# Convolutional neural networks

Course announcements

- Homework 5 is due tonight.
  - Any questions about the homework?
  - How many of you have looked at/started/finished homework 5?

- Homework 6 will be posted tonight and will be due April 22nd.

- Take-home quiz 9 has been posted and is due April 12th.

Overview of today's lecture

- Some notes on optimization.

- Convolutional neural networks.

- Training ConvNets.

Slide credits

Most of these slides were adapted from:
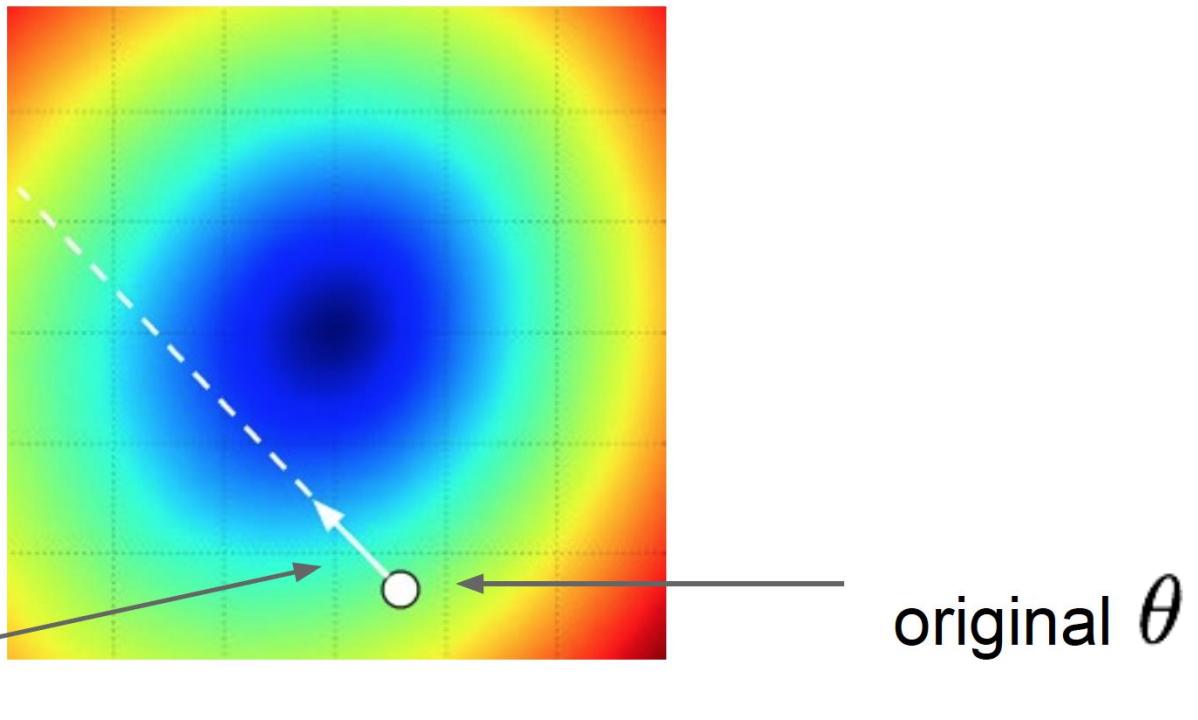
- Noah Snavely (Cornell University).

- Fei-Fei Li (Stanford University).

- Andrej Karpathy (Stanford University).

Some notes on optimization

Summary

- Always use mini-batch gradient descent

- Incorrectly refer to it as "doing SGD" as everyone else

  (or call it batch gradient descent)

- The mini-batch size is a hyperparameter, but it is not very common to cross-validate over it (usually based on practical concerns, e.g. space/time efficiency)

Learning rates

original θ

negative gradient direction

$$\theta \leftarrow \theta - \eta \frac{\partial \mathcal{L}}{\partial \theta}$$

Step size: learning rate
Too big: will miss the minimum
Too small: slow convergence

Learning rate scheduling

- Use different learning rate at each iteration.

- Most common choice:

$$\eta_t = \frac{\eta_0}{\sqrt{t}}$$

Need to select initial learning rate η_0, important!!!

- More modern choice: Adaptive learning rates.
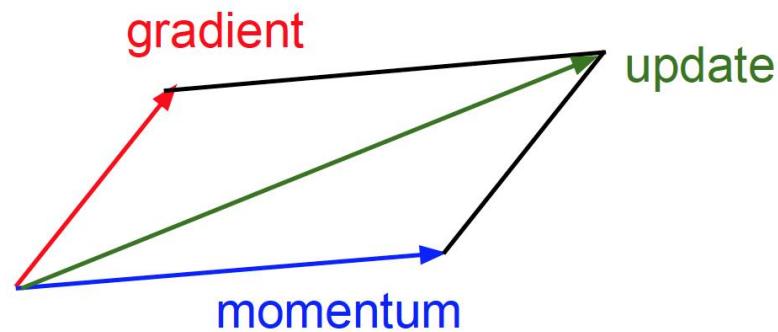
$$\eta_t = G\left(\left\{\frac{\partial L}{\partial \theta}\right\}_{i=0}^{t}\right)$$

Many choices for G (Adam, Adagrad, Adadelta).

Momentum Update

$$\theta \leftarrow \theta - \eta \frac{\partial \mathcal{L}}{\partial \theta}$$

$$\Delta\theta \leftarrow w \frac{\partial L}{\partial \theta} + (1 - w)\Delta\theta$$

gradient

update

momentum
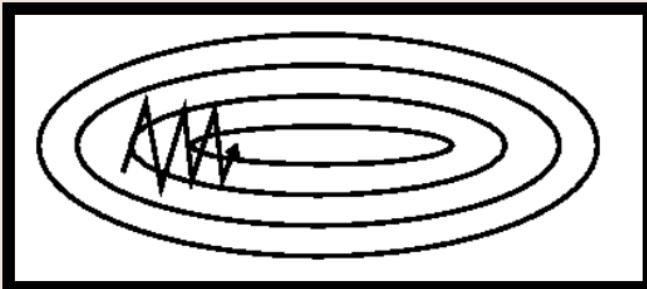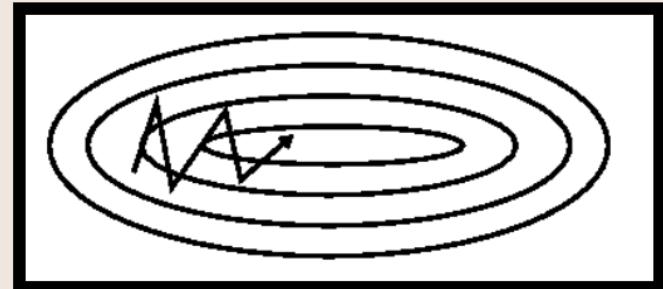
Take direction history into account!

```
weights_grad = evaluate_gradient(loss_fun, data, weights)
vel = vel * 0.9 - step_size * weights_grad
weights += vel
```

(Fig. 2a)

(Fig. 2b)

Many other ways to perform optimization…

- Second order methods that use the Hessian (or its approximation): BFGS, **LBFGS**, etc.

- Currently, the lesson from the trenches is that well-tuned SGD+Momentum is very hard to beat for CNNs.

- No consensus on Adam etc.: Seem to give faster performance to worse local minima.

Derivatives

- Given f(x), where x is vector of inputs
  - Compute gradient of f at x: $\nabla f(x)$

  How do we do differentiation?

Numerical differentiation

Numerical differentiation

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

$$f(x+h) = f(x) + h\frac{df(x)}{dx}$$

Numerical differentiation

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

$$f(x+h) = f(x) + h\frac{df(x)}{dx}$$

Numerical differentiation is:

– Approximate.

– Slow.

– Numerically unstable.

– Easy to write.

Symbolic differentiation

Symbolic differentiation

- What Mathematica does: Automatically derive *analytical* expressions for derivative.

Symbolic differentiation

- What Mathematica does: Automatically derive *analytical* expressions for derivative.

- Often results in very redundant (and expensive to evaluate) expressions.

$$D[\mathbf{Log}[1 + \mathbf{Exp}[w * x + b]], w]$$

$$\text{Out[11]} = \frac{e^{b+wx}\, w}{1 + e^{b+wx}}$$

$$\text{In[19]} := D[\mathbf{Log}[1 + \mathbf{Exp}[w2 * \mathbf{Log}[1 + \mathbf{Exp}[w1 * x + b1]] + b2]], w1]$$

$$\text{Out[19]} = \frac{e^{b1+b2+w1\,x+w2\,\mathrm{Log}\left[1+e^{b1+w1\,x}\right]}\, w2\, x}{\left(1 + e^{b1+w1\,x}\right)\left(1 + e^{b2+w2\,\mathrm{Log}\left[1+e^{b1+w1\,x}\right]}\right)}$$

- Often intractable.

Automatic differentiation (autodiff)

Automatic differentiation (autodiff)

- An autodiff system will convert the program into a sequence of primitive operations which have specified routines for computing derivatives.

- In this representation, backprop can be done in a completely mechanical way.

Sequence of primitive operations:

Original program:

$$z = wx + b$$

$$y = \frac{1}{1 + \exp(-z)}$$

$$\mathcal{L} = \frac{1}{2}(y - t)^2$$

$$t_1 = wx$$

$$z = t_1 + b$$

$$t_3 = -z$$

$$t_4 = \exp(t_3)$$

$$t_5 = 1 + t_4$$

$$y = 1/t_5$$

$$t_6 = y - t$$

$$t_7 = t_6^2$$

$$\mathcal{L} = t_7/2$$

In summary

- Numerical gradient: easy to implement, bad to use.

- Symbolic gradient: sometimes useful, often intractable.

- Automatic gradient: exact, fast, error-prone.

In practice: Use symbolic gradient for small/trivial programs. Almost always use analytic gradient, but check correctness of implementation with numerical gradient.

- This is called a <u>gradient check.</u>

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

Motivation

Products

Helping the Blind

https://www.facebook.com/zuck/videos/10102801434799001/
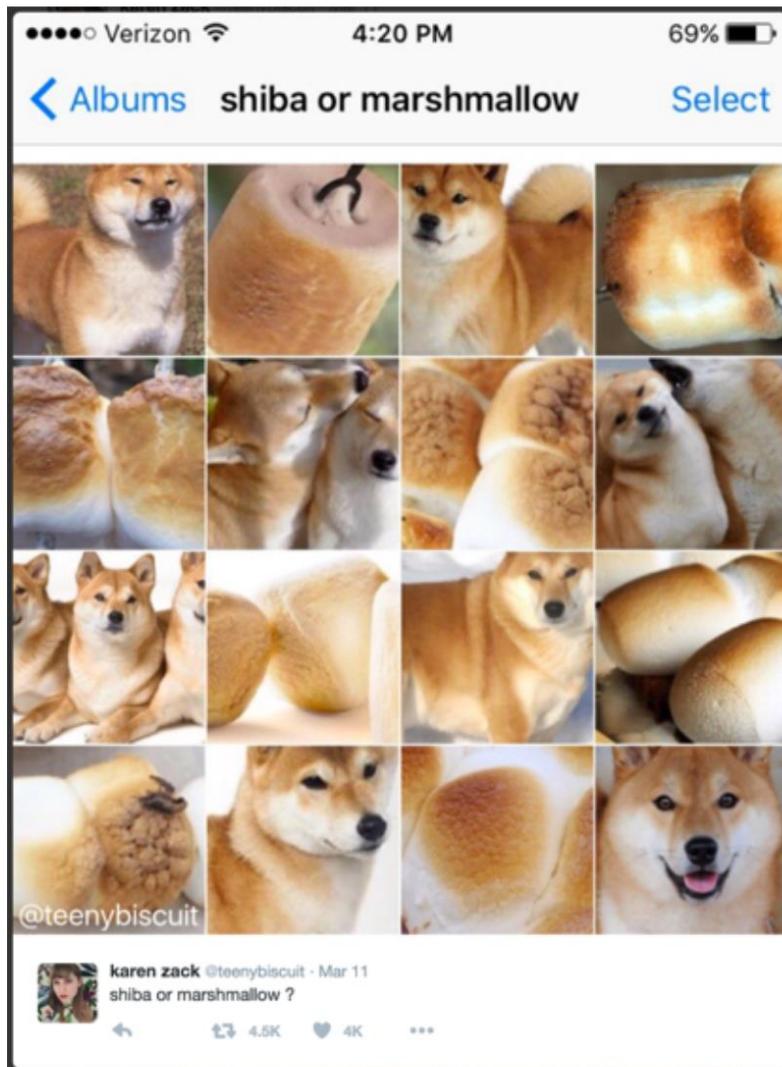
(Unrelated) Dog vs Food

[Karen Zack, @teenybiscuit]

(Unrelated) Dog vs Food

[Karen Zack, @teenybiscuit]

CNNs in 2012: "SuperVision" (aka "AlexNet")

"AlexNet" — Won the ILSVRC2012 Challenge

Major breakthrough: 15.3% Top-5 error on ILSVRC2012 (Next best: 25.7%)
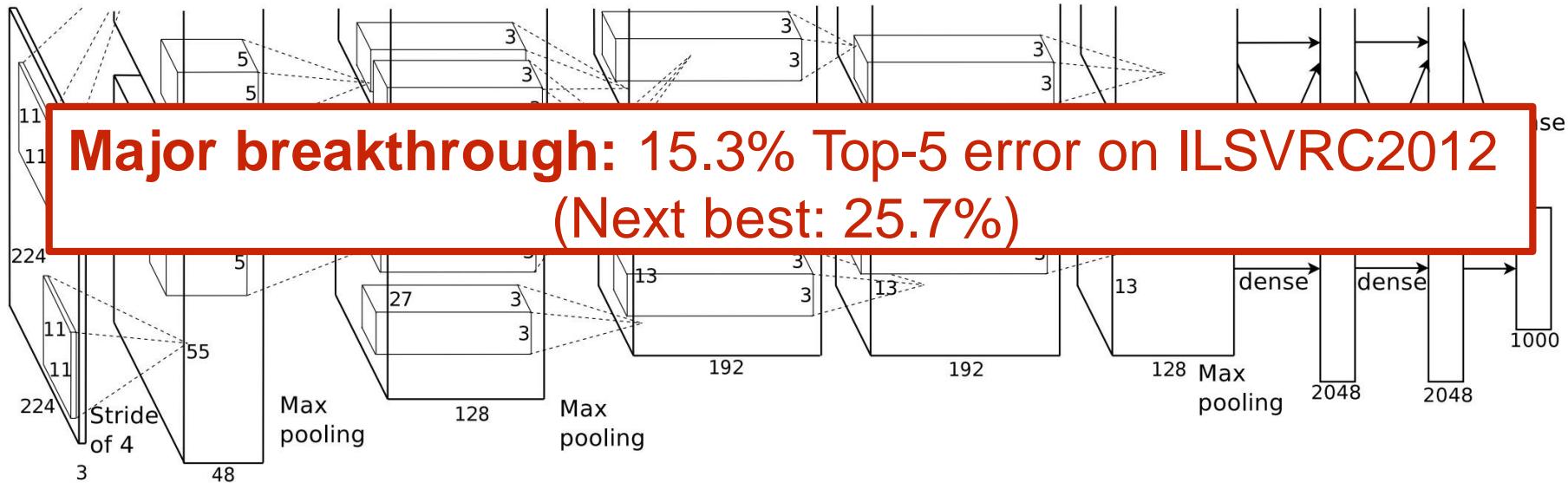
Figure 2: An illustration of the architecture of our CNN, explicitly showing the delineation of responsibilities between the two GPUs. One GPU runs the layer-parts at the top of the figure while the other runs the layer-parts at the bottom. The GPUs communicate only at certain layers. The network's input 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*
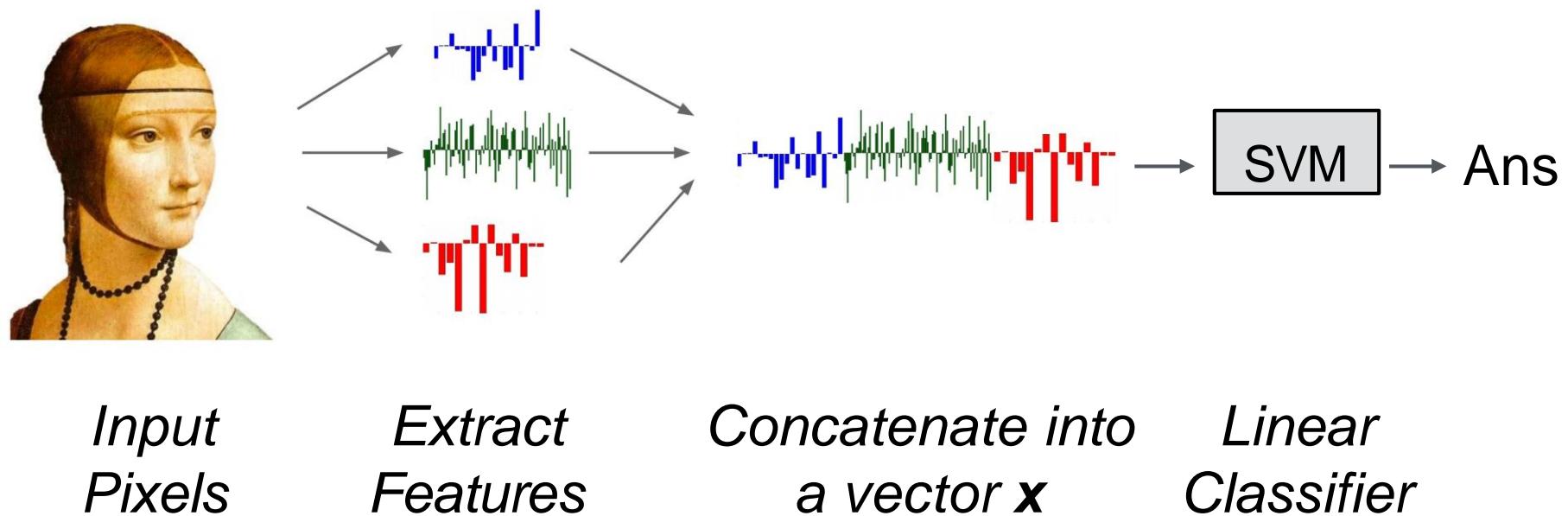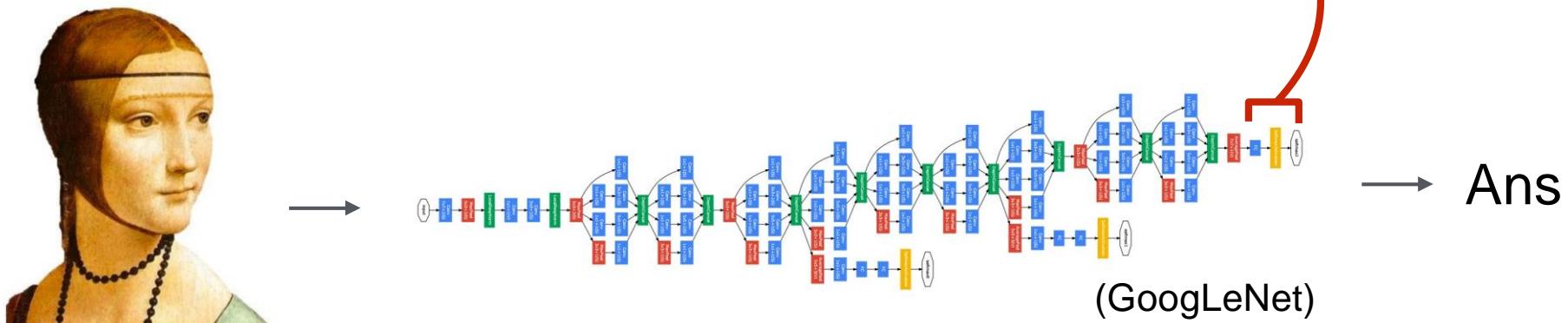
Figure: Karpathy 2016

The last layer of (most) CNNs are linear classifiers

This piece is just a linear classifier


→ Ans

(GoogLeNet)

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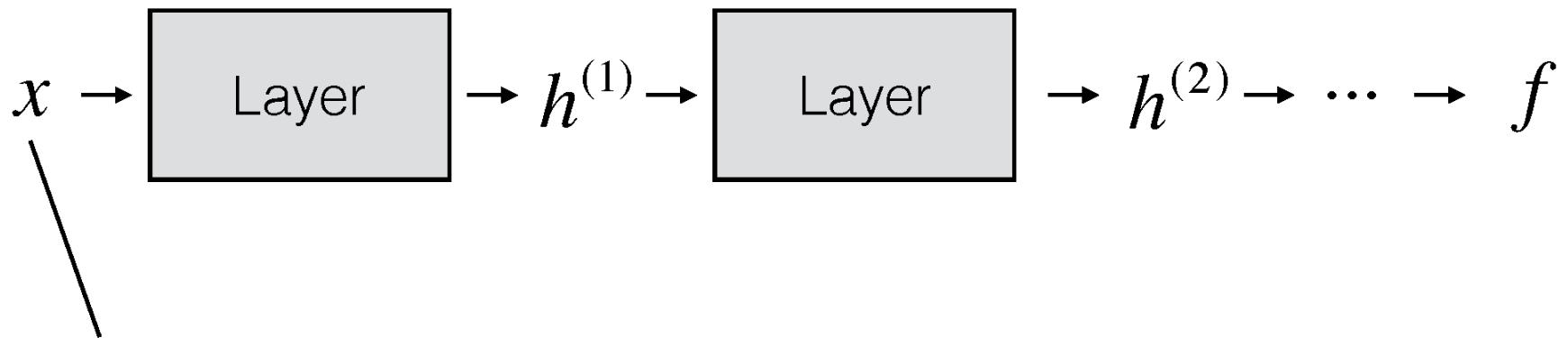
Perform everything with a big neural network, trained end-to-end

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

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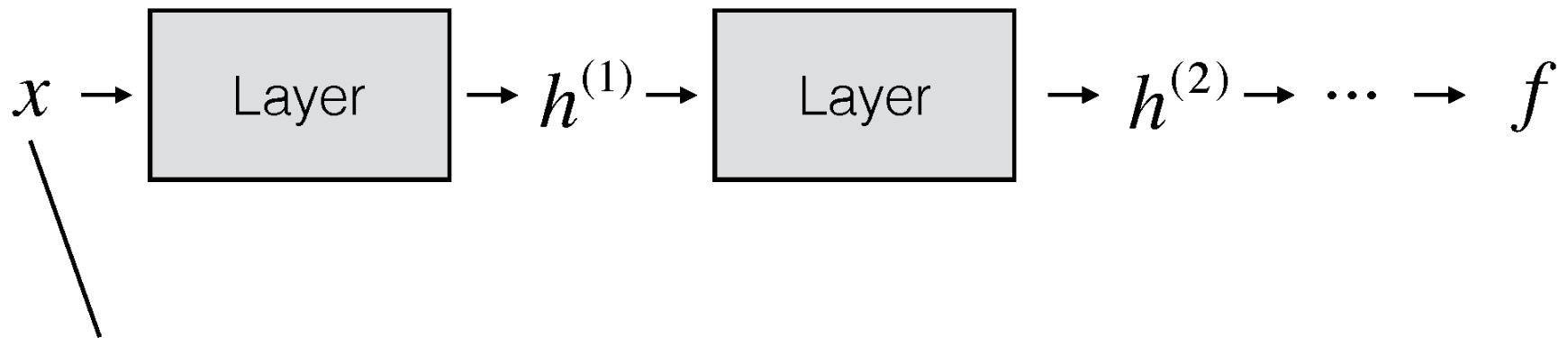
They're just neural networks with
3D activations and weight sharing

What shape should the activations have?

