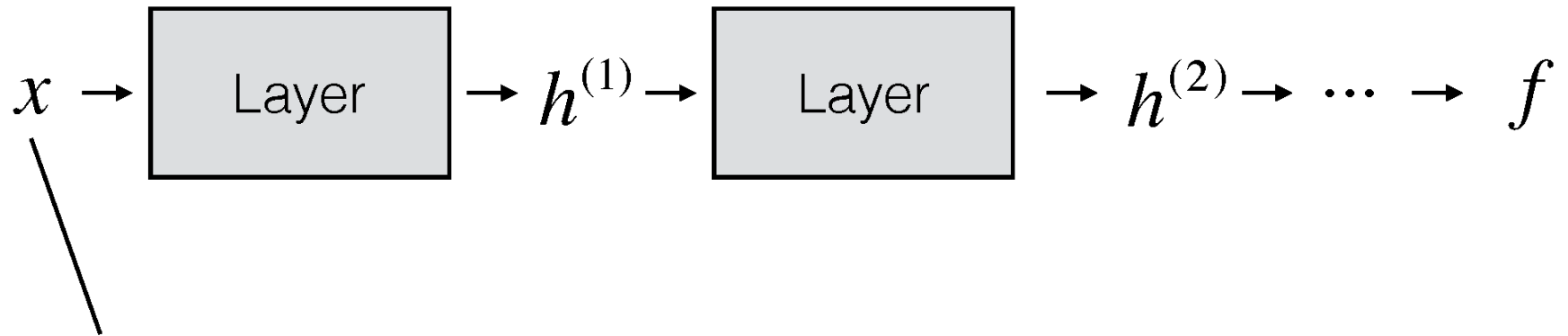
*Perform everything with a big neural
network, trained end-to-end*

Key: perform enough processing so that by the time you get to the end of the network, the classes are linearly separable

ConvNets

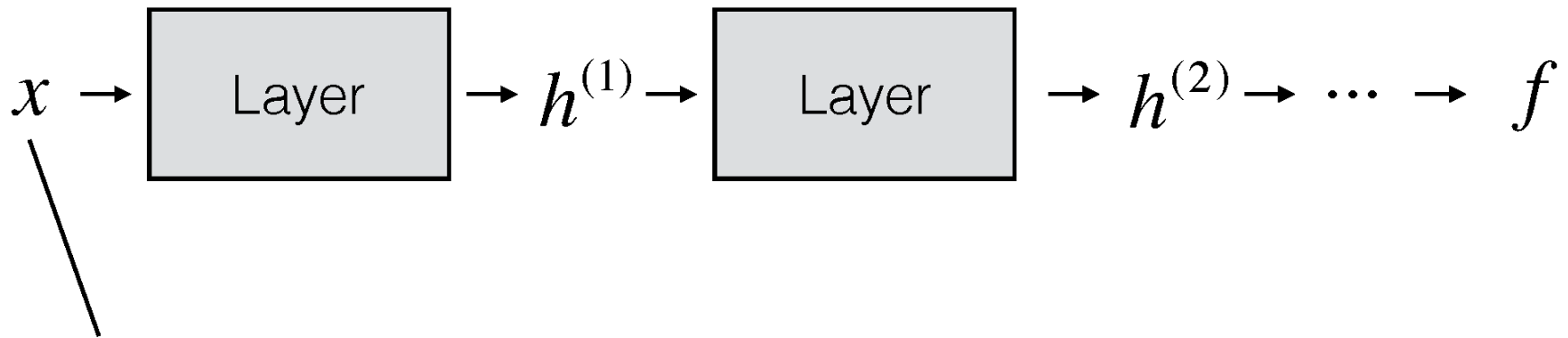
They're just neural networks with
3D activations and weight sharing

What shape should the activations have?



- The input is an image, which is 3D (RGB channel, height, width)

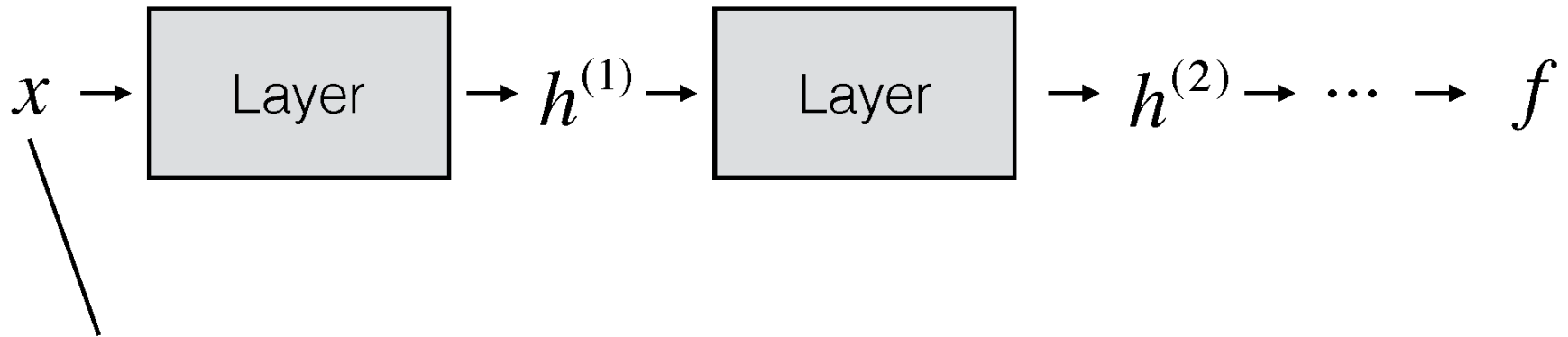
What shape should the activations have?



- The input is an image, which is 3D (RGB channel, height, width)

- We could flatten it to a 1D vector, but then we lose structure

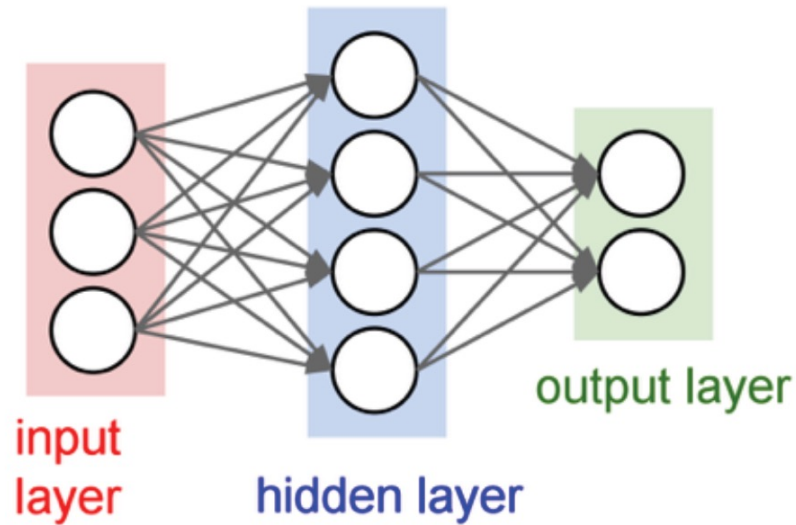
What shape should the activations have?



- The input is an image, which is 3D (RGB channel, height, width)
- We could flatten it to a 1D vector, but then we lose structure
- What about keeping everything in 3D?

3D Activations

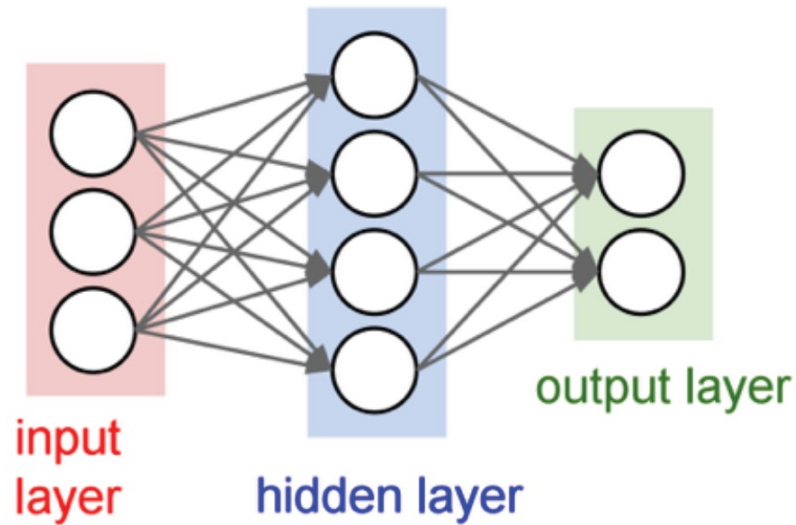
before:



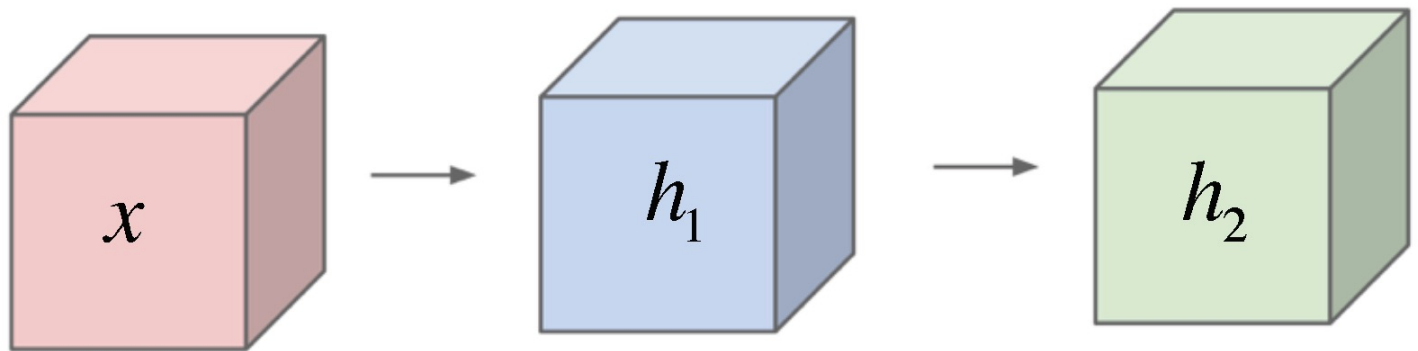
(1D vectors)

3D Activations

before:



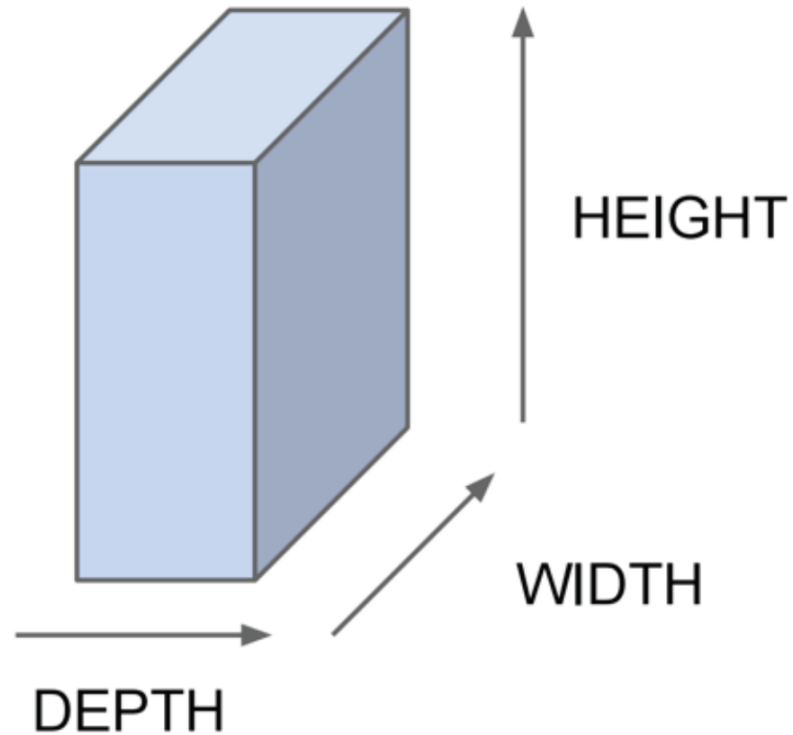
now:



(3D arrays)

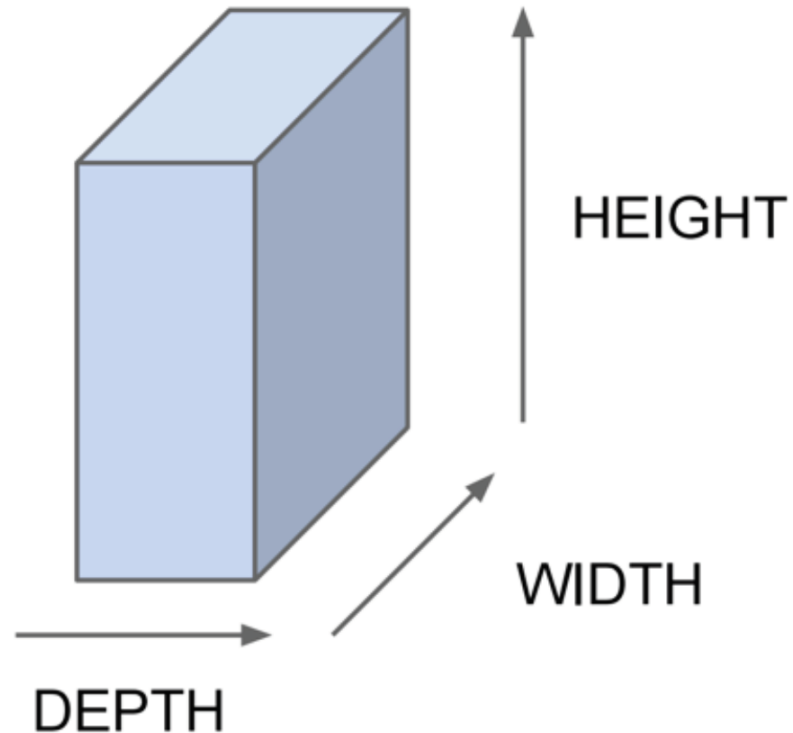
3D Activations

All Neural Net activations arranged in **3 dimensions**:



3D Activations

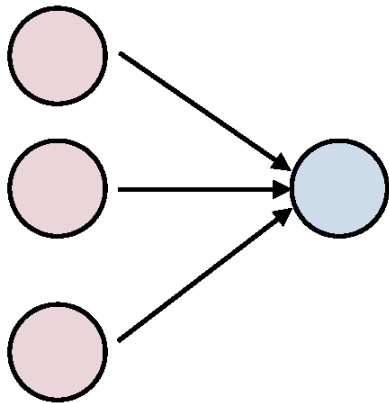
All Neural Net activations arranged in **3 dimensions**:



For example, a CIFAR-10 image is a $3 \times 32 \times 32$ volume (3 depth — RGB channels, 32 height, 32 width)

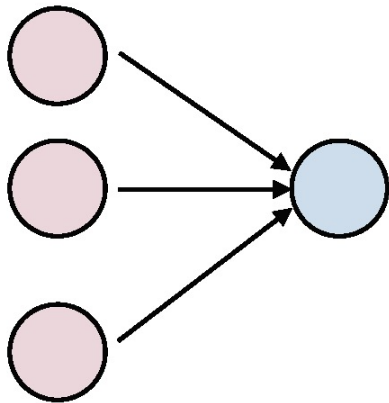
3D Activations

1D Activations:

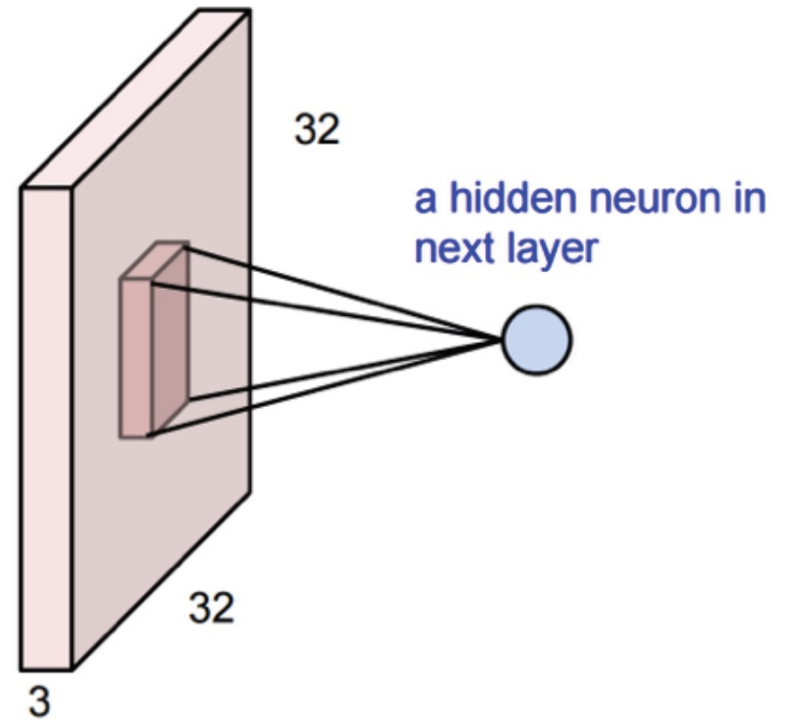


3D Activations

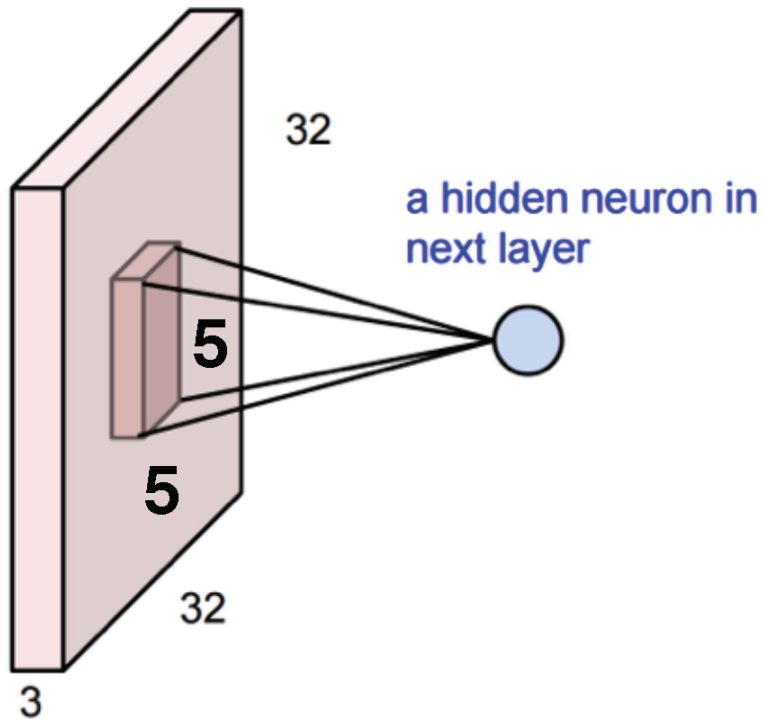
1D Activations:



3D Activations:

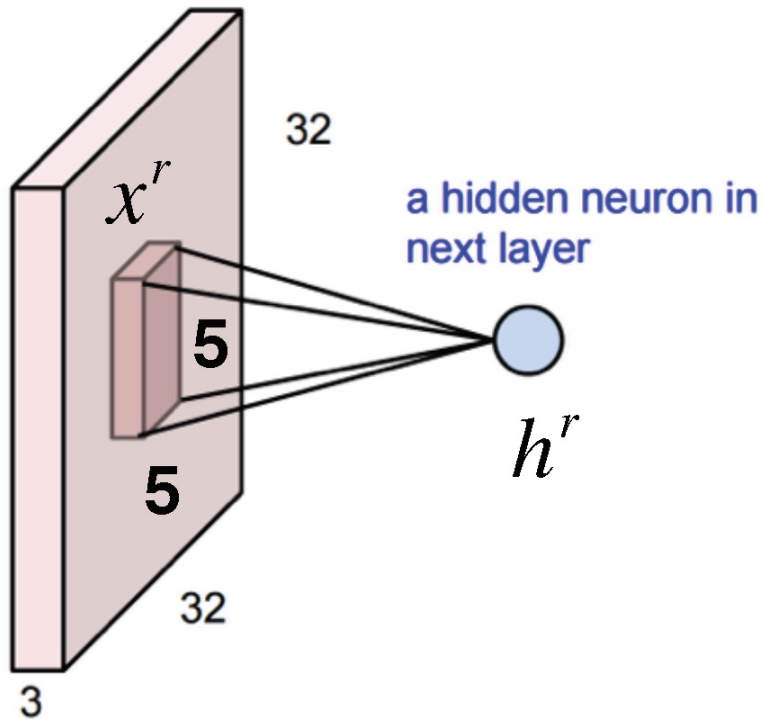


3D Activations



- The input is $3 \times 32 \times 32$
- This neuron depends on a $3 \times 5 \times 5$ chunk of the input
- The neuron also has a $3 \times 5 \times 5$ set of weights and a bias (scalar)

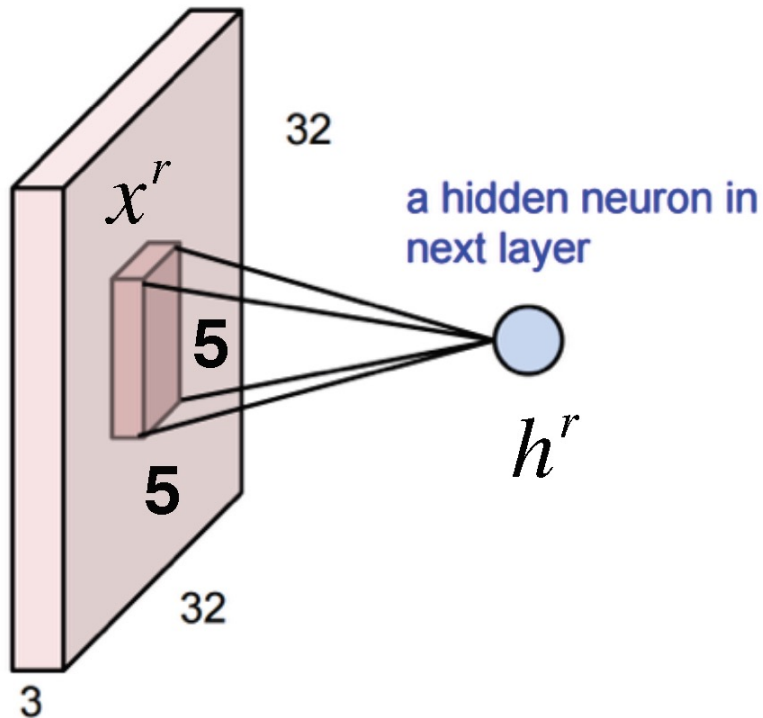
3D Activations



Example: consider the region of the input " x^r "

With output neuron h^r

3D Activations



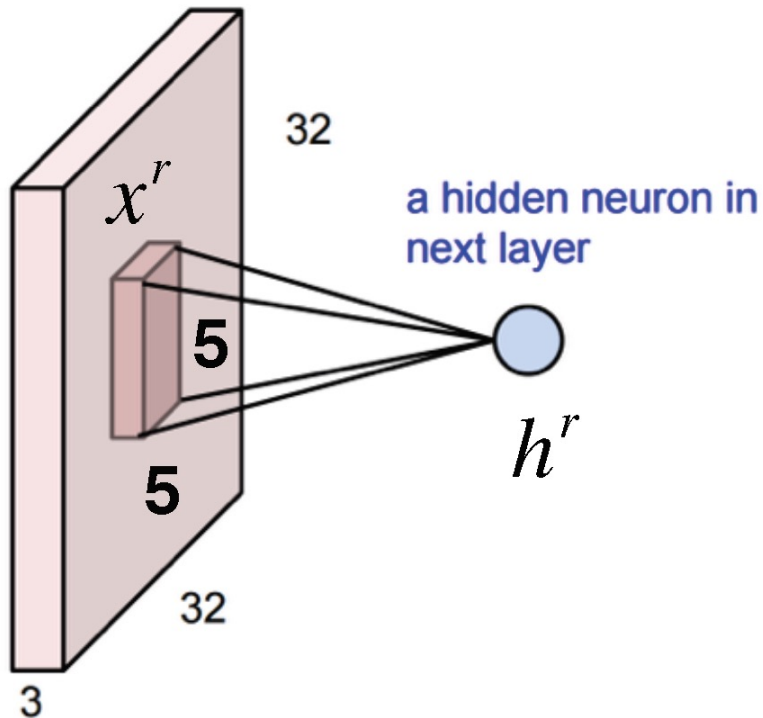
Example: consider the region of the input “ x^r ”

With output neuron h^r

Then the output is:

$$h^r = \sum_{ijk} x^r_{ijk} W_{ijk} + b$$

3D Activations



Example: consider the region of the input " x^r "

