# CS5670: Computer Vision

## Noah Snavely

## Lecture 26: CNN Structure and Training

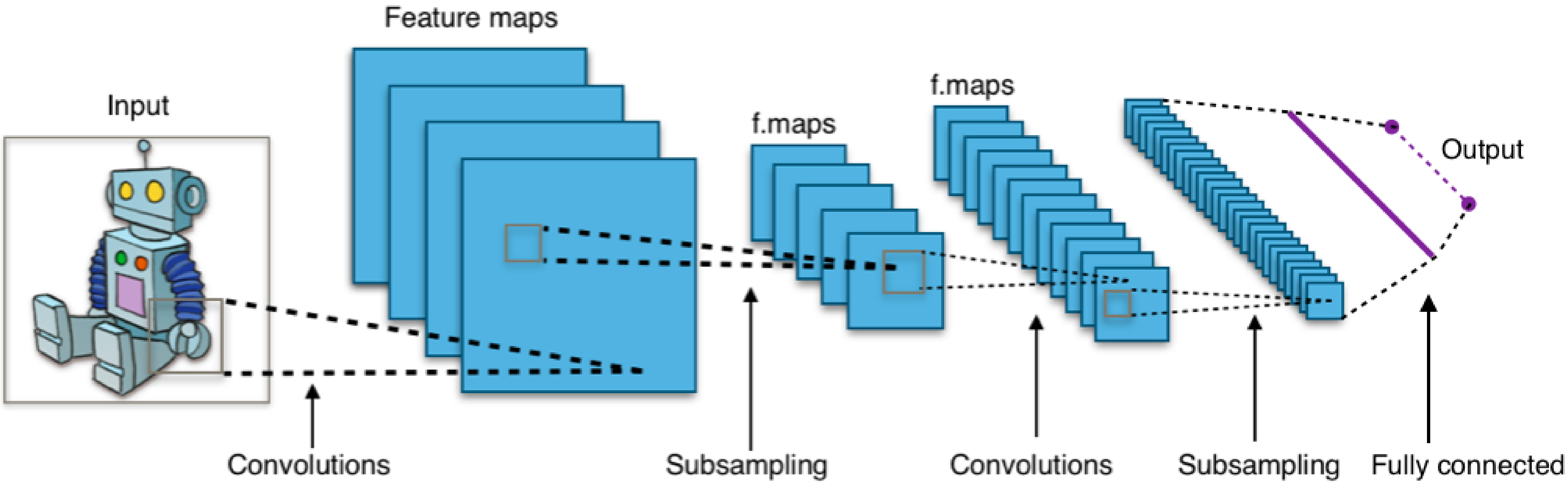
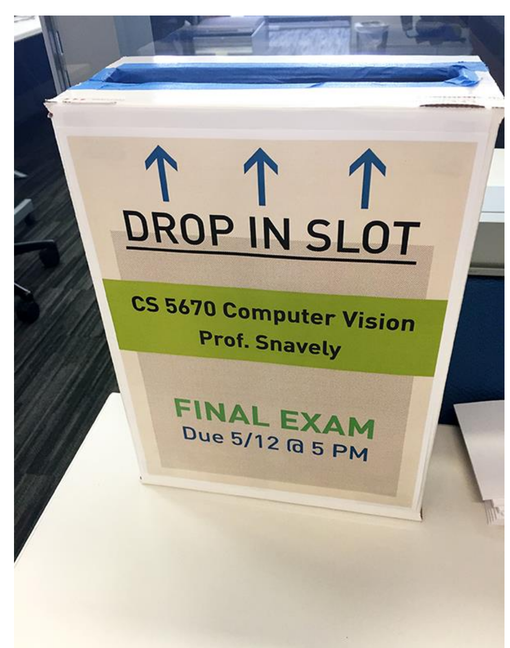



Image credit: Aphex34, [CC BY-SA 4.0 (http://creativecommons.org/licenses/by-sa/4.0)]

Slides from Andrej