$$x \rightarrow \boxed{\text{Layer}} \rightarrow h^{(1)} \rightarrow \boxed{\text{Layer}} \rightarrow h^{(2)} \rightarrow \cdots \rightarrow f$$
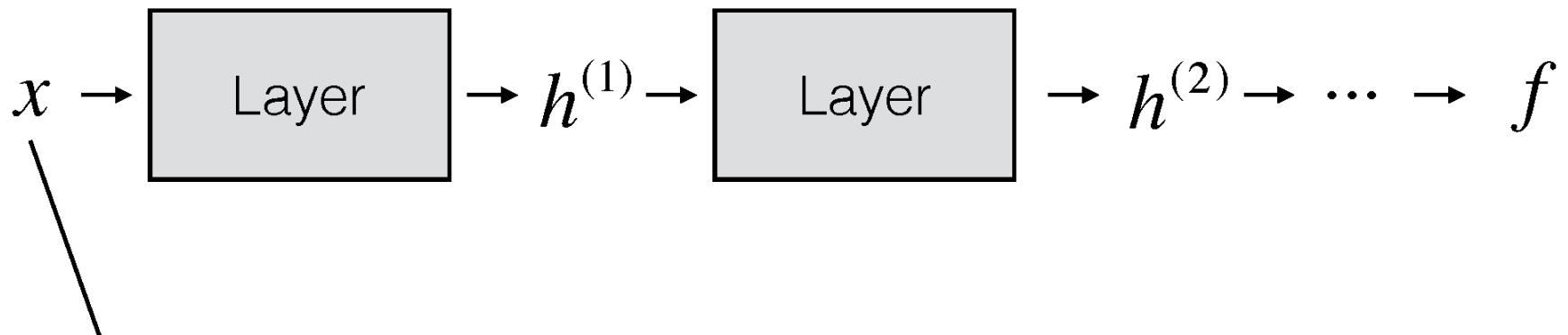
- The input is an image, which is 3D
(RGB channel, height, width)

What shape should the activations have?

$$x \rightarrow \boxed{\text{Layer}} \rightarrow h^{(1)} \rightarrow \boxed{\text{Layer}} \rightarrow h^{(2)} \rightarrow \cdots \rightarrow f$$

- 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?
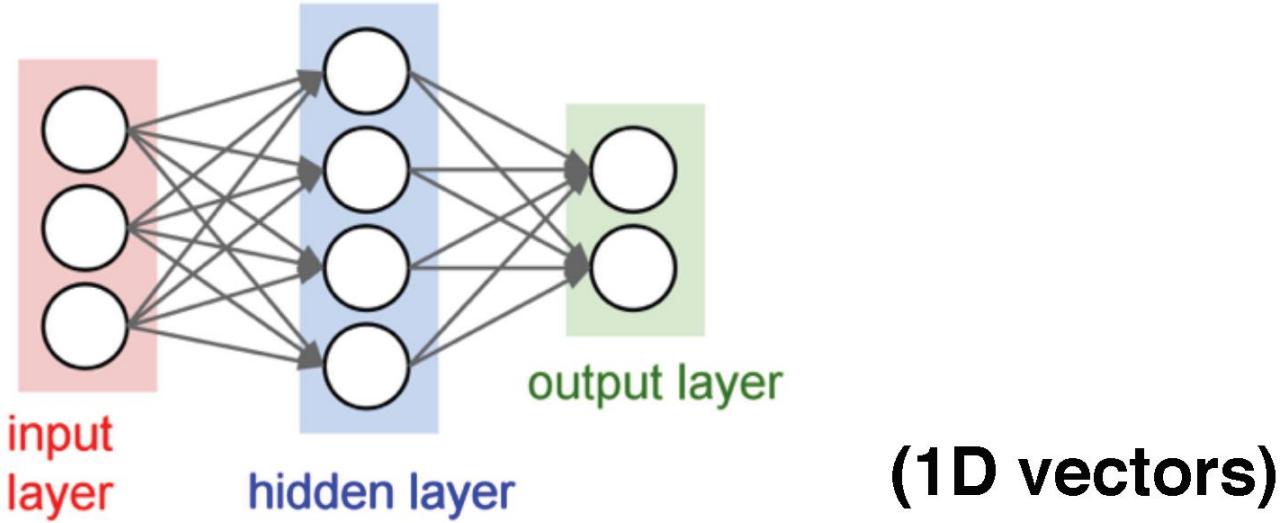
$$x \rightarrow \boxed{\text{Layer}} \rightarrow h^{(1)} \rightarrow \boxed{\text{Layer}} \rightarrow h^{(2)} \rightarrow \cdots \rightarrow f$$

- 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?

ConvNets

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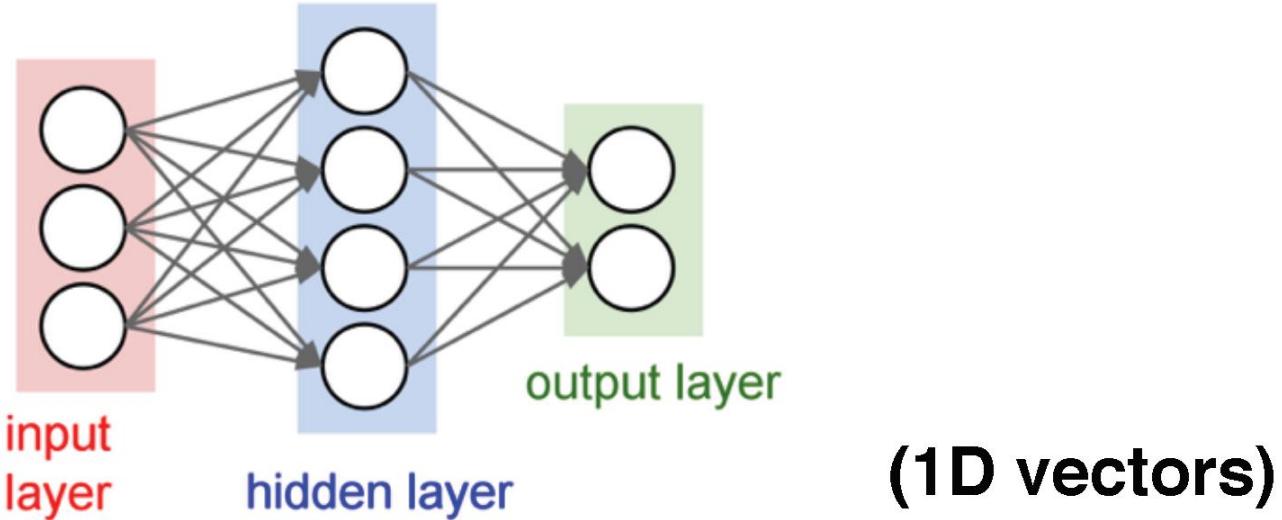
3D Activations

before:
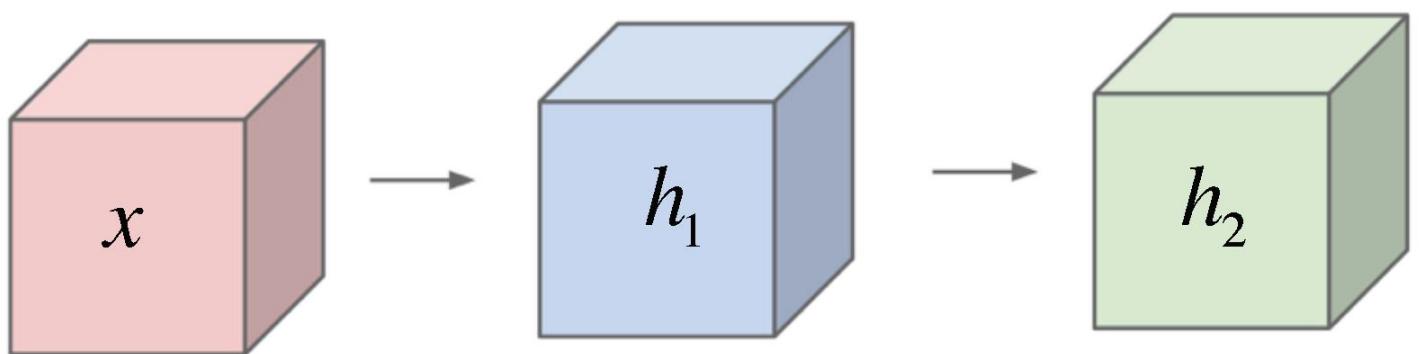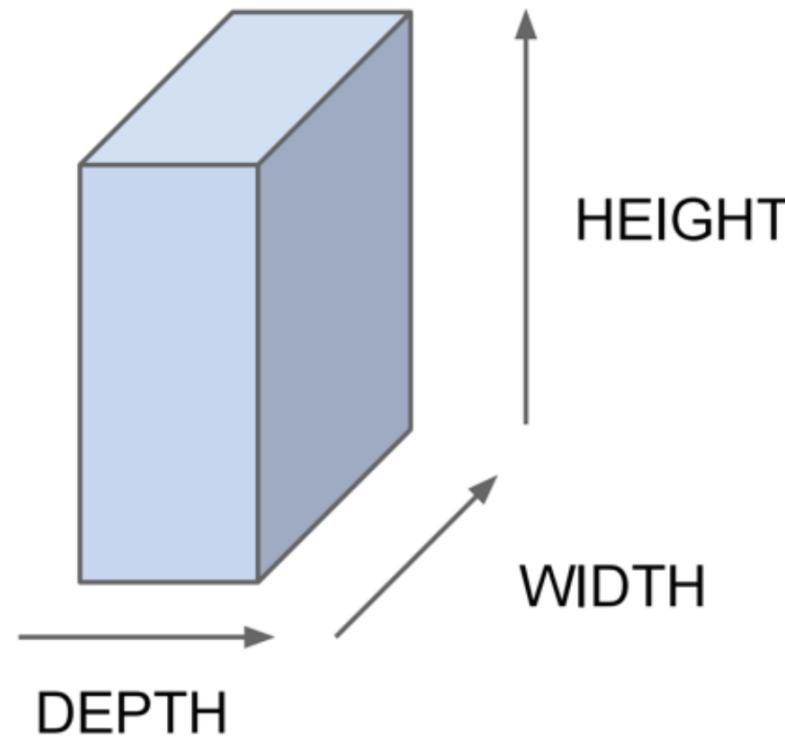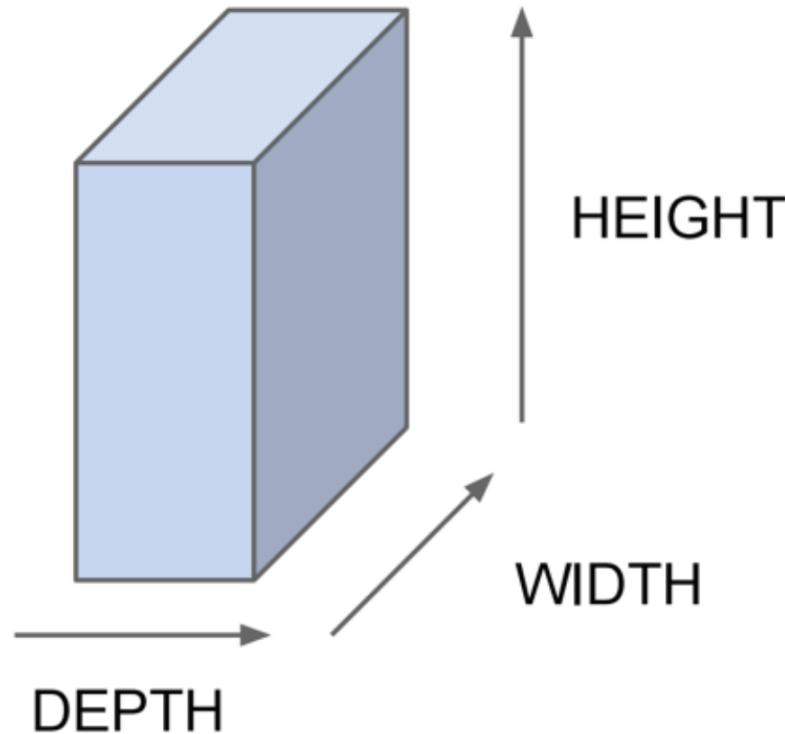
(1D vectors)

*Figure: Andrej Karpathy*

3D Activations

before:

input layer — hidden layer — output layer

(1D vectors)

now:

$x \rightarrow h_1 \rightarrow h_2$

(3D arrays)

Figure: Andrej Karpathy

3D Activations

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


HEIGHT

WIDTH

DEPTH

Figure: Andrej Karpathy

3D Activations

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

For example, a CIFAR-10 image is a 3x32x32 volume (3 depth — RGB channels, 32 height, 32 width)

Figure: Andrej Karpathy

3D Activations

1D Activations:

3D Activations

**1D Activations:**

3D Activations:

32

a hidden neuron in next layer

32

3
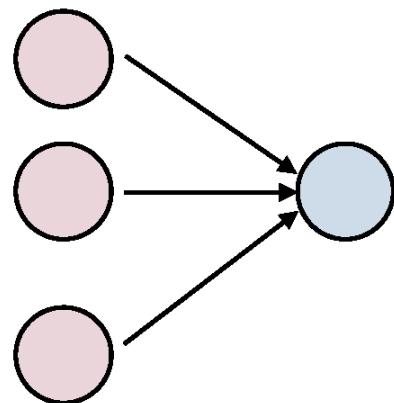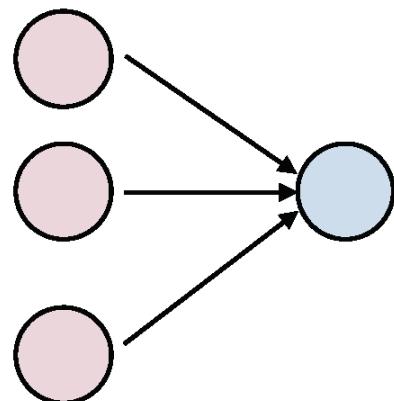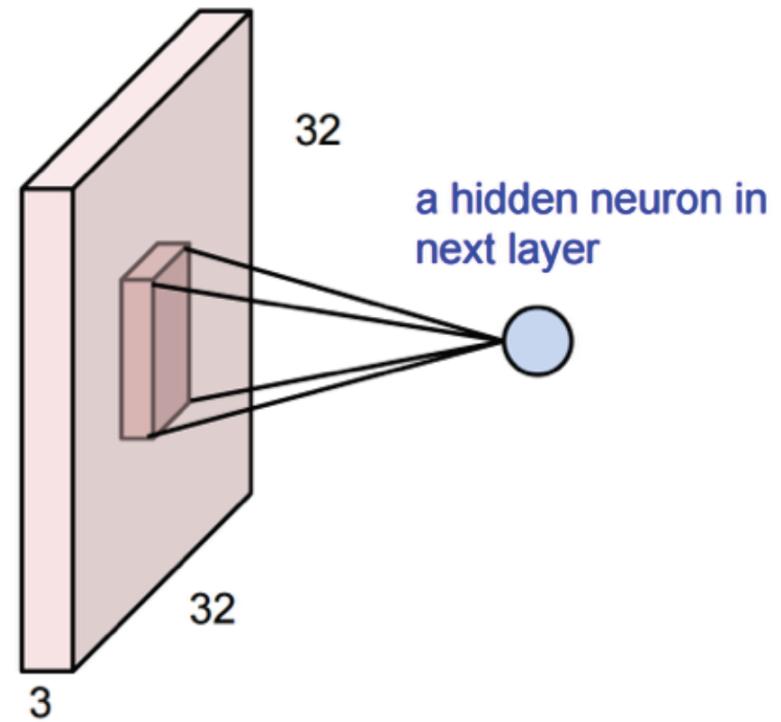
Figure: Andrej Karpathy
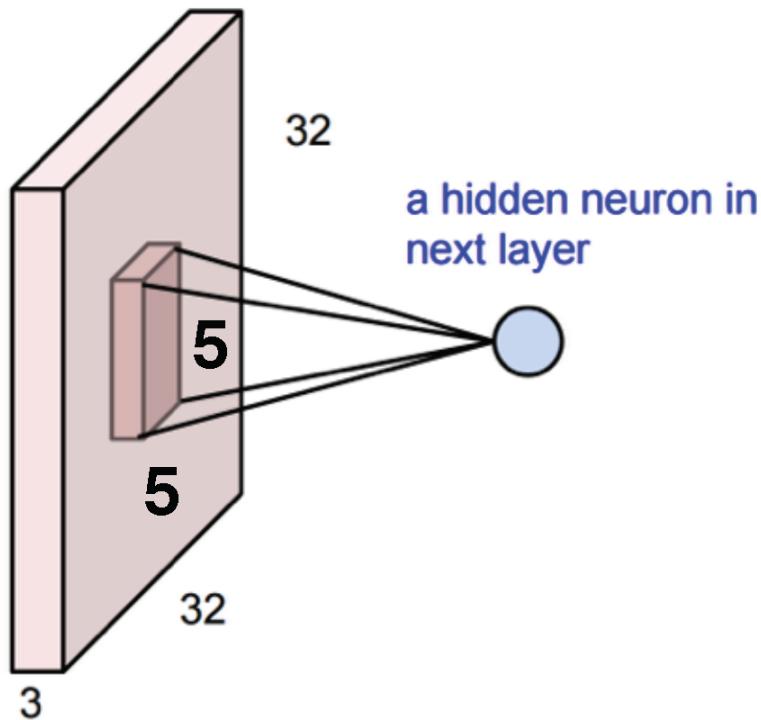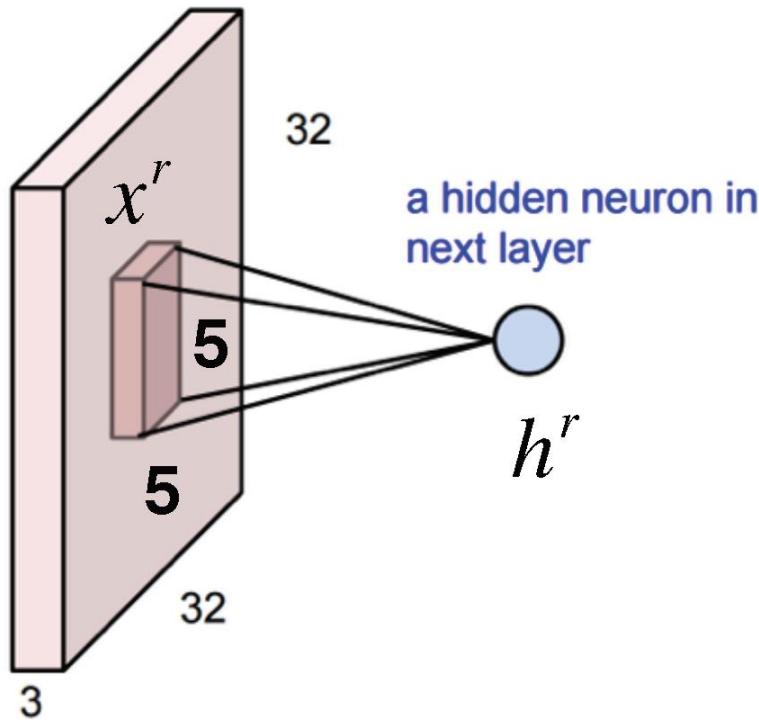
3D Activations

- The input is 3x32x32

- This neuron depends on a 3x5x5 chunk of the input

- The neuron also has a 3x5x5 set of weights and a bias (scalar)
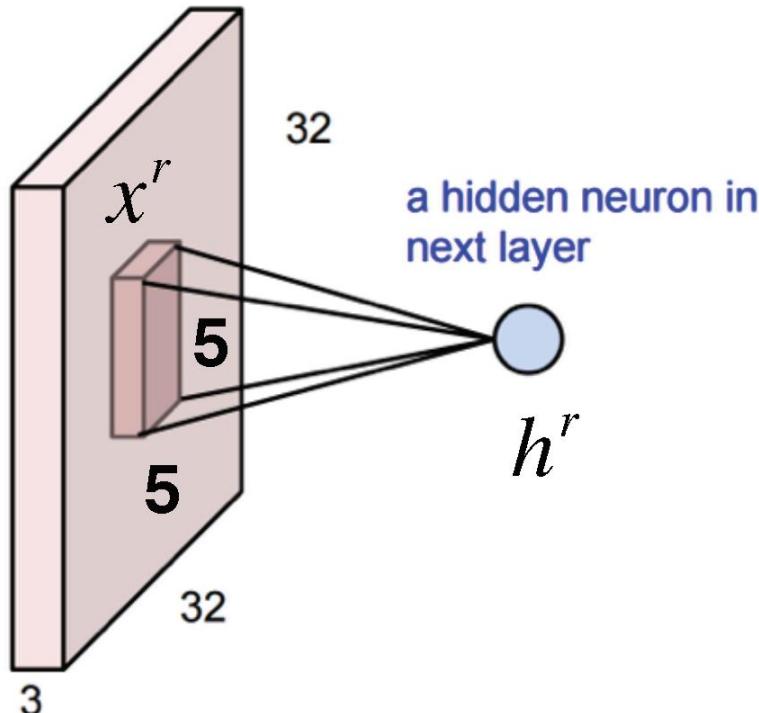
Figure: Andrej Karpathy

3D Activations

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

With output neuron h^r

Figure: Andrej Karpathy

3D Activations

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

With output neuron h^r

Then the output is:

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

Figure: Andrej Karpathy
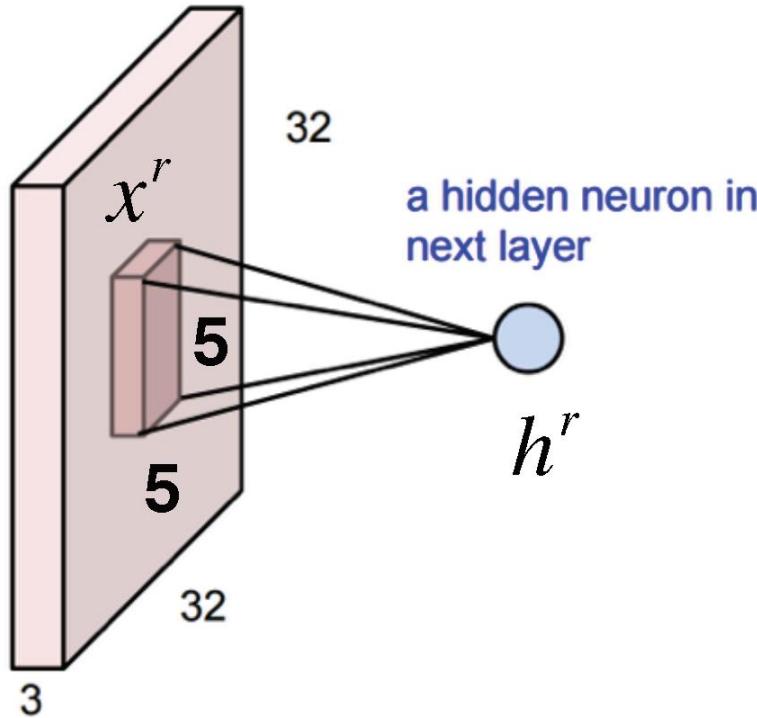
3D Activations

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

With output neuron h^r
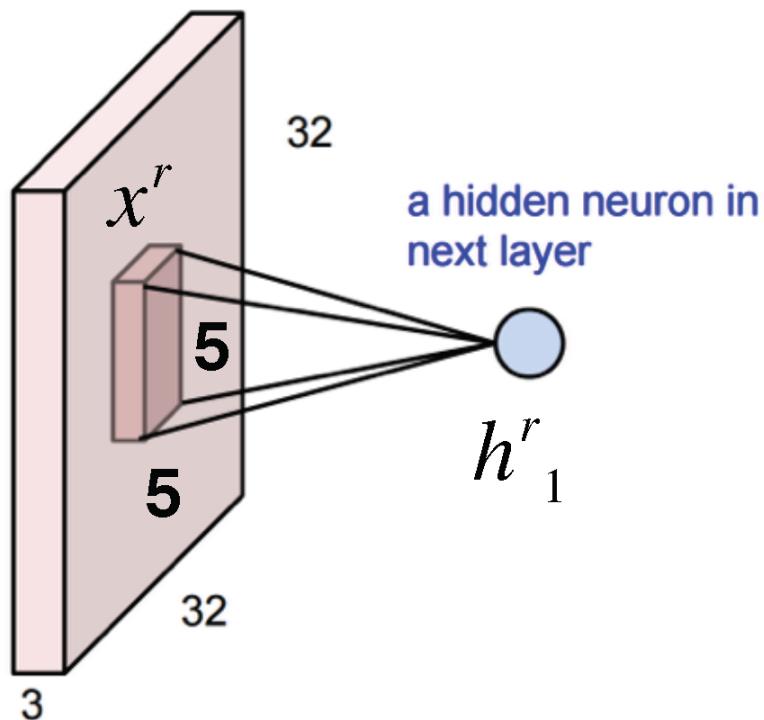
Then the output is:

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

Sum over 3 axes

Figure: Andrej Karpathy

3D Activations

x^r

32

a hidden neuron in
next layer

5

5

32

3

h^r_1
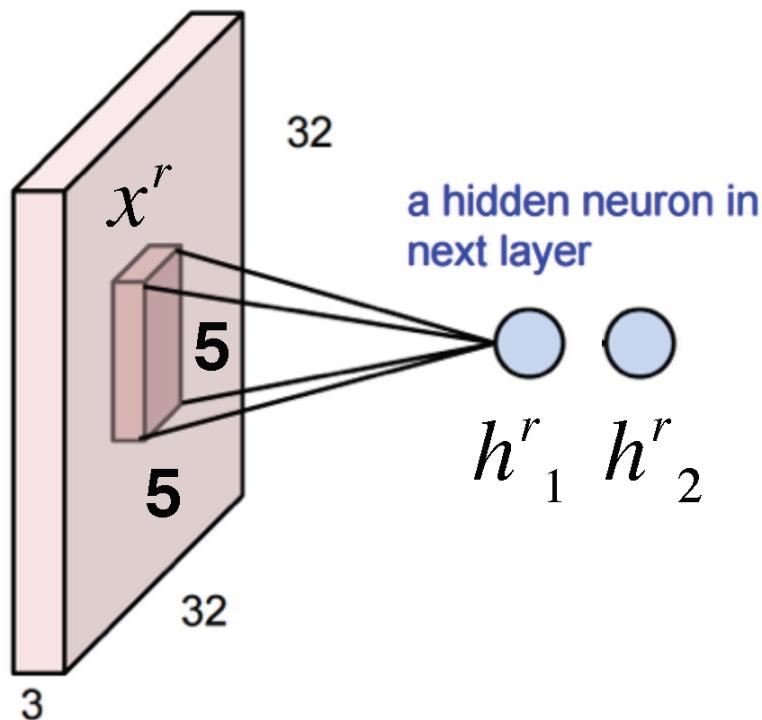
Figure: Andrej Karpathy
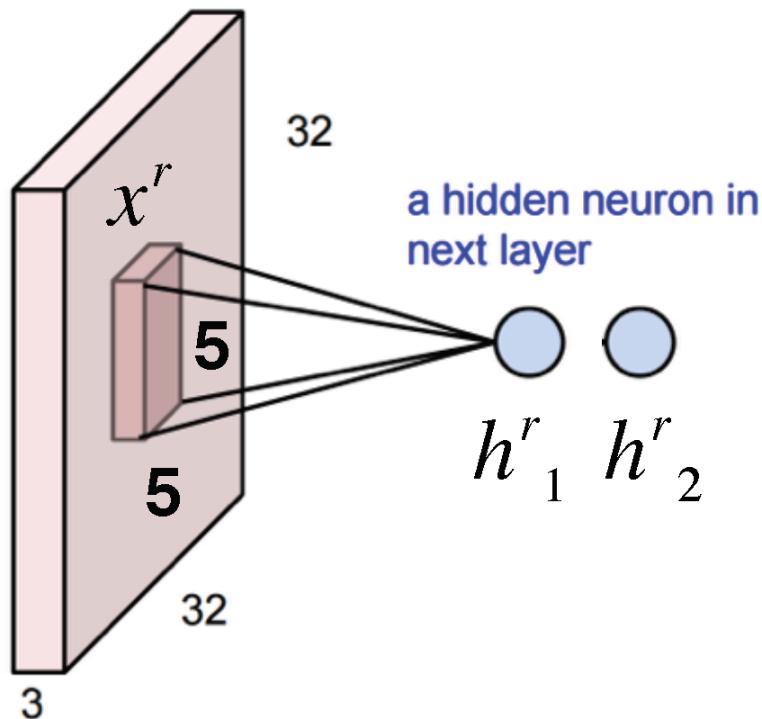
3D Activations

Figure: Andrej Karpathy
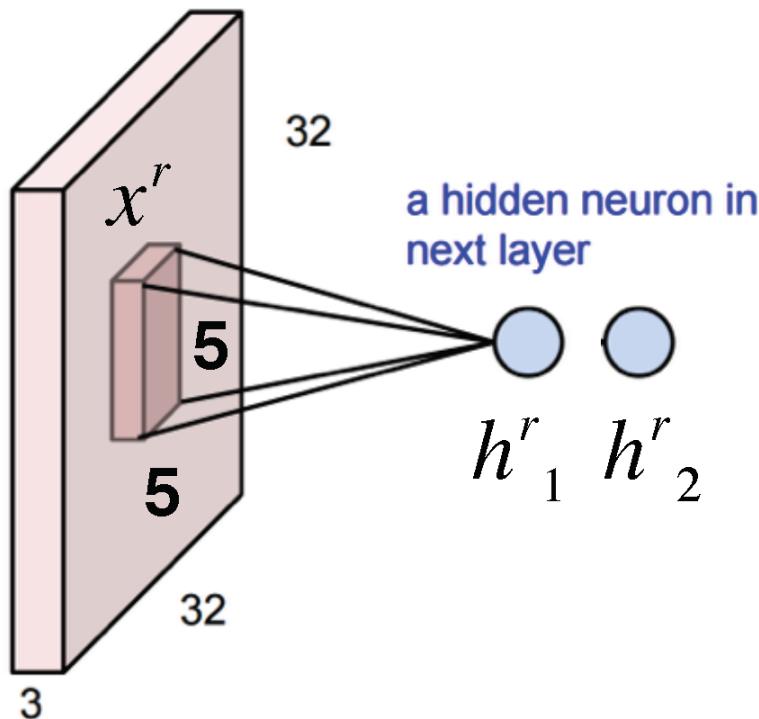
3D Activations

With **2** output neurons

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

$$h^r_{\ 2} = \sum_{ijk} x^r_{\ ijk} W_{2ijk} + b_2$$
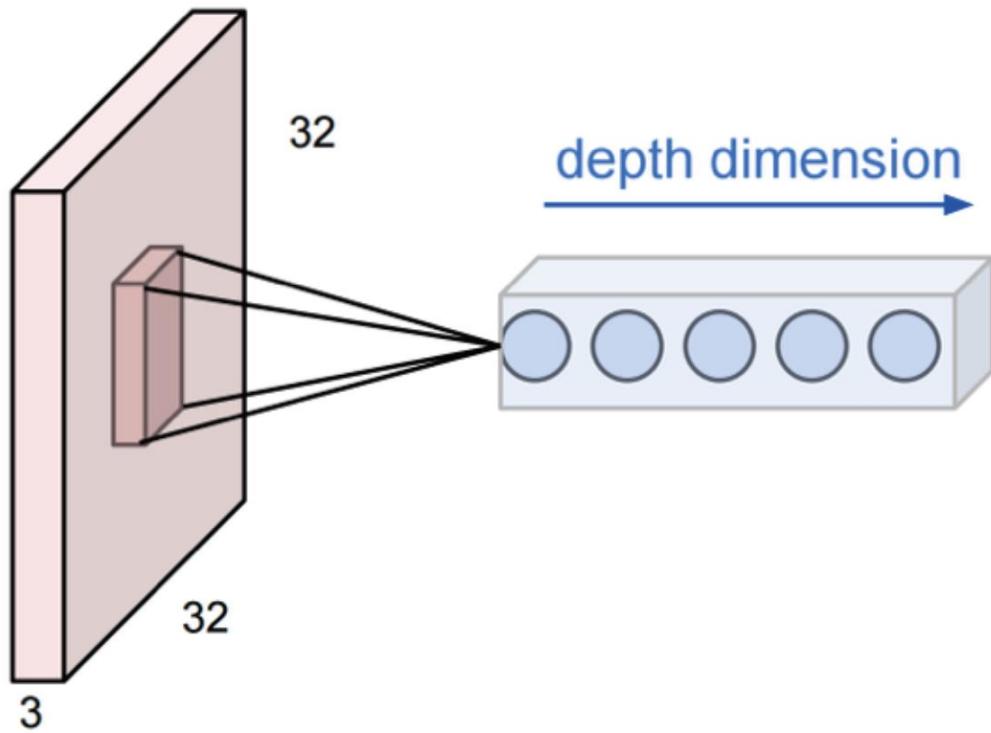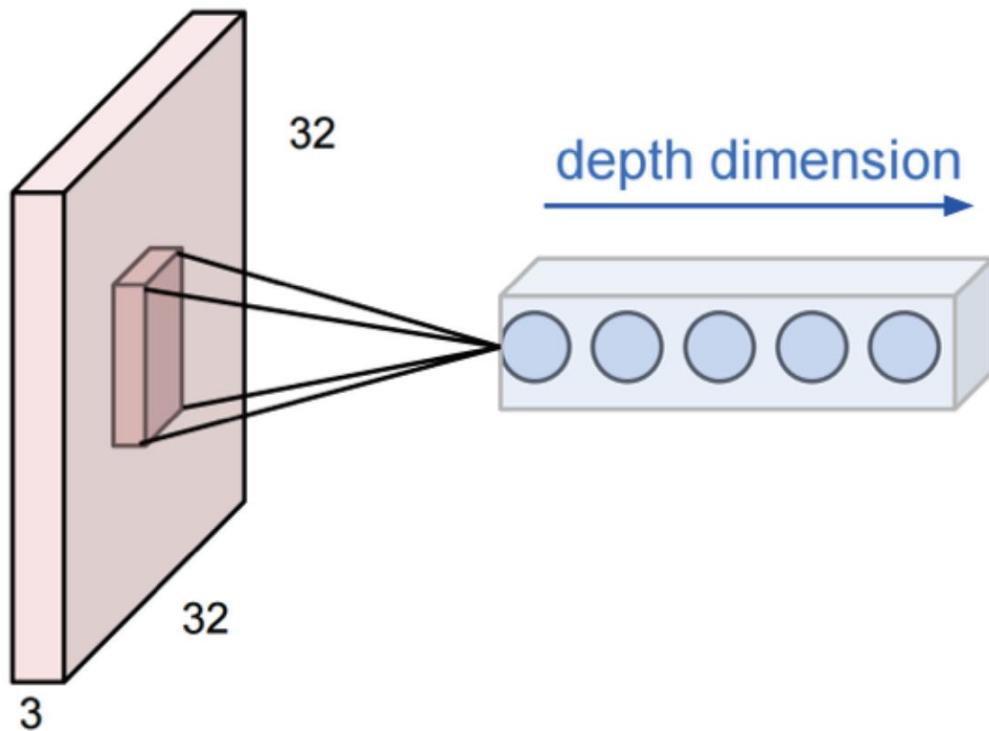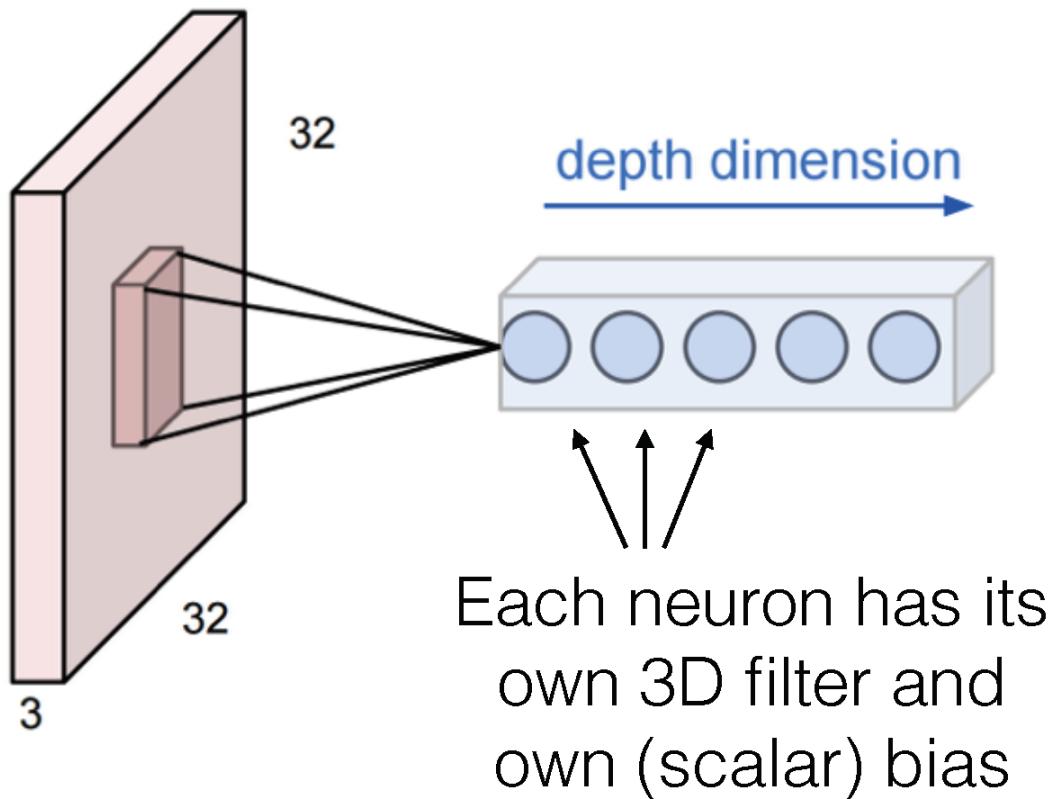
Figure: Andrej Karpathy

3D Activations

With **2** output neurons

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

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

Figure: Andrej Karpathy

# 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]

Figure: Andrej Karpathy

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

Figure: Andrej Karpathy
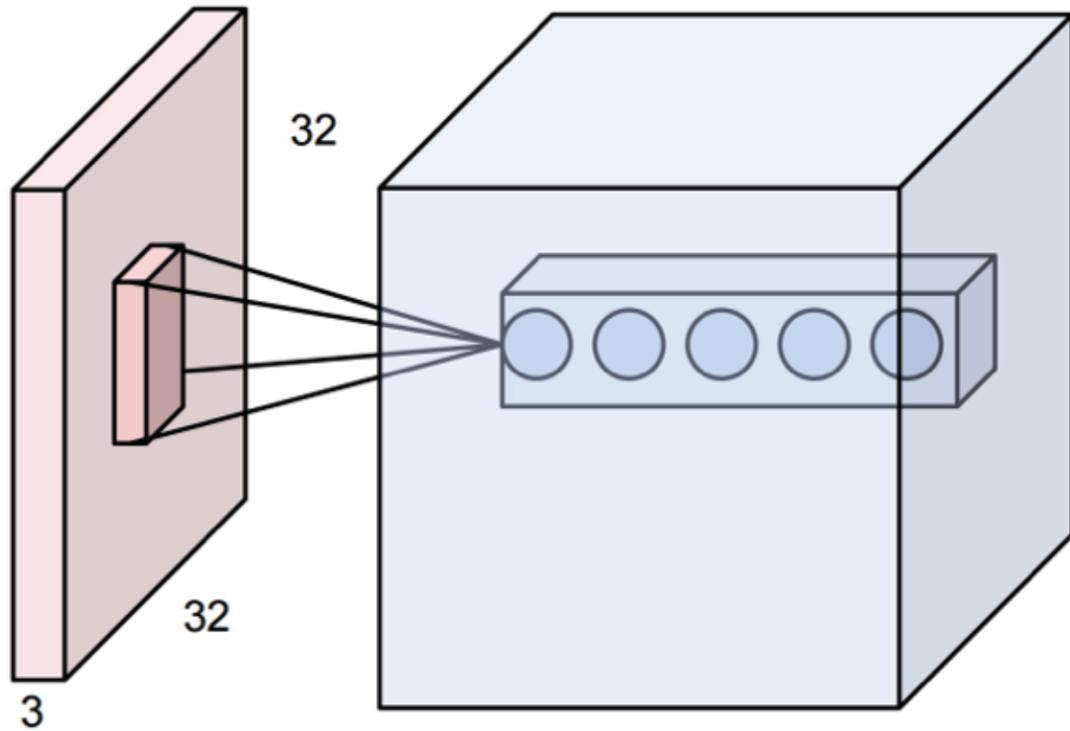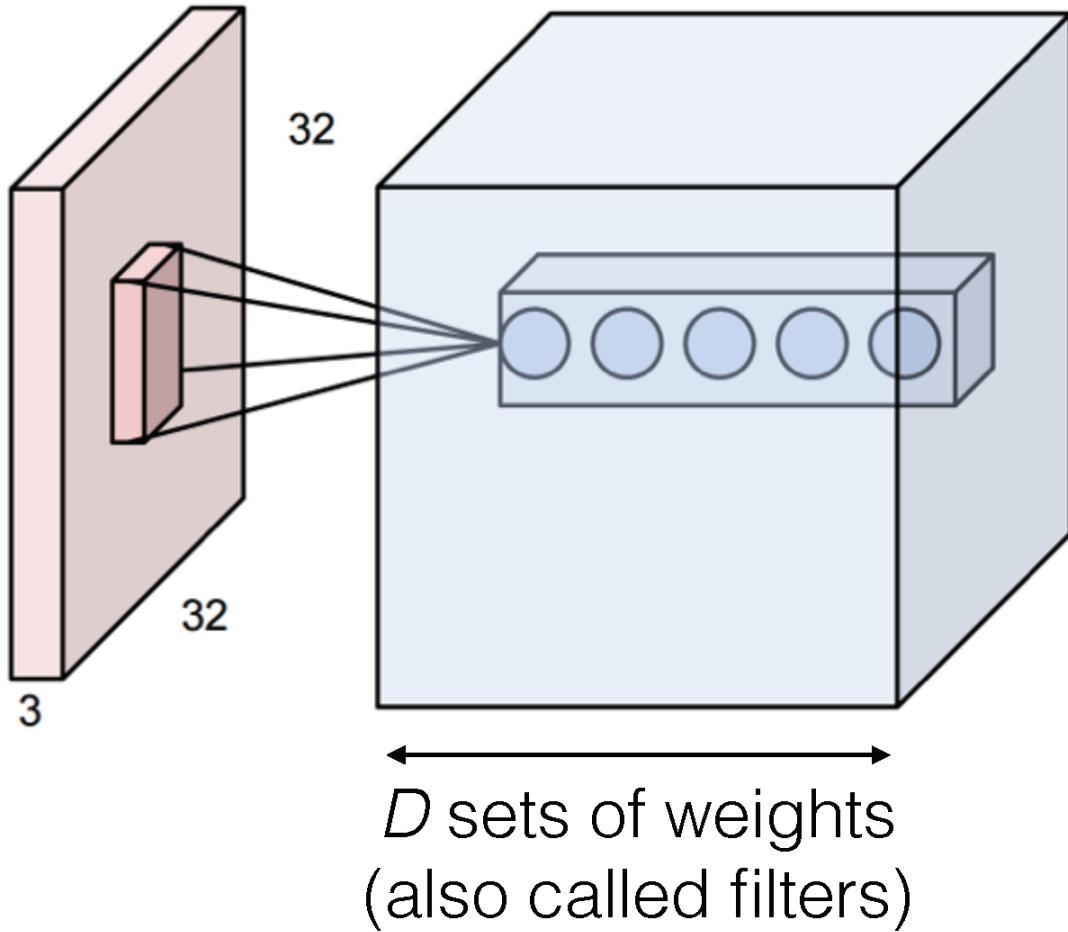
3D Activations

Now repeat this across the input

D sets of weights (also called filters)

Figure: Andrej Karpathy

3D Activations

32

32

3

D sets of weights
(also called filters)
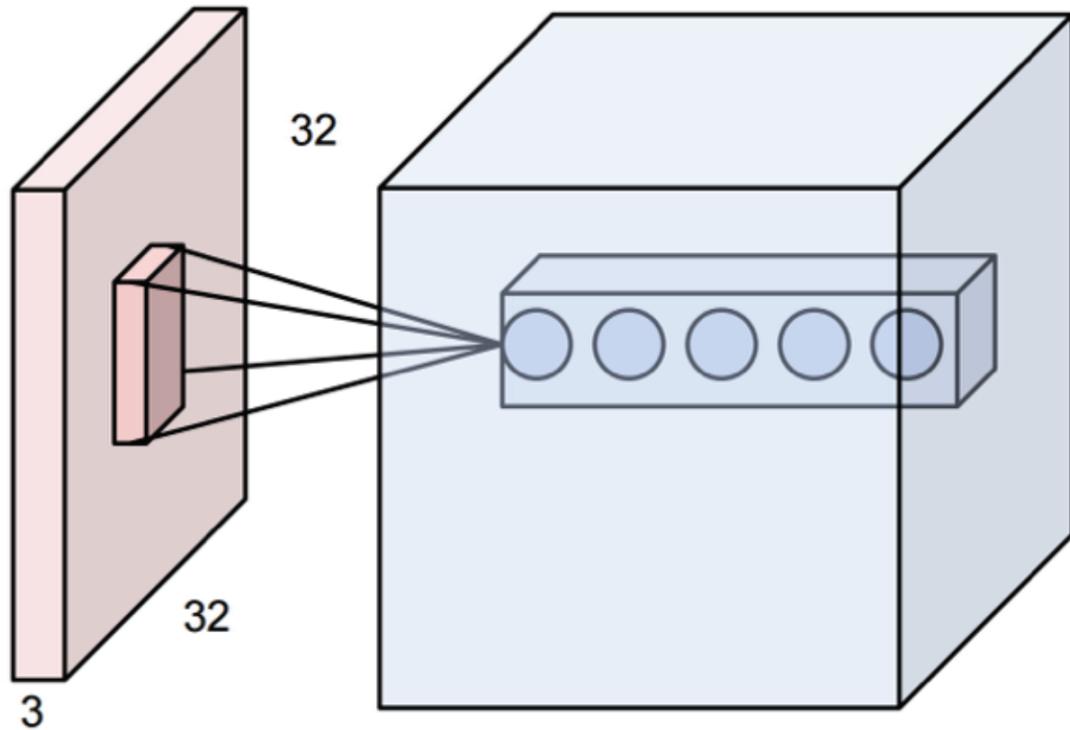
Now repeat this across the input

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

Figure: Andrej Karpathy

3D Activations

32

32

3

D sets of weights
(also called filters)

Figure: Andrej Karpathy

3D Activations

32

32

3

D sets of weights
(also called filters)