With output neuron h^r

Then the output is:

$$h^r = \sum_{ijk} x^r_{ijk} W_{ijk} + b$$



Sum over 3 axes

3D Activations

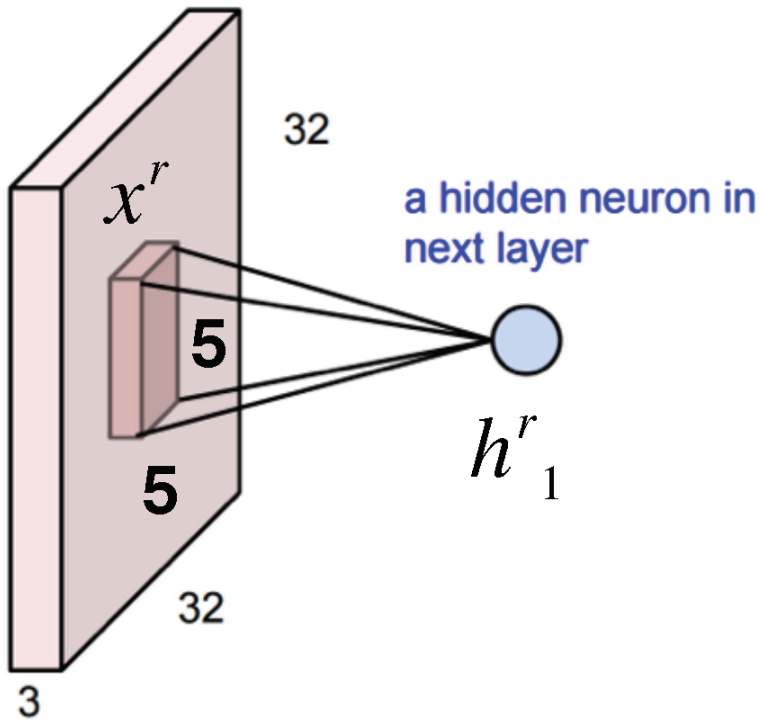


Figure: Andrej Karpathy

3D Activations

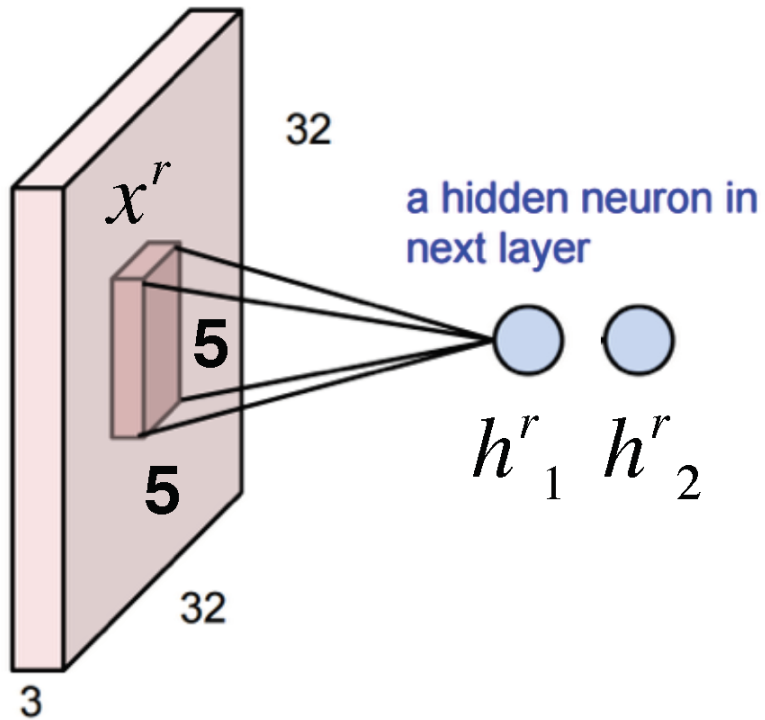
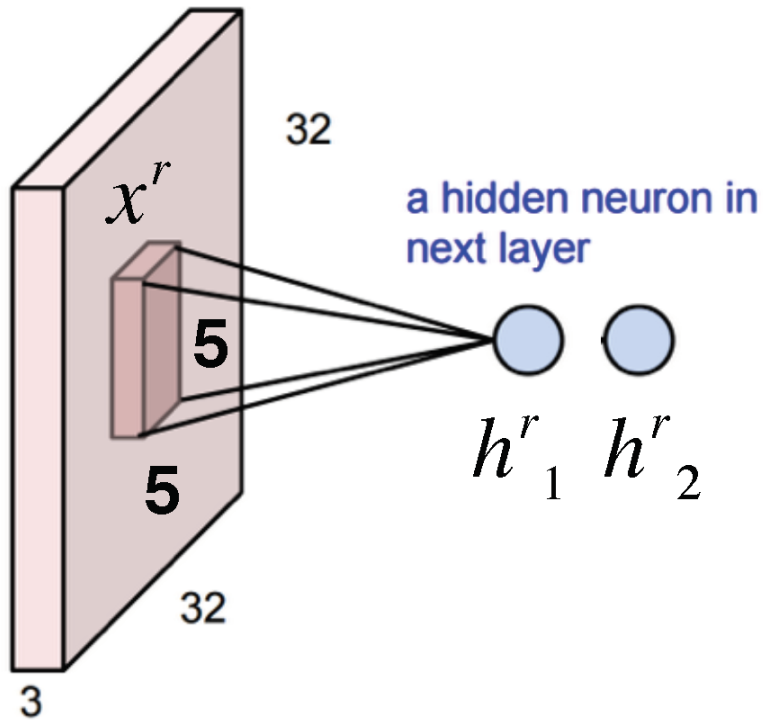


Figure: Andrej Karpathy

3D Activations

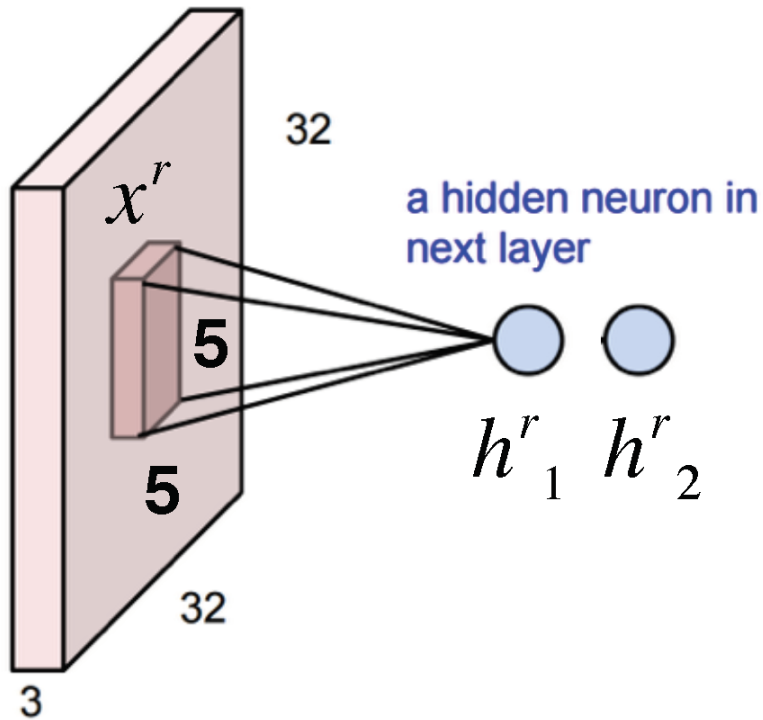


With **2** output neurons

$$h^r_1 = \sum_{ijk} x^r_{ijk} W_{1ijk} + b_1$$

$$h^r_2 = \sum_{ijk} x^r_{ijk} W_{2ijk} + b_2$$

3D Activations



With **2** output neurons

$$h^r_1 = \sum_{ijk} x^r_{ijk} W_{1ijk} + b_{1}$$

$$h^r_2 = \sum_{ijk} x^r_{ijk} W_{2ijk} + b_{2}$$

3D Activations

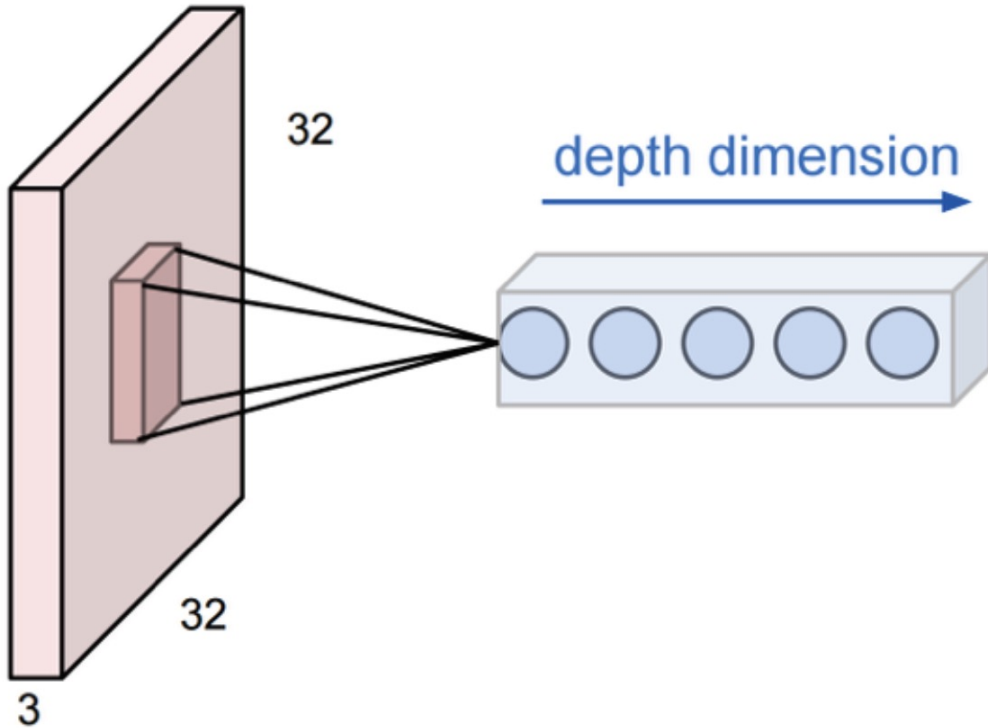
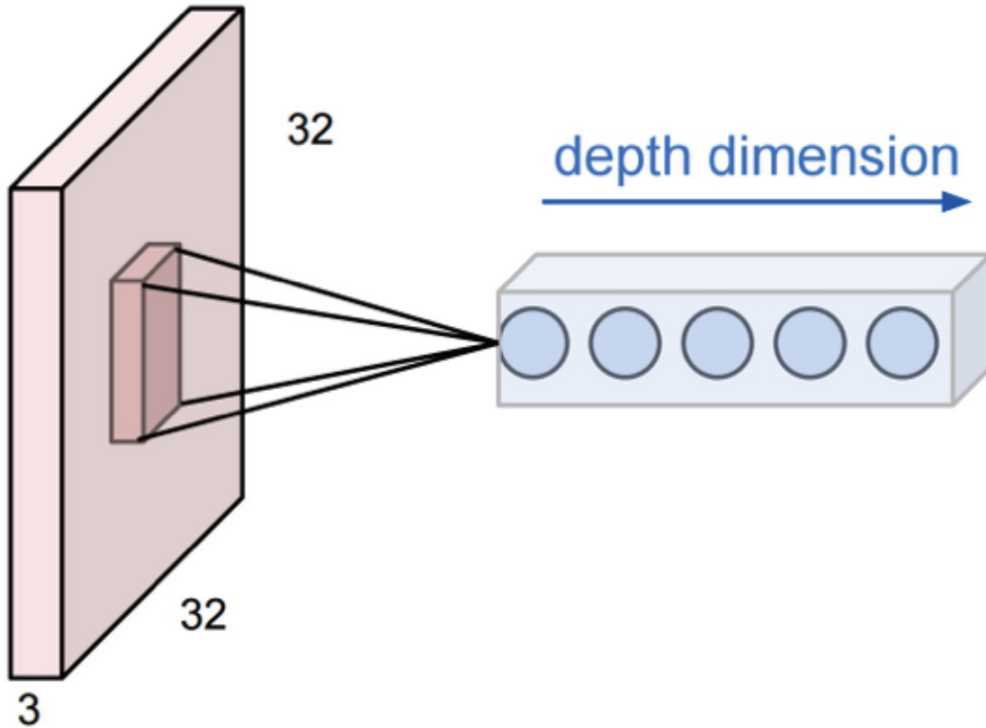


Figure: Andrej Karpathy

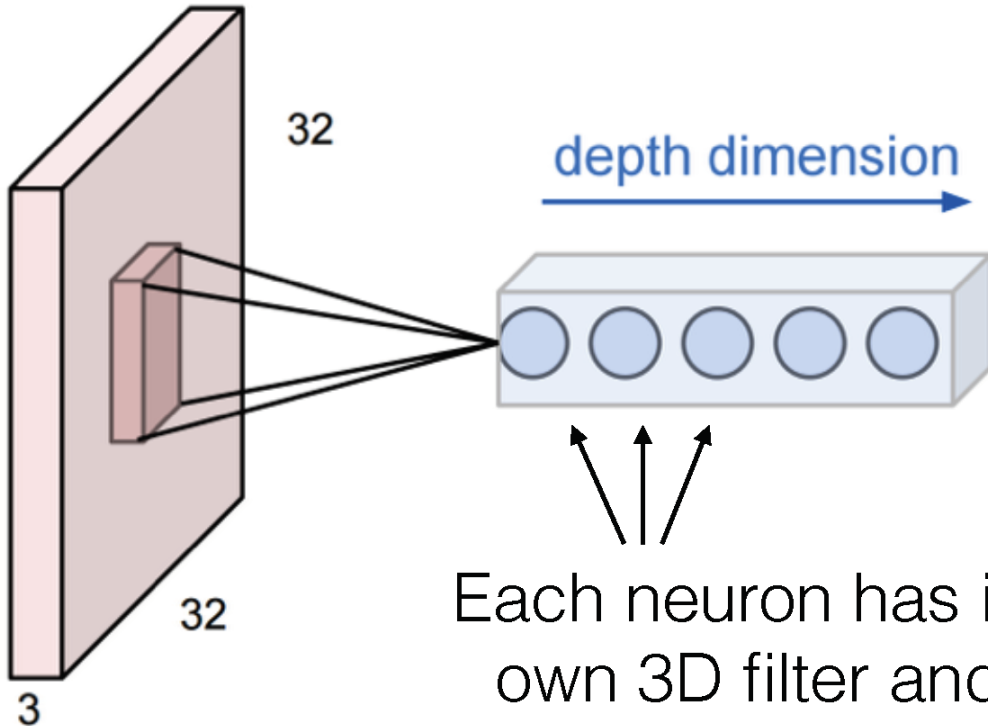
3D Activations



We can keep adding more outputs

These form a column in the output volume:
[depth x 1 x 1]

3D Activations

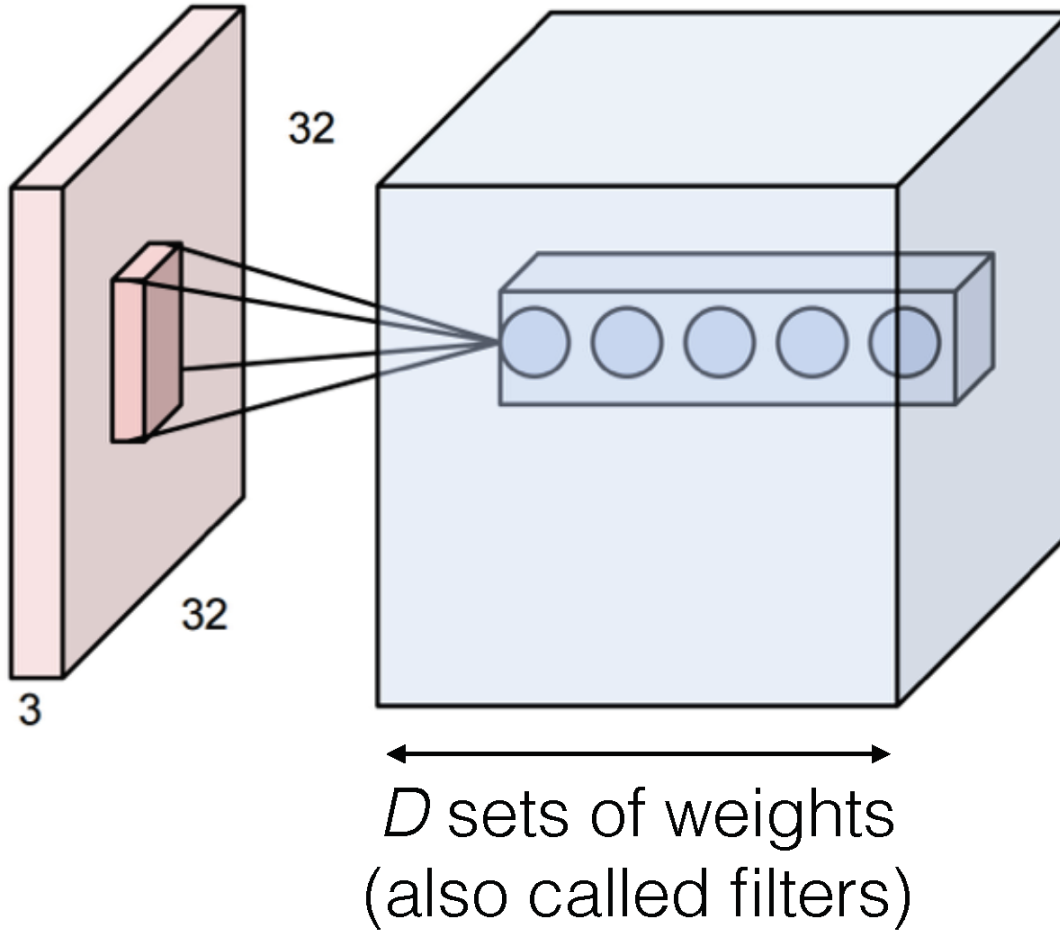


We can keep adding more outputs

These form a column in the output volume:
[depth x 1 x 1]

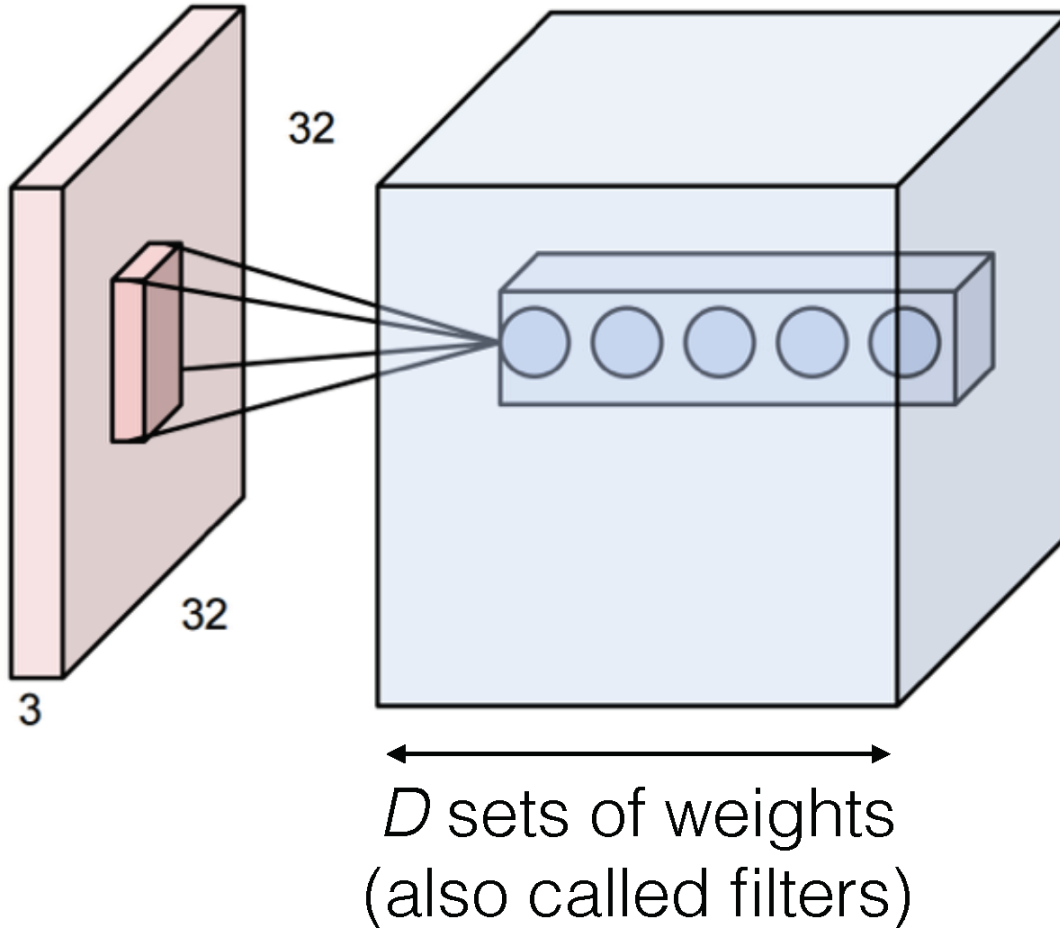
Each neuron has its own 3D filter and own (scalar) bias

3D Activations



Now repeat this
across the input

3D Activations



Now repeat this
across the input

Weight sharing:
Each filter shares
the same weights
(but each depth
index has its own
set of weights)

3D Activations

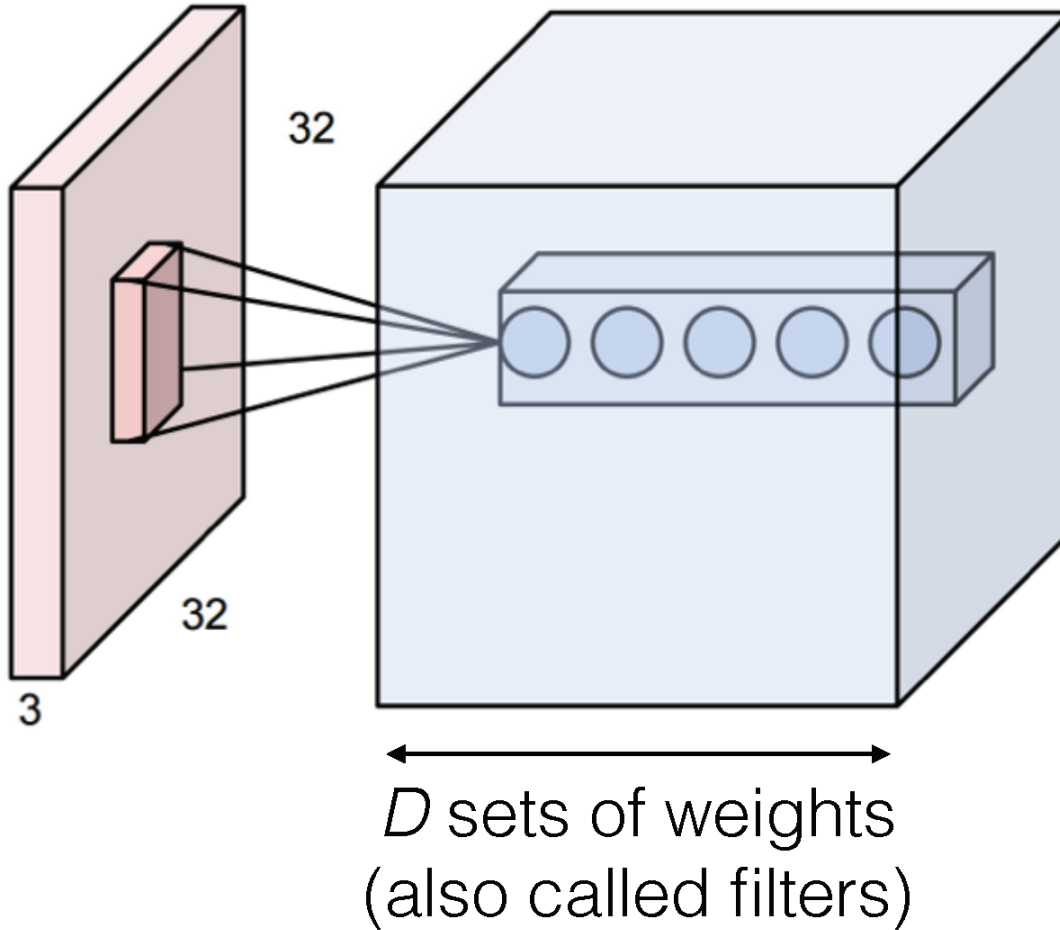
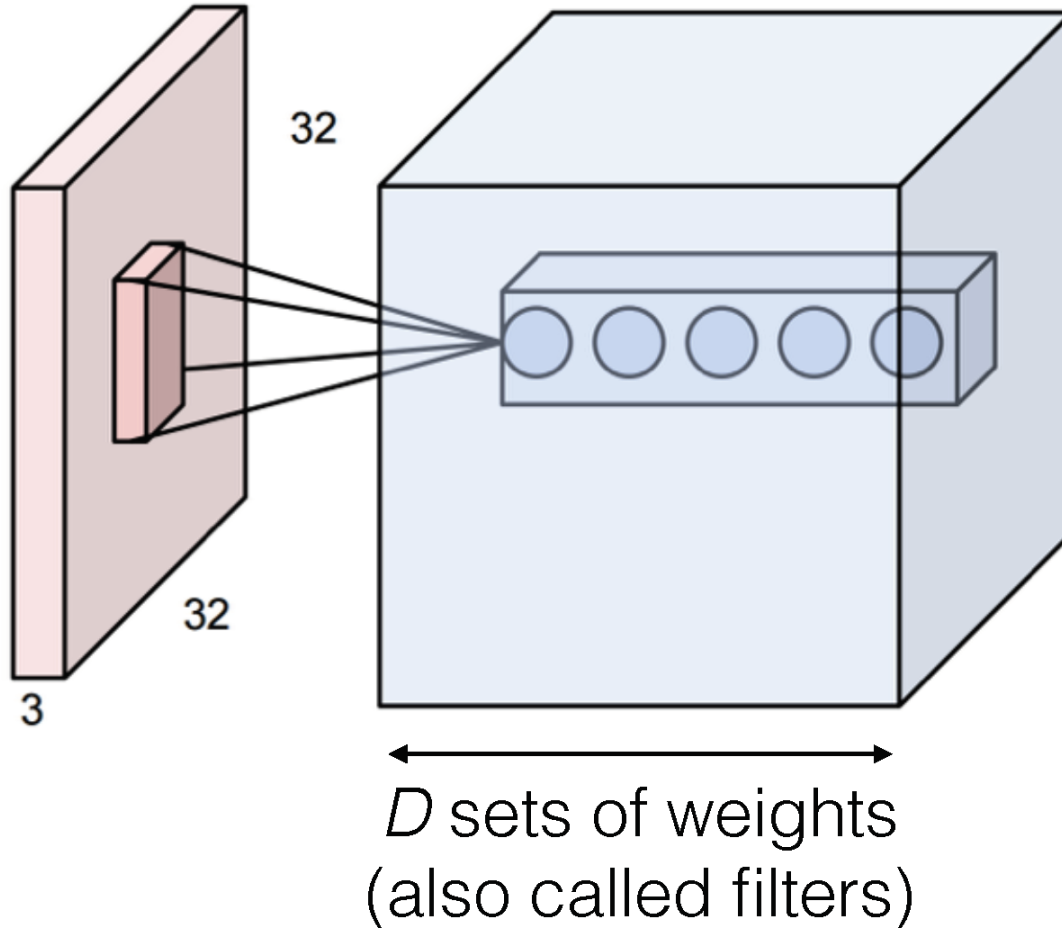