With weight
sharing,
this is called
convolution

Figure: Andrej Karpathy
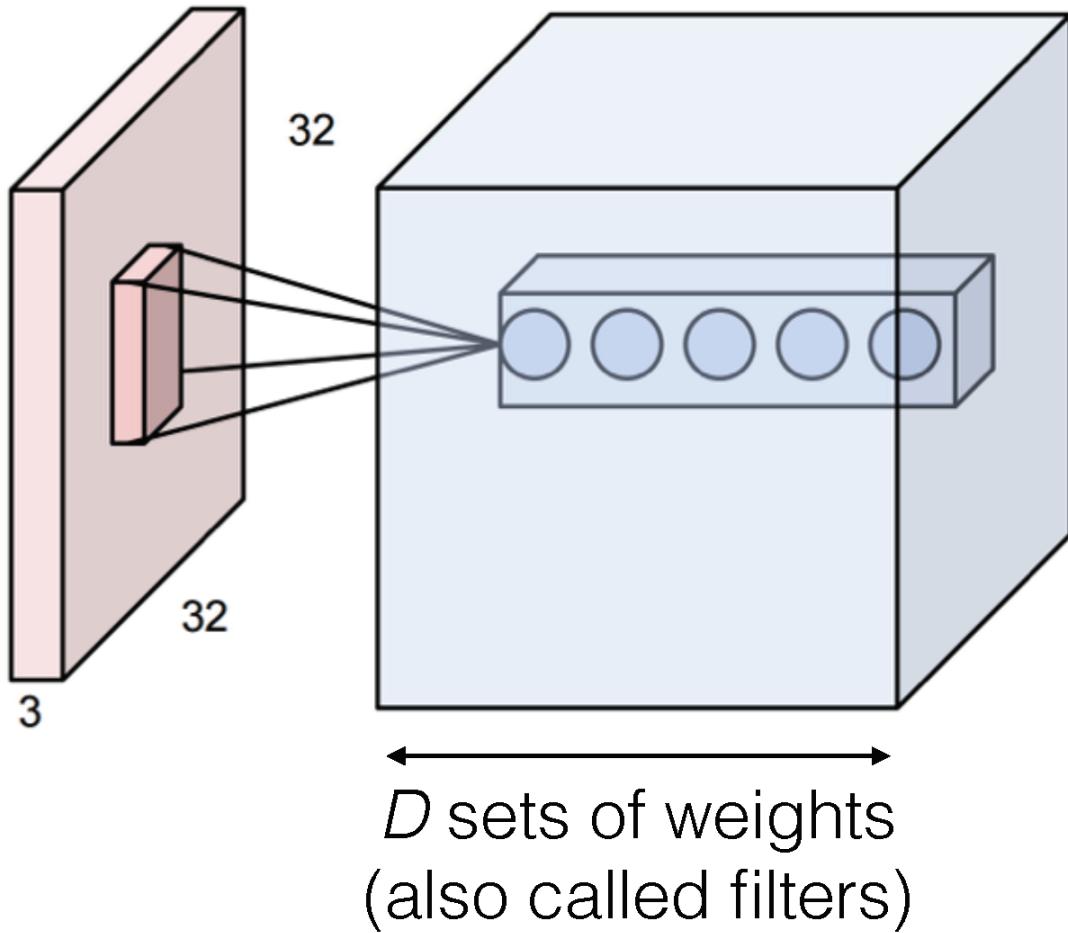
3D Activations

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

Without weight sharing, this is called a **locally connected layer**
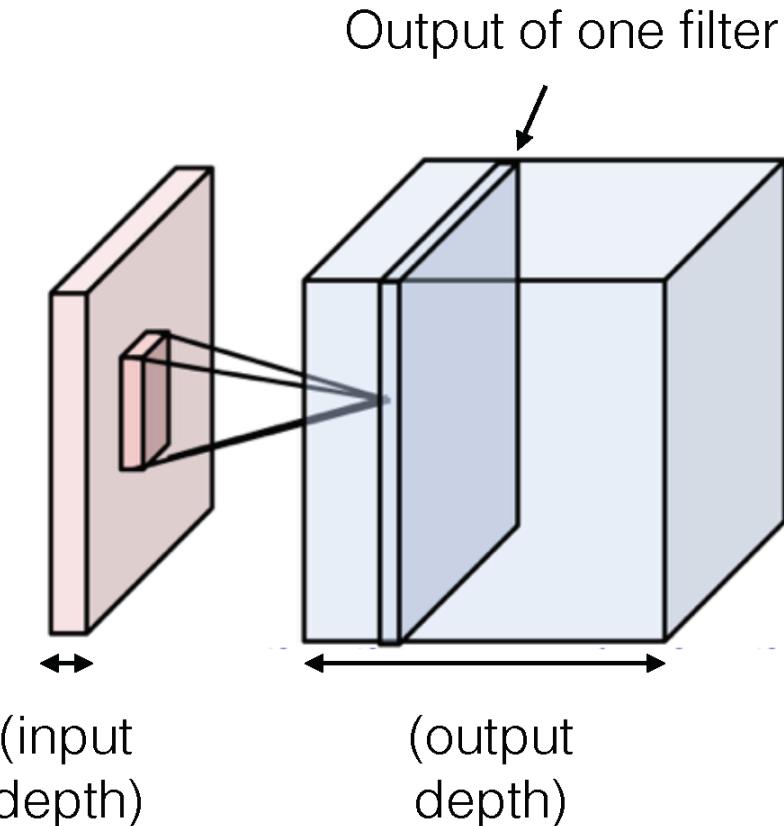
D sets of weights (also called filters)

Figure: Andrej Karpathy

3D Activations

Output of one filter

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)

(input depth)

(output depth)

3D Activations

Output of one filter

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)

(input depth)

(output depth)

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)

Activations:

Figure: Andrej Karpathy

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)

Activations:

Figure: Andrej Karpathy
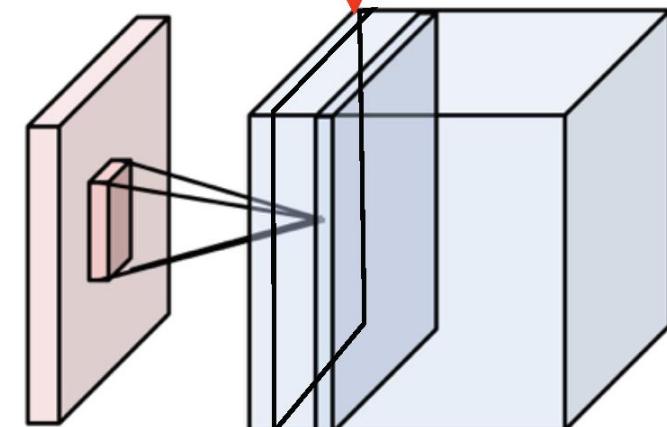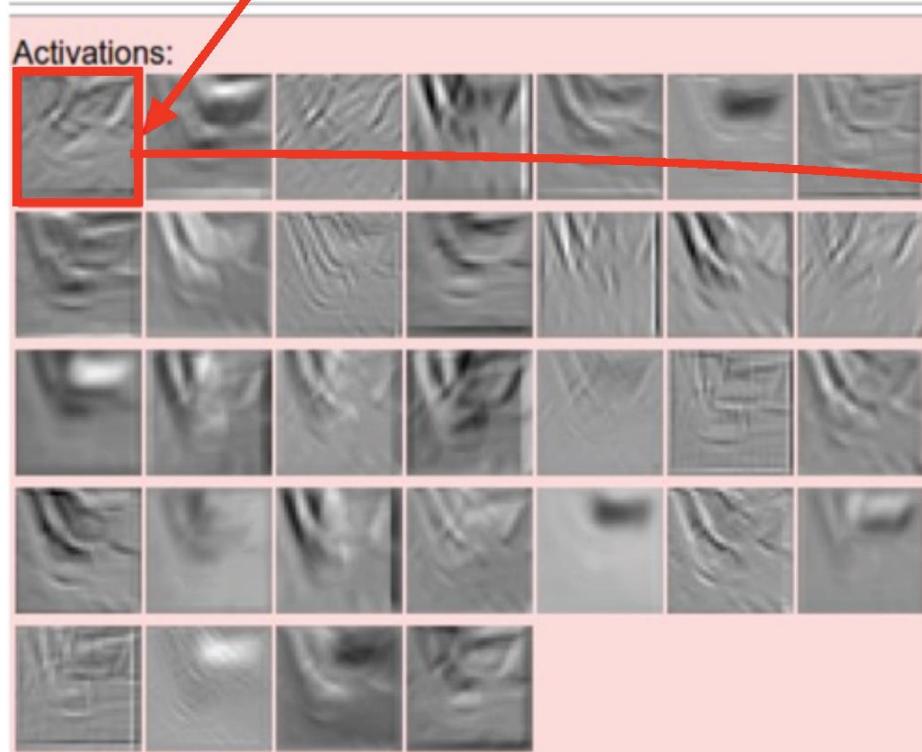
3D Activations

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

(Input)

one filter = one depth slice (or activation map)

(32 filters, each 3x5x5)

Activations:

Figure: Andrej Karpathy

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)

Activations:

Figure: Andrej Karpathy

(Recap)

A **ConvNet** is a sequence of convolutional layers, interspersed with activation functions (and possibly other layer types)
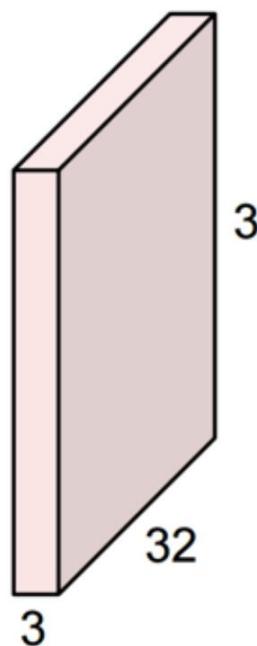
32
32
3

CONV,
ReLU
e.g. 6
5x5x3
filters

28
28
6

CONV,
ReLU
e.g. 10
5x5x**6**
filters

24
24
10

CONV,
ReLU

....

(Recap)

Convolution Layer

32x32x3 image

32 height

32 width

3 depth

(Recap)

Convolution Layer

32x32x3 image

32

32

3

5x5x3 filter

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

(Recap)

Convolution Layer

32x32x3 image
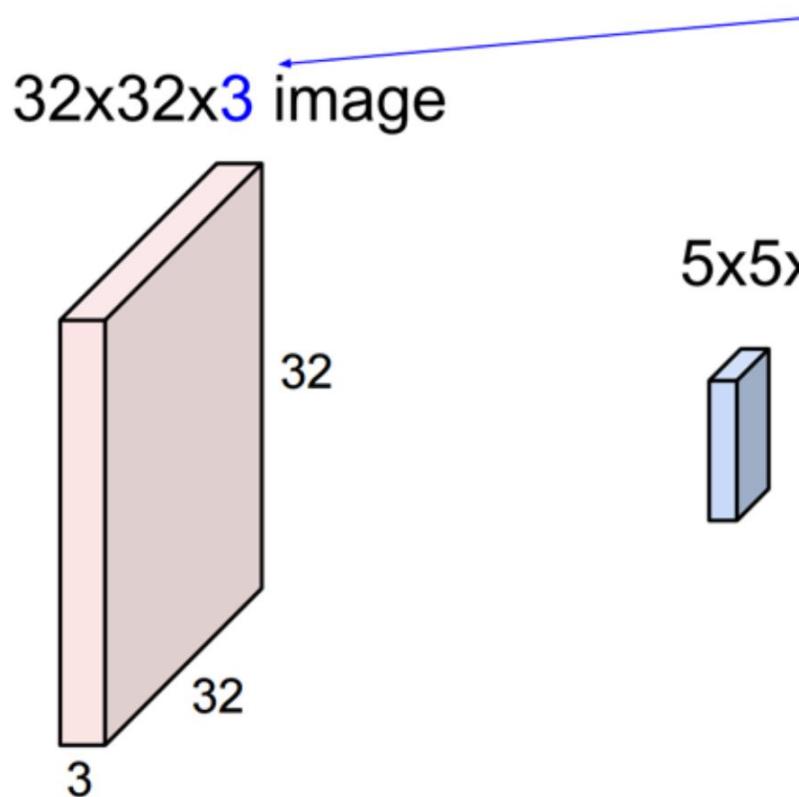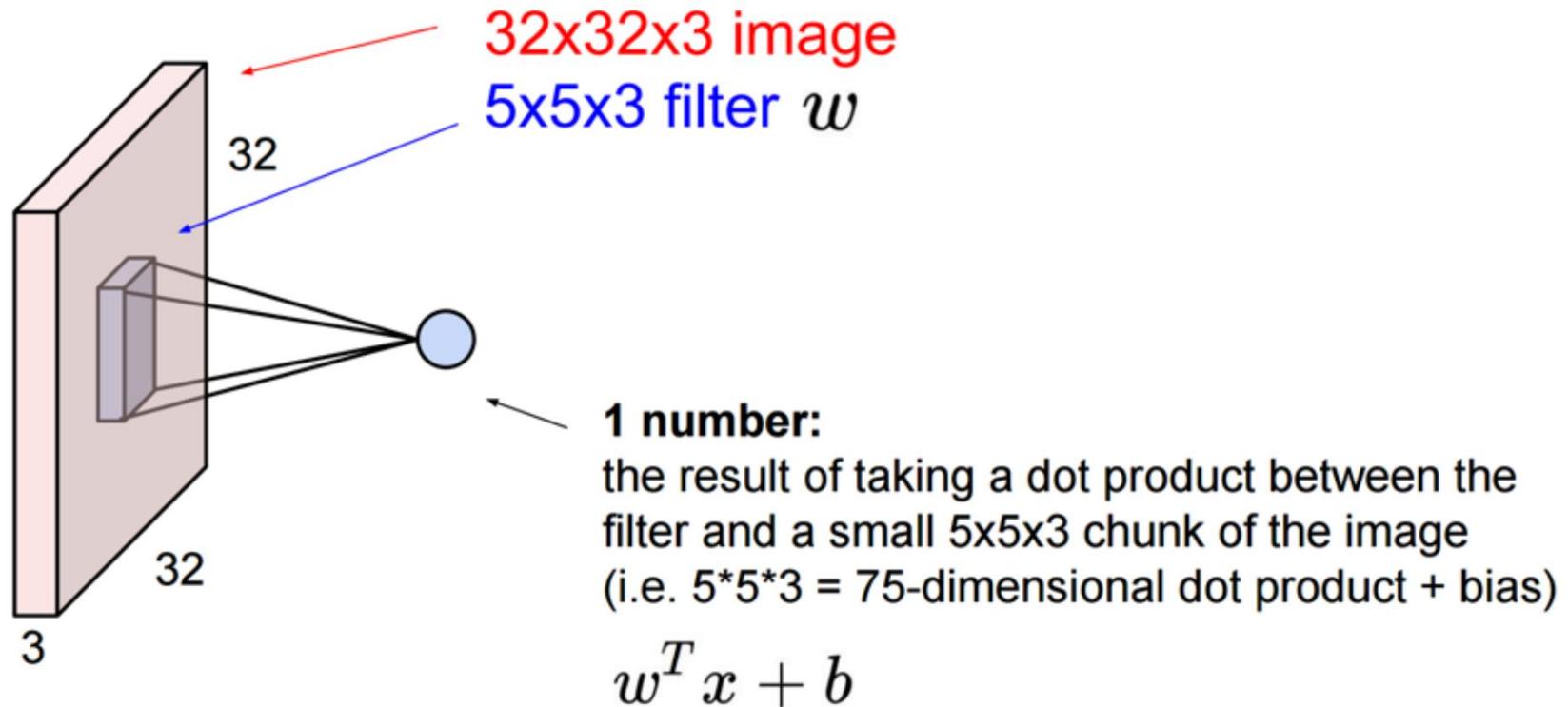
Filters always extend the full depth of the input volume

32

32

3

**5x5x3 filter**

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

(Recap)

Convolution Layer

32x32x3 image
5x5x3 filter w

32

32

3

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)

$$w^T x + b$$

(Recap)

Convolution Layer

32x32x3 image
5x5x3 filter

convolve (slide) over all
spatial locations

activation map

32

32

3

28

28

1
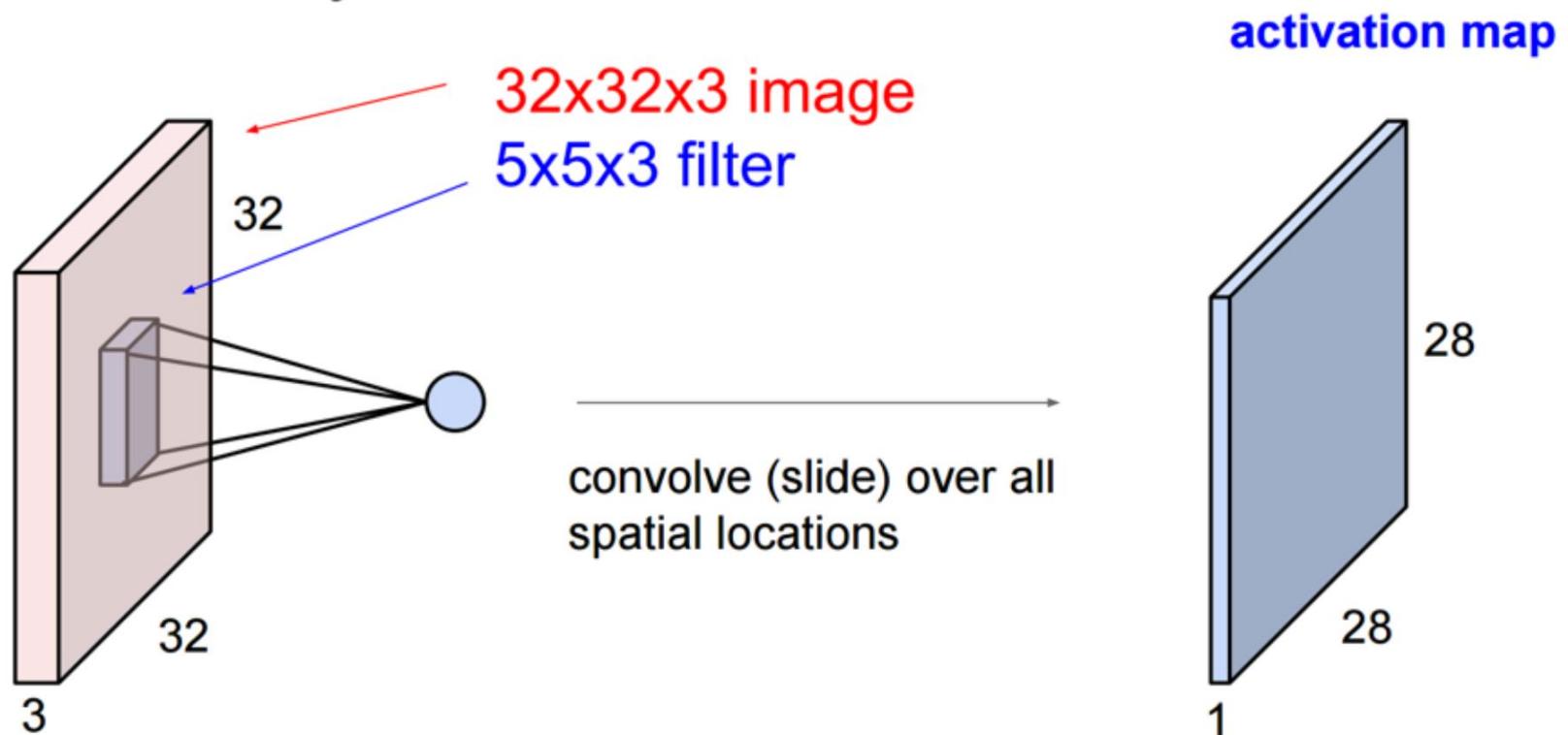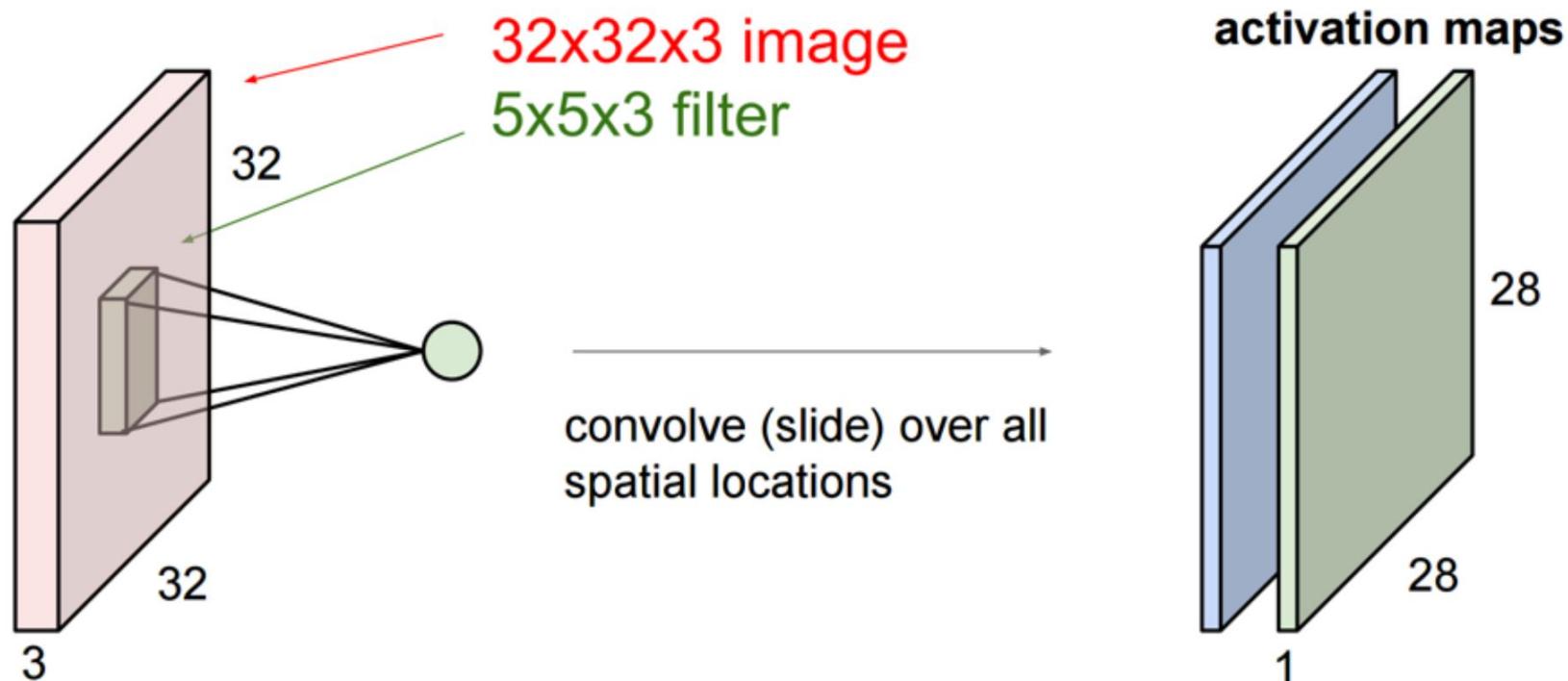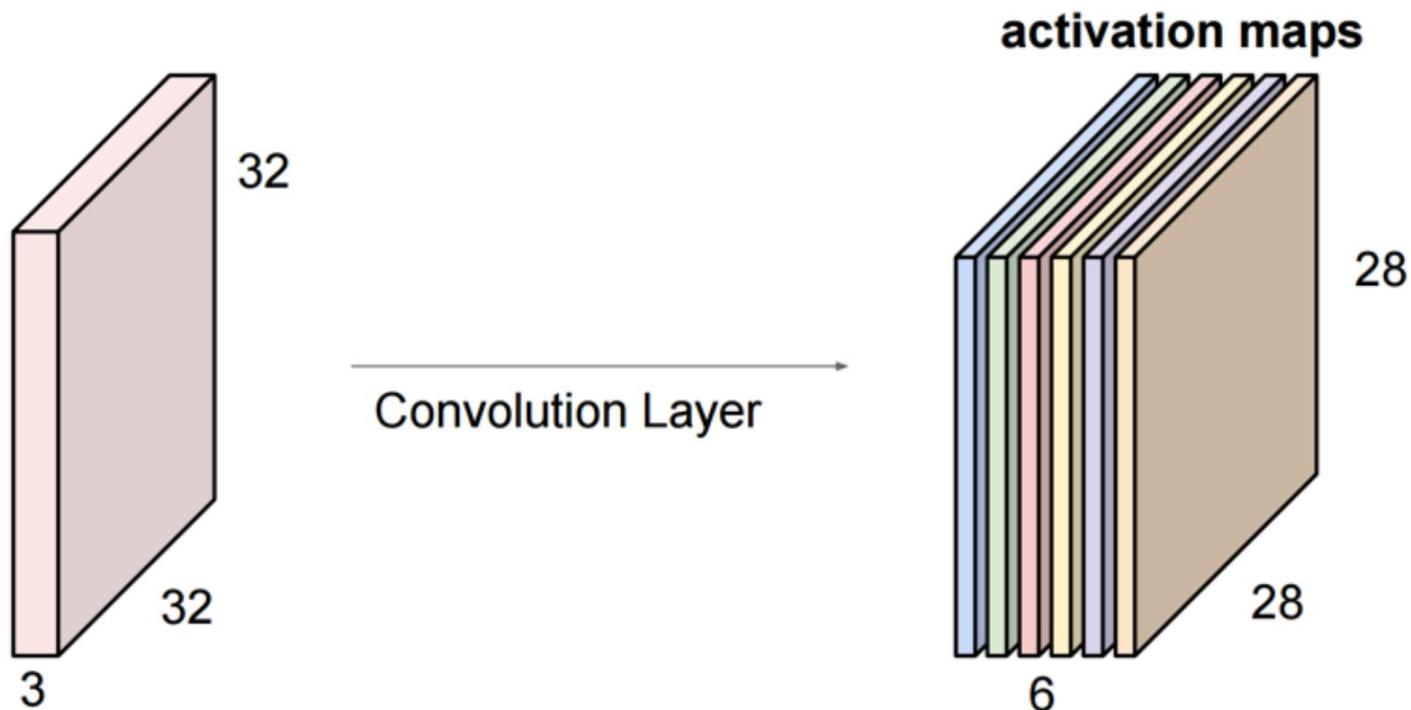
(Recap)

Convolution Layer

consider a second, green filter

32x32x3 image
5x5x3 filter

activation maps

convolve (slide) over all spatial locations

(Recap)

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

activation maps

32

32

3

Convolution Layer

28

28

6
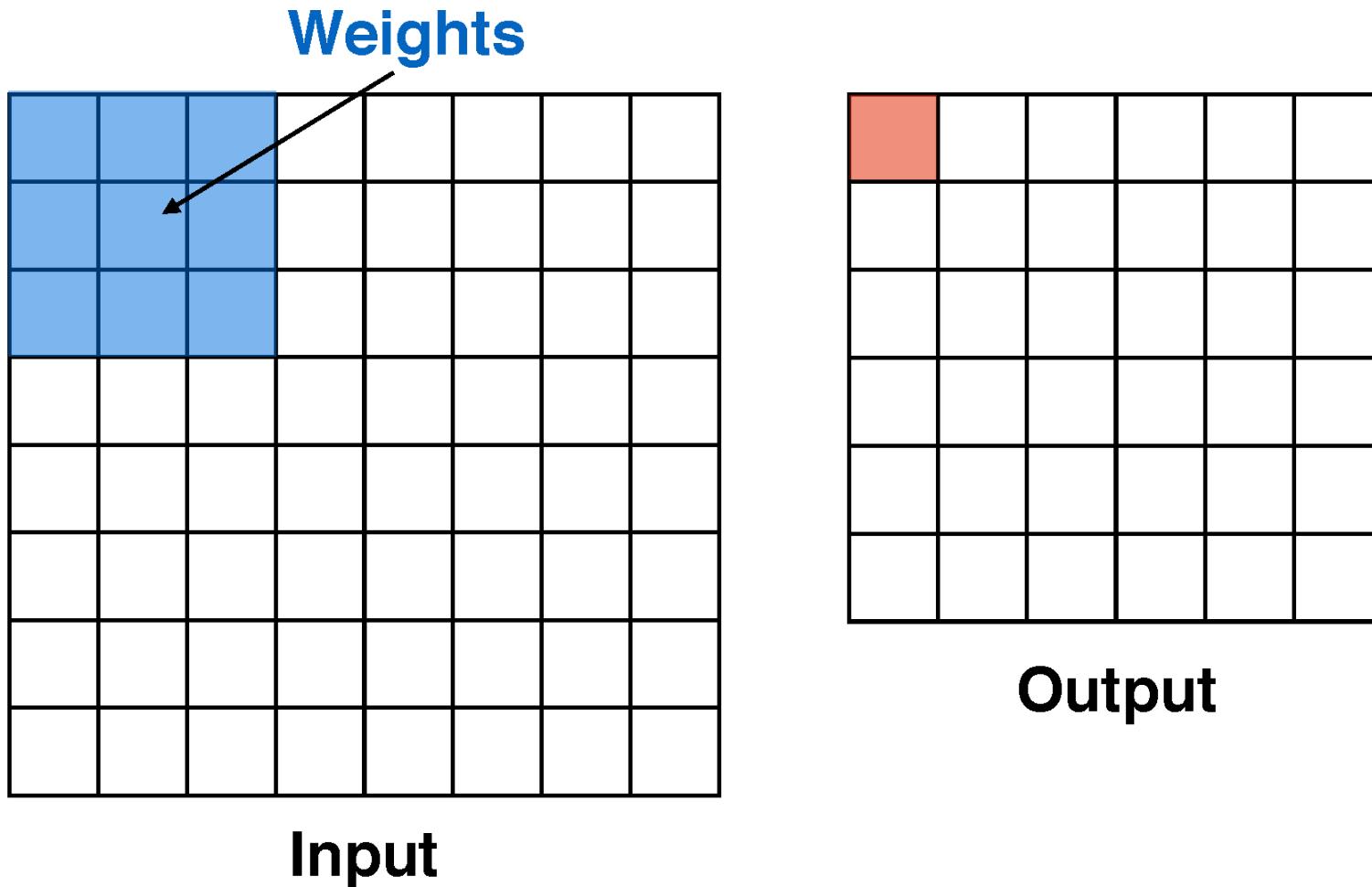
We stack these up to get a "new image" of size 28x28x6!

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

Weights

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

Output

Convolution: Stride

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

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

$$h^r = \sum_{ijk} x^r_{ijk} W_{ijk} + b$$
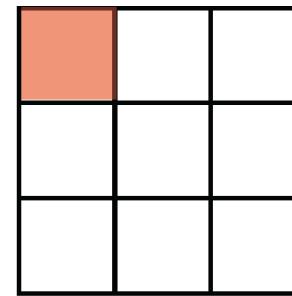
(channel, row, column)
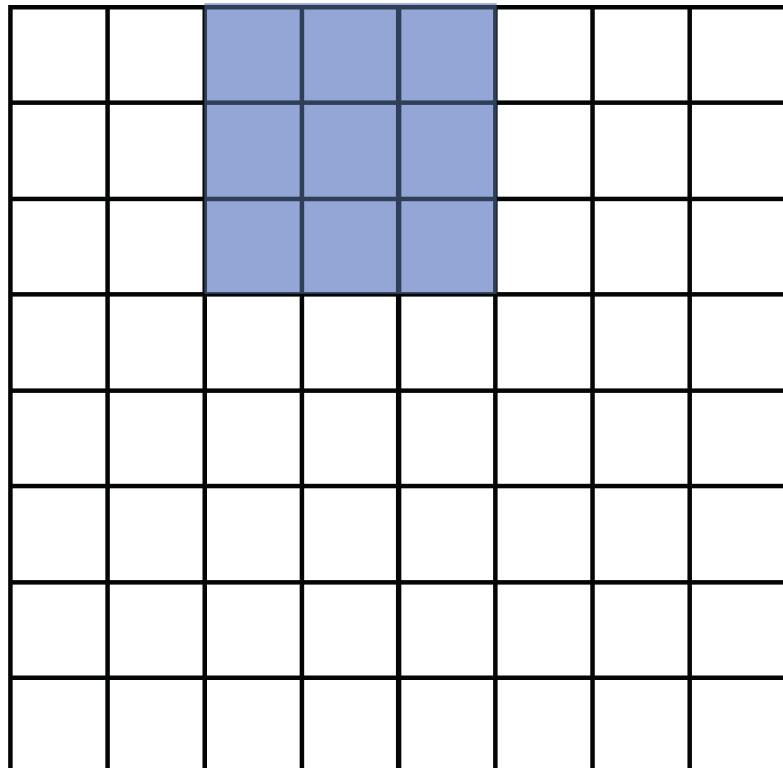
Input

Convolution: Stride

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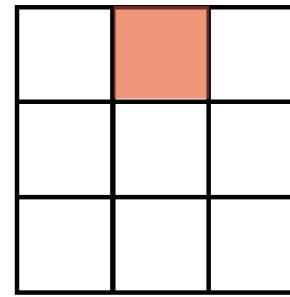
Input

Output

Convolution: Stride

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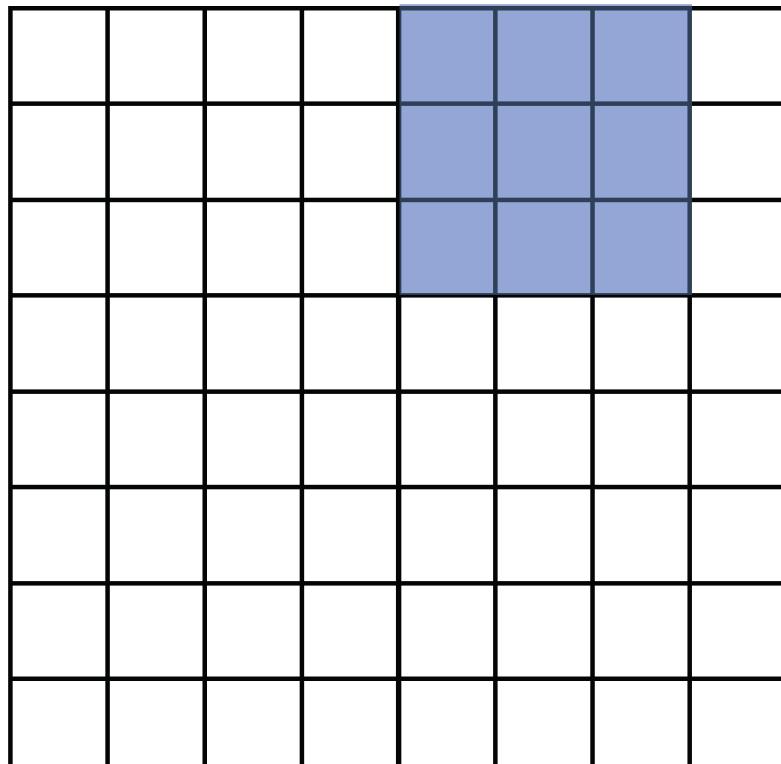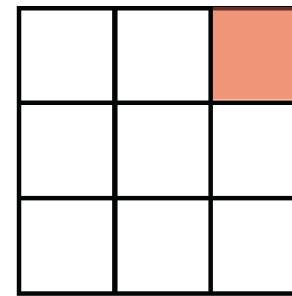
Input

Output

Convolution: Stride

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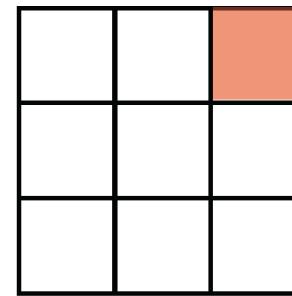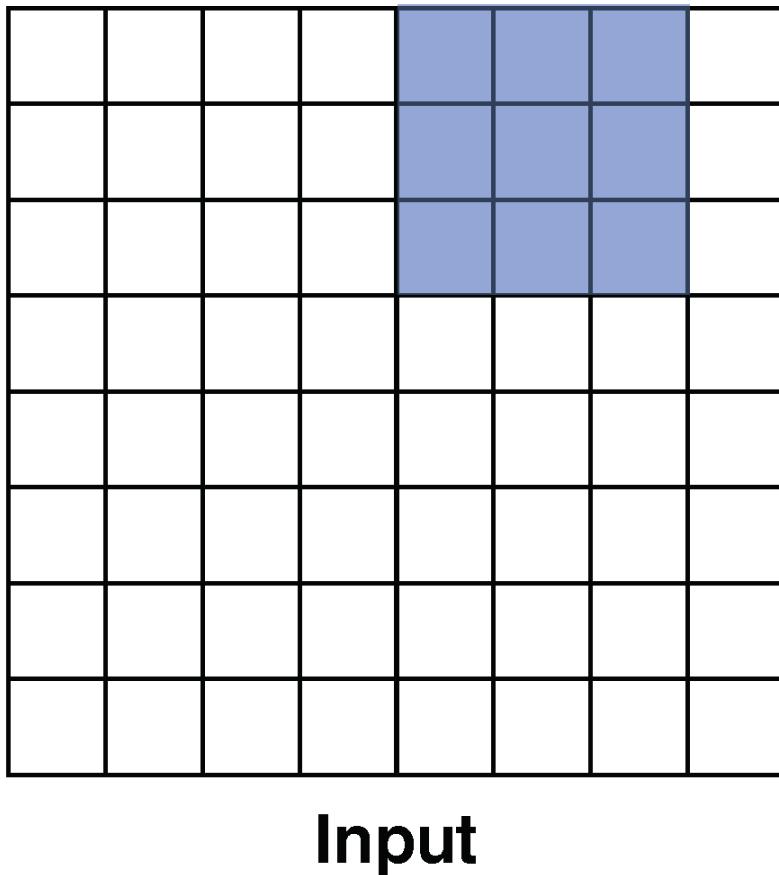
Input

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**

Input

Output

Convolution: Padding

We can also pad the input with zeros.
Here, **pad = 1, stride = 2**

Input

Output

Convolution: Padding

We can also pad the input with zeros.
Here, **pad = 1, stride = 2**

Input

Output

Convolution: Padding

We can also pad the input with zeros.
Here, **pad = 1, stride = 2**

Input

Output

Convolution:
How big is the output?

stride s

kernel k

width w_{in}

p

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?

stride s

kernel k

p width w_{in} p

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

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

$$= \left\lfloor \frac{w_{in} + 2 - 3}{1} \right\rfloor + 1$$

$$= w_{in}$$

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?
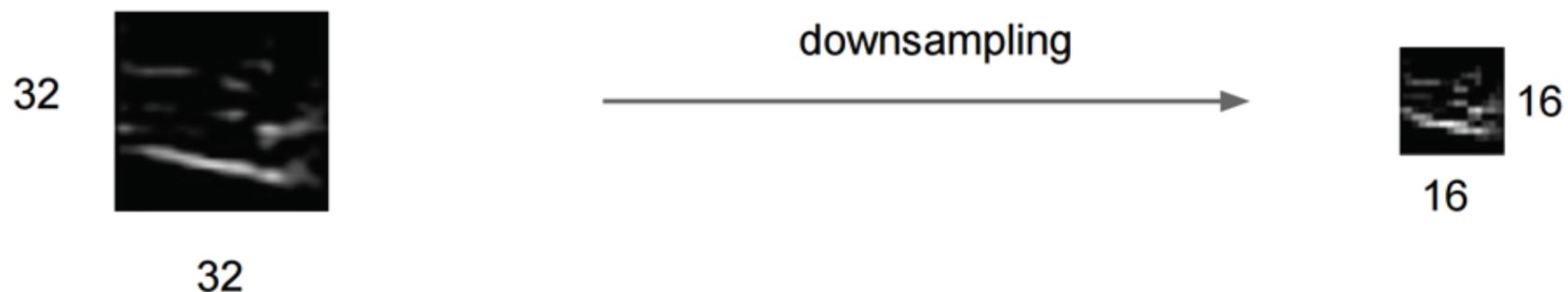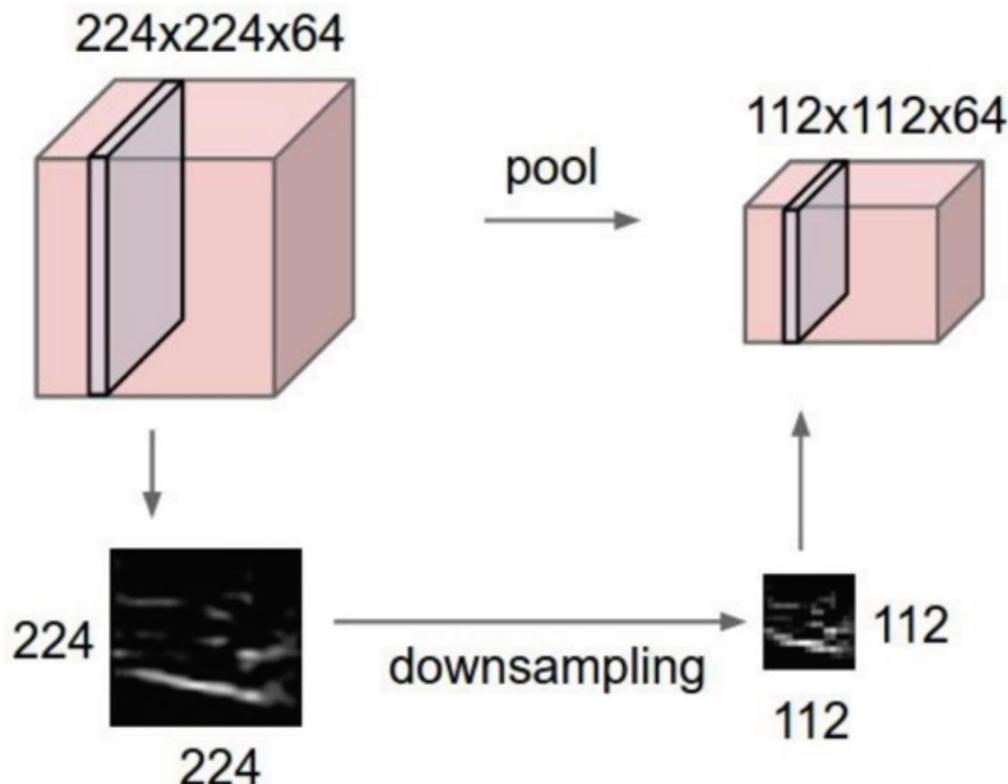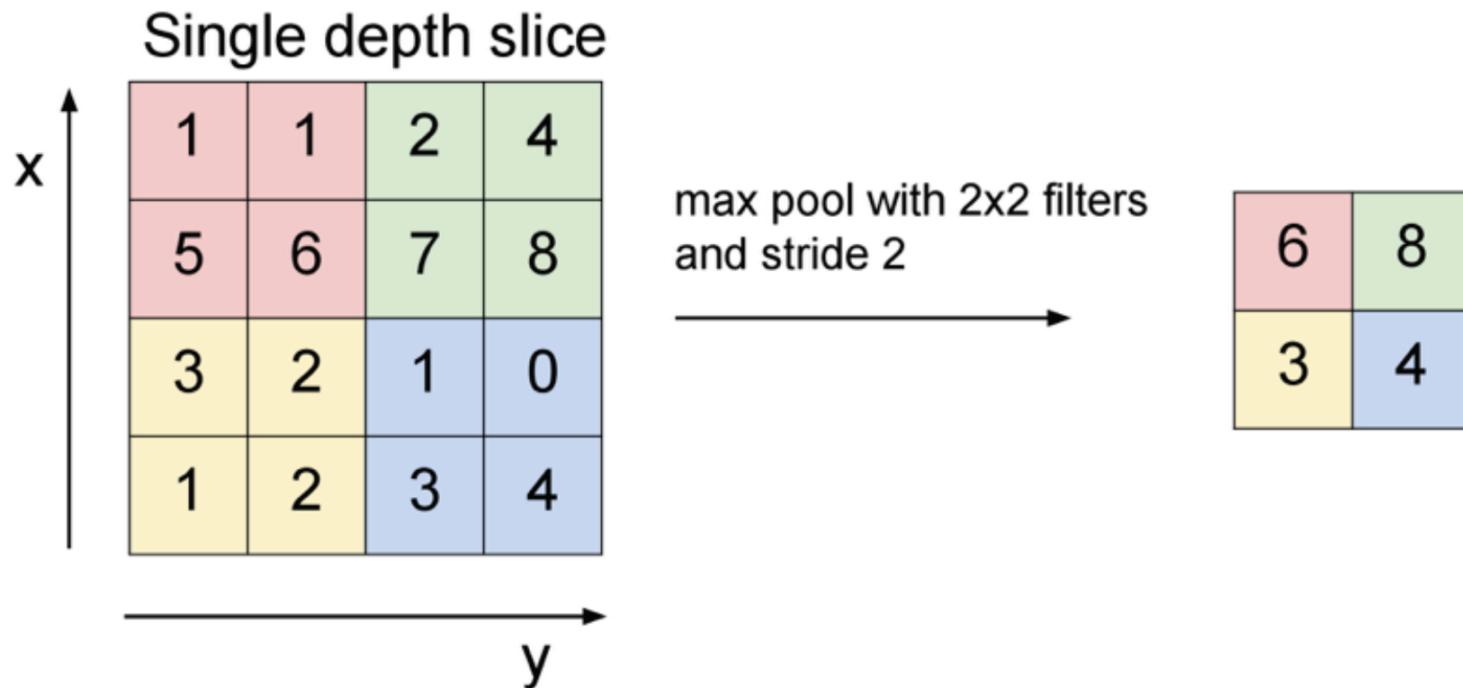
Figure: Andrej Karpathy