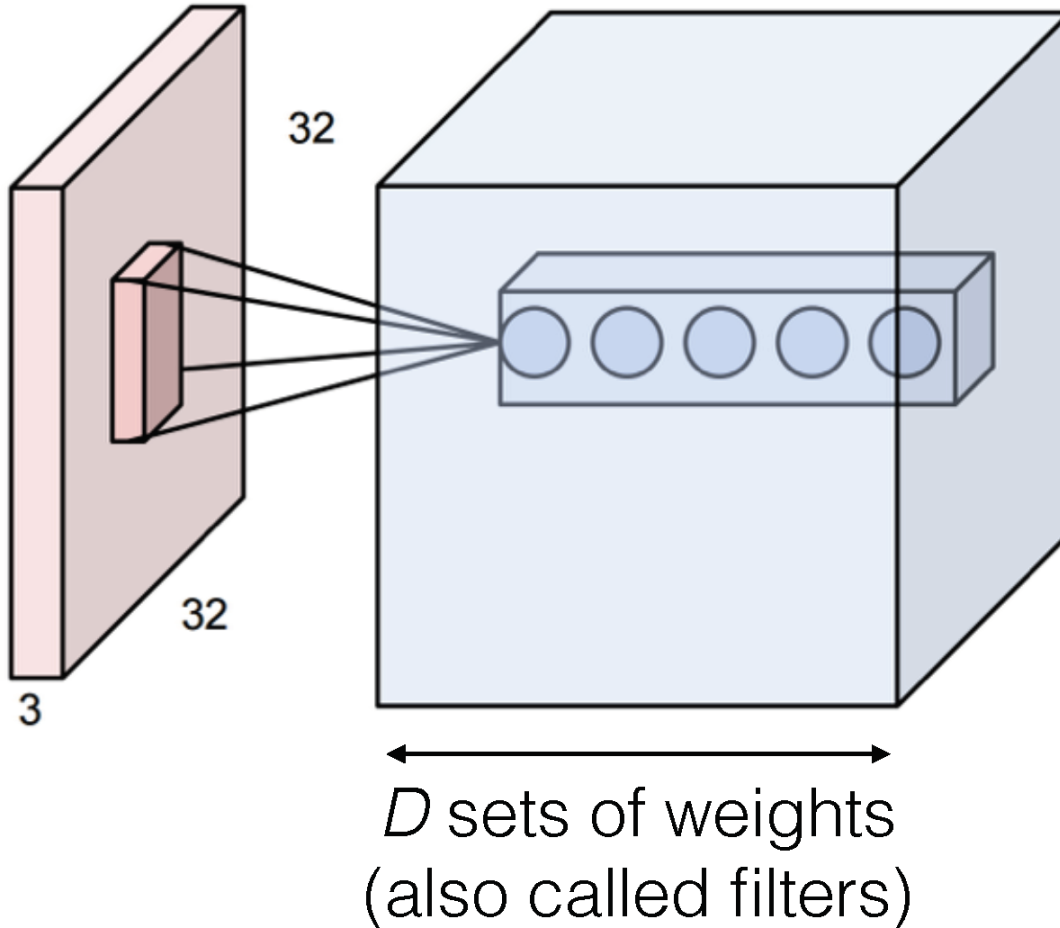
Figure: Andrej Karpathy

3D Activations



With weight sharing,
this is called **convolution**

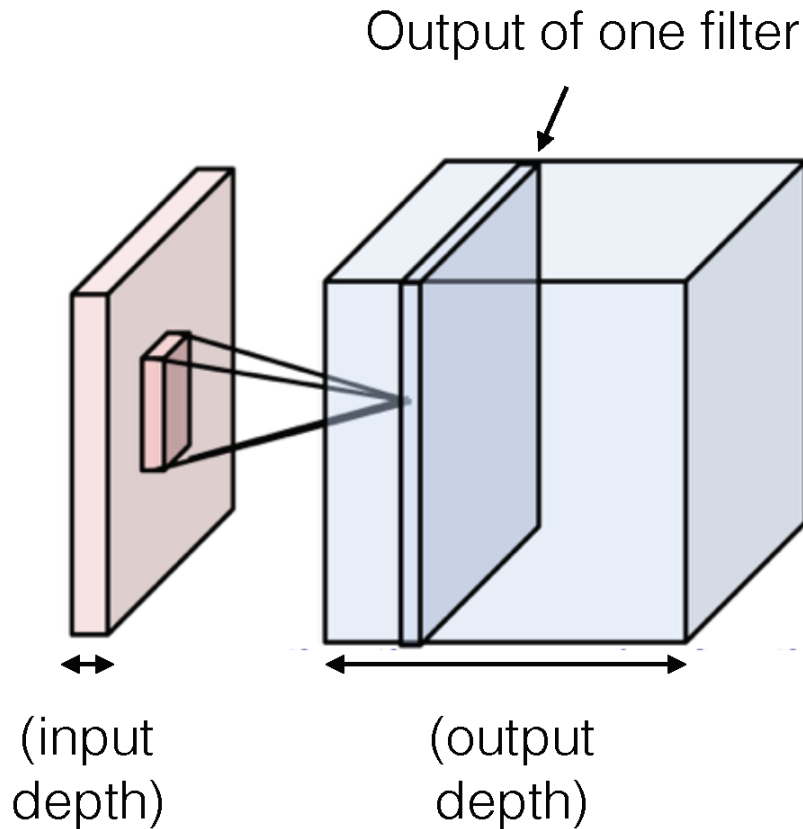
3D Activations



With weight sharing,
this is called
convolution

Without weight sharing,
this is called a
locally connected layer

3D Activations

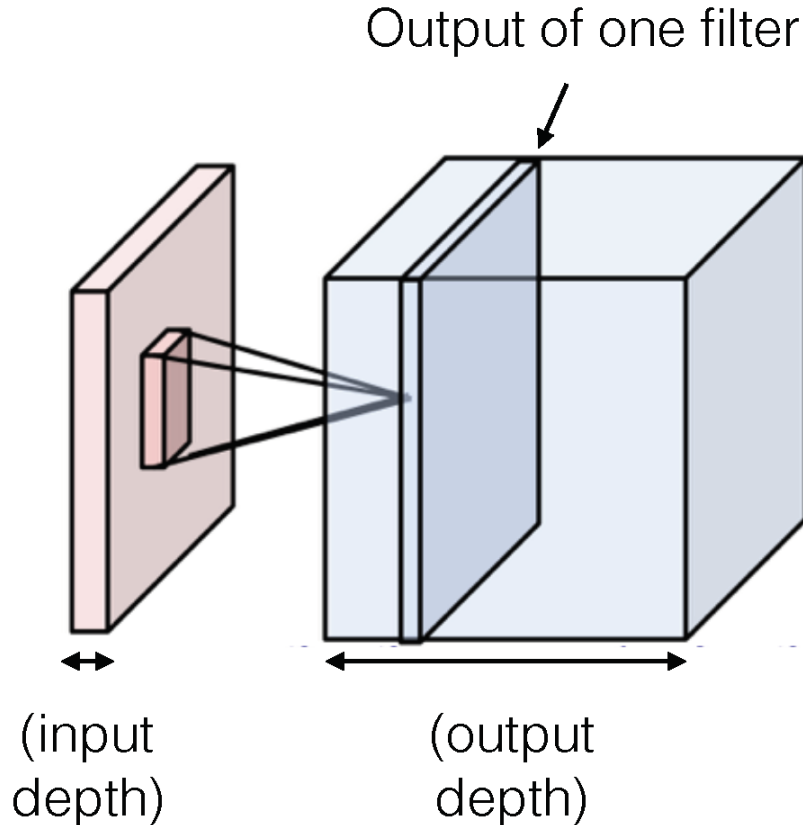


One set of weights gives one slice in the output

To get a 3D output of depth D , use D different filters

In practice, ConvNets use many filters (~ 64 to 1024)

3D Activations



One set of weights gives one slice in the output

To get a 3D output of depth D , use D different filters

In practice, ConvNets use many filters (~ 64 to 1024)

All together, the weights are **4** dimensional:
(output depth, input depth, kernel height, kernel width)

3D Activations

We can unravel the 3D cube and show each layer separately:

(Input)



one filter = one depth slice (or activation map) (32 filters, each 3x5x5)

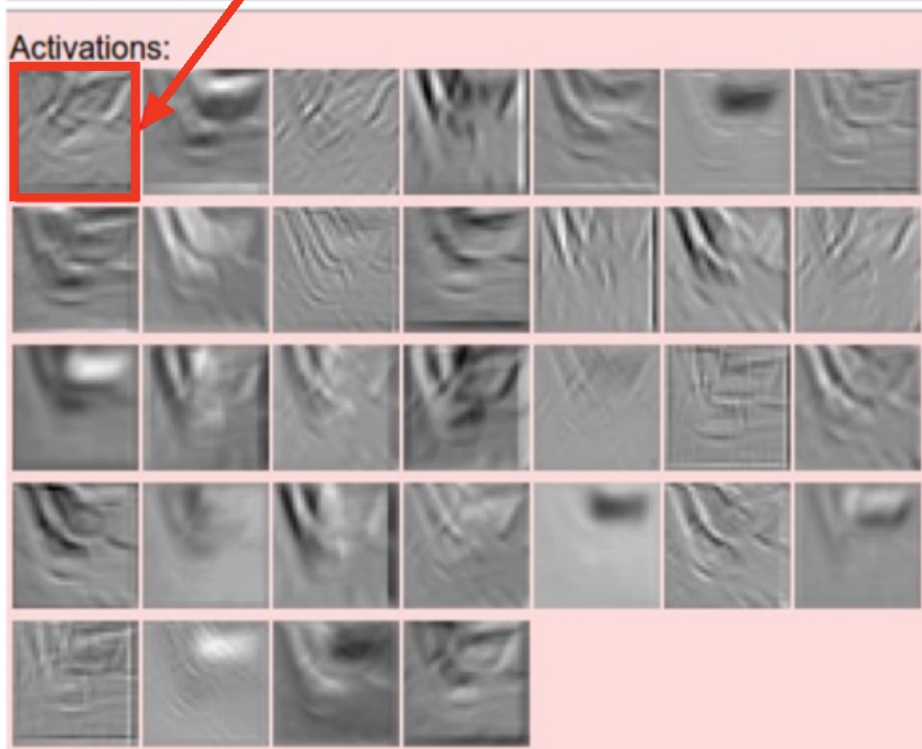


Figure: Andrej Karpathy

3D Activations

We can unravel the 3D cube and show each layer separately:

(Input)

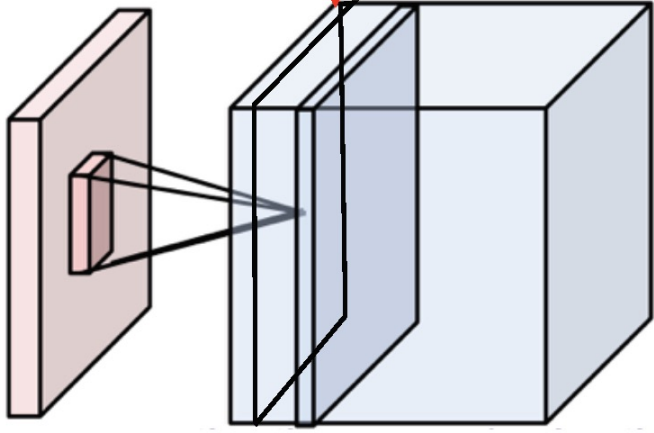
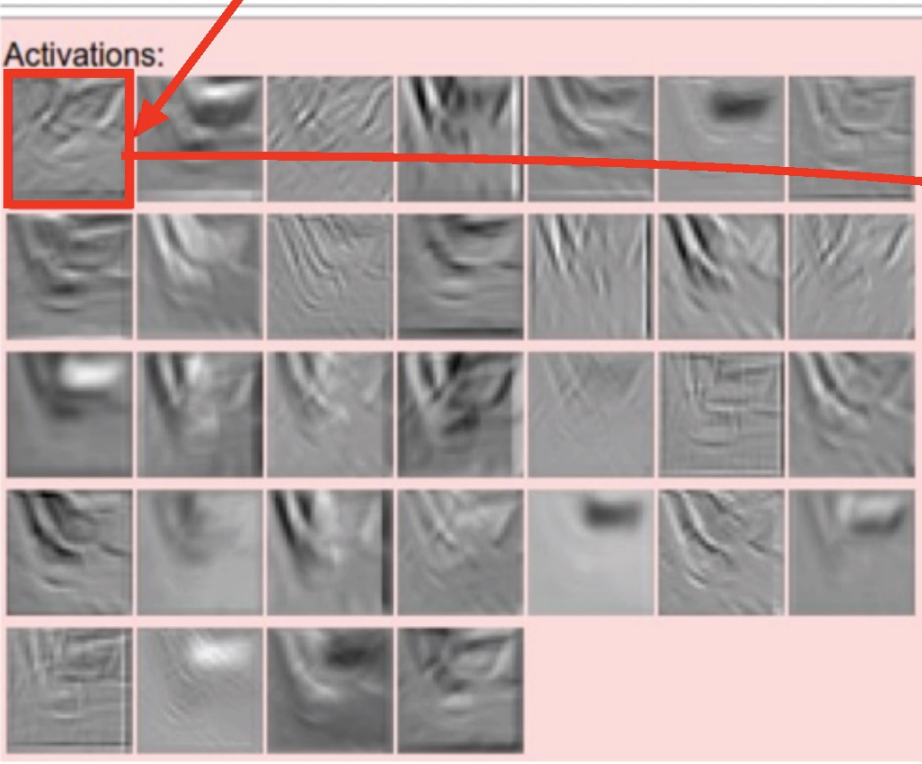
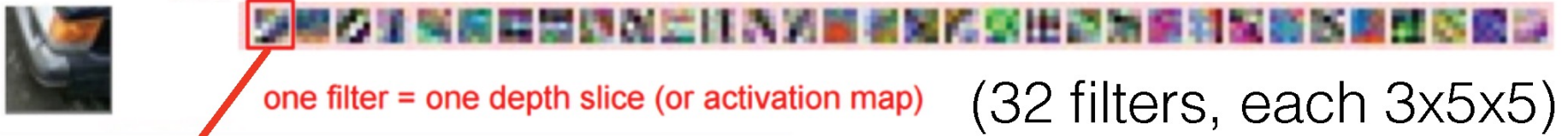


Figure: Andrej Karpathy

3D Activations

We can unravel the 3D cube and show each layer separately:

(Input)

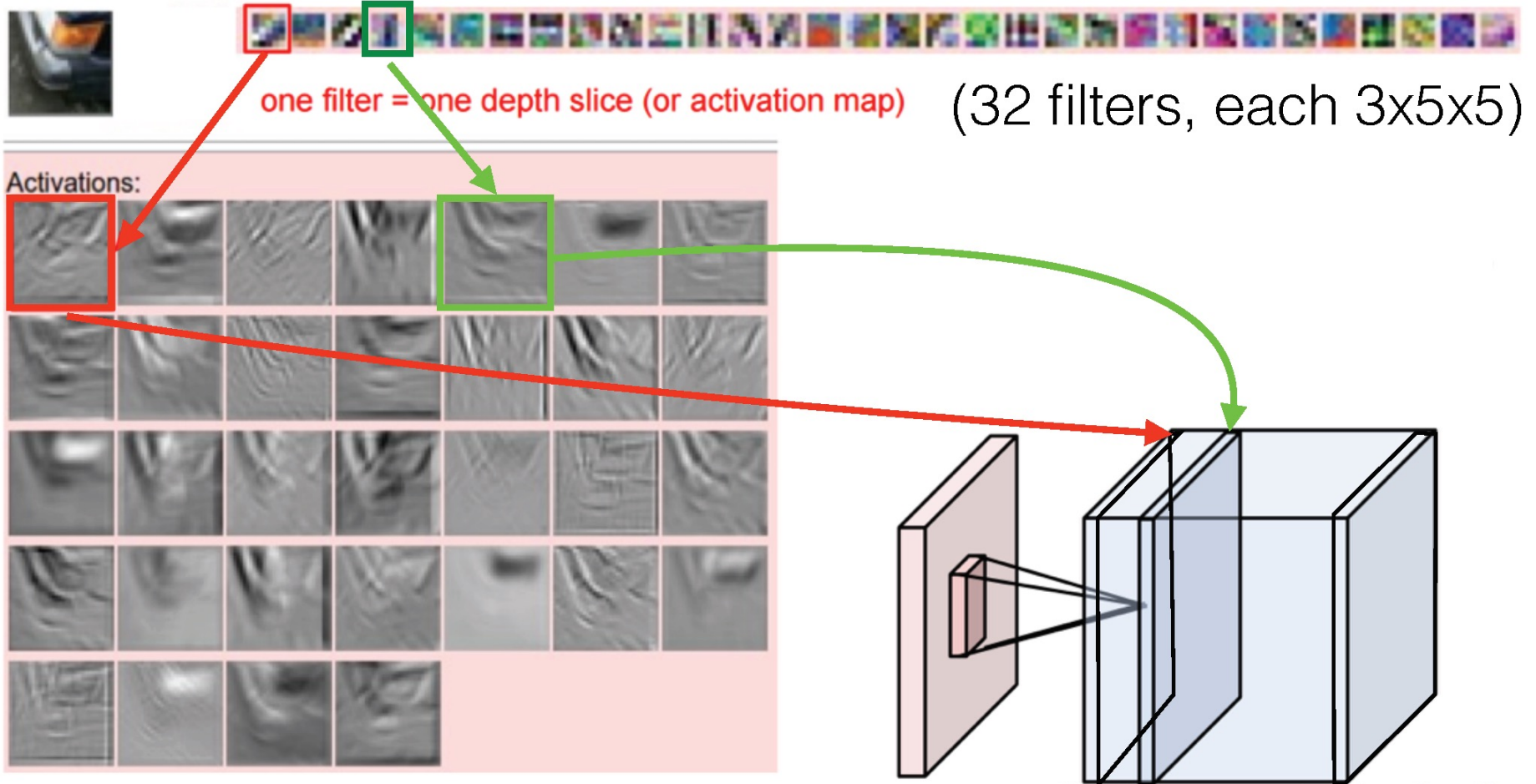
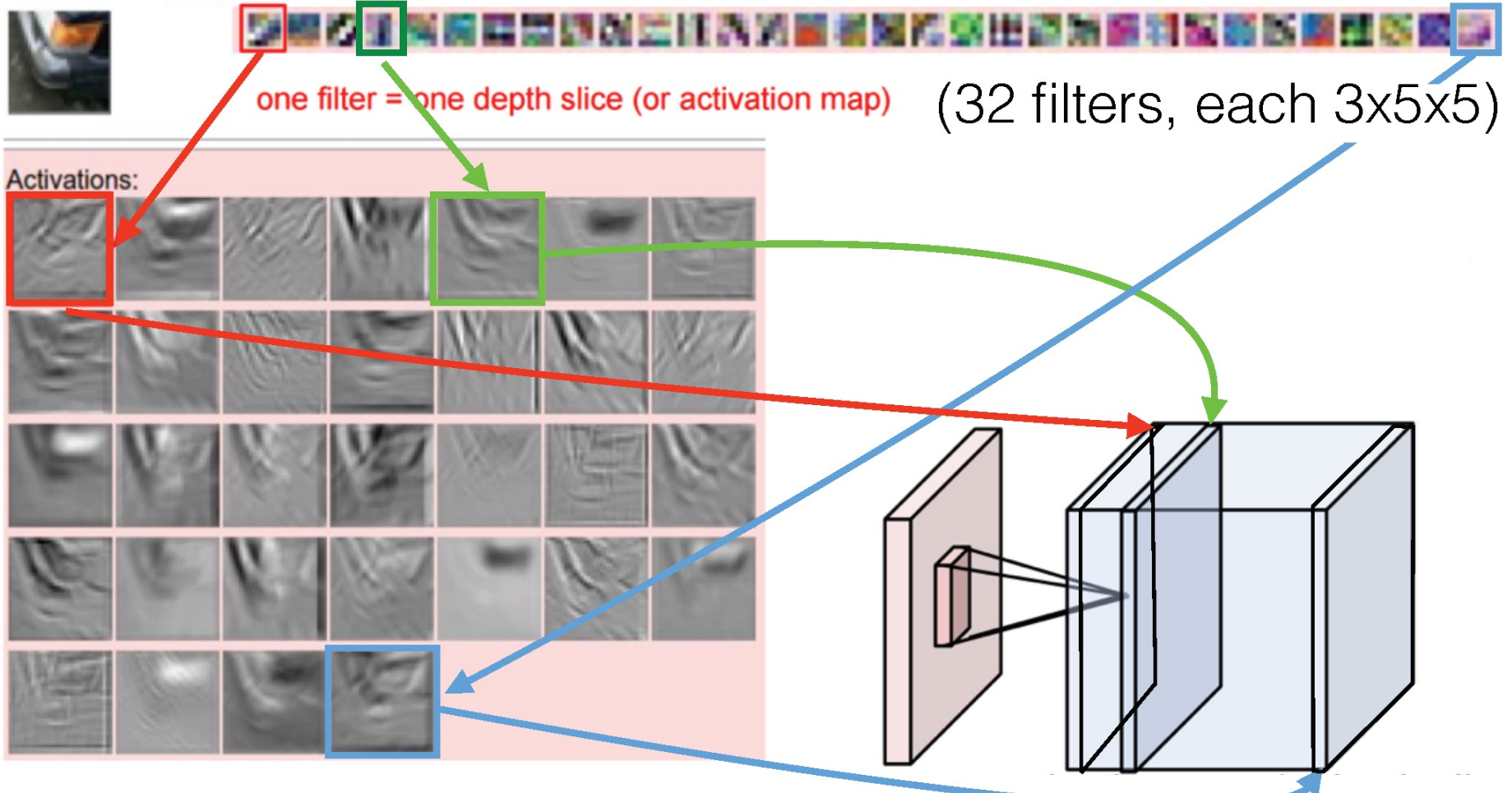


Figure: Andrej Karpathy

3D Activations

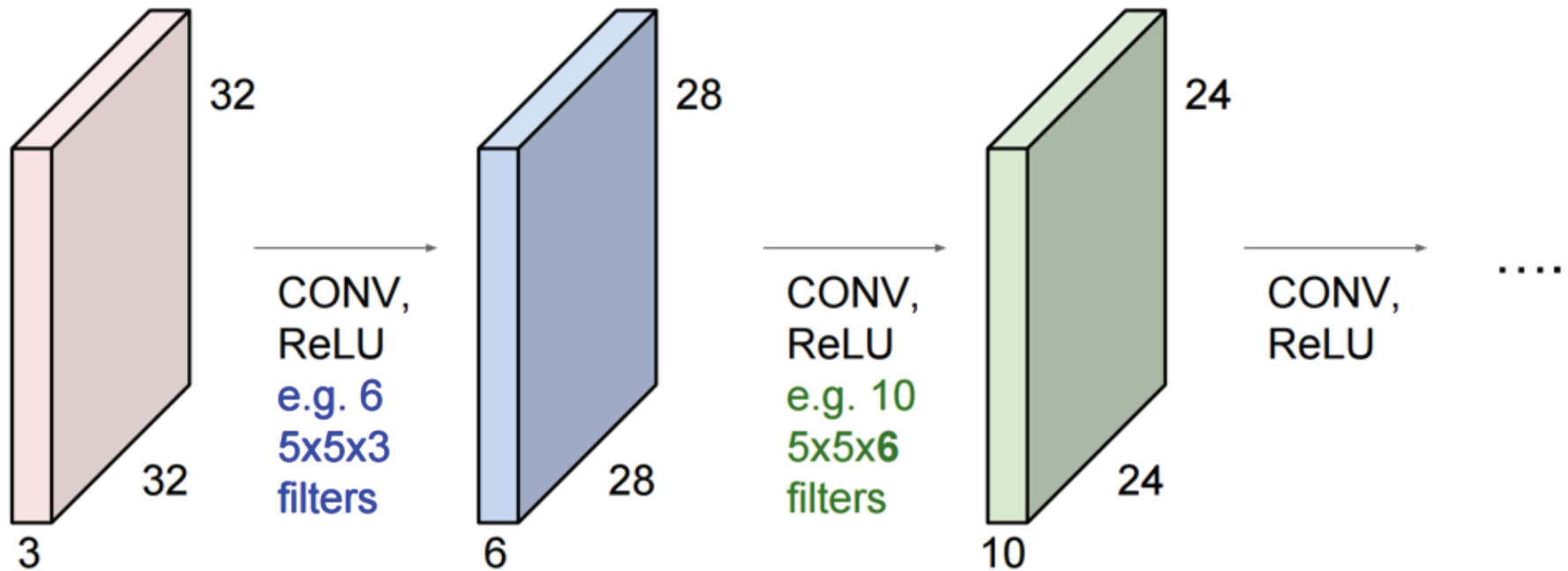
We can unravel the 3D cube and show each layer separately:

(Input)



(Recap)

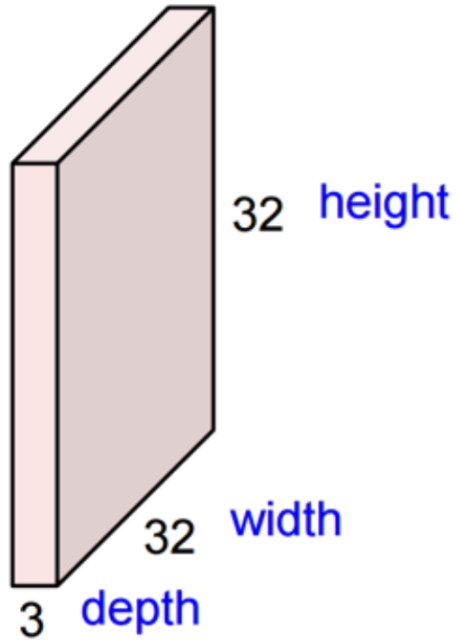
A **ConvNet** is a sequence of convolutional layers, interspersed with activation functions (and possibly other layer types)



(Recap)

Convolution Layer

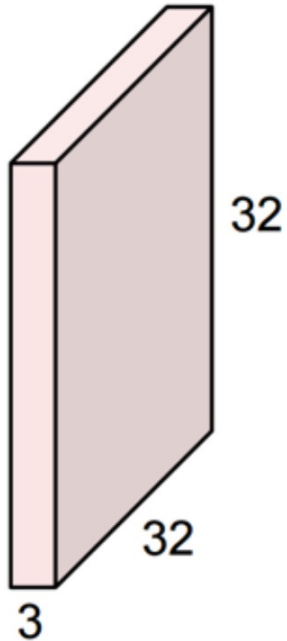
32x32x3 image



(Recap)

Convolution Layer

32x32x3 image



5x5x3 filter

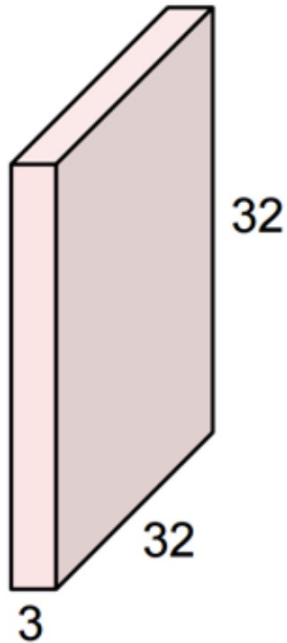


Convolve the filter with the image
i.e. “slide over the image spatially,
computing dot products”

(Recap)

Convolution Layer

32x32x3 image



Filters always extend the full depth of the input volume

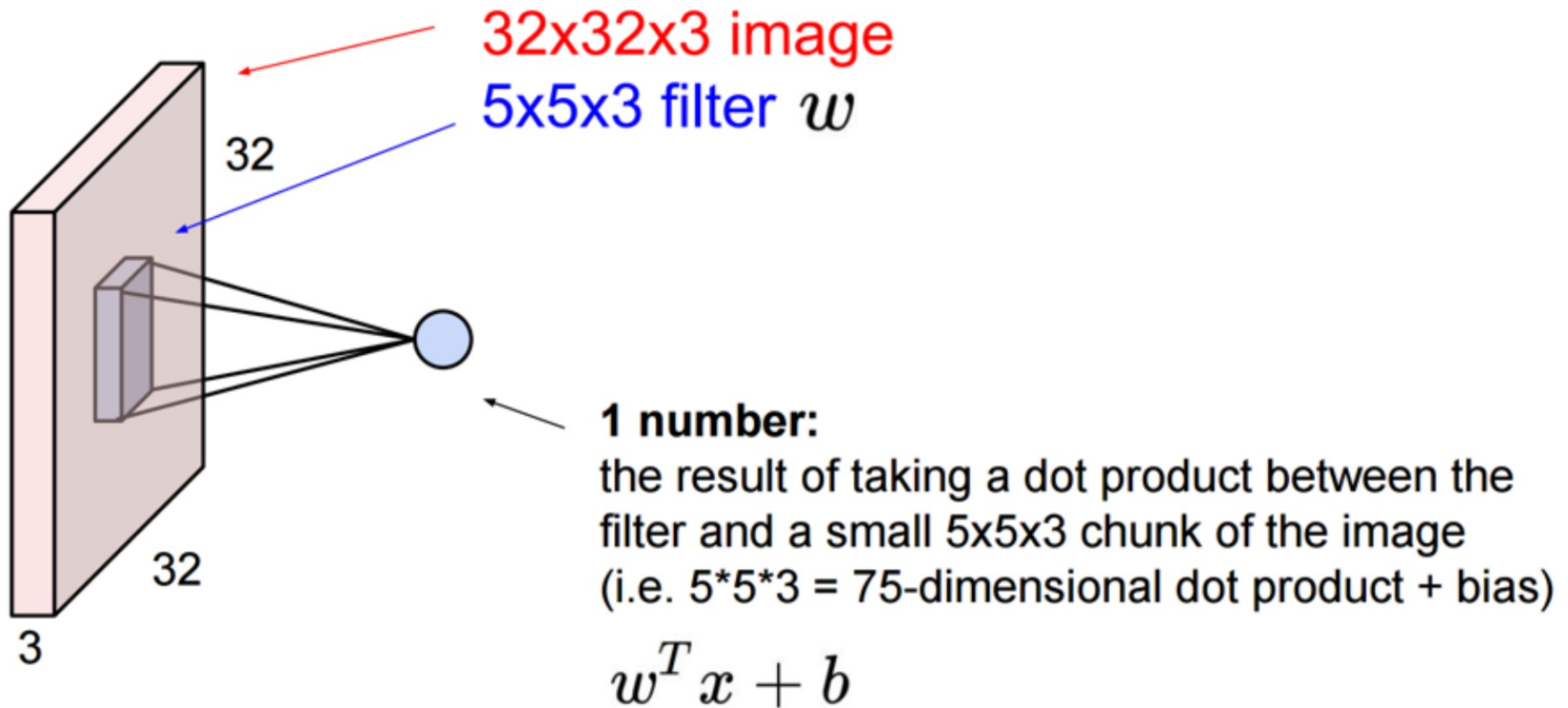
5x5x3 filter



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

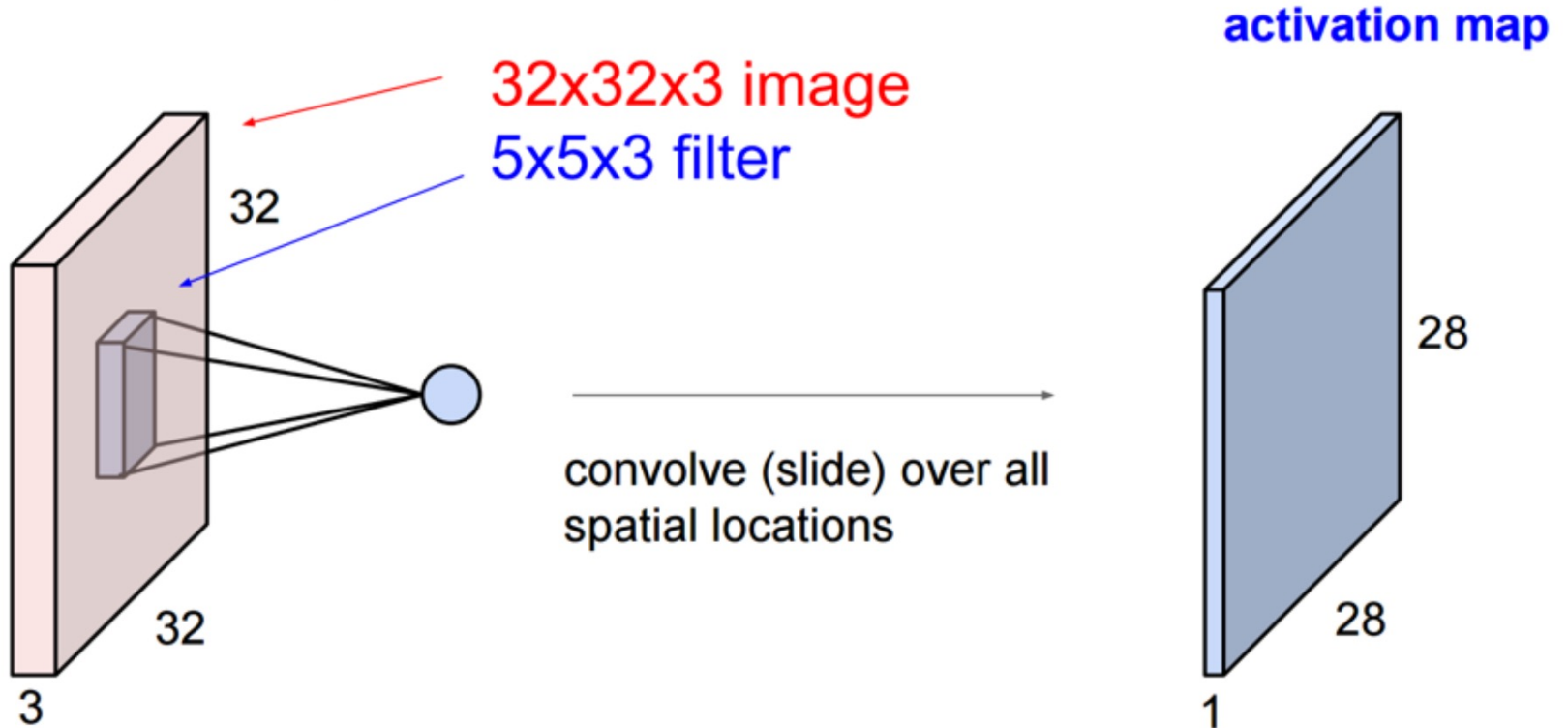
(Recap)

Convolution Layer



(Recap)

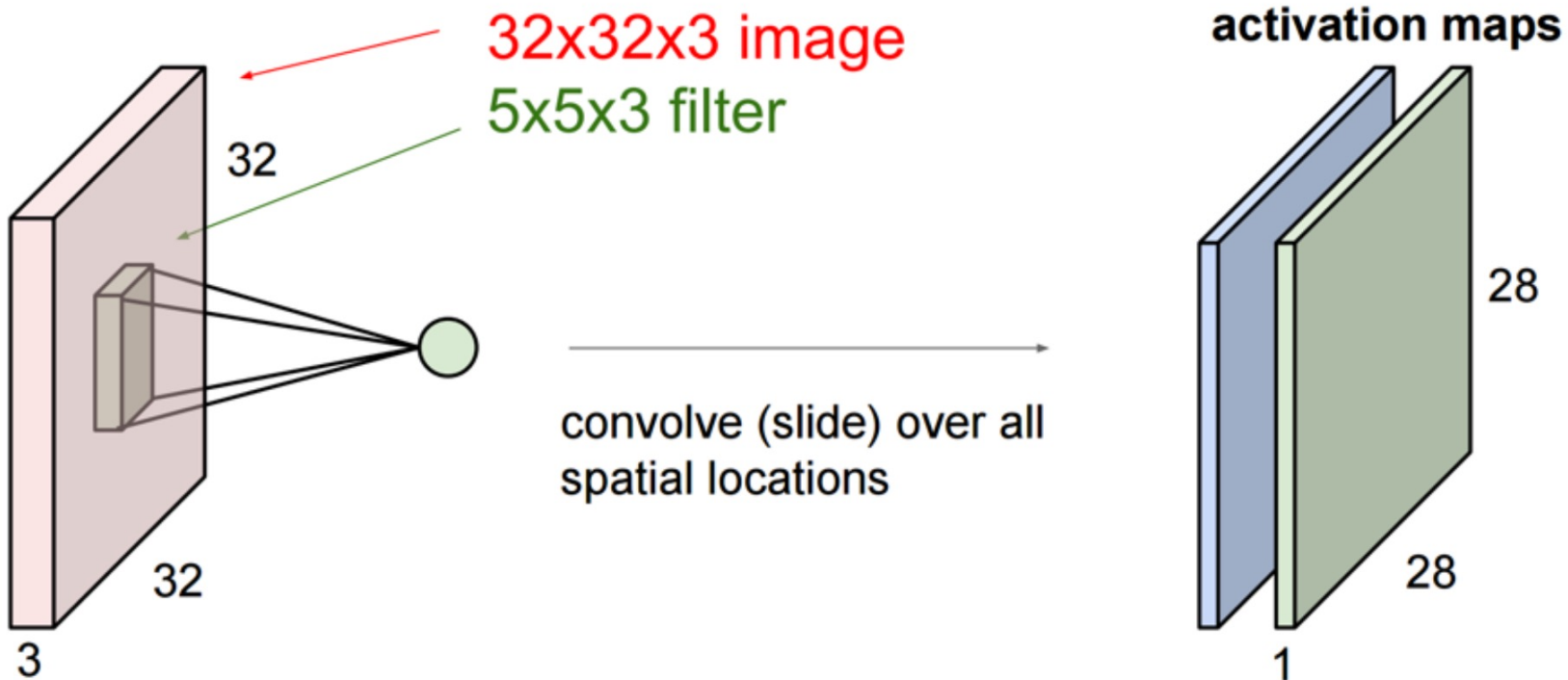
Convolution Layer



(Recap)

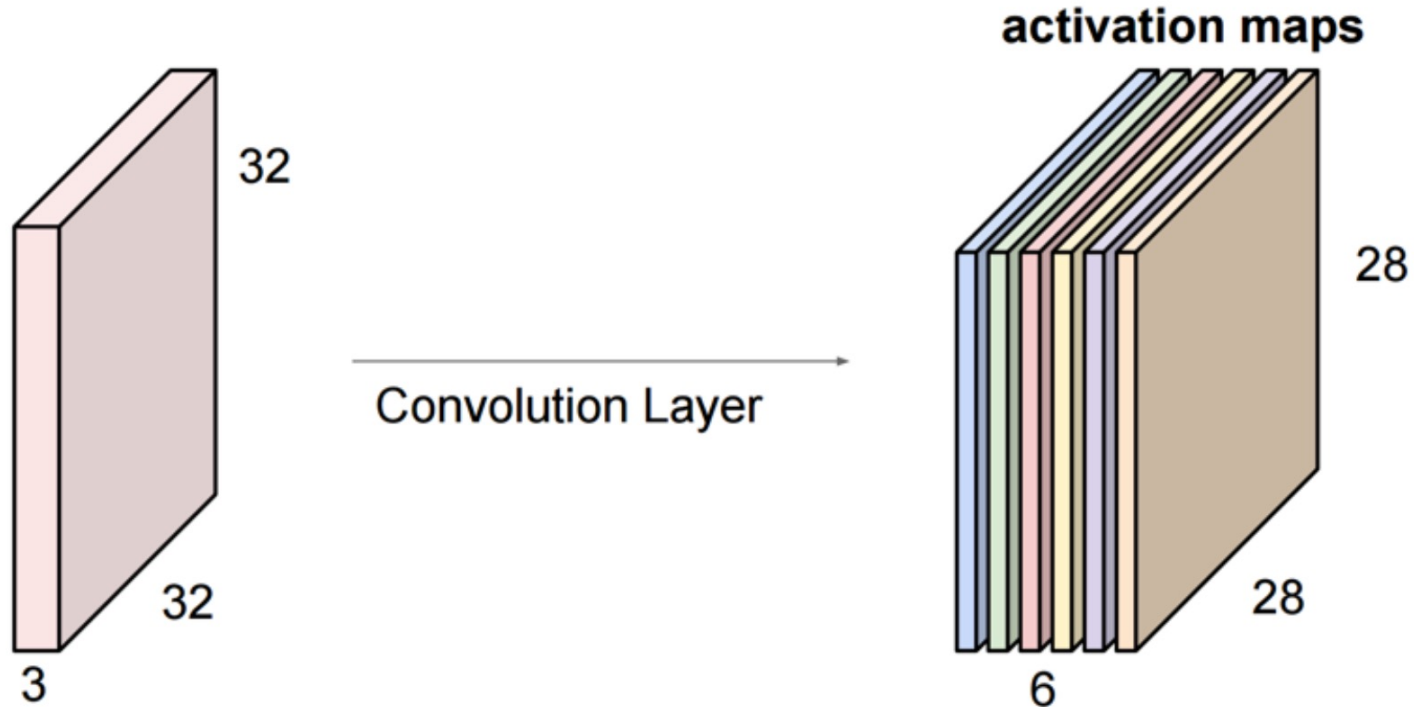
Convolution Layer

consider a second, **green** filter



(Recap)

For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:



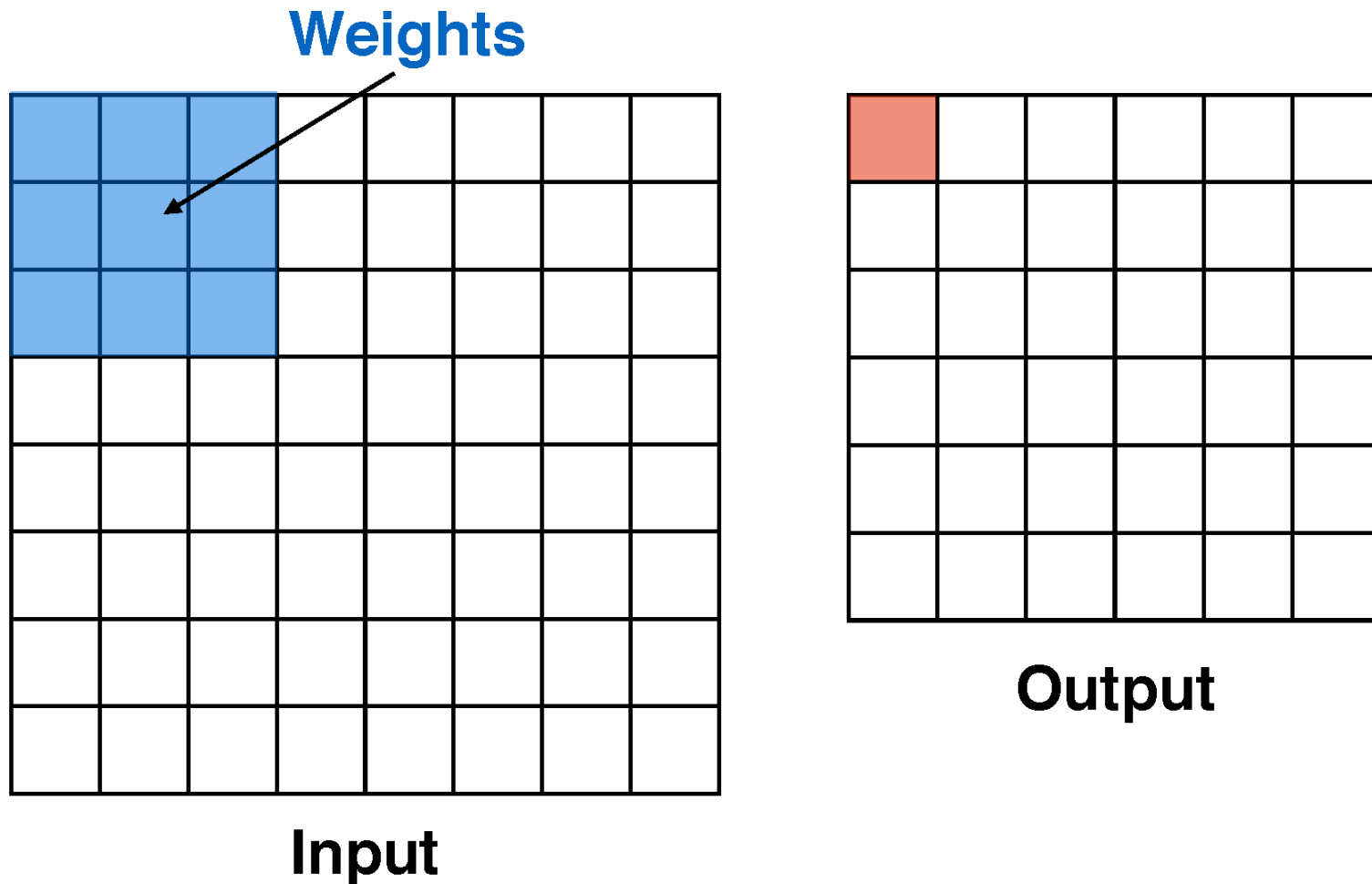
We stack these up to get a “new image” of size 28x28x6!

Demos

- <http://cs231n.stanford.edu/>
- <http://cs.stanford.edu/people/karpathy/convnetjs/demo/mnist.html>