Pooling

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

Max Pooling

Single depth slice

max pool with 2x2 filters and stride 2

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

Figure: Andrej Karpathy

Example ConvNet

CONV ReLU CONV ReLU POOL

truck
car
airplane
ship
horse

Figure: Andrej Karpathy
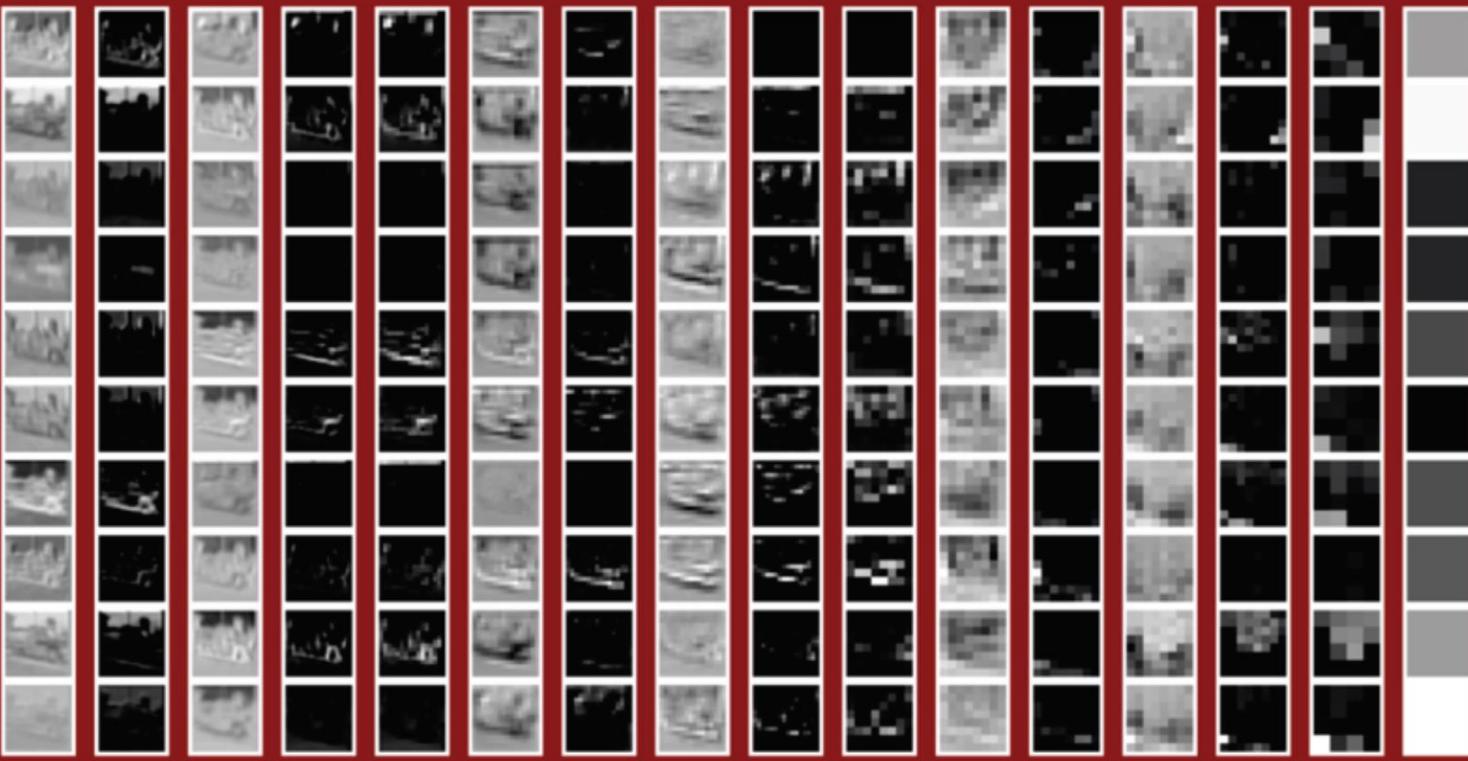
Example ConvNet

Figure: Andrej Karpathy
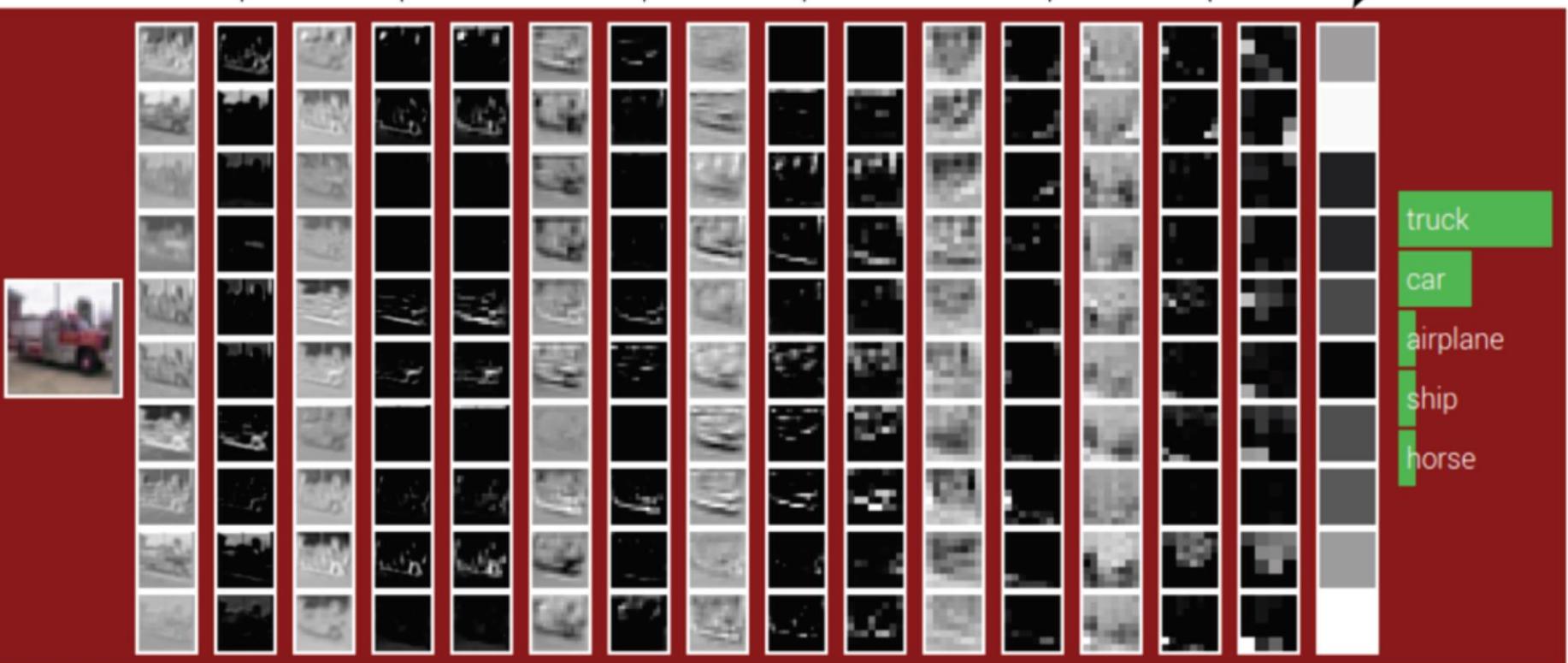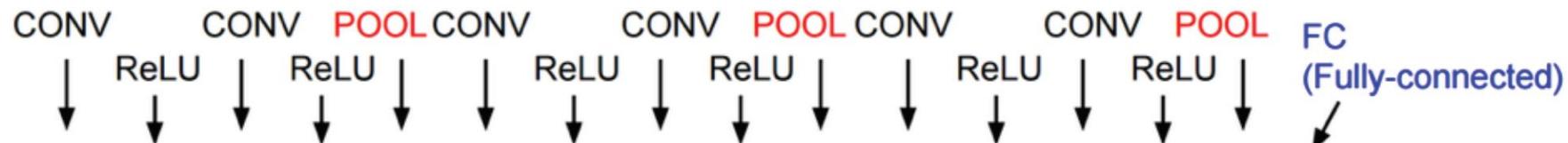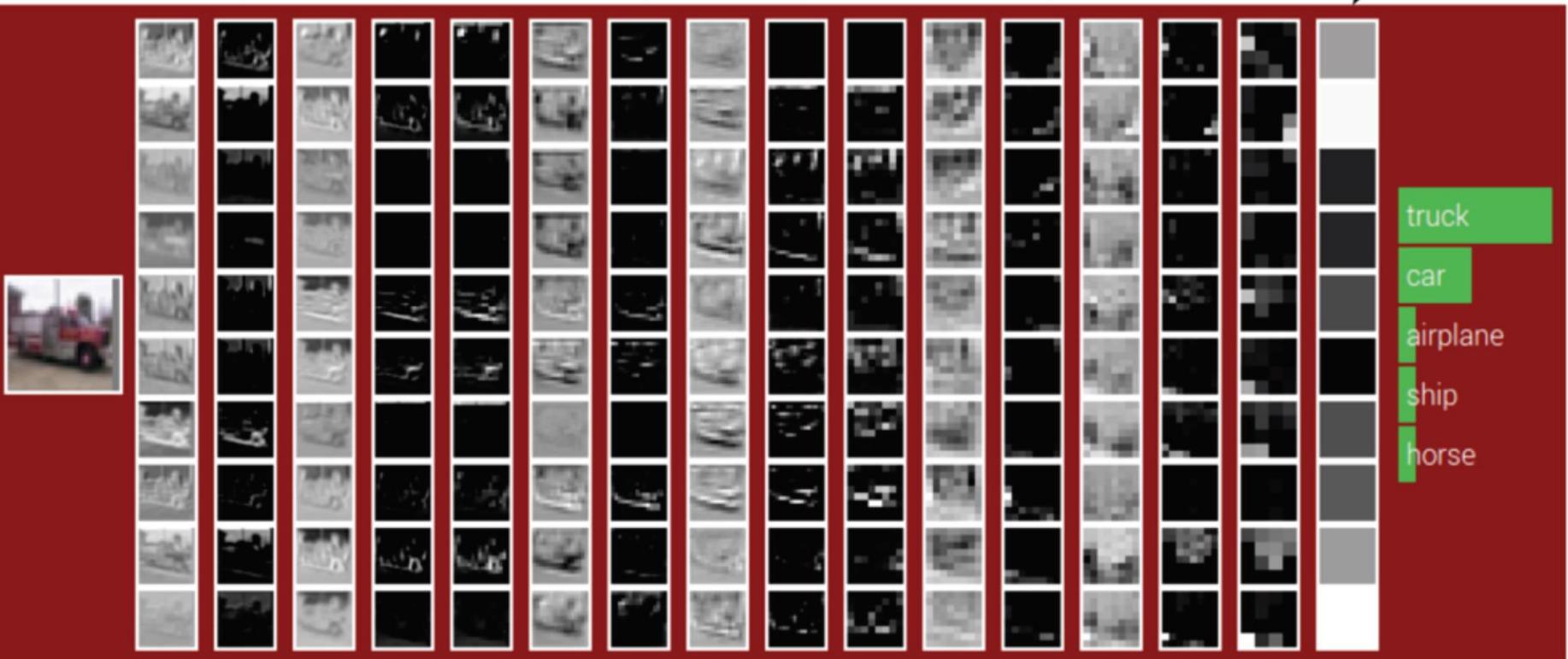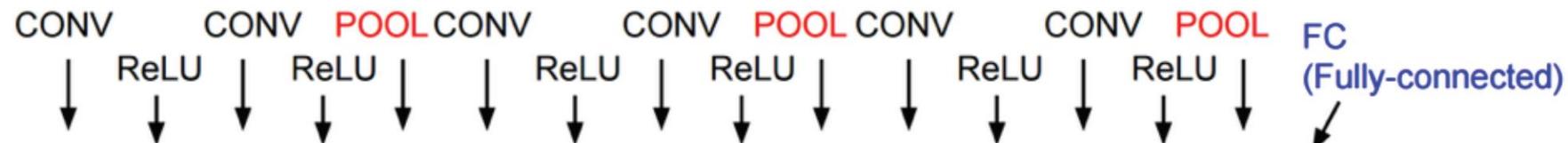
Example ConvNet

Figure: Andrej Karpathy

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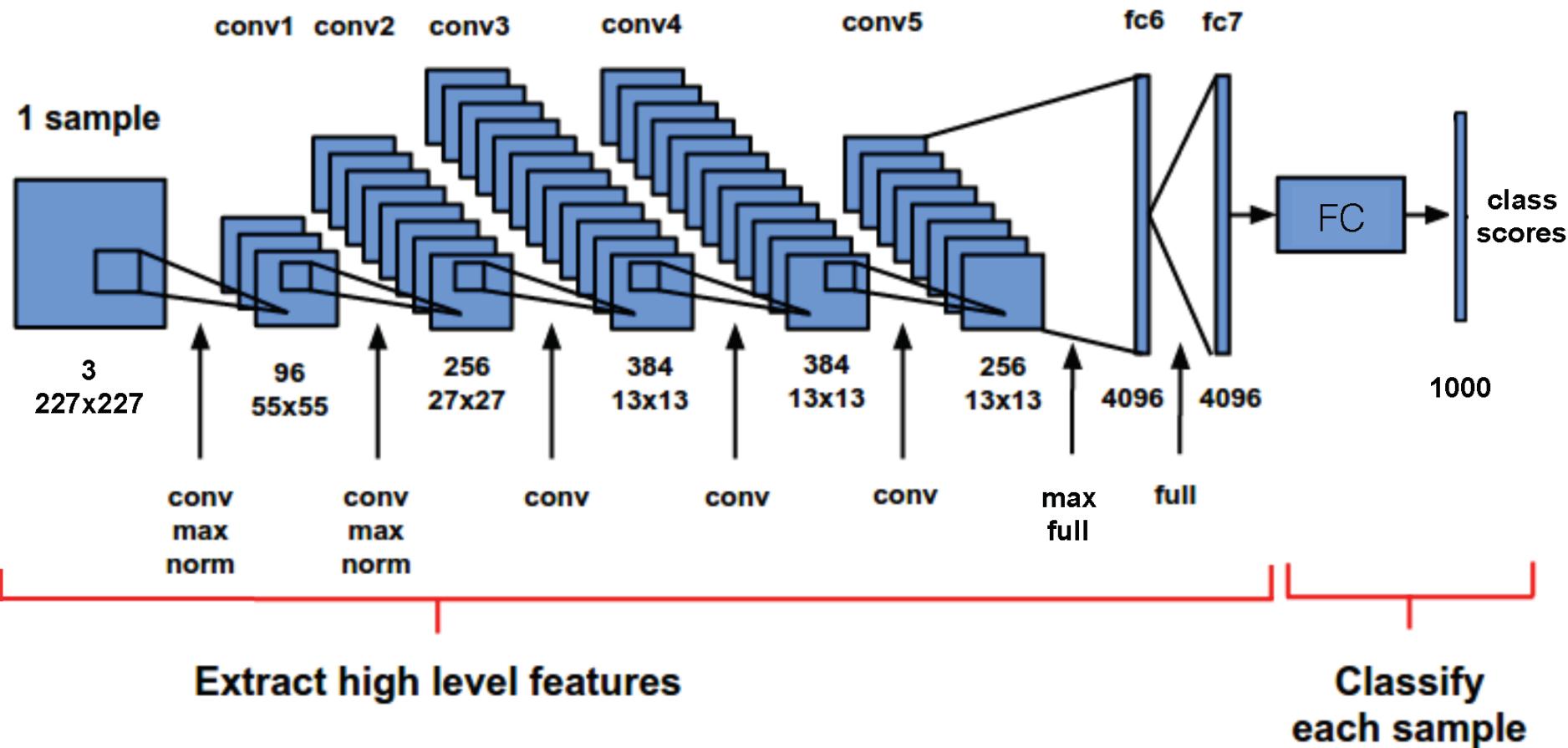
10x3x3 conv filters, stride 1, pad 1
2x2 pool filters, stride 2

Figure: Andrej Karpathy

Example: AlexNet [Krizhevsky 2012]

Figure: [Karnowski 2015] *(with corrections)*
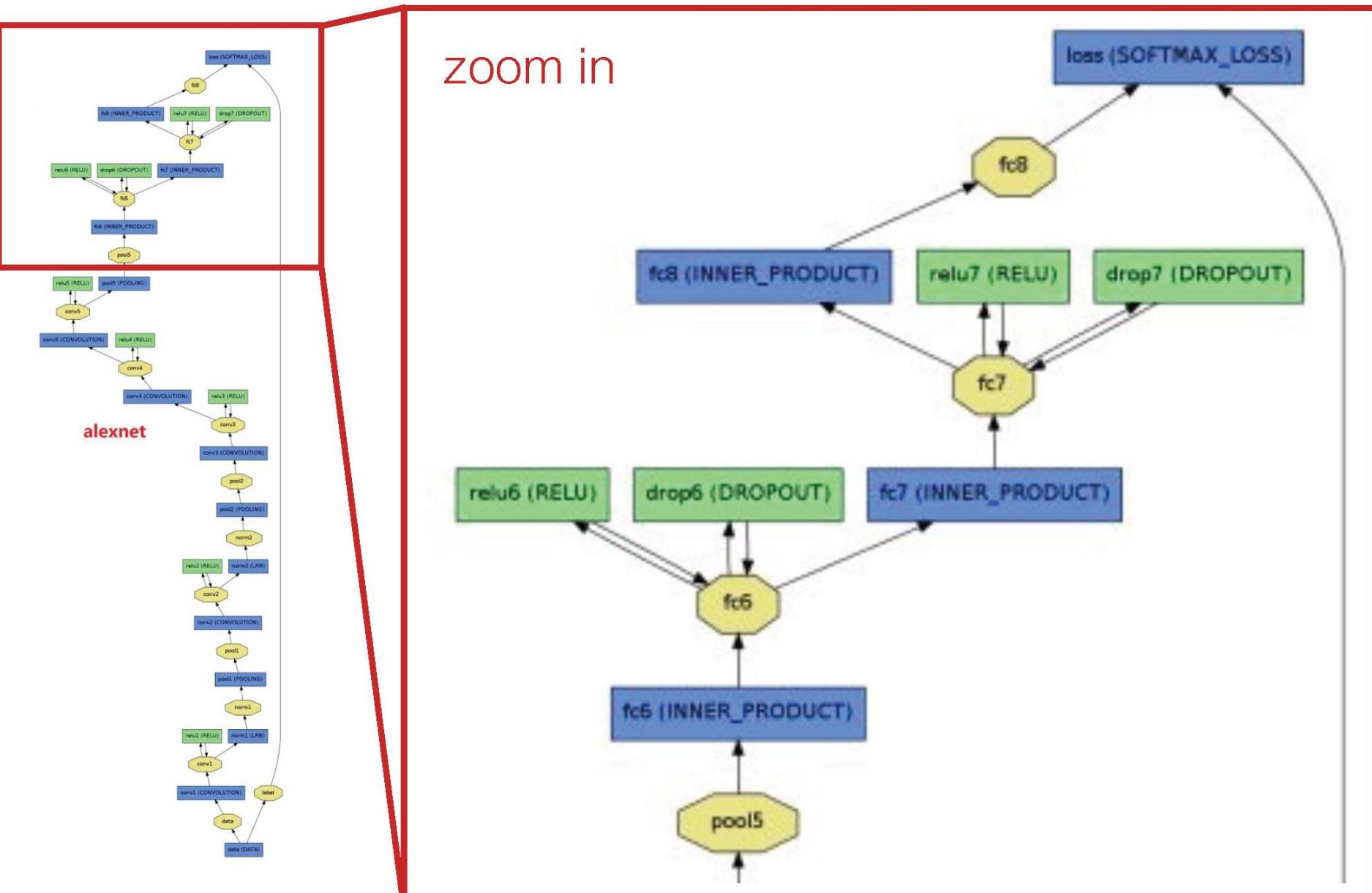
"max": max pooling
"norm": local response normalization
"full": fully connected

Example: AlexNet [Krizhevsky 2012]

zoom in

alexnet
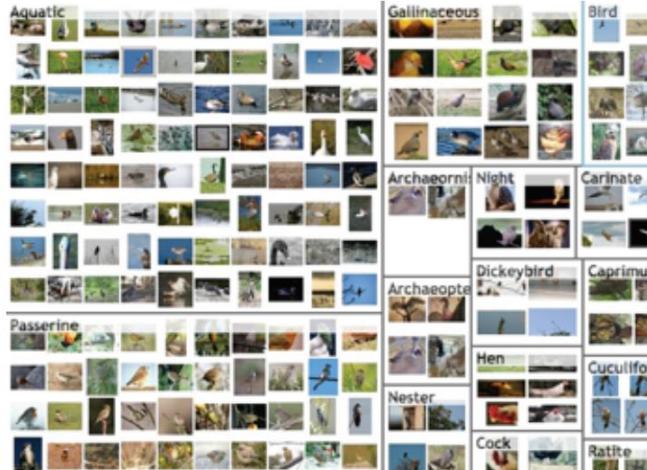
Training ConvNets

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
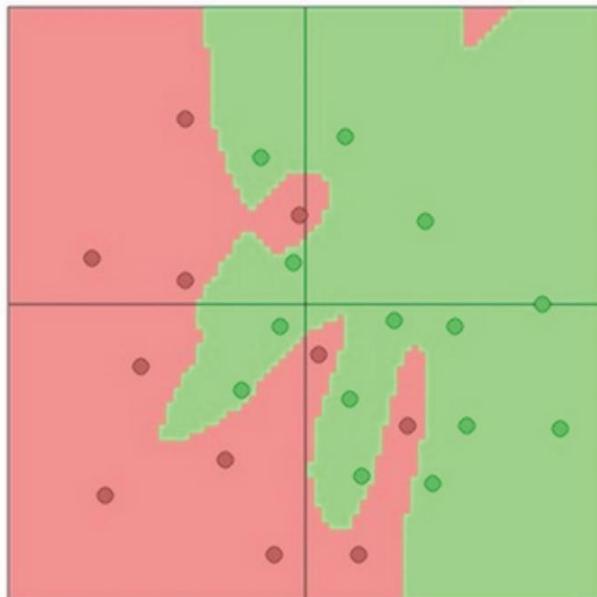
Regularization

Regularization reduces overfitting:

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

[Andrej Karpathy http://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html]
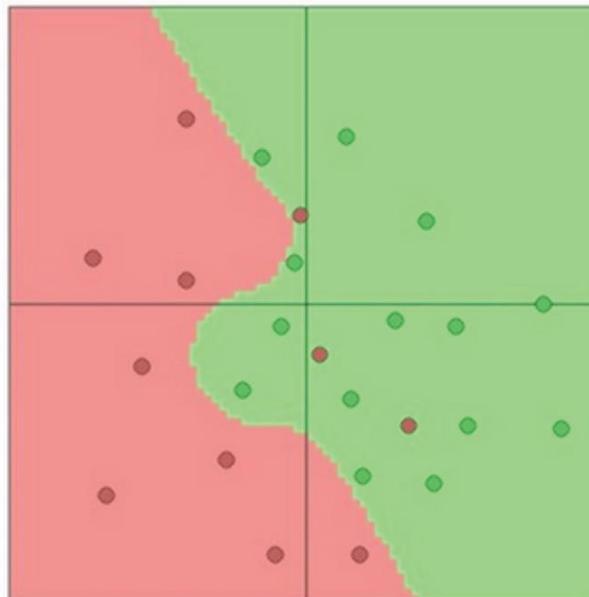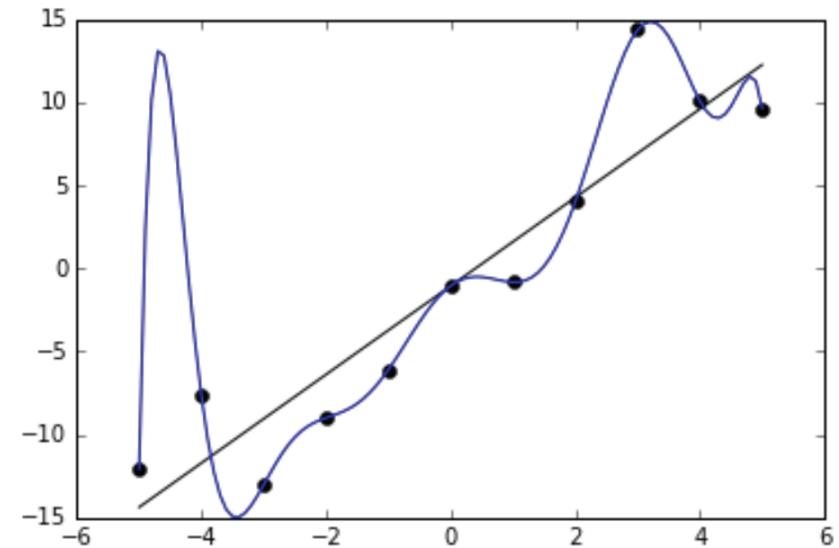
Overfitting

Overfitting: modeling noise in the training set instead of the "true" underlying relationship
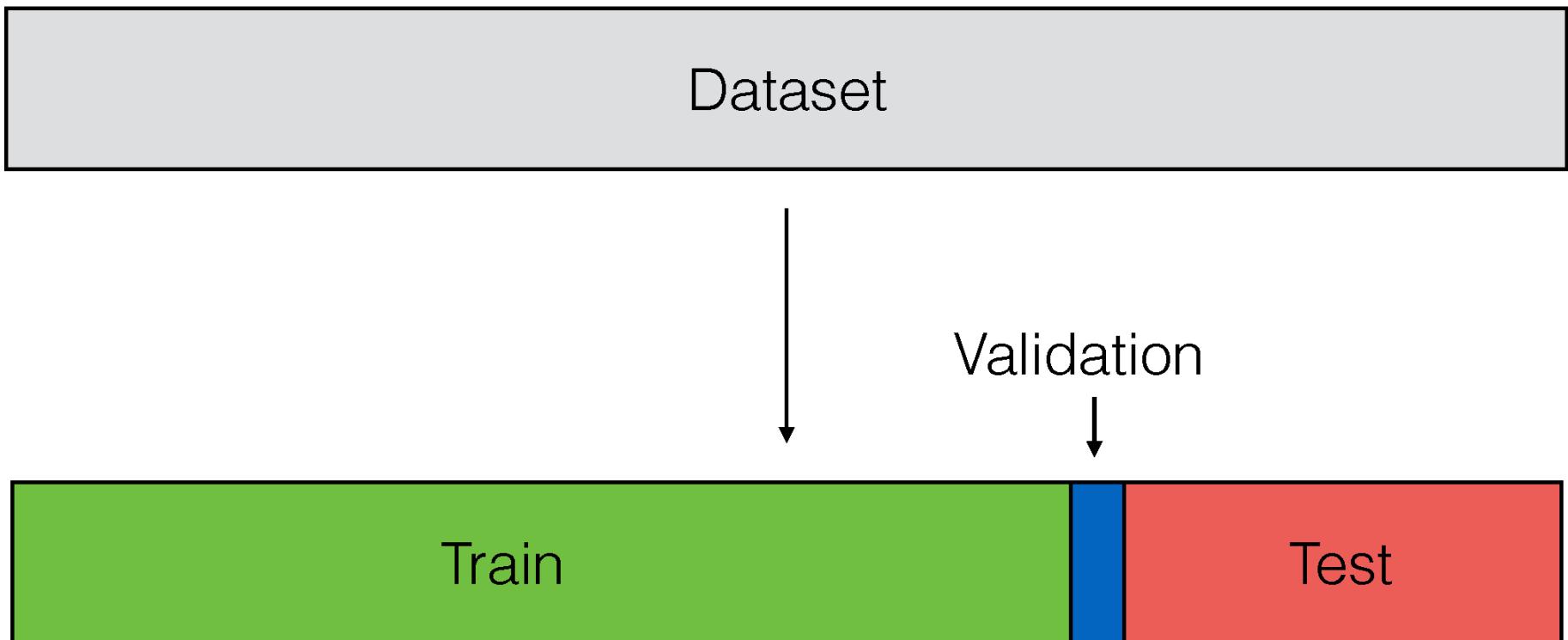
Underfitting: insufficiently modeling the relationship in the training set