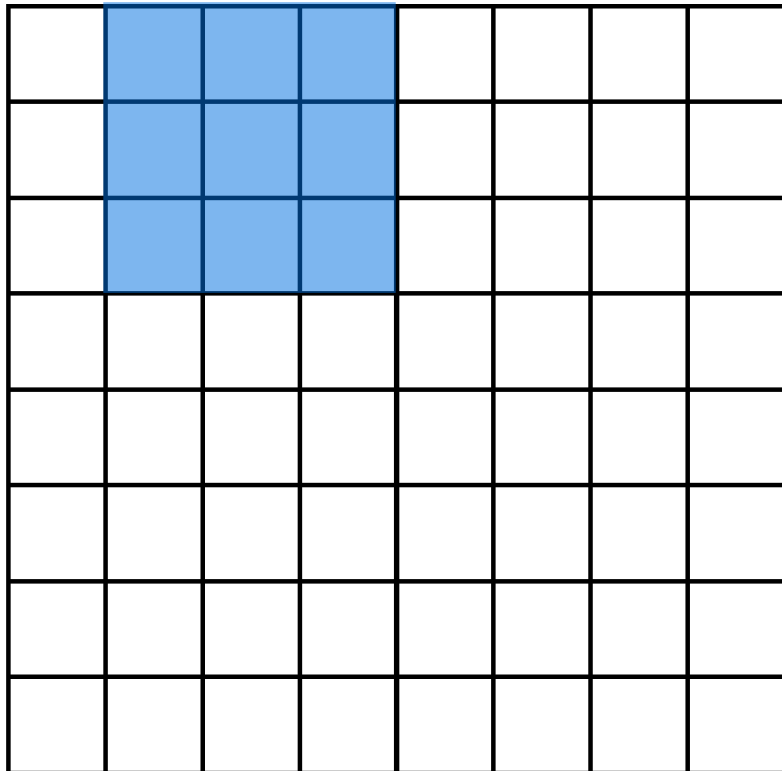
Convolution: Stride

During convolution, the weights “slide” along the input to generate each output

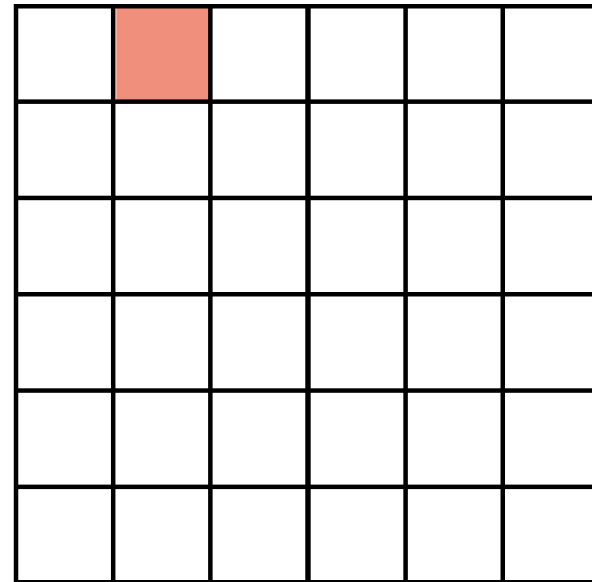


Convolution: Stride

During convolution, the weights “slide” along the input to generate each output



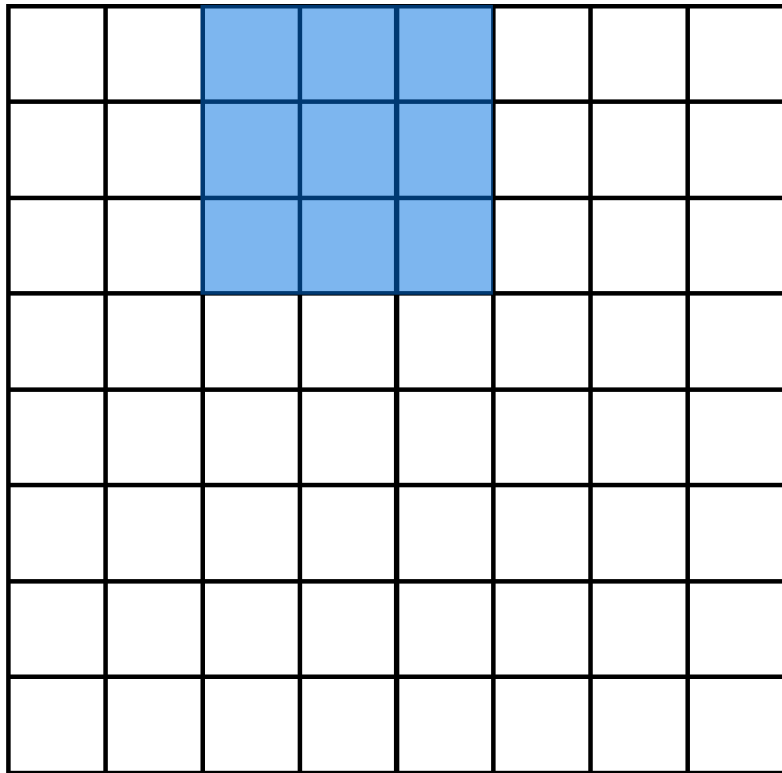
Input



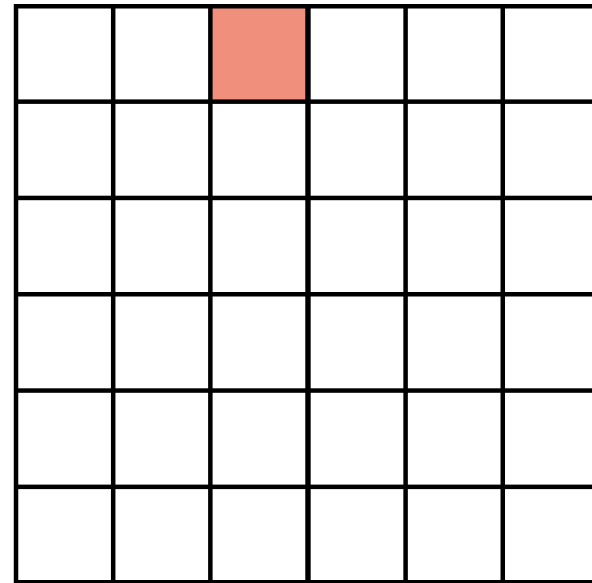
Output

Convolution: Stride

During convolution, the weights “slide” along the input to generate each output



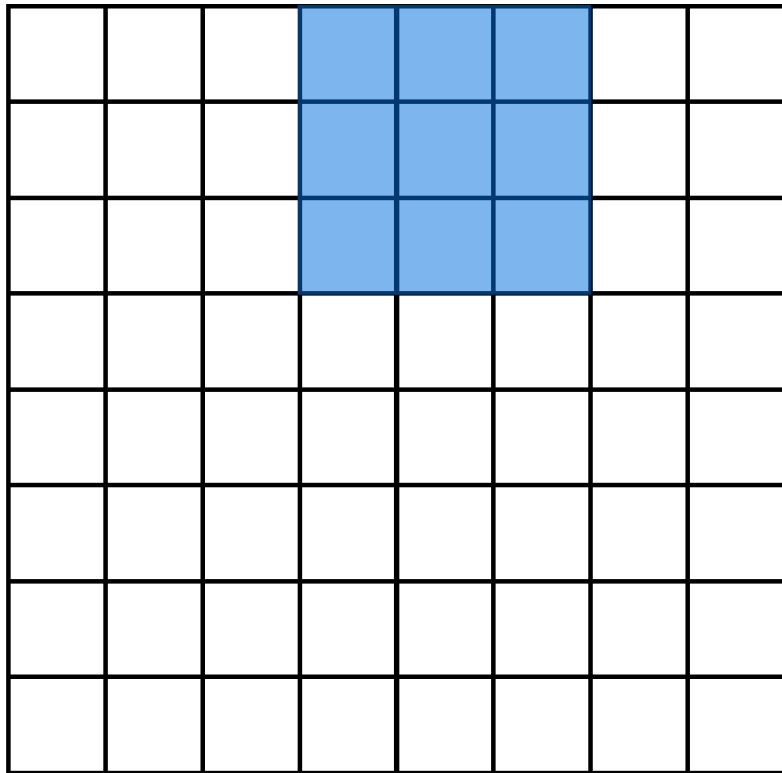
Input



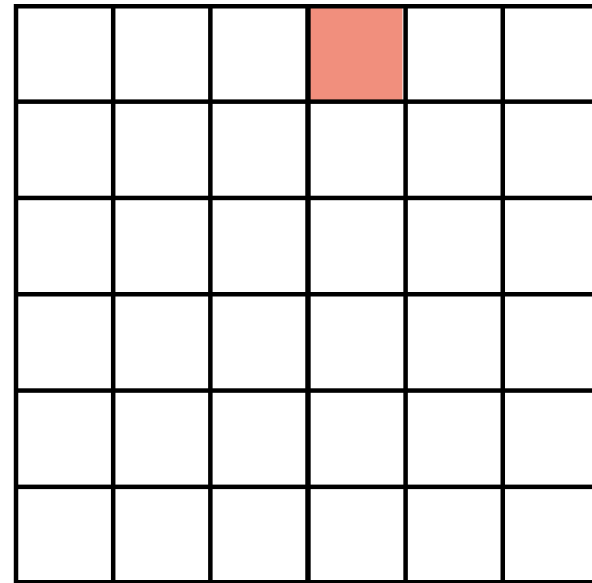
Output

Convolution: Stride

During convolution, the weights “slide” along the input to generate each output



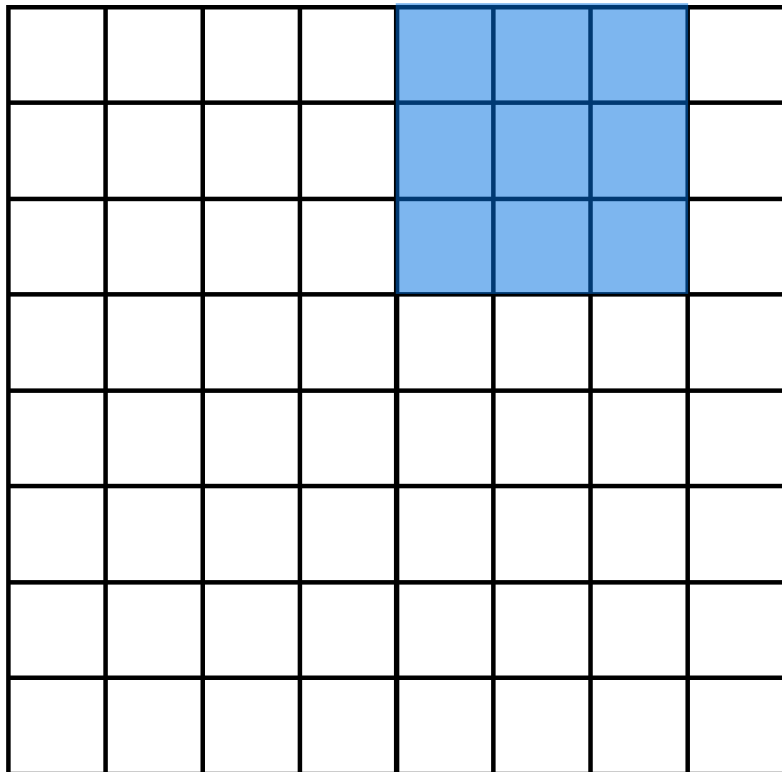
Input



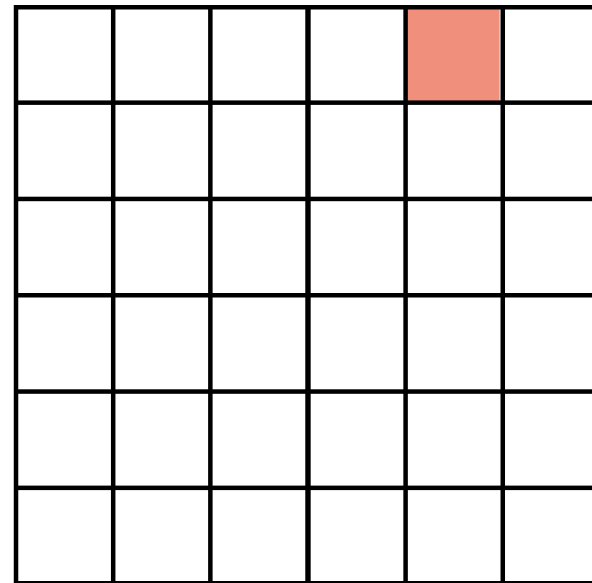
Output

Convolution: Stride

During convolution, the weights “slide” along the input to generate each output



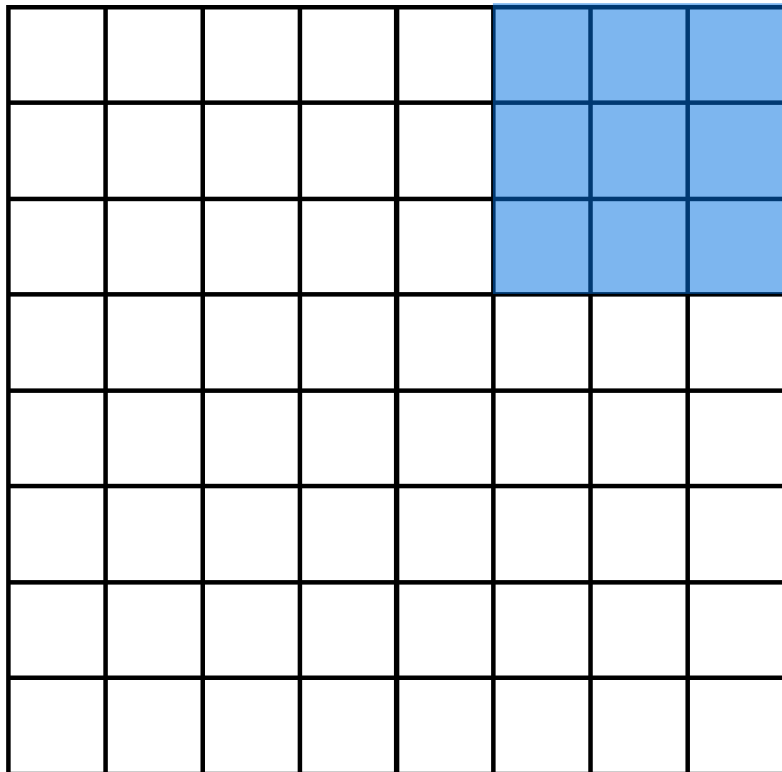
Input



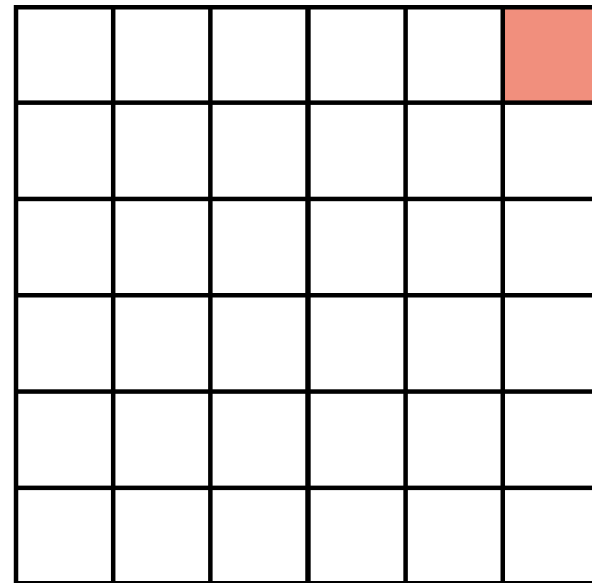
Output

Convolution: Stride

During convolution, the weights “slide” along the input to generate each output



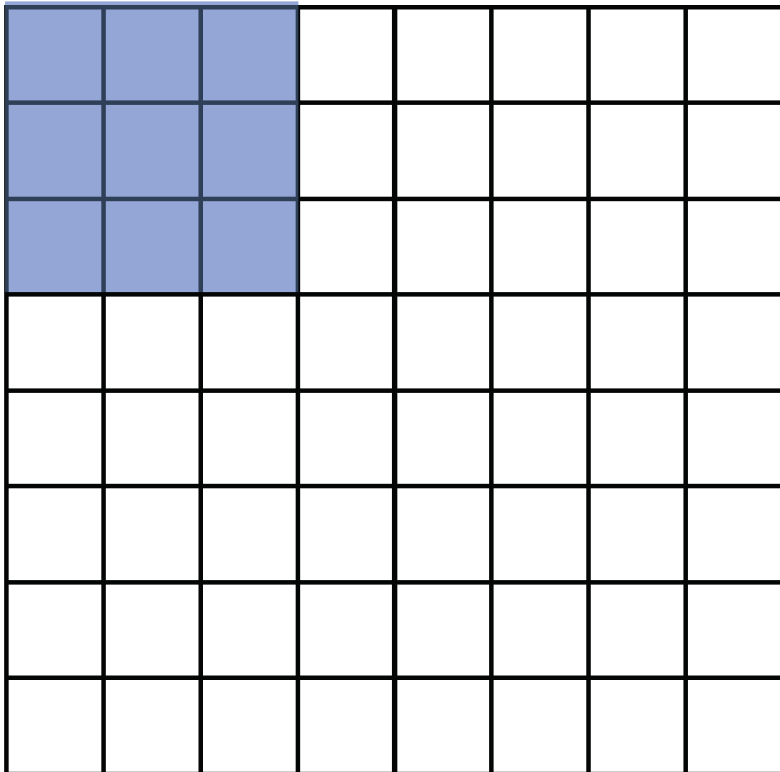
Input



Output

Convolution: Stride

During convolution, the weights “slide” along the input to generate each output



Input

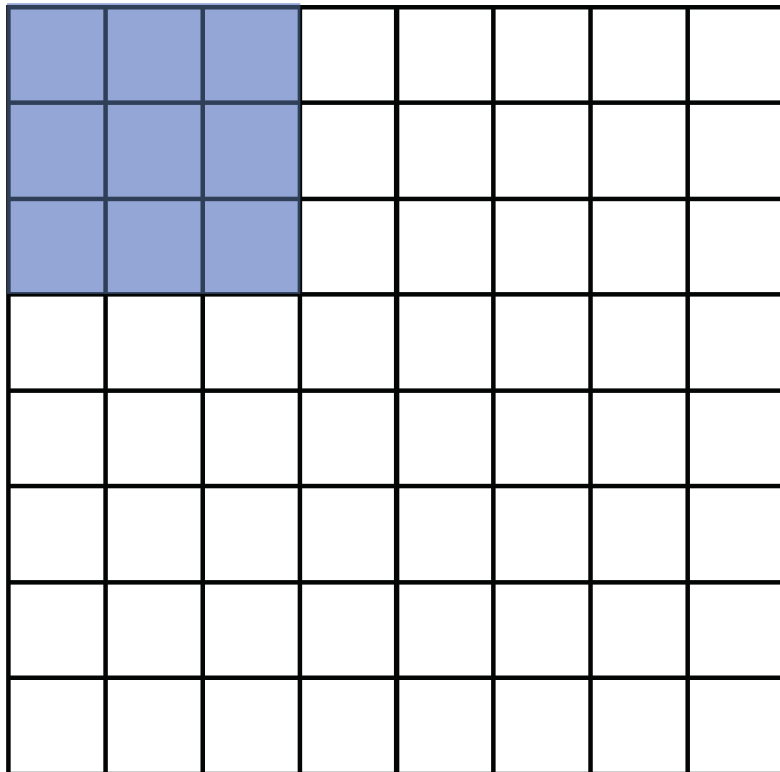
Recall that at each position, we are doing a **3D** sum:

$$h^r = \sum_{ijk} x^r_{ijk} W_{ijk} + b$$

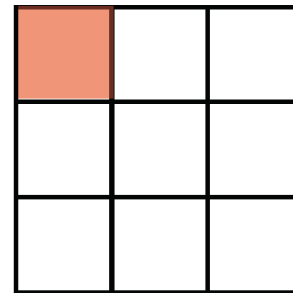
(channel, row, column)

Convolution: Stride

But we can also convolve with a **stride**, e.g. stride = 2



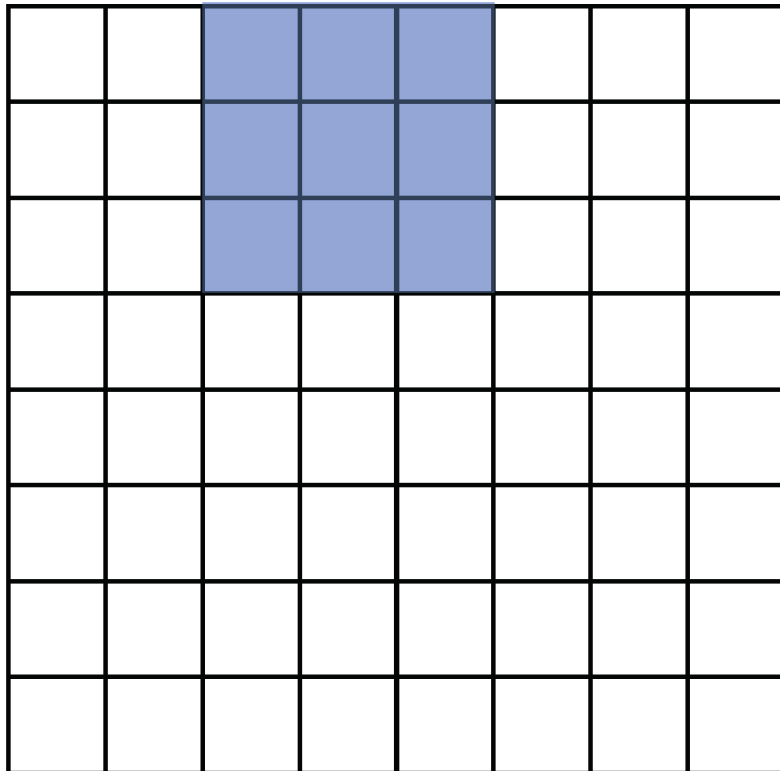
Input



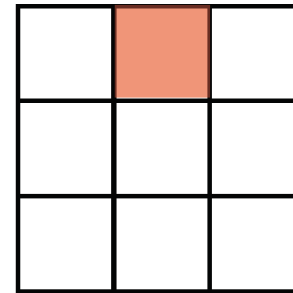
Output

Convolution: Stride

But we can also convolve with a **stride**, e.g. stride = 2



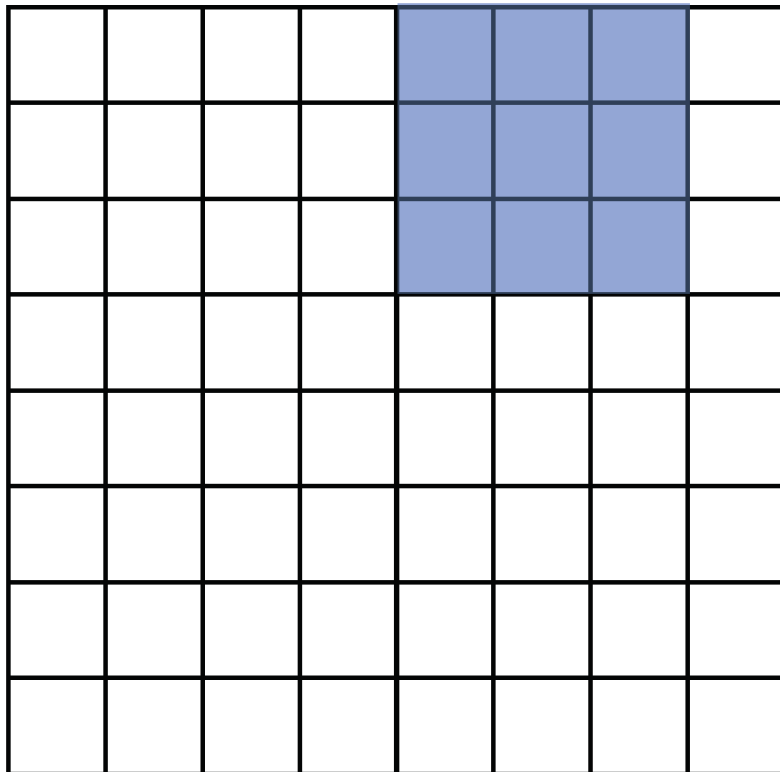
Input



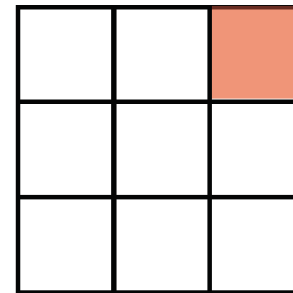
Output

Convolution: Stride

But we can also convolve with a **stride**, e.g. stride = 2



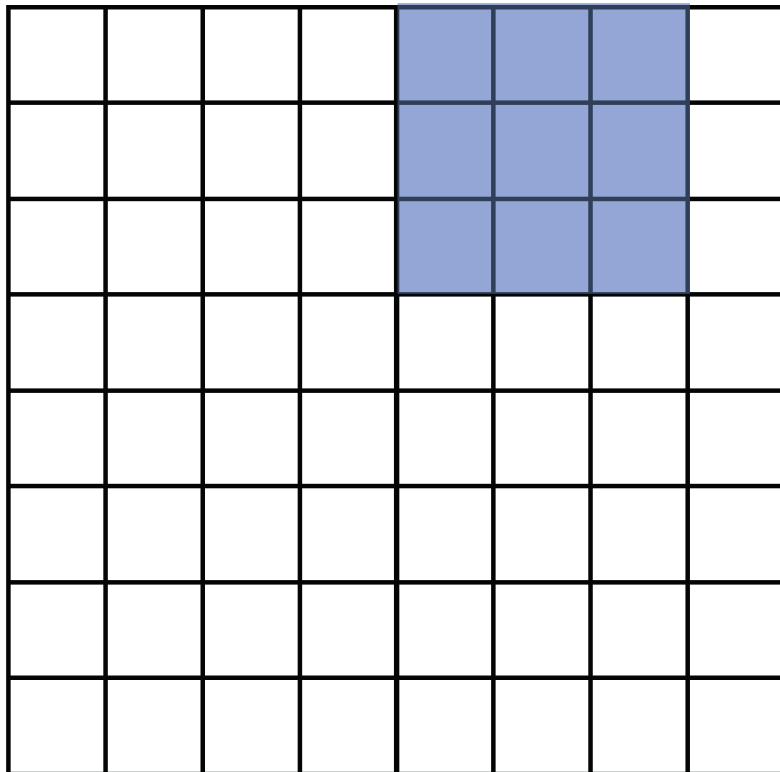
Input



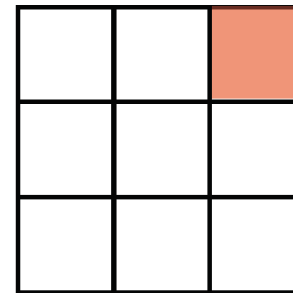
Output

Convolution: Stride

But we can also convolve with a **stride**, e.g. stride = 2



Input



Output

- Notice that with certain strides, we may not be able to cover all of the input
- The output is also half the size of the input

Convolution: Padding

We can also pad the input with zeros.

Here, **pad = 1, stride = 2**

0	0	0	0	0	0	0	0	0
0								0
0								0
0								0
0								0
0								0
0								0
0								0
0								0
0	0	0	0	0	0	0	0	0

Input

Output

Convolution: Padding

We can also pad the input with zeros.

Here, **pad = 1**, **stride = 2**

0	0	0	0	0	0	0	0	0
0								0
0								0
0								0
0								0
0								0
0								0
0								0
0								0
0	0	0	0	0	0	0	0	0

Input

Output

Convolution: Padding

We can also pad the input with zeros.

Here, **pad = 1**, **stride = 2**

0	0	0	0	0	0	0	0	0
0								0
0								0
0								0
0								0
0								0
0								0
0								0
0								0
0	0	0	0	0	0	0	0	0

Input

Output

Convolution: Padding

We can also pad the input with zeros.

Here, **pad = 1**, **stride = 2**

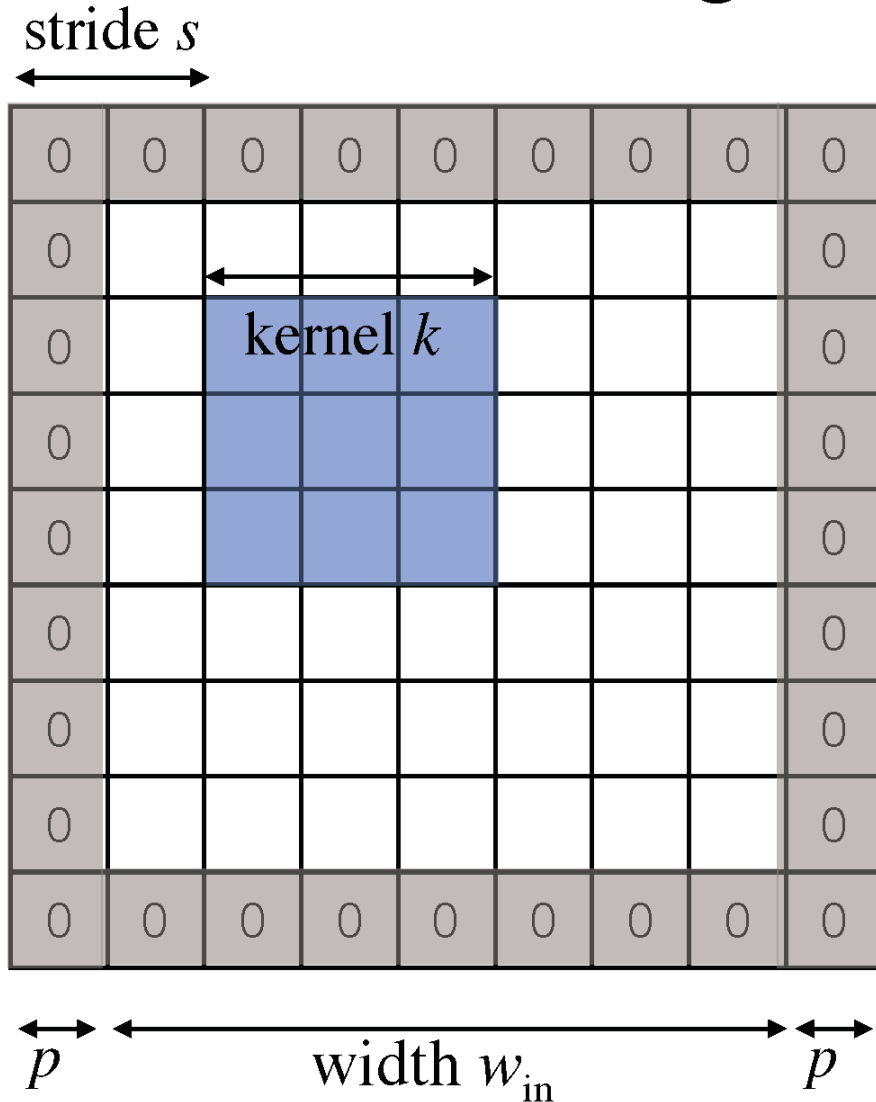
0	0	0	0	0	0	0	0	0
0								0
0								0
0								0
0								0
0								0
0								0
0								0
0	0	0	0	0	0	0	0	0

Input

Output

Convolution:

How big is the output?

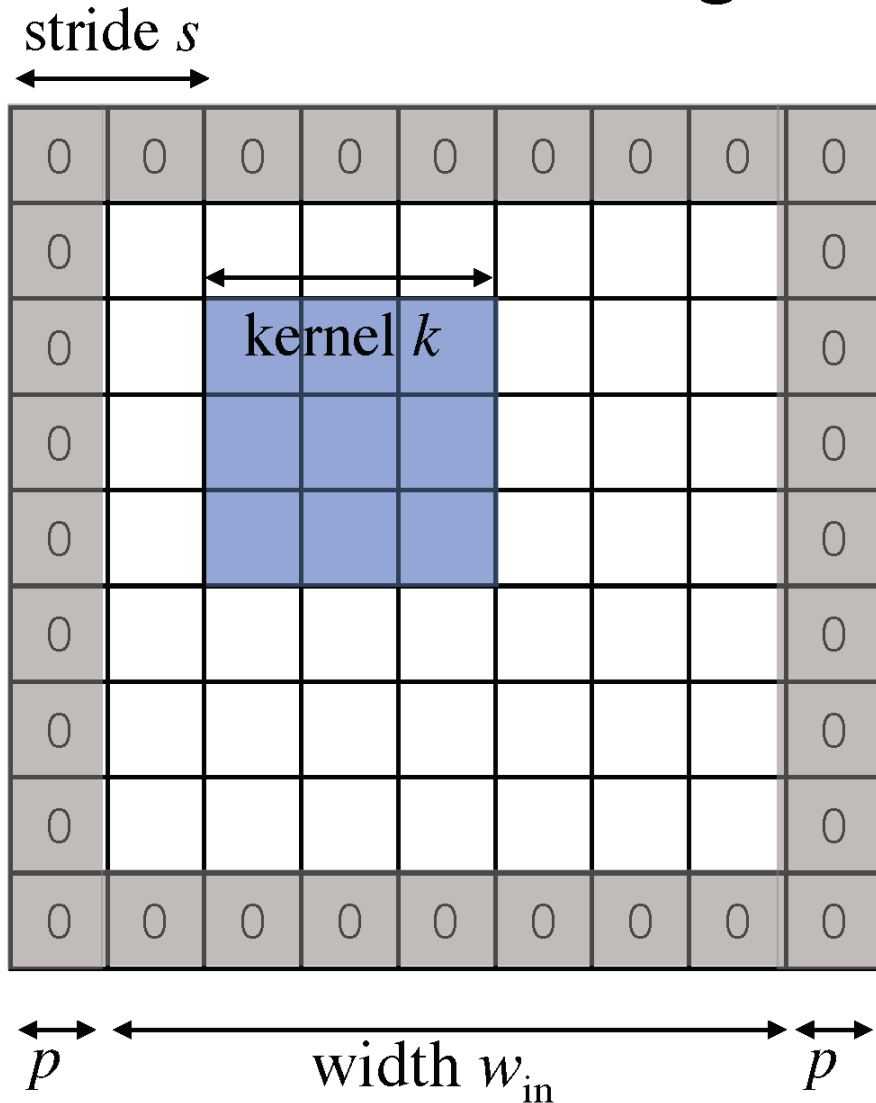


In general, the output has size:

$$w_{out} = \left\lfloor \frac{w_{in} + 2p - k}{s} \right\rfloor + 1$$

Convolution:

How big is the output?



Example: $k=3$, $s=1$, $p=1$

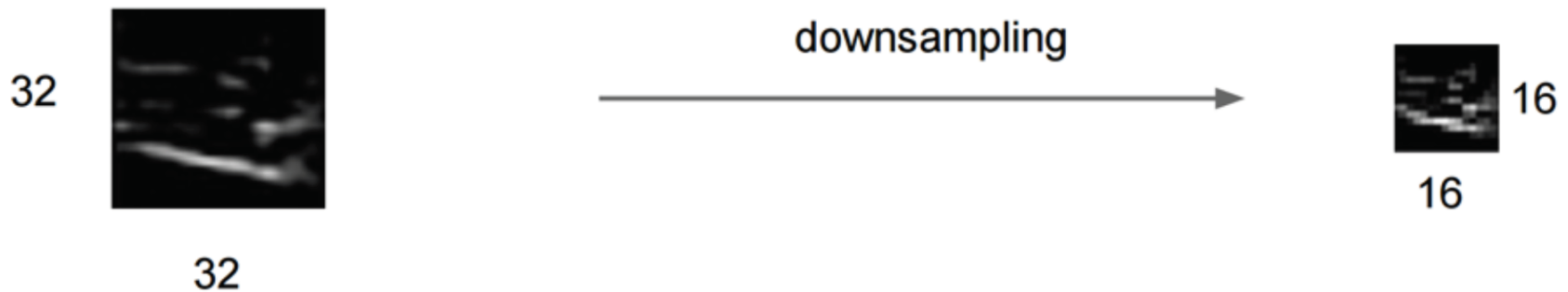
$$\begin{aligned}w_{out} &= \left\lfloor \frac{w_{in} + 2p - k}{s} \right\rfloor + 1 \\ &= \left\lfloor \frac{w_{in} + 2 - 3}{1} \right\rfloor + 1 \\ &= w_{in}\end{aligned}$$

VGGNet [Simonyan 2014]
uses filters of this shape

Pooling

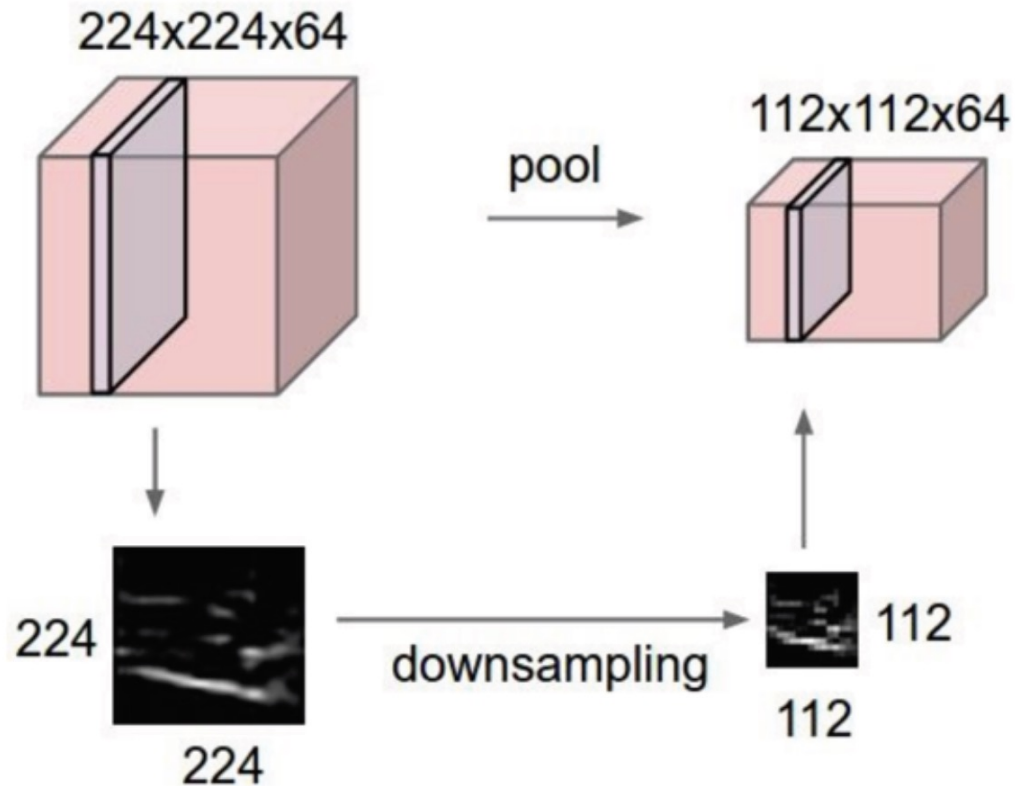
For most ConvNets, **convolution** is often followed by **pooling**:

- Creates a smaller representation while retaining the most important information
- The “max” operation is the most common
- Why might “avg” be a poor choice?

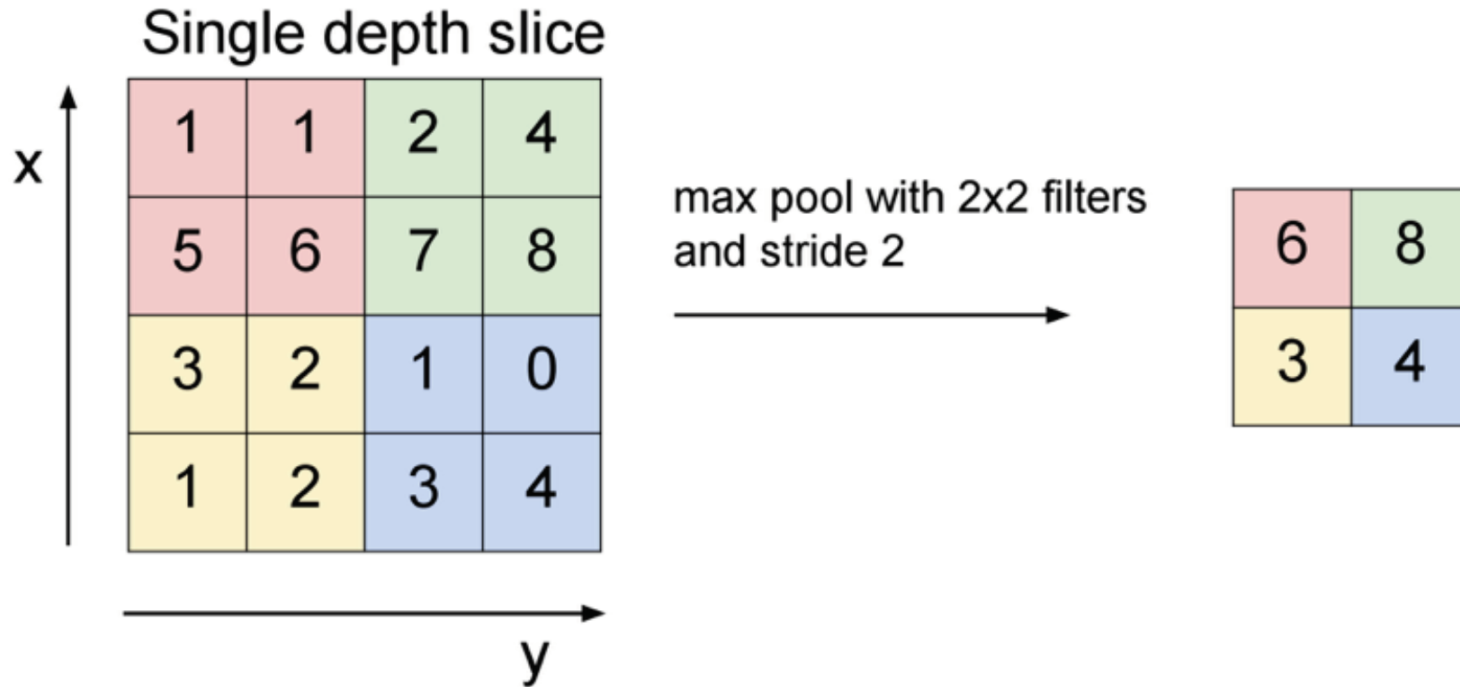


Pooling

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



Max Pooling



What's the backprop rule for max pooling?

- In the forward pass, store the index that took the max
- The backprop gradient is the input gradient at that index

Example ConvNet

CONV CONV POOL
↓ ReLU ↓ ReLU ↓

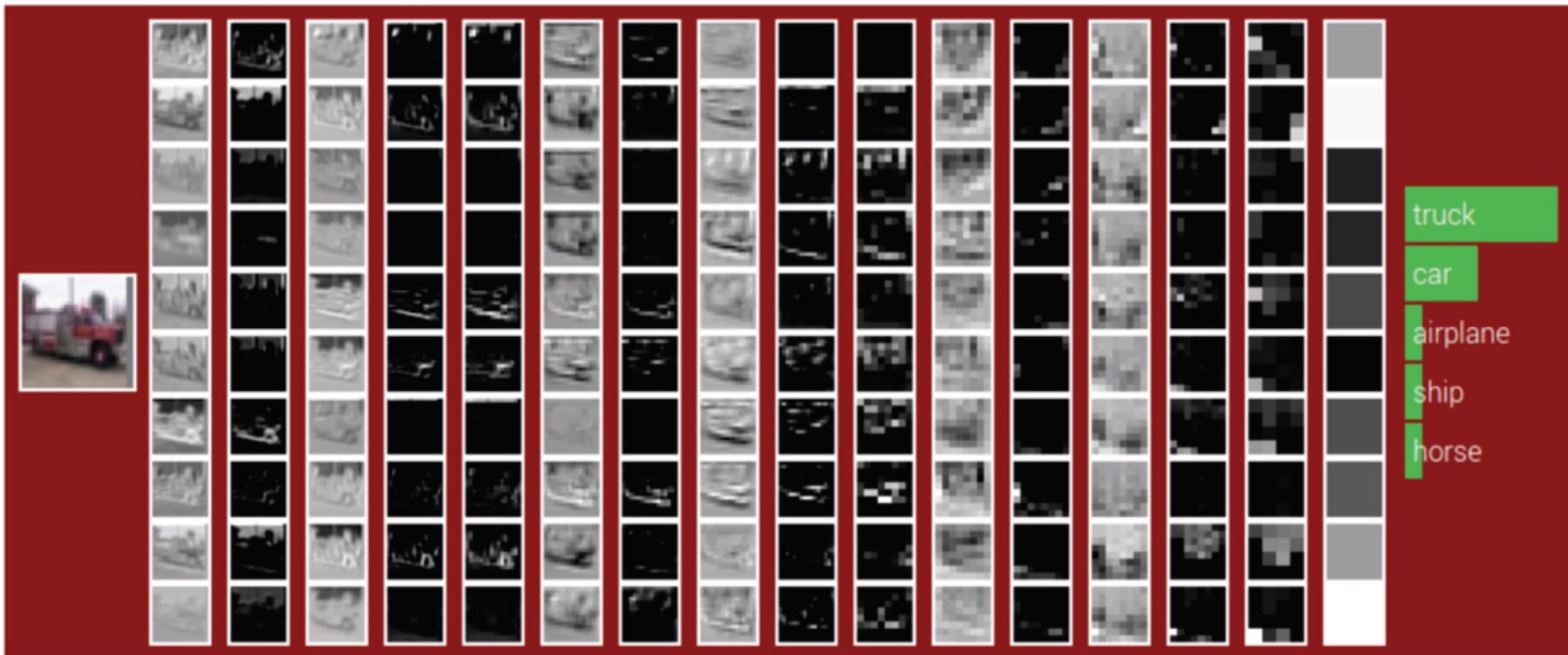


Figure: Andrej Karpathy

Example ConvNet

CONV CONV POOL CONV CONV POOL CONV CONV POOL
↓ ReLU ↓ ReLU ↓ ReLU ↓ ReLU ↓ ReLU ↓ ReLU ↓ ReLU ↓

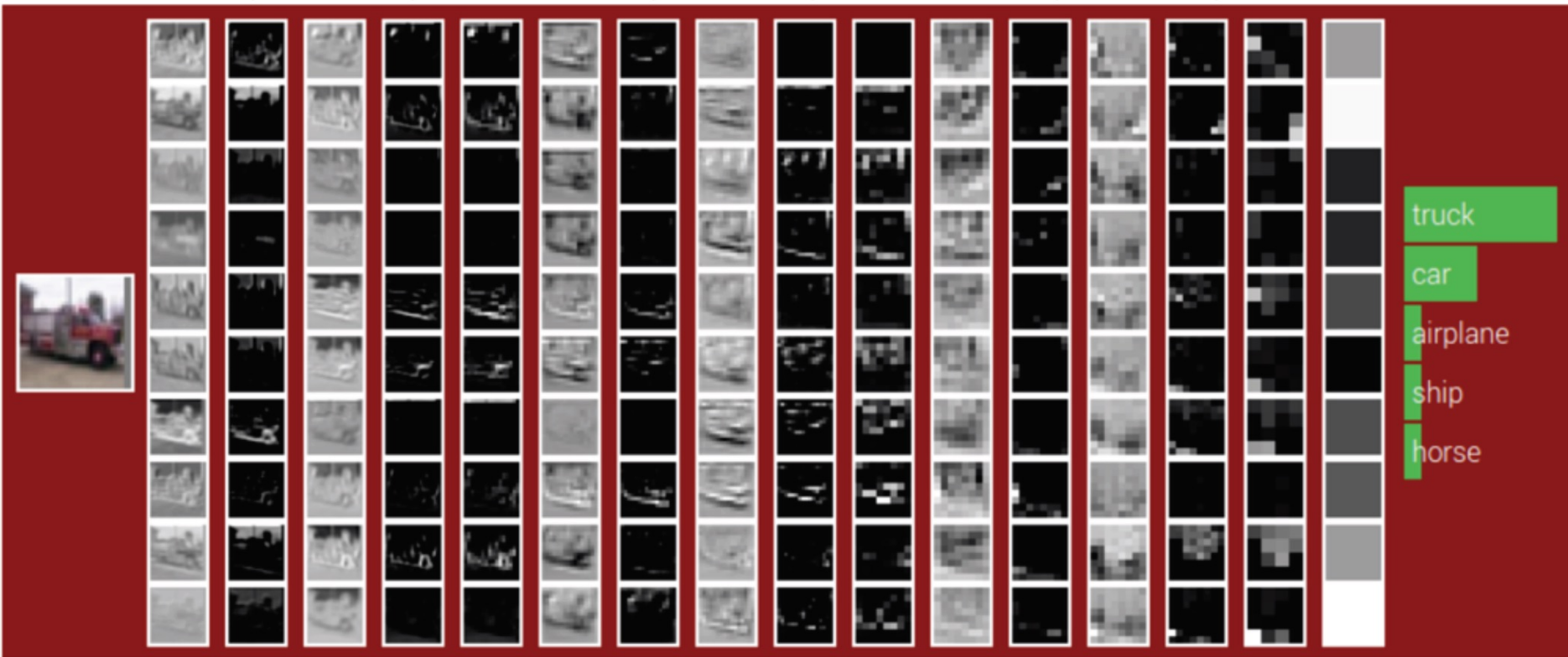


Figure: Andrej Karpathy

Example ConvNet

CONV CONV POOL CONV CONV POOL CONV CONV POOL FC
↓ ReLU ↓ ReLU ↓ ReLU ↓ ReLU ↓ ReLU ↓ ReLU ↓ ReLU ↓ ReLU ↓ (Fully-connected)

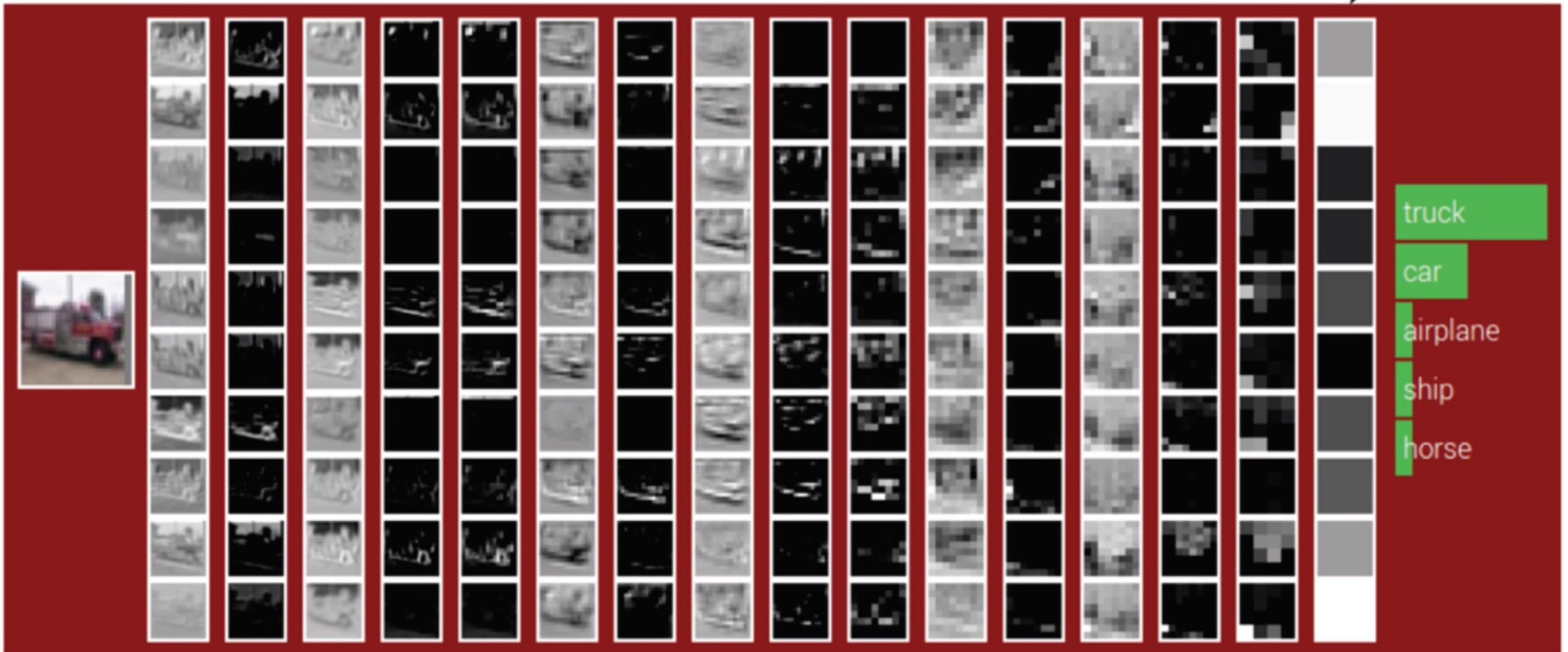
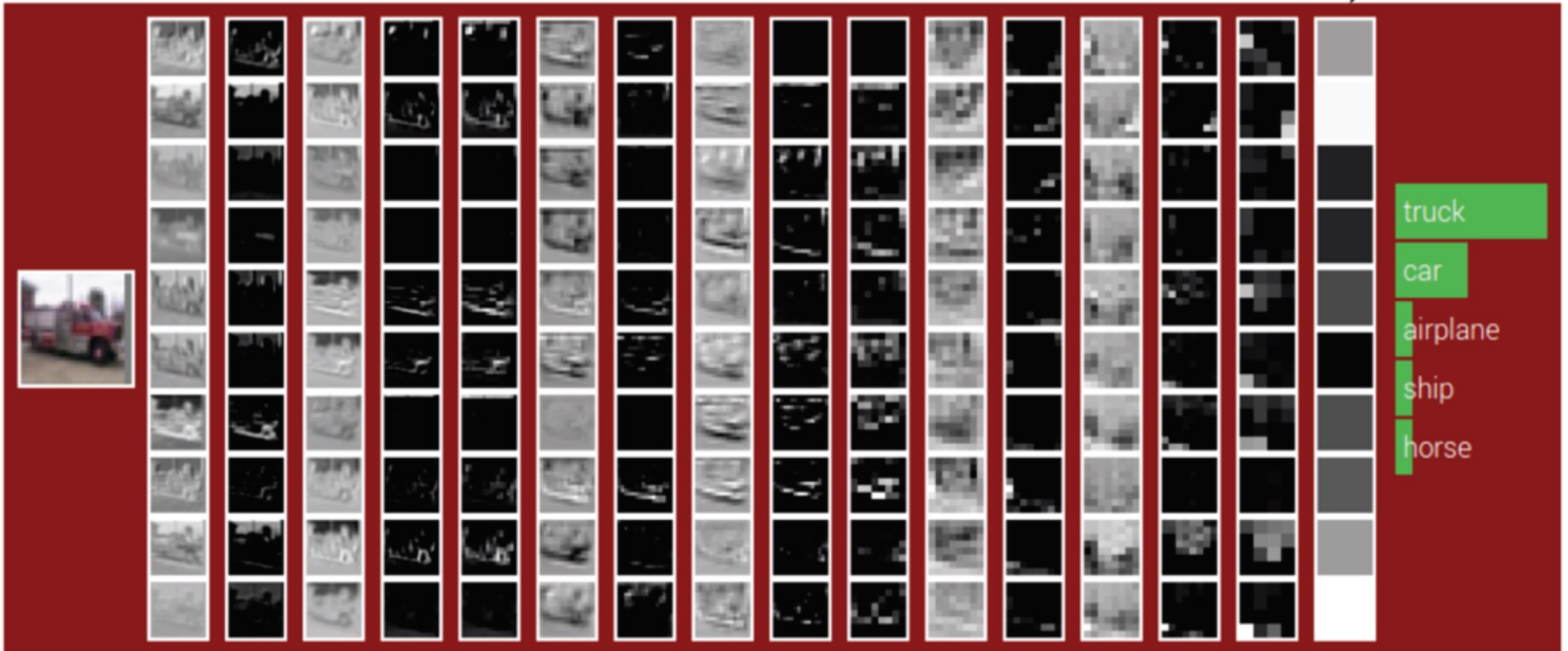


Figure: Andrej Karpathy

Example ConvNet

CONV CONV POOL CONV CONV POOL CONV CONV POOL FC
(Fully-connected)

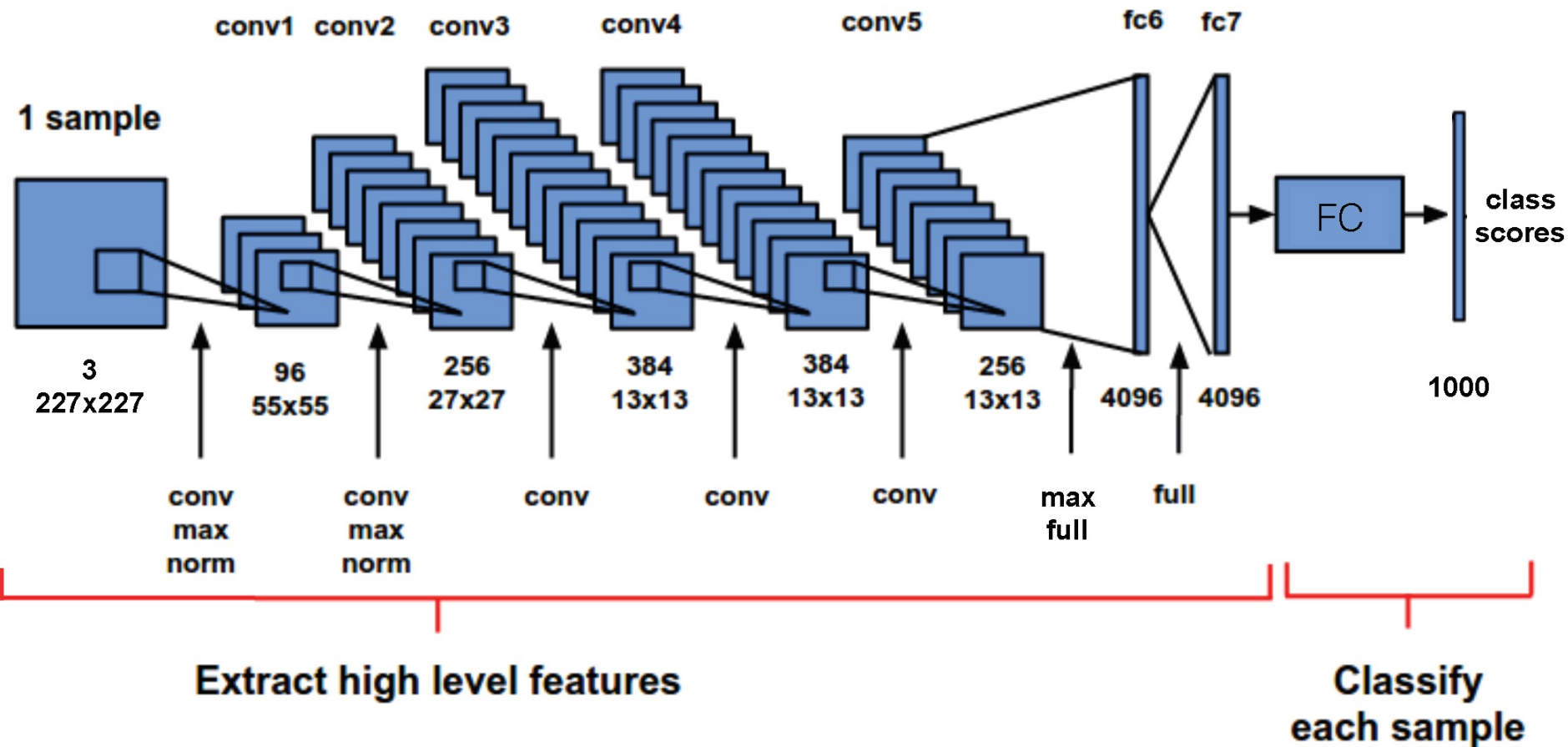
↓ ReLU ↓ ReLU ↓ ReLU ↓ ReLU ↓ ReLU ↓ ReLU ↓ ReLU ↓



10x3x3 conv filters, stride 1, pad 1
2x2 pool filters, stride 2

Figure: Andrej Karpathy

Example: AlexNet [Krizhevsky 2012]



“max”: max pooling

“norm”: local response normalization

“full”: fully connected

Figure: [Karnowski 2015] (with corrections)

Training ConvNets

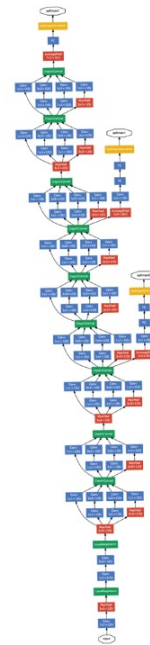
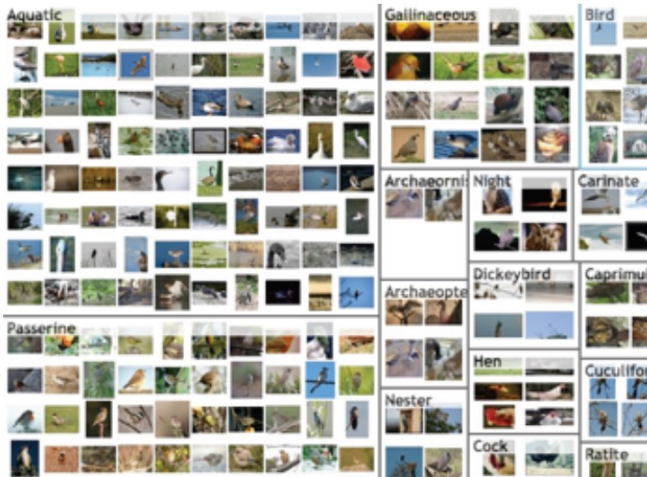
How do you actually train these things?

Roughly speaking:

Gather
labeled data

Find a ConvNet
architecture

Minimize
the loss



Training a convolutional neural network

- Split and preprocess your data
- Choose your network architecture
- Initialize the weights
- Find a learning rate and regularization strength
- Minimize the loss and monitor progress
- Fiddle with knobs

Mini-batch Gradient Descent

Loop:

1. Sample a batch of training data (~100 images)
2. Forwards pass: compute loss (avg. over batch)
3. Backwards pass: compute gradient
4. Update all parameters

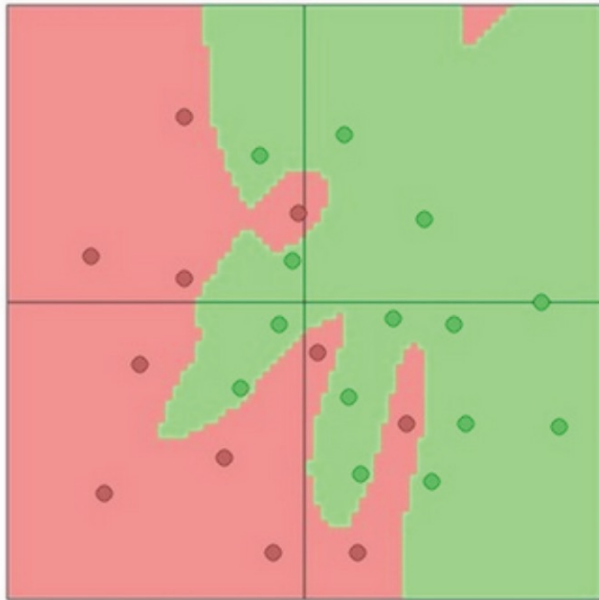
Note: usually called “stochastic gradient descent” even though SGD has a batch size of 1

Regularization

Regularization reduces overfitting:

$$L = L_{\text{data}} + L_{\text{reg}} \quad L_{\text{reg}} = \lambda \frac{1}{2} \|W\|_2^2$$

$\lambda = 0.001$



$\lambda = 0.01$



$\lambda = 0.1$

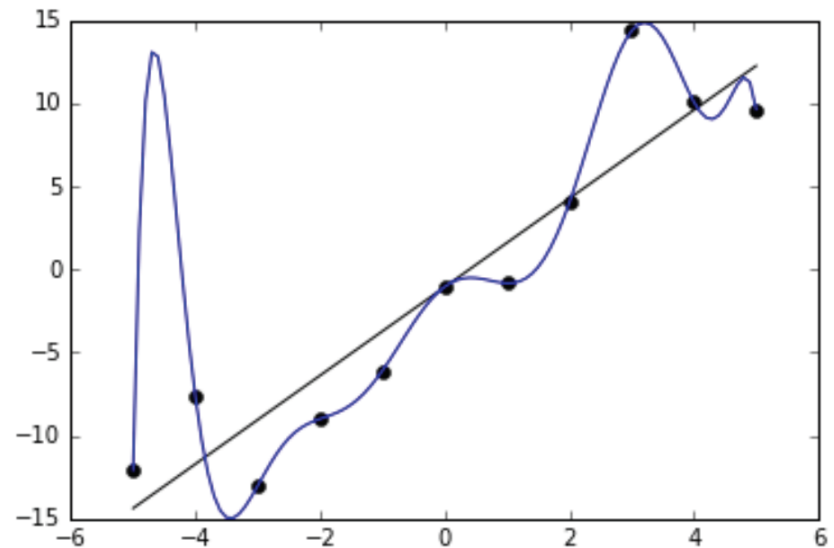


Overfitting

Overfitting: modeling noise in the training set instead of the “true” underlying relationship