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

Validation

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

Validation
↓

Train		Test

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:
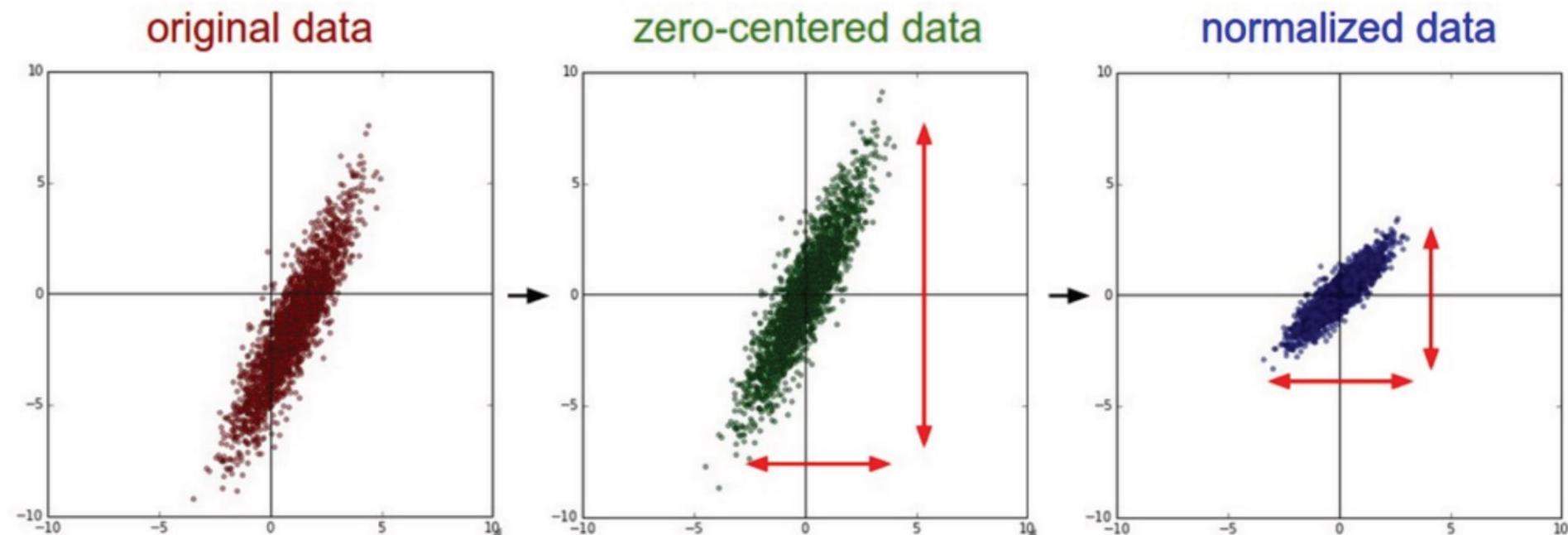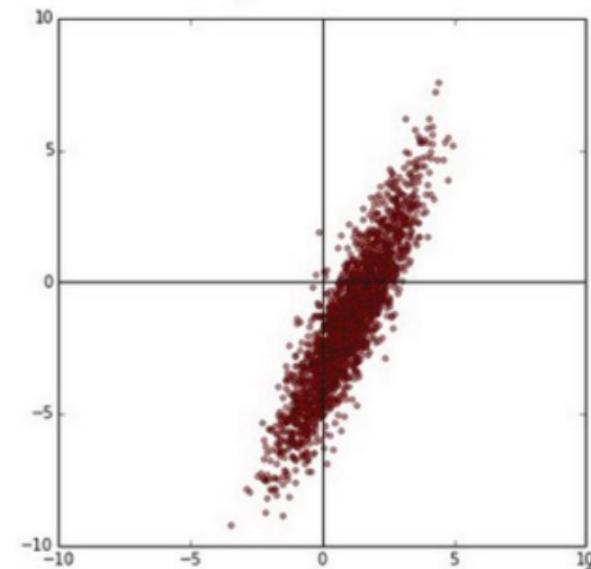
Figure: Andrej Karpathy
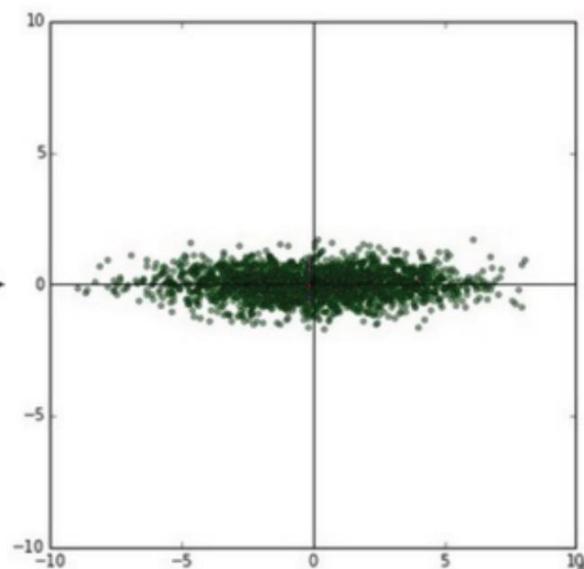
(1) Data preprocessing

In practice, you may also see **PCA** and **Whitening** of the data:
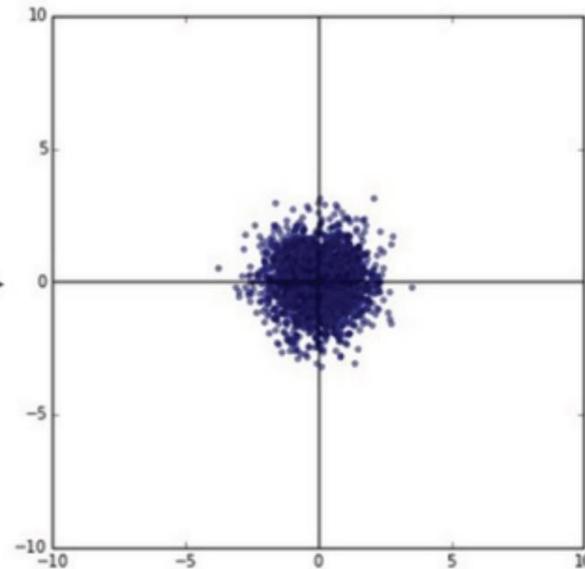
original data   decorrelated data   whitened data
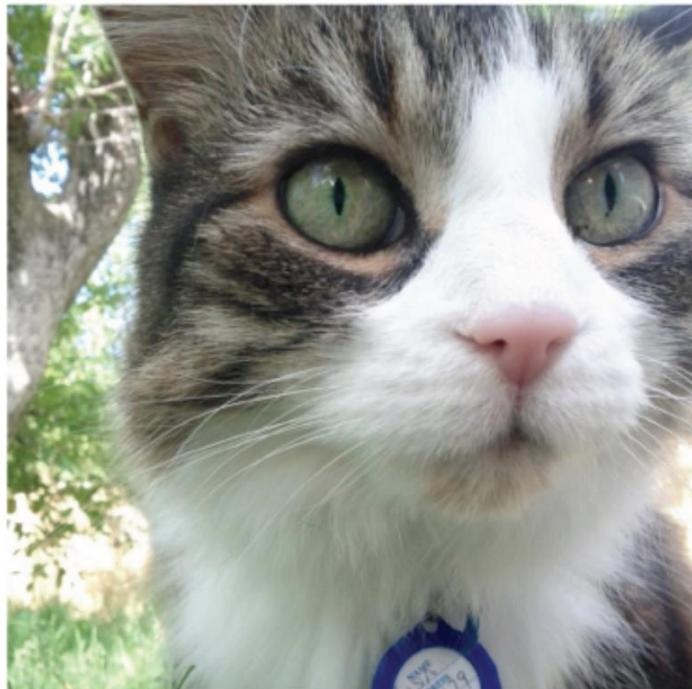
(data has diagonal covariance matrix)   (covariance matrix is the identity matrix)
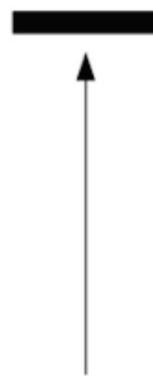
Slide: Andrej Karpathy

(1) Data preprocessing

For ConvNets, typically only the mean is subtracted.

An input image (256x256)   Minus sign   The mean input image
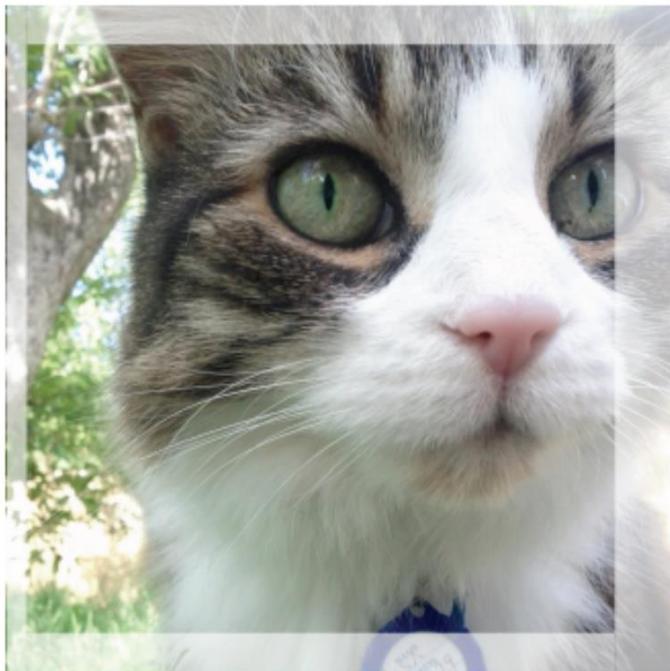
A per-channel mean also works (one value per R,G,B).

(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
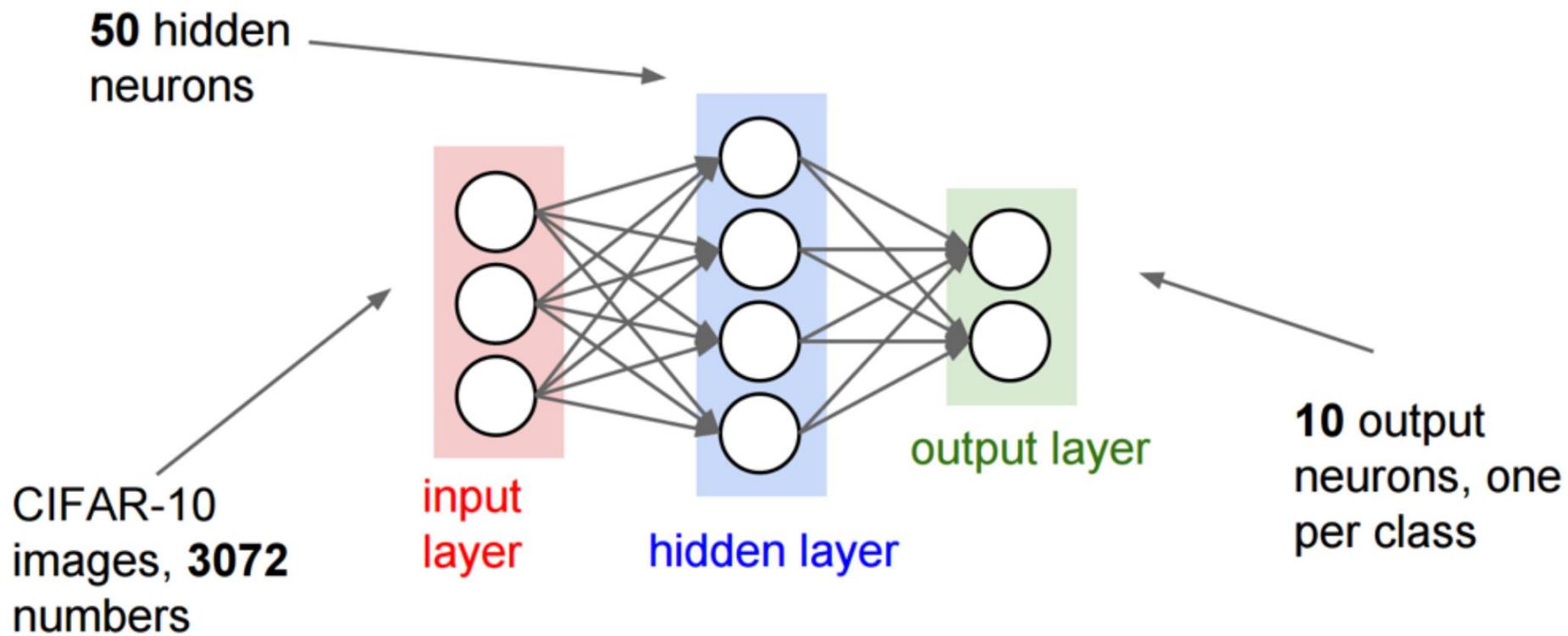
Randomly reflect horizontally

Perform the augmentation live during training

Figure: Alex Krizhevsky

(2) Choose your architecture

Toy example: one hidden layer of size 50

50 hidden neurons

CIFAR-10 images, **3072** numbers

input layer

hidden layer

output layer

10 output neurons, one per class

(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)
```

Slide: Andrej Karpathy

(3) Check that the loss is reasonable

```python
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
```

```python
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)
print loss
```

disable regularization

returns the loss and the
gradient for all parameters

Slide: Andrej Karpathy

(3) Check that the loss is reasonable

```python
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
```

```python
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)
print loss
```

crank up regularization

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

Slide: Andrej Karpathy

(4) Overfit a small portion of the data

100% accuracy on the training set (good)

```
Finished epoch 1 / 200: cost 2.302603, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 2 / 200: cost 2.302258, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 3 / 200: cost 2.301849, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 4 / 200: cost 2.301196, train: 0.650000, val 0.650000, lr 1.000000e-03
Finished epoch 5 / 200: cost 2.300044, train: 0.650000, val 0.650000, lr 1.000000e-03
Finished epoch 6 / 200: cost 2.297864, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 7 / 200: cost 2.293595, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 8 / 200: cost 2.285096, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 9 / 200: cost 2.268094, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 10 / 200: cost 2.234787, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 11 / 200: cost 2.173187, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 12 / 200: cost 2.076862, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 13 / 200: cost 1.974090, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 14 / 200: cost 1.895885, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 15 / 200: cost 1.820876, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 16 / 200: cost 1.737430, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 17 / 200: cost 1.642356, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 18 / 200: cost 1.535239, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 19 / 200: cost 1.421527, train: 0.600000, val 0.600000, lr 1.000000e-03

Finished epoch 195 / 200: cost 0.002694, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 196 / 200: cost 0.002674, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 197 / 200: cost 0.002655, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 198 / 200: cost 0.002635, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 199 / 200: cost 0.002617, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 200 / 200: cost 0.002597, train: 1.000000, val 1.000000, lr 1.000000e-03
finished optimization. best validation accuracy: 1.000000
```

Slide: Andrej Karpathy

(4) Find a learning rate

Let's start with small regularization and find the learning rate that makes the loss decrease:

```python
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)
```

(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
```
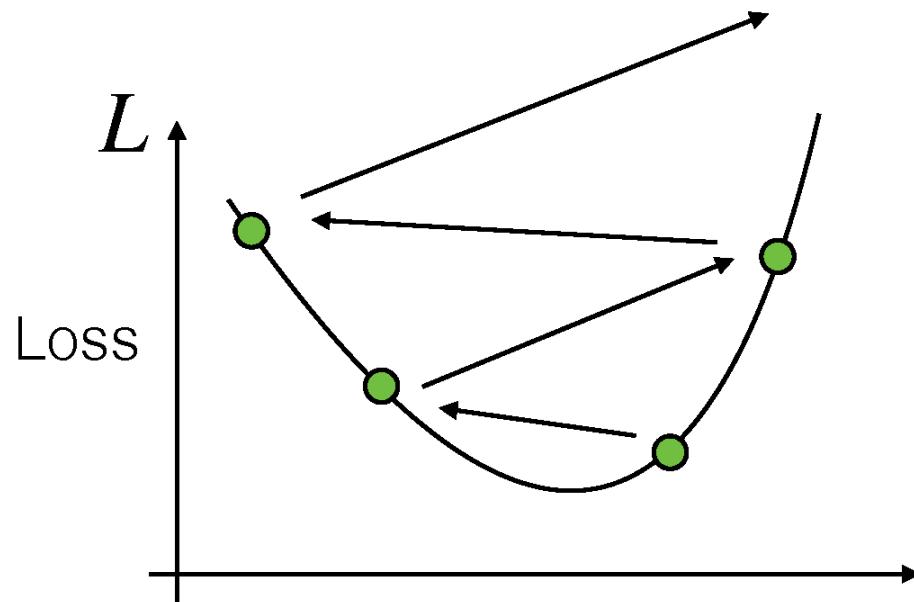
Loss barely changes   **Why is the accuracy 20%?**

(learning rate is too low or regularization too high)

Slide: Andrej Karpathy

(4) Find a learning rate

Learning rate: 1e6 — what could go wrong?

L

Loss

A weight somewhere in the network

(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 * 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
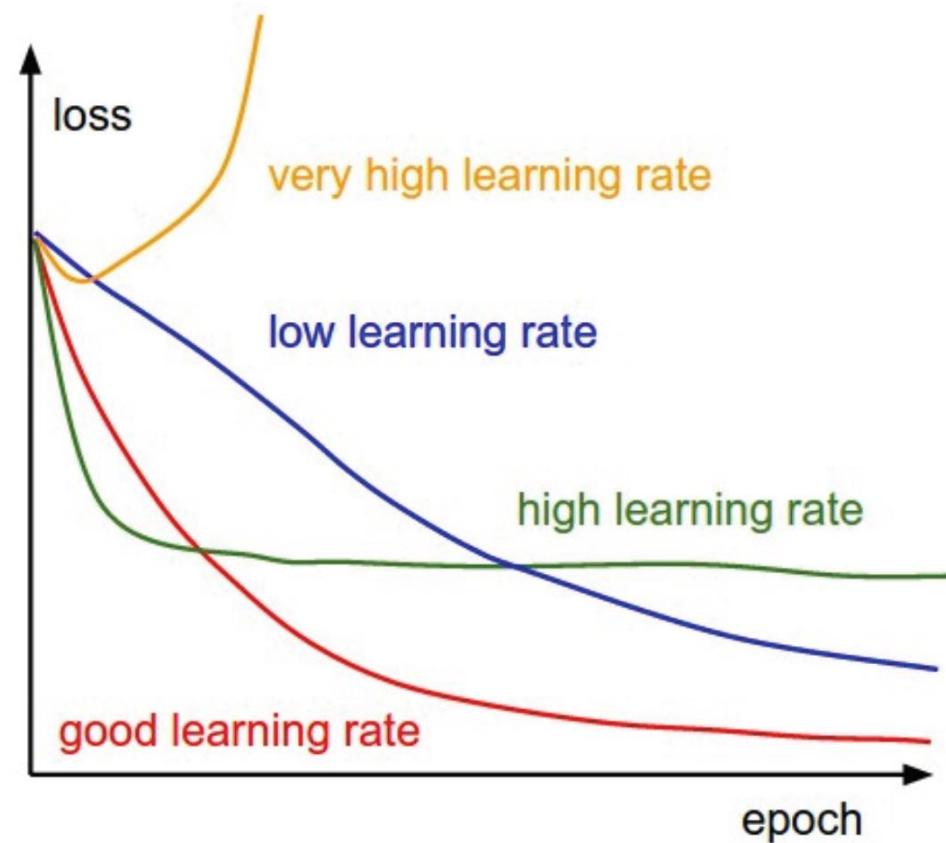
Figure: Andrej Karpathy
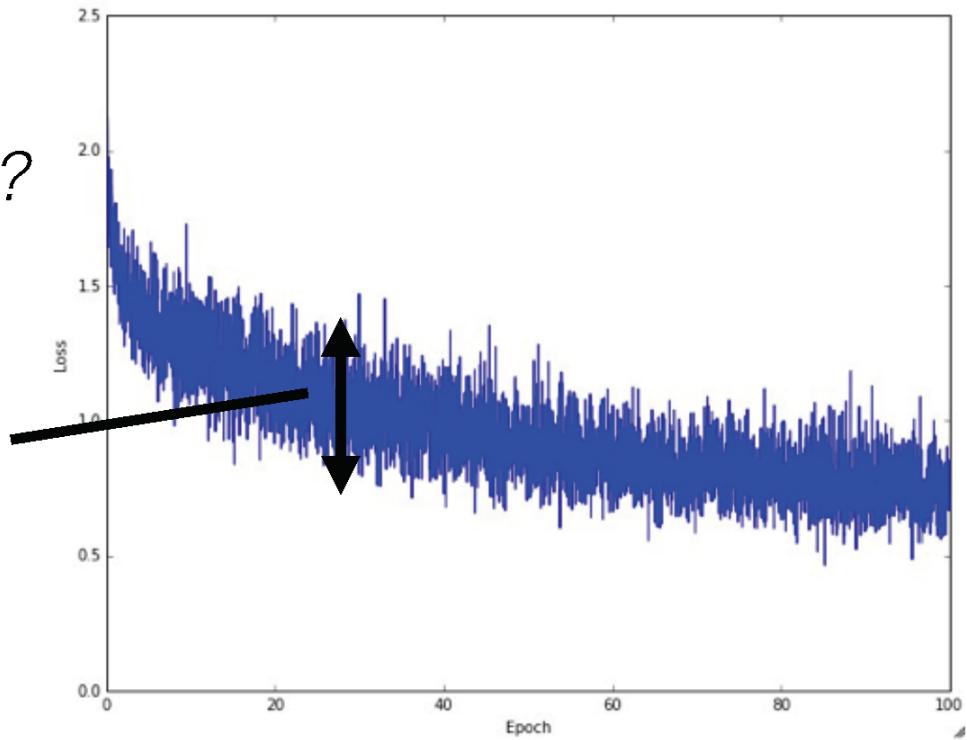
(4) Find a learning rate

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
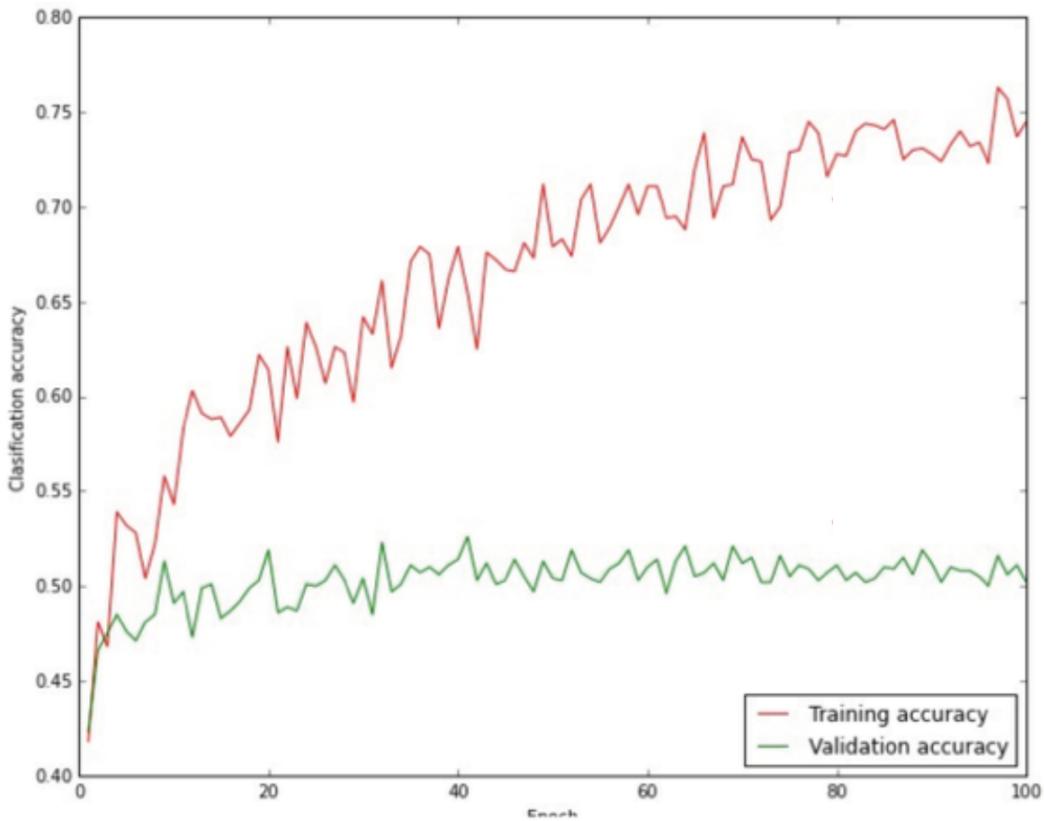
Possibly too linear
(learning rate too small)

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)

Figure: Andrej Karpathy

# (4) Find a learning rate

Visualize the weights

Noisy weights: possibly regularization not strong enough
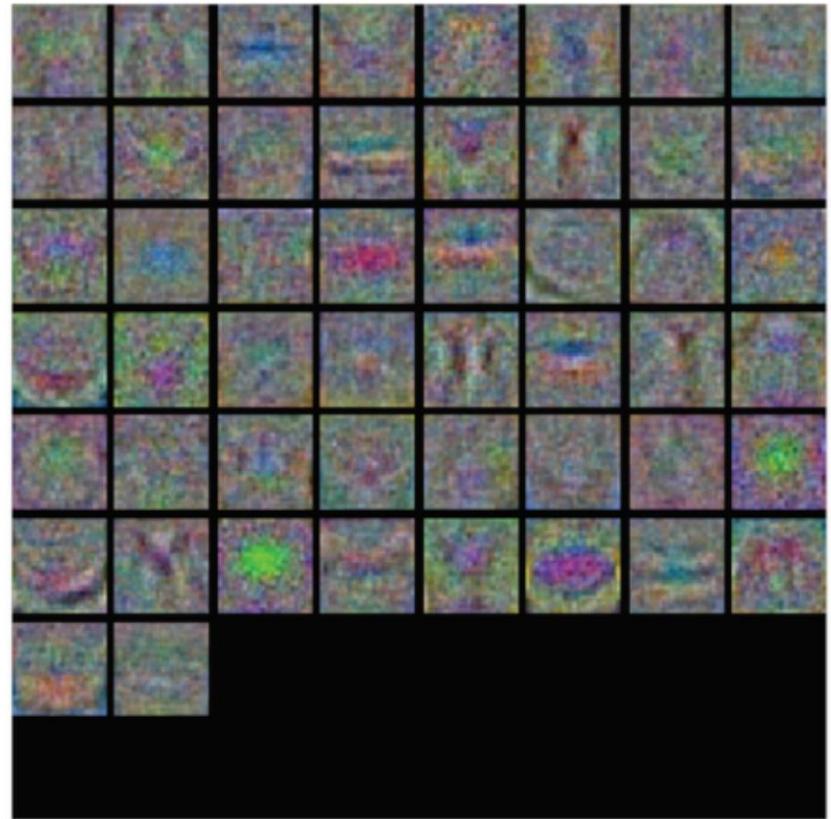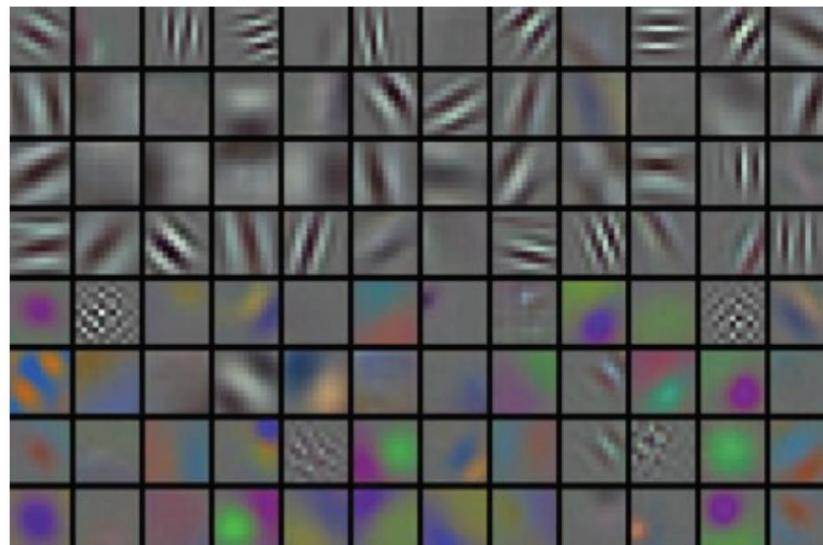
Figure: Andrej Karpathy

(4) Find a learning rate

Visualize the weights

Nice clean weights:
training is proceeding well
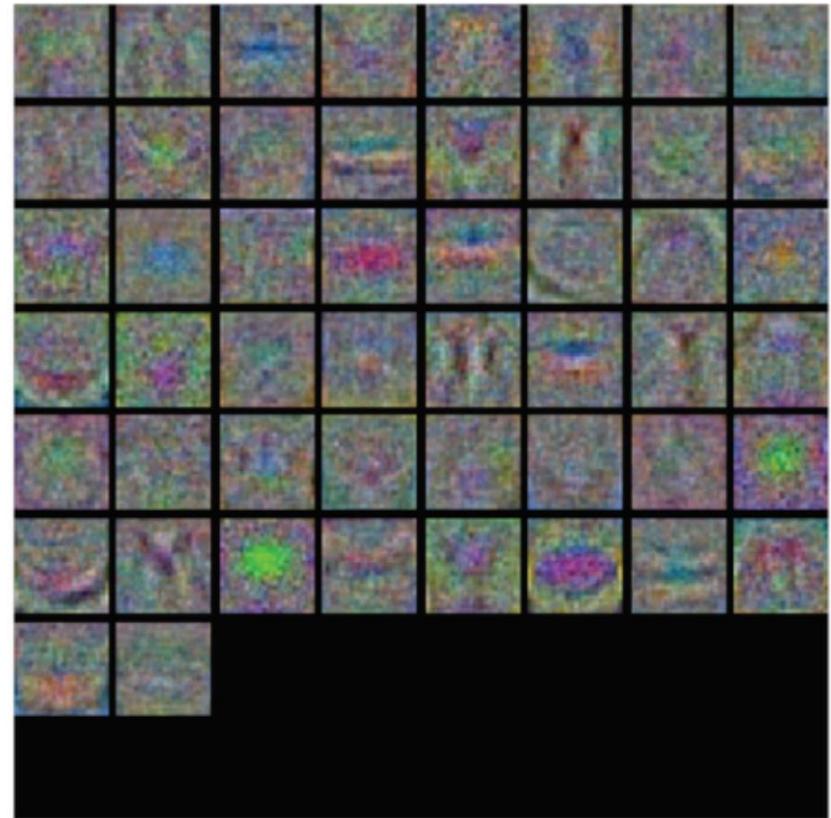
Figure: Alex Krizhevsky , Andrej Karpathy

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/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
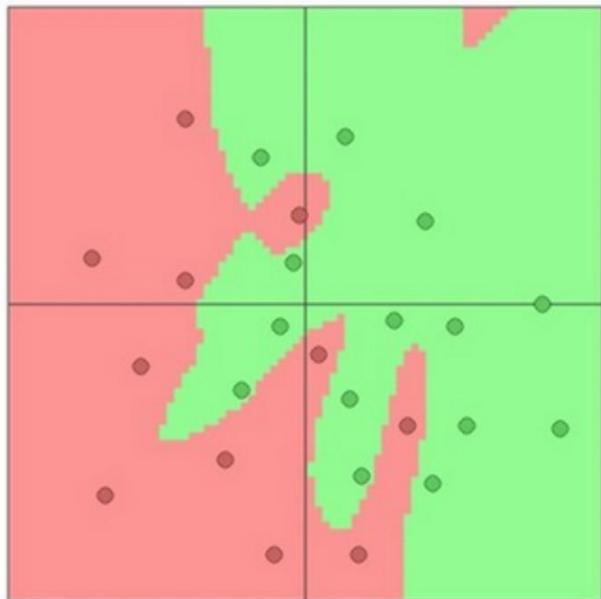
(Recall) Regularization reduces overfitting
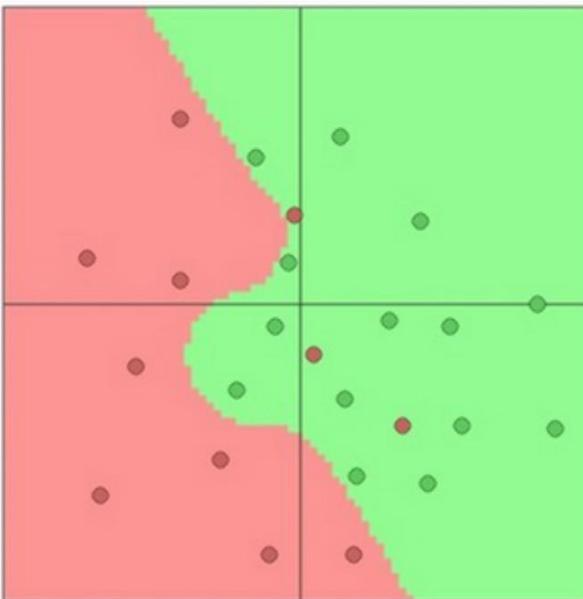
$$L = L_{\text{data}} + L_{\text{reg}} \qquad\qquad L_{\text{reg}} = \lambda \frac{1}{2} \|W\|_2^2$$

$\lambda = 0.001$ $\lambda = 0.01$ $\lambda = 0.1$

[Andrej Karpathy http://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html]
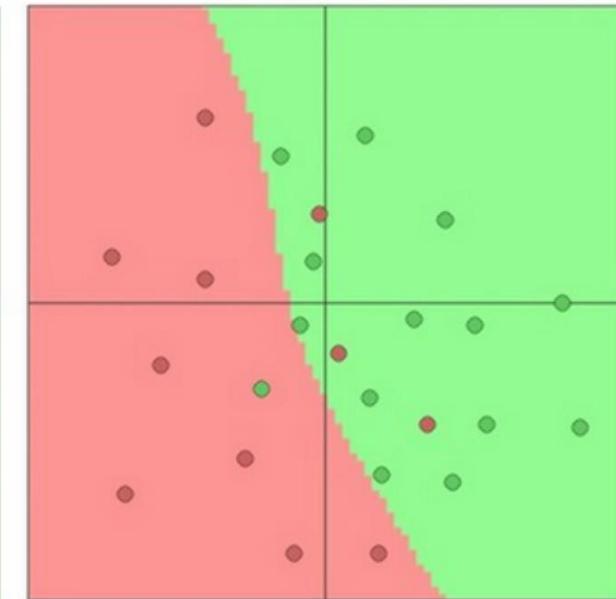
Example Regularizers

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:

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

Gradient descent step:
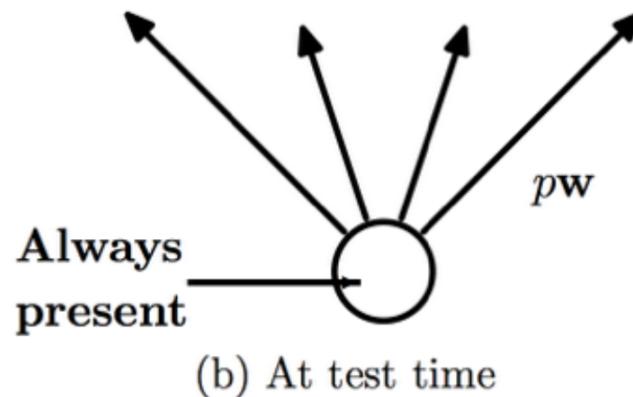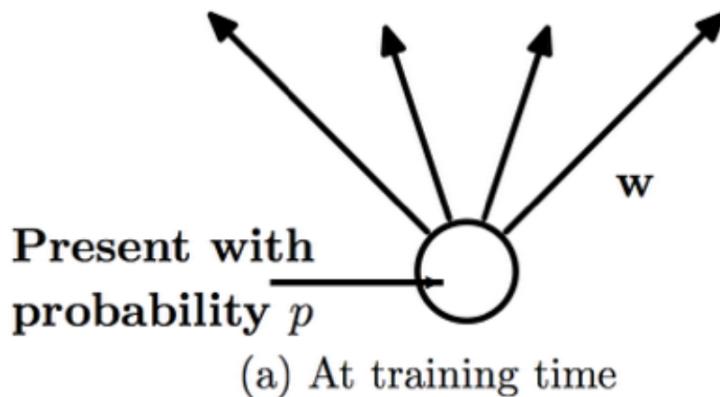
$$W \leftarrow W - \alpha\lambda W - \frac{\partial L_{\text{data}}}{\partial W}$$

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

Note: biases are sometimes excluded from regularization

Dropout
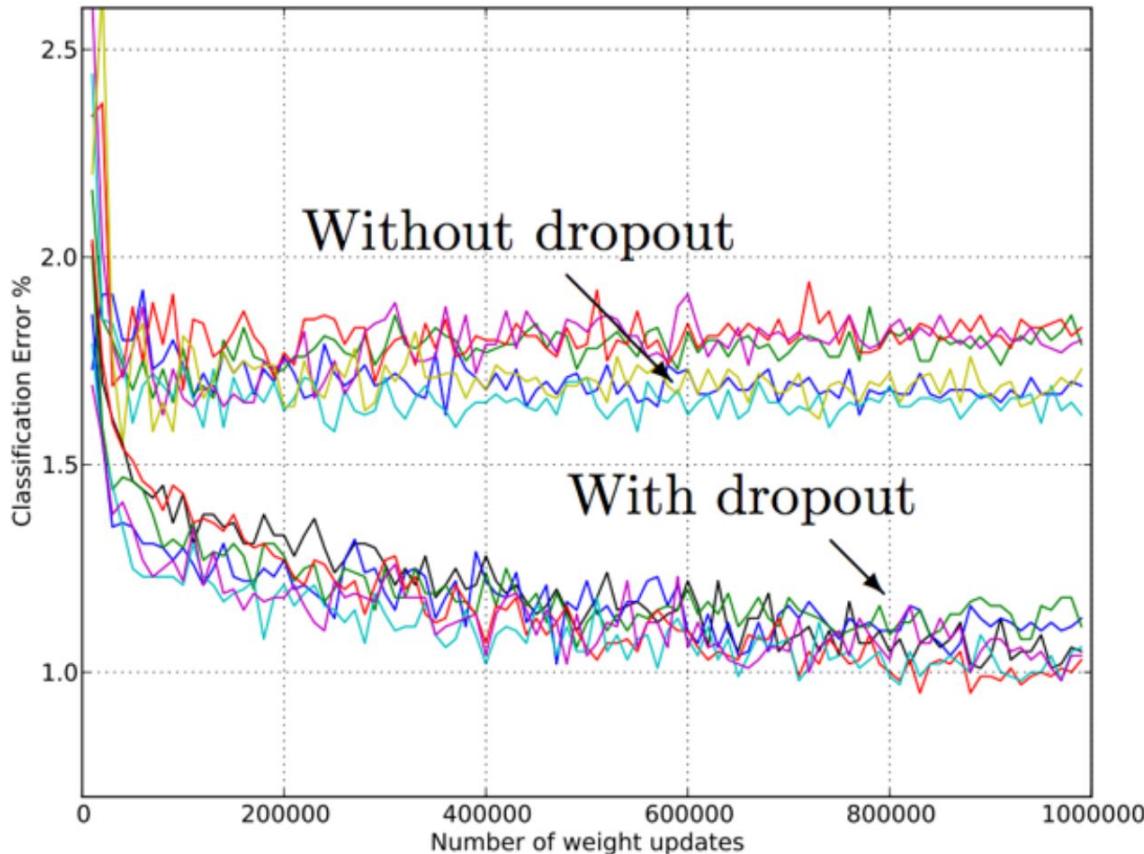
**Simple but powerful technique to reduce overfitting:**

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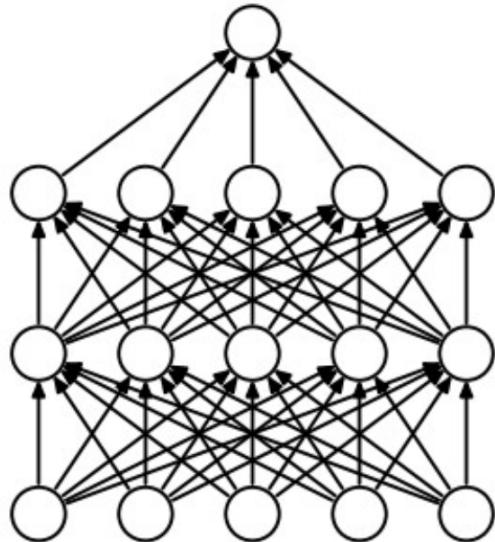
Dropout

Simple but powerful technique to reduce overfitting:
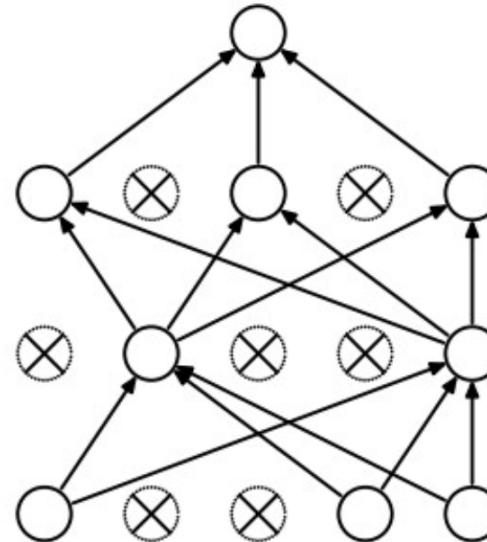

[Srivasta et al, "Dropout: A Simple Way to Prevent Neural Networks from Overfitting", JMLR 2014]

Dropout

Simple but powerful technique to reduce overfitting:


(a) Standard Neural Net
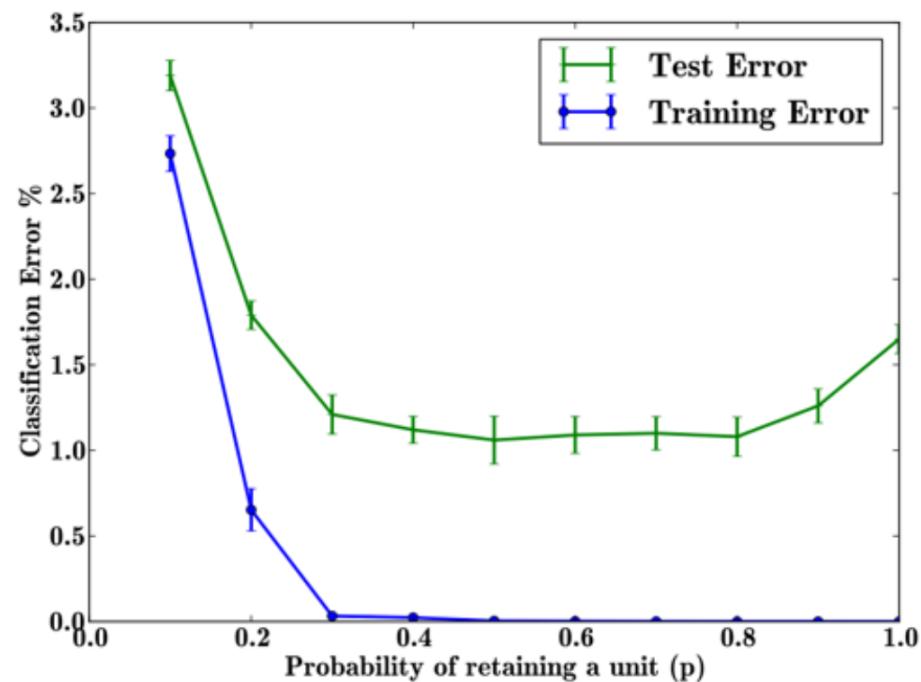
(b) After applying dropout.

Note: Dropout can be interpreted as an approximation to taking the geometric mean of an ensemble of exponentially many models
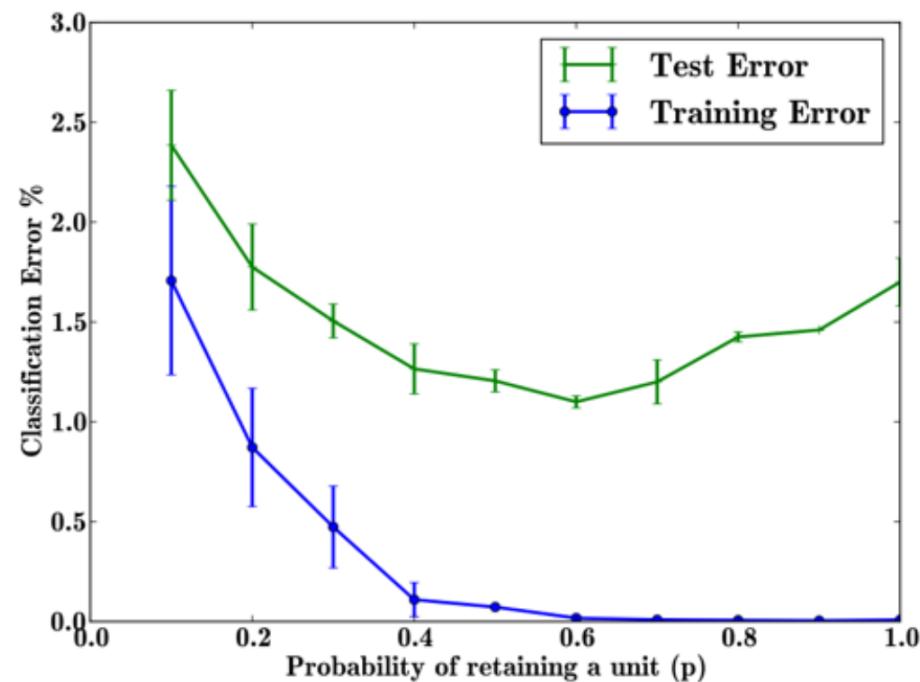
[Srivasta et al, "Dropout: A Simple Way to Prevent Neural Networks from Overfitting", JMLR 2014]

Dropout

How much dropout?   Around p = 0.5

(a) Keeping *n* fixed.

(b) Keeping *pn* fixed.
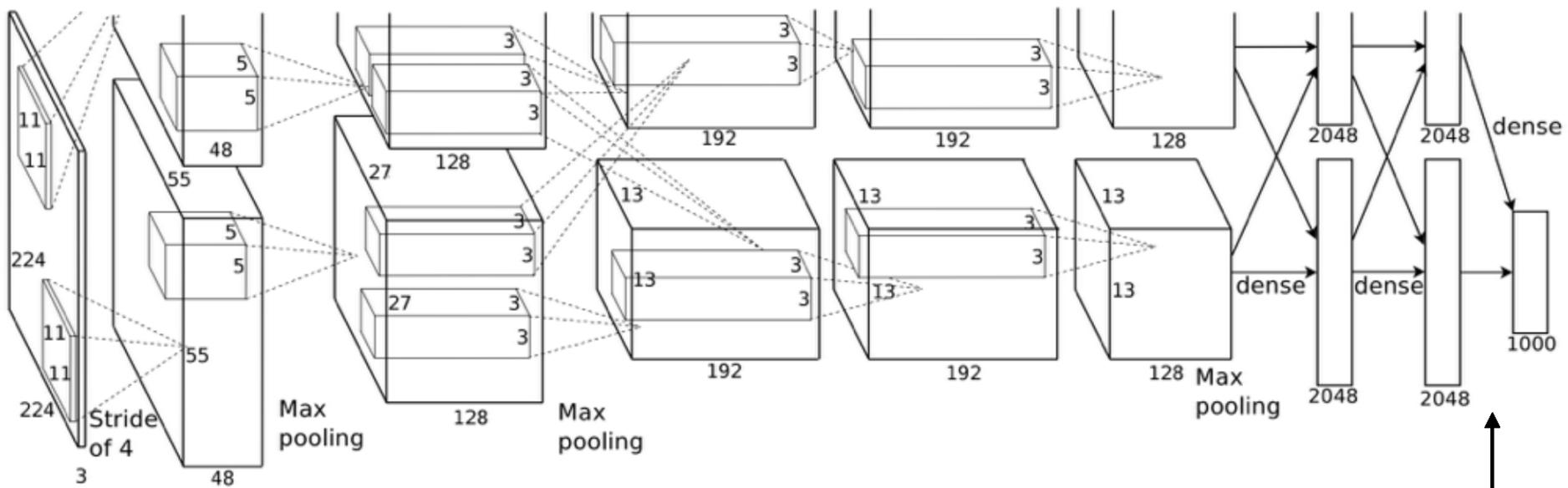
[Srivasta et al, "Dropout: A Simple Way to Prevent Neural Networks from Overfitting", JMLR 2014]

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

```python
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)
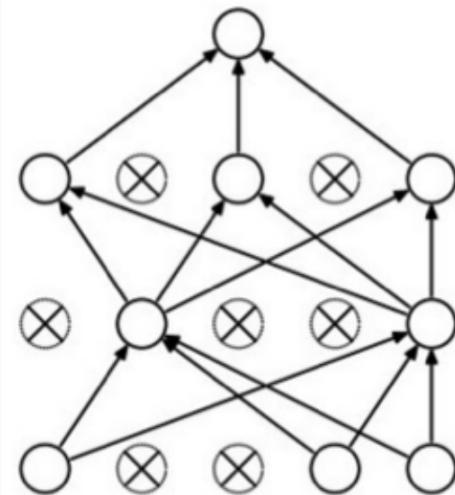
Figure: Andrej Karpathy

Dropout

Test time: scale the activations

Expected value of a neuron h with dropout:

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

```python
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

Figure: Andrej Karpathy

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

References

Basic reading: No standard textbooks yet! Some good resources:

- https://sites.google.com/site/deeplearningsummerschool/
- http://www.deeplearningbook.org/
- http://www.cs.toronto.edu/~hinton/absps/NatureDeepReview.pdf