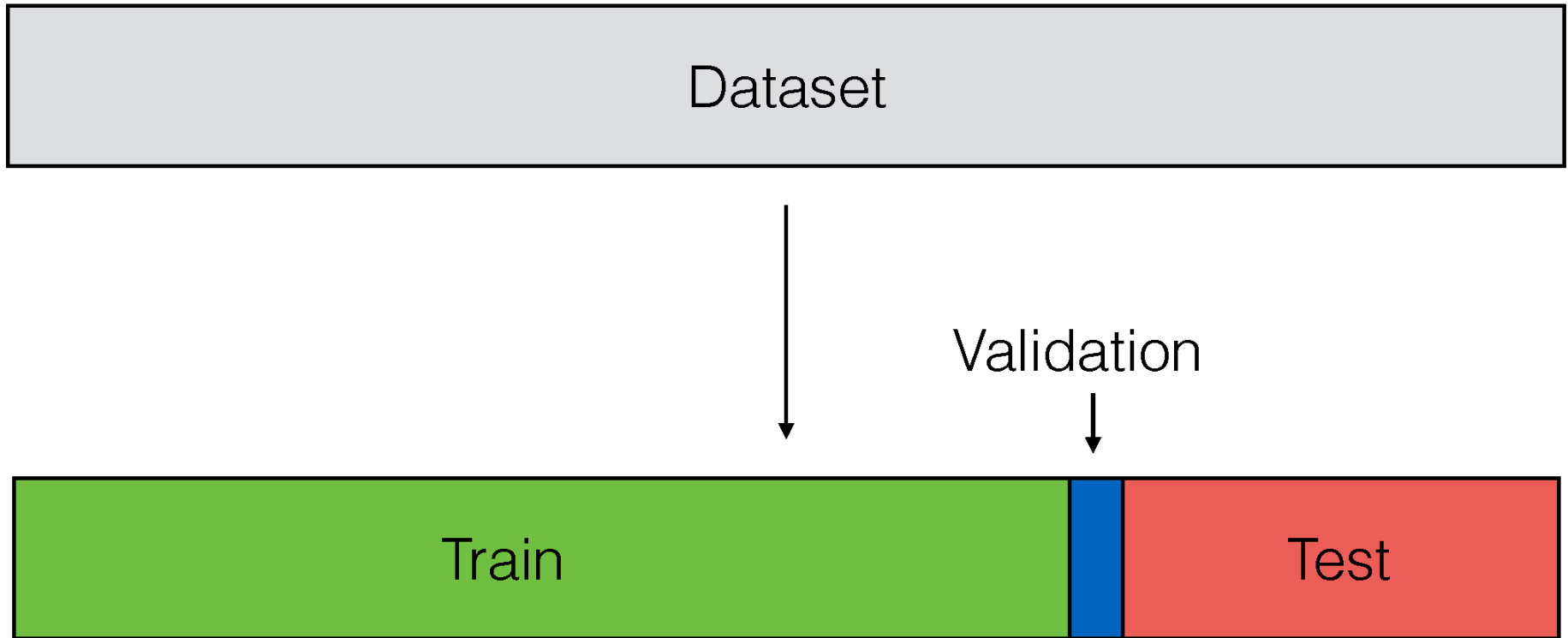
Underfitting: insufficiently modeling the relationship in the training set

General rule: models that are “bigger” or have more capacity are more likely to overfit



(0) Dataset split

Split your data into “train”, “validation”, and “test”:



(0) Dataset split



Train: gradient descent and fine-tuning of parameters

Validation: determining hyper-parameters (learning rate, regularization strength, etc) and picking an architecture

Test: estimate real-world performance
(e.g. accuracy = fraction correctly classified)

(0) Dataset split



Be careful with false discovery:

To avoid false discovery, once we have used a test set once, we should *not use it again* (but nobody follows this rule, since it's expensive to collect datasets)

Instead, try and avoid looking at the test score until the end

(1) Data preprocessing

Preprocess the data so that learning is better conditioned:

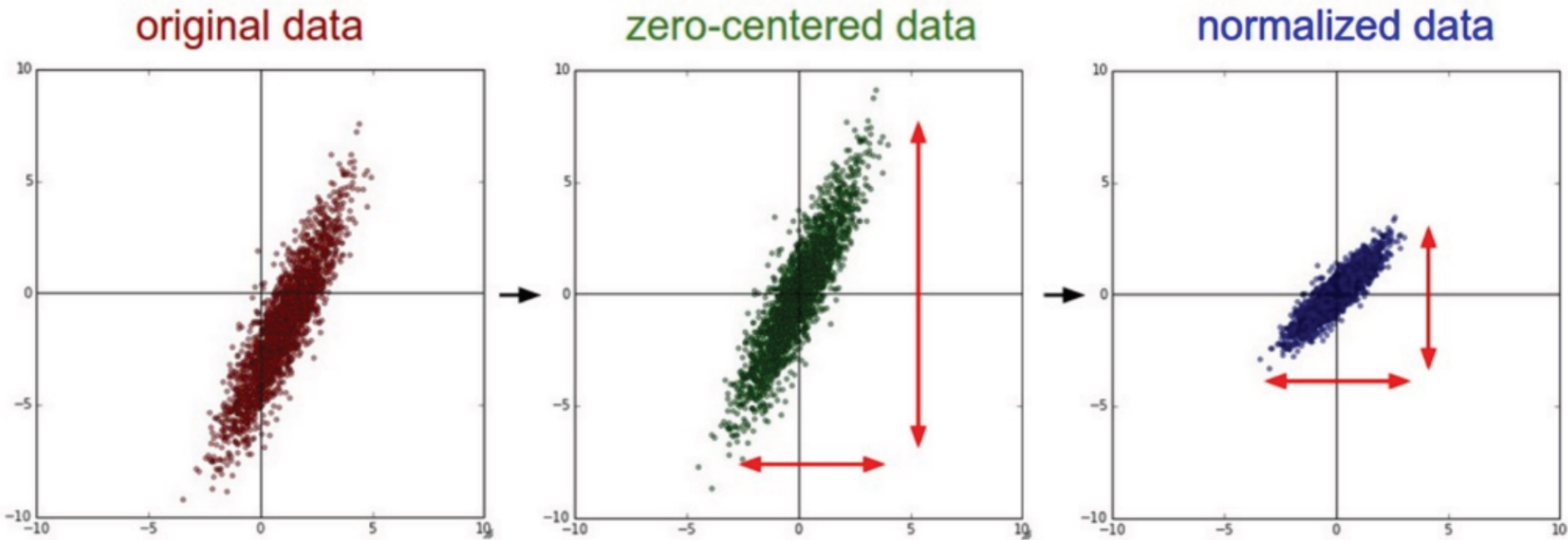
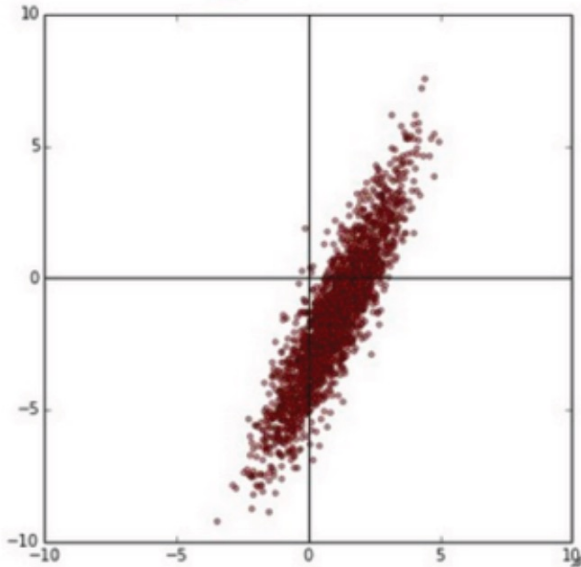


Figure: Andrej Karpathy

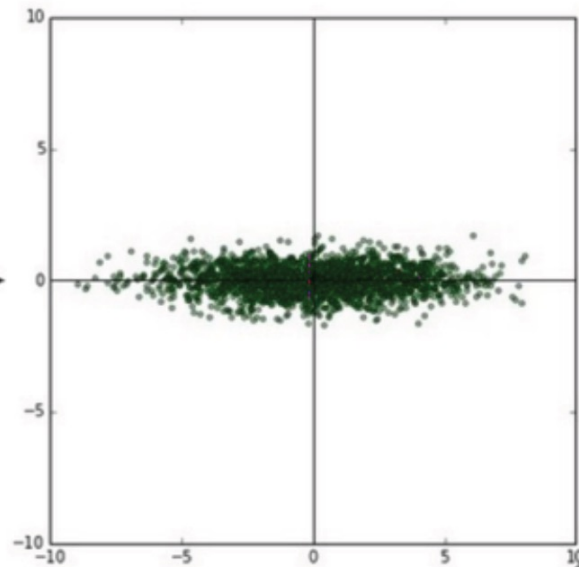
(1) Data preprocessing

In practice, you may also see **PCA** and **Whitening** of the data:

original data

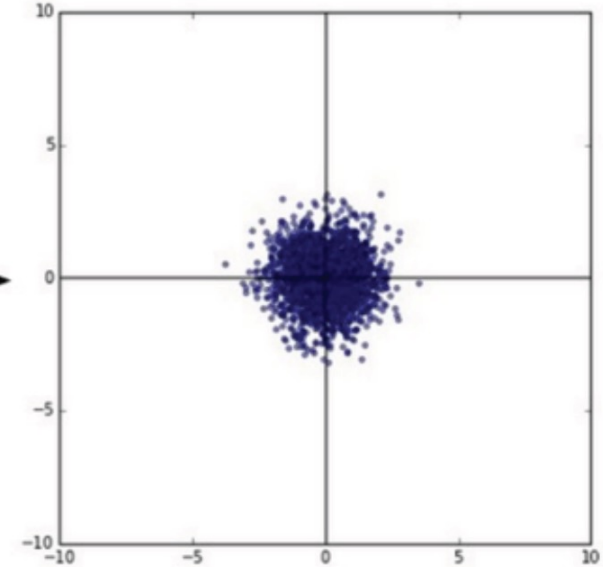


decorrelated data



(data has diagonal covariance matrix)

whitened data



(covariance matrix is the identity matrix)

(1) Data preprocessing

For ConvNets, typically only the mean is subtracted.



An input image (256x256)



Minus sign

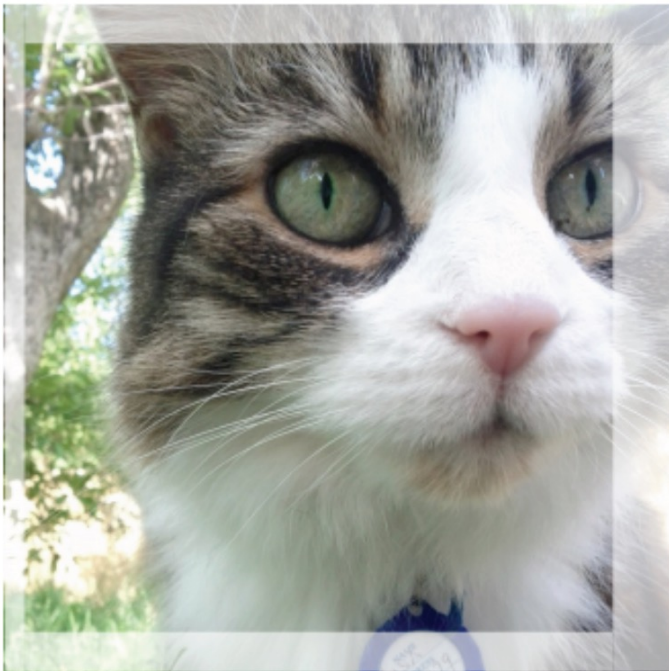


The mean input image

A per-channel mean also works (one value per R,G,B).

(1) Data preprocessing

Augment the data — extract random crops from the input, with slightly jittered offsets. Without this, typical ConvNets (e.g. [Krizhevsky 2012]) overfit the data.



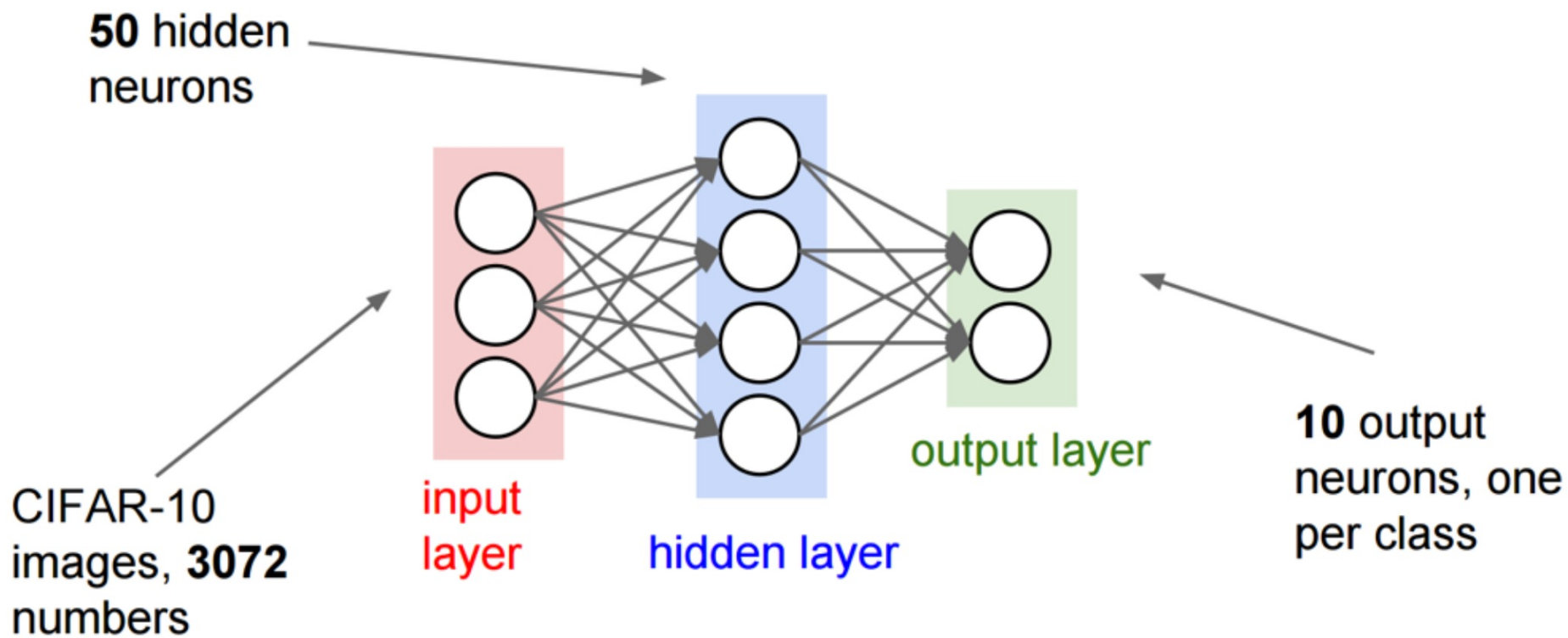
E.g. 224x224 patches
extracted from 256x256 images

Randomly reflect horizontally

Perform the augmentation live
during training

(2) Choose your architecture

Toy example: one hidden layer of size 50



(3) Initialize your weights

Set the weights to small random numbers:

```
W = np.random.randn(D, H) * 0.001
```

(matrix of small random numbers drawn from a Gaussian distribution)

(the magnitude is important and this is not optimal — more on this later)

Set the bias to zero (or small nonzero):

```
b = np.zeros(H)
```

(3) Check that the loss is reasonable

```
def init_two_layer_model(input_size, hidden_size, output_size):  
    # initialize a model  
    model = {}  
    model['W1'] = 0.0001 * np.random.randn(input_size, hidden_size)  
    model['b1'] = np.zeros(hidden_size)  
    model['W2'] = 0.0001 * np.random.randn(hidden_size, output_size)  
    model['b2'] = np.zeros(output_size)  
    return model
```

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes  
loss, grad = two_layer_net(X_train, model, y_train, 0.0) # disable regularization  
print loss
```

returns the loss and the
gradient for all parameters

(3) Check that the loss is reasonable

```
def init_two_layer_model(input_size, hidden_size, output_size):  
    # initialize a model  
    model = {}  
    model['W1'] = 0.0001 * np.random.randn(input_size, hidden_size)  
    model['b1'] = np.zeros(hidden_size)  
    model['W2'] = 0.0001 * np.random.randn(hidden_size, output_size)  
    model['b2'] = np.zeros(output_size)  
    return model
```

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes  
loss, grad = two_layer_net(X_train, model, y_train, 1e3) # crank up regularization  
print loss
```

 loss went up, good. (sanity check)

(4) Overfit a small portion of the data

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
X_tiny = X_train[:20] # take 20 examples ←
y_tiny = y_train[:20]
best_model, stats = trainer.train(X_tiny, y_tiny, X_tiny, y_tiny,
                                  model, two_layer_net,
                                  num_epochs=200, reg=0.0,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = False,
                                  learning_rate=1e-3, verbose=True)
```

Details:

'sgd': vanilla gradient descent (no momentum etc)

learning_rate_decay = 1: constant learning rate

sample_batches = False (full gradient descent, no batches)

epochs = 200: number of passes through the data

(4) Find a learning rate

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=1e-6, verbose=True)
```

```
Finished epoch 1 / 10: cost 2.302576, train: 0.080000, val 0.103000, lr 1.000000e-06
Finished epoch 2 / 10: cost 2.302582, train: 0.121000, val 0.124000, lr 1.000000e-06
Finished epoch 3 / 10: cost 2.302558, train: 0.119000, val 0.138000, lr 1.000000e-06
Finished epoch 4 / 10: cost 2.302519, train: 0.127000, val 0.151000, lr 1.000000e-06
Finished epoch 5 / 10: cost 2.302517, train: 0.158000, val 0.171000, lr 1.000000e-06
Finished epoch 6 / 10: cost 2.302518, train: 0.179000, val 0.172000, lr 1.000000e-06
Finished epoch 7 / 10: cost 2.302466, train: 0.180000, val 0.176000, lr 1.000000e-06
Finished epoch 8 / 10: cost 2.302452, train: 0.175000, val 0.185000, lr 1.000000e-06
Finished epoch 9 / 10: cost 2.302459, train: 0.206000, val 0.192000, lr 1.000000e-06
Finished epoch 10 / 10: cost 2.302420, train: 0.190000, val 0.192000, lr 1.000000e-06
finished optimization. best validation accuracy: 0.192000
```

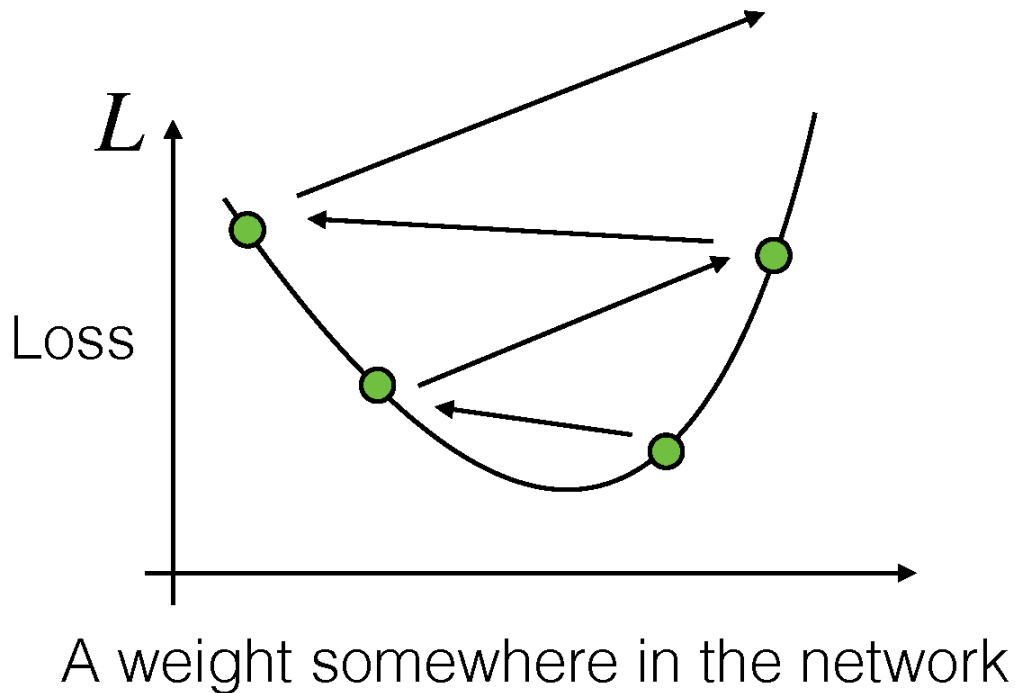
Loss barely changes

Why is the accuracy 20%?

(learning rate is too low or regularization too high)

(4) Find a learning rate

Learning rate: $1e6$ — what could go wrong?



(4) Find a learning rate

Coarse to fine search

First stage: only a few epochs (passes through the data) to get a rough idea

Second stage: longer running time, finer search

Tip: if loss $> 3 * \text{original loss}$, quit early
(learning rate too high)

(4) Find a learning rate

Normally, you don't have the budget for lots of cross-validation —> visualize as you go

Plot the loss

For very small learning rates, the loss decreases linearly and slowly

(Why linearly?)

Larger learning rates tend to look more exponential

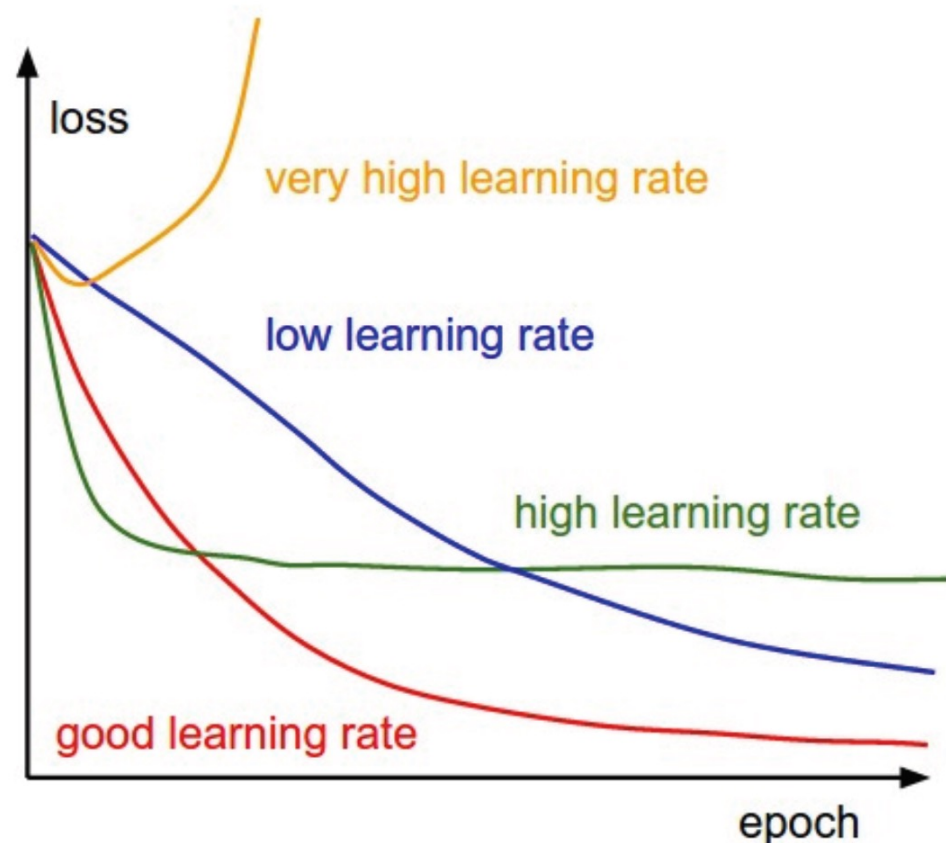


Figure: Andrej Karpathy

(4) Find a learning rate

Normally, you don't have the budget for lots of cross-validation —> visualize as you go

Typical training loss:

Why is it varying so rapidly?

The width of the curve is related to the batchsize — if too noisy, increase the batch size

Possibly too linear
(learning rate too small)

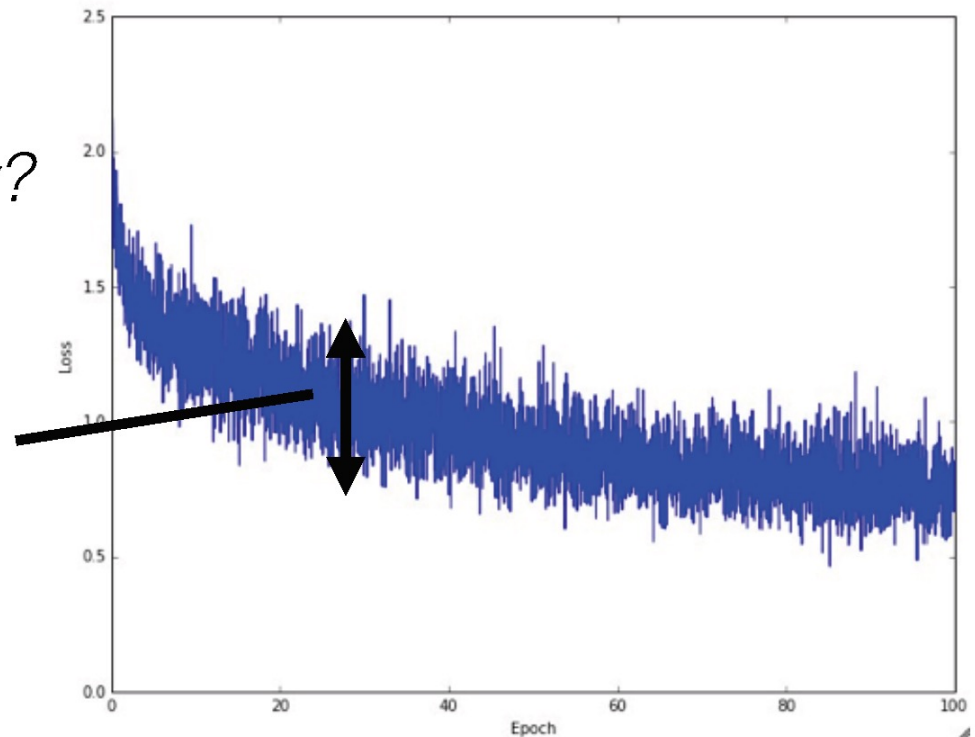
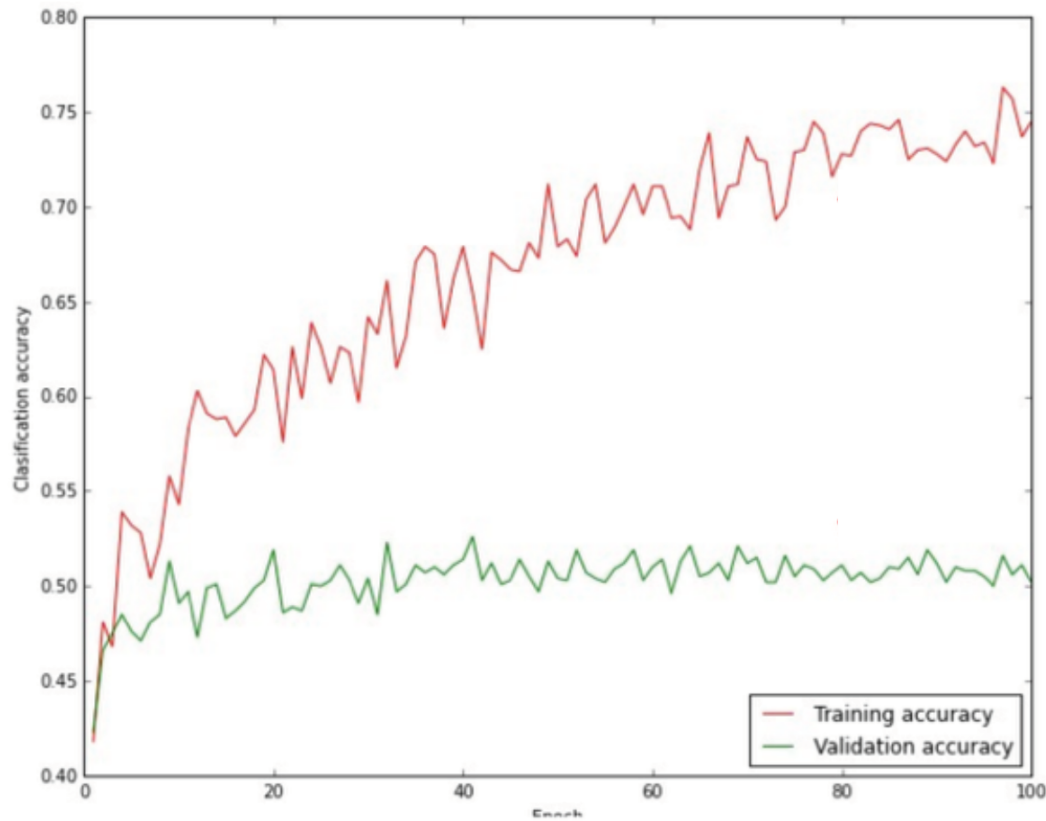


Figure: Andrej Karpathy

(4) Find a learning rate

Visualize the accuracy



Big gap: overfitting
(increase regularization)

No gap: underfitting
(increase model capacity,
make layers bigger
or decrease regularization)

(4) Find a learning rate

Visualize the weights

Noisy weights: possibly regularization not strong enough

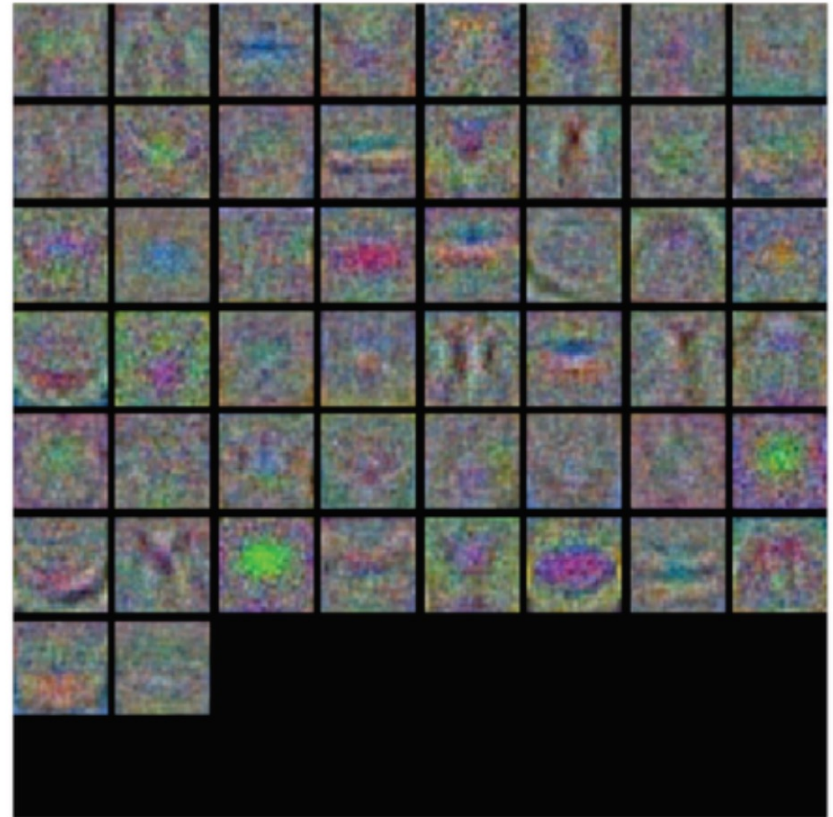
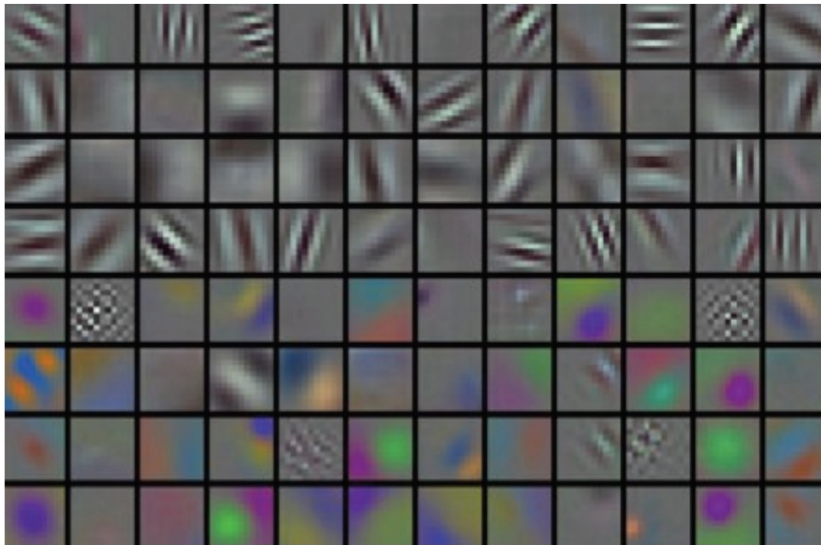


Figure: Andrej Karpathy

(4) Find a learning rate

Visualize the weights



Nice clean weights:
training is proceeding well

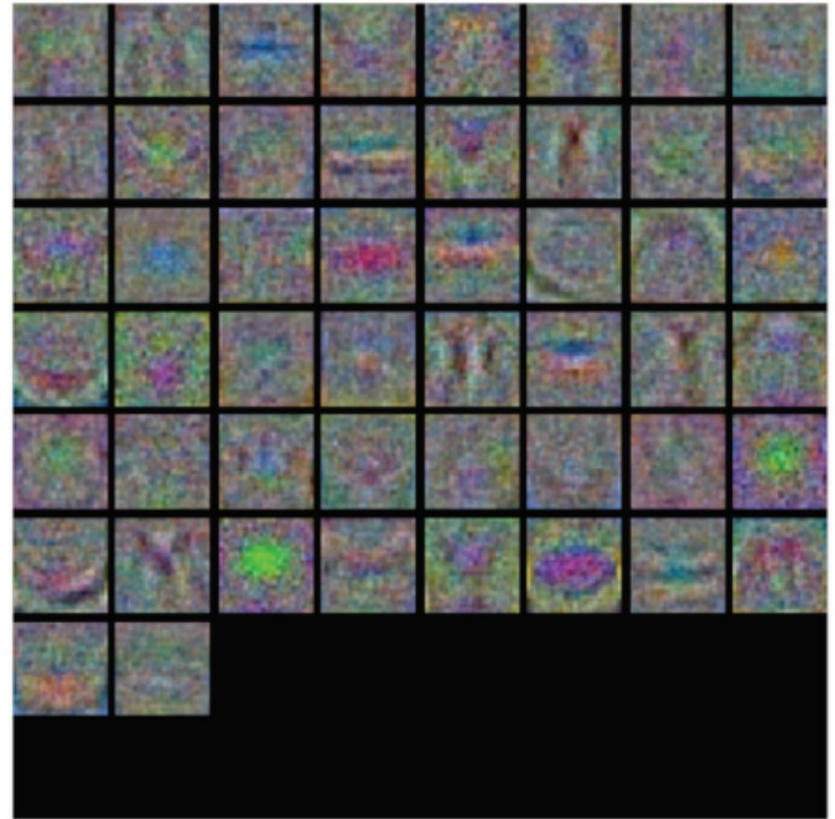


Figure: Alex Krizhevsky , Andrej Karpathy

Learning rate schedule

How do we change the learning rate over time?

Various choices:

- Step down by a factor of 0.1 every 50,000 mini-batches (used by SuperVision [Krizhevsky 2012])
- Decrease by a factor of 0.97 every epoch (used by GoogLeNet [Szegedy 2014])
- Scale by $\sqrt{1-t/\text{max_t}}$ (used by BVLC to re-implement GoogLeNet)
- Scale by $1/t$
- Scale by $\exp(-t)$

Summary of things to fiddle

- Network architecture
- Learning rate, decay schedule, update type
- Regularization (L2, L1, maxnorm, dropout, ...)
- Loss function (softmax, SVM, ...)
- Weight initialization

Neural network
parameters



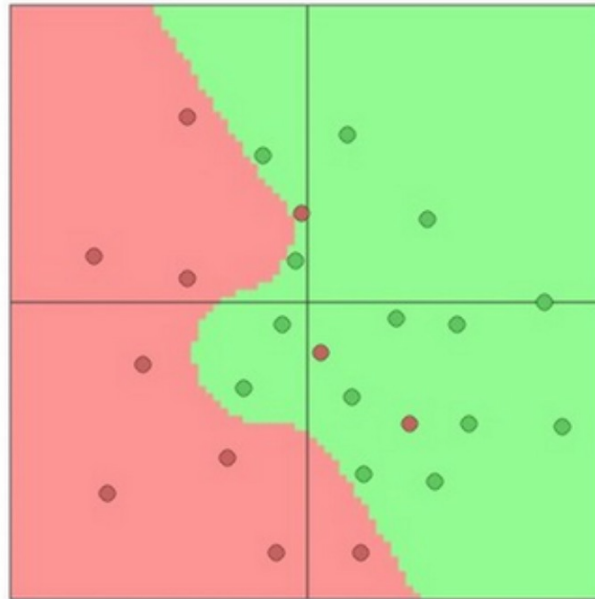
(Recall) Regularization reduces overfitting

$$L = L_{\text{data}} + L_{\text{reg}} \quad L_{\text{reg}} = \lambda \frac{1}{2} \|W\|_2^2$$

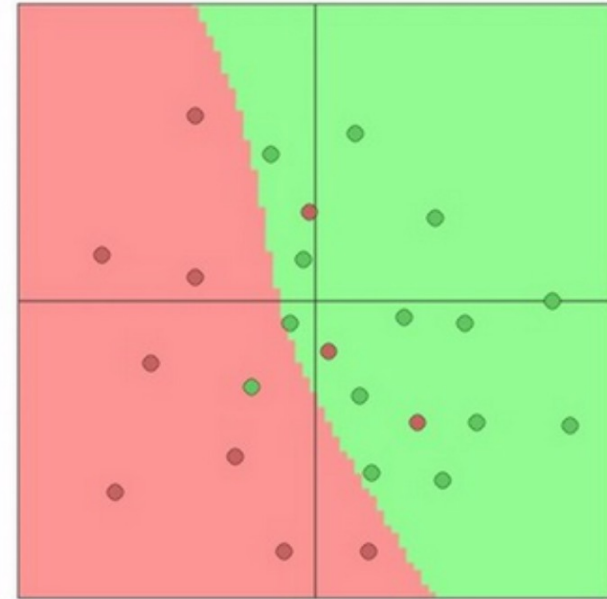
$\lambda = 0.001$



$\lambda = 0.01$



$\lambda = 0.1$



Example Regularizers

L2 regularization

$$L_{\text{reg}} = \lambda \frac{1}{2} \|W\|_2^2$$

(L2 regularization encourages small weights)

L1 regularization

$$L_{\text{reg}} = \lambda \|W\|_1 = \lambda \sum_{ij} |w_{ij}|$$

(L1 regularization encourages sparse weights: weights are encouraged to reduce to exactly zero)

“Elastic net”

$$L_{\text{reg}} = \lambda_1 \|W\|_1 + \lambda_2 \|W\|_2^2$$

(combine L1 and L2 regularization)

Max norm

Clamp weights to some max norm

$$\|W\|_2^2 \leq c$$

“Weight decay”

Regularization is also called “weight decay” because the weights “decay” each iteration:

$$L_{\text{reg}} = \lambda \frac{1}{2} \|W\|_2^2 \quad \longrightarrow \quad \frac{\partial L}{\partial W} = \lambda W$$

Gradient descent step:

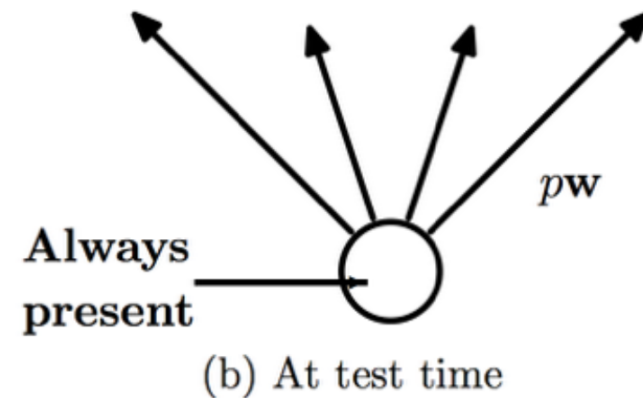
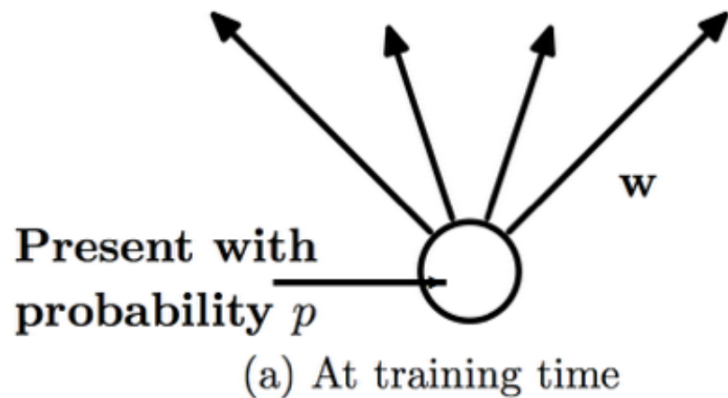
$$W \leftarrow W - \alpha \lambda W - \frac{\partial L_{\text{data}}}{\partial W}$$

Weight decay: $\alpha \lambda$ (weights always decay by this amount)

Note: biases are sometimes excluded from regularization

Dropout

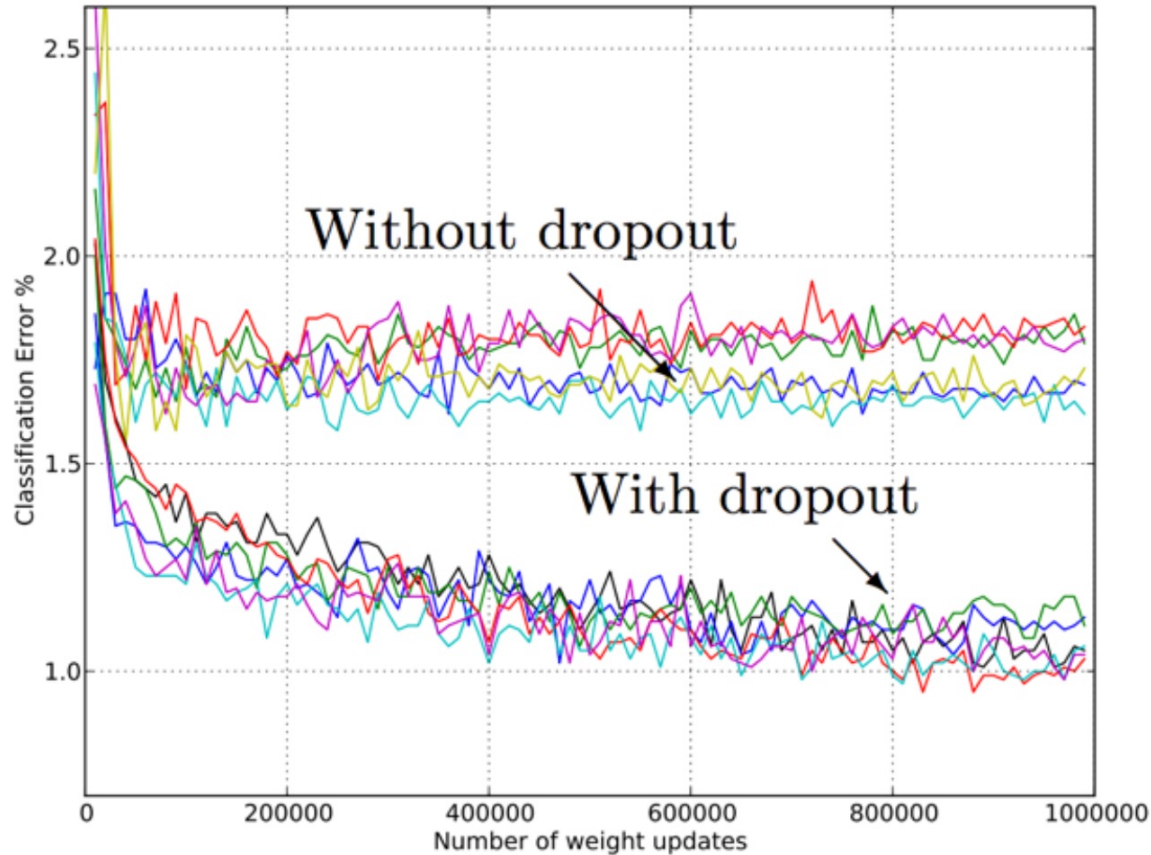
Simple but powerful technique to reduce overfitting:



[Srivasta et al, “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”, JMLR 2014]

Dropout

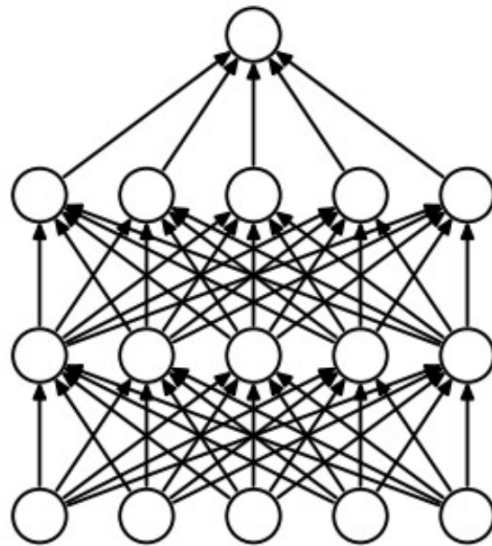
Simple but powerful technique to reduce overfitting:



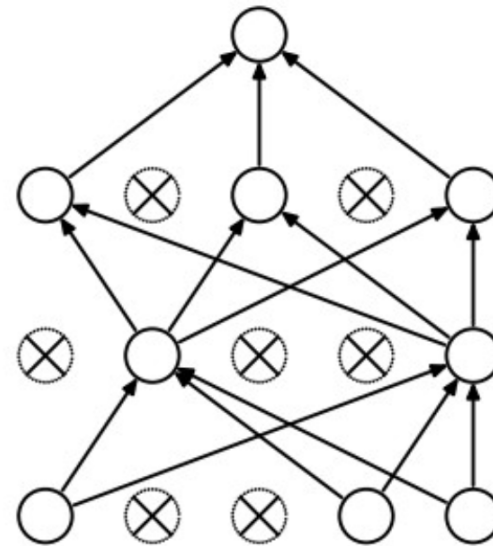
[Srivasta et al, "Dropout: A Simple Way to Prevent Neural Networks from Overfitting", JMLR 2014]

Dropout

Simple but powerful technique to reduce overfitting:



(a) Standard Neural Net



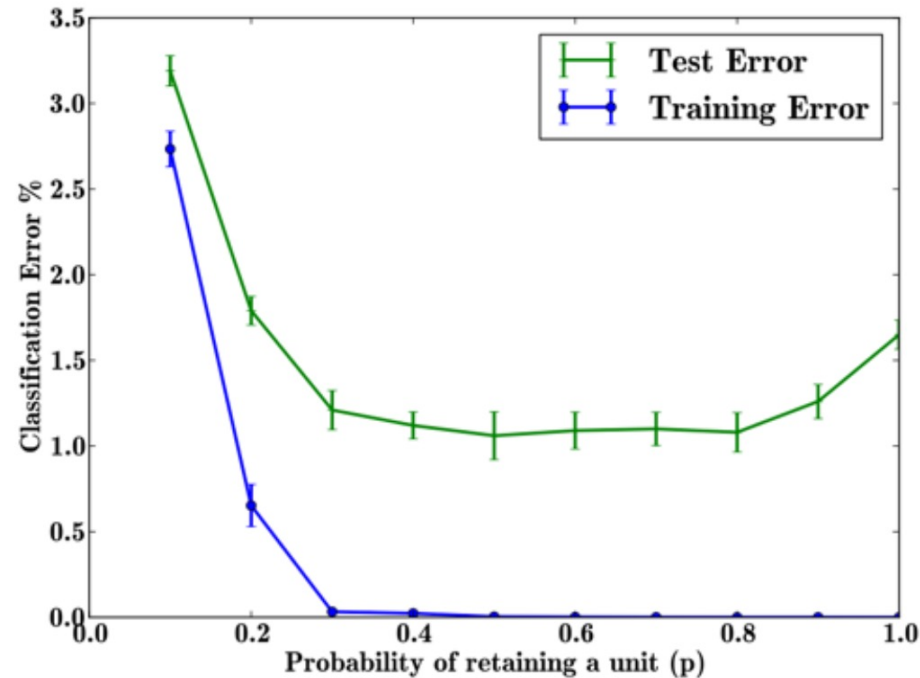
(b) After applying dropout.

Note: Dropout can be interpreted as an approximation to taking the geometric mean of an ensemble of exponentially many models

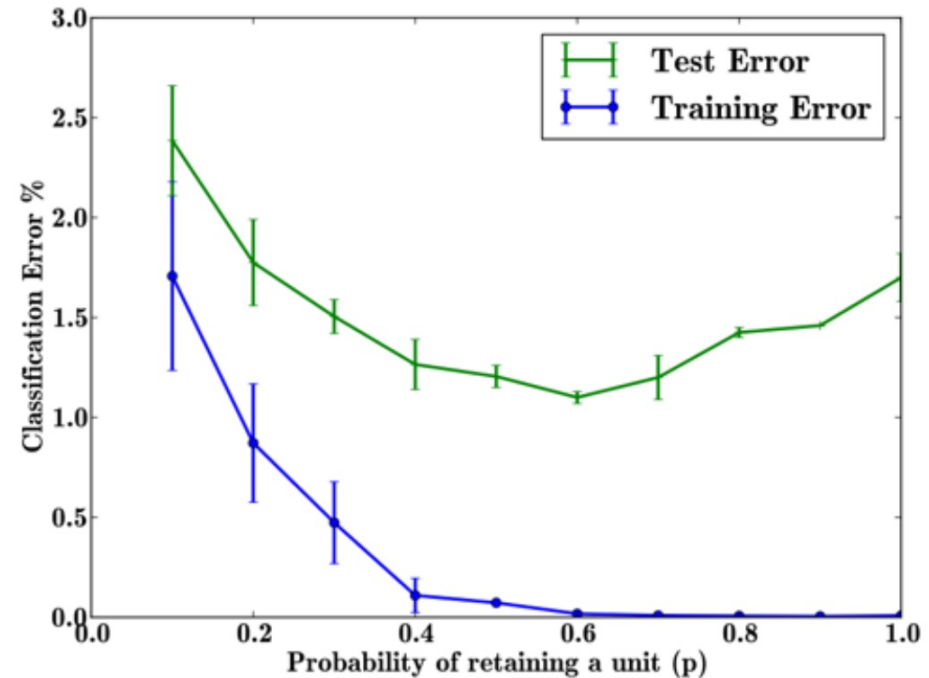
[Srivasta et al, “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”, JMLR 2014]

Dropout

How much dropout? Around $p = 0.5$



(a) Keeping n fixed.



(b) Keeping pn fixed.

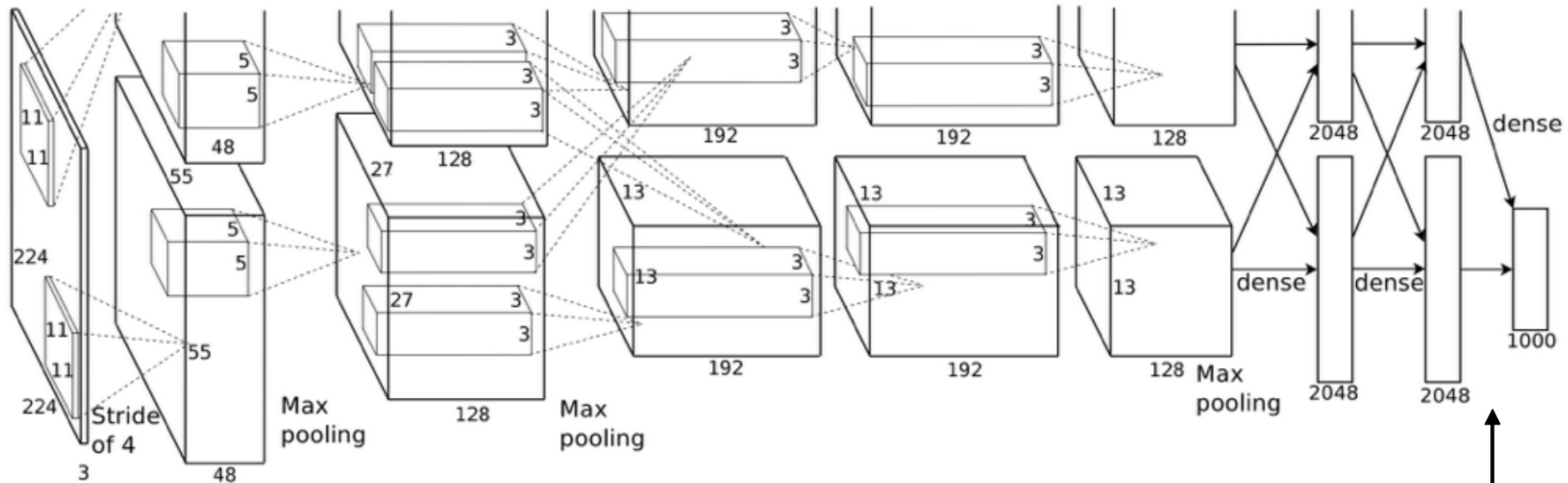
[Srivasta et al, "Dropout: A Simple Way to Prevent Neural Networks from Overfitting", JMLR 2014]

Dropout

Case study: [Krizhevsky 2012]

“Without dropout, our network exhibits substantial overfitting.”

Dropout here



But not here — why?

[Krizhevsky et al, “ImageNet Classification with Deep Convolutional Neural Networks”, NIPS 2012]

Dropout

```
p = 0.5 # probability of keeping a unit active. higher = less dropout
```

```
def train_step(X):
```

```
    """ X contains the data """
```

```
    # forward pass for example 3-layer neural network
```

```
    H1 = np.maximum(0, np.dot(W1, X) + b1)
```

```
    U1 = np.random.rand(*H1.shape) < p # first dropout mask
```

```
    H1 *= U1 # drop!
```

```
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
```

```
    U2 = np.random.rand(*H2.shape) < p # second dropout mask
```

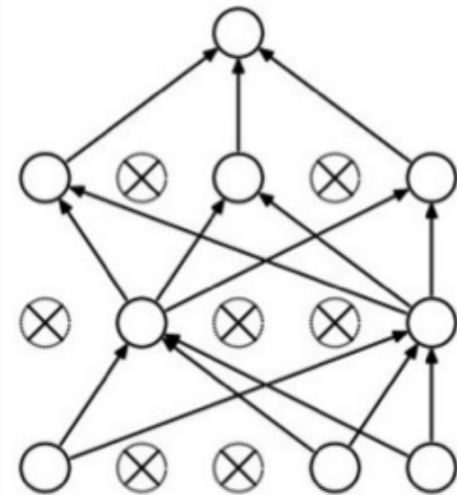
```
    H2 *= U2 # drop!
```

```
    out = np.dot(W3, H2) + b3
```

```
    # backward pass: compute gradients... (not shown)
```

```
    # perform parameter update... (not shown)
```

Example forward pass with a 3-layer network using dropout



(note, here X is a single input)

Dropout

Test time: scale the activations

Expected value of a neuron h with dropout:

$$E[h] = ph + (1 - p)0 = ph$$

```
def predict(X):  
    # ensembled forward pass  
    H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations  
    H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations  
    out = np.dot(W3, H2) + b3
```

We want to keep the same expected value

Summary

- Preprocess the data (subtract mean, sub-crops)
- Initialize weights carefully
- Use Dropout
- Use SGD + Momentum
- Fine-tune from ImageNet
- Babysit the network as it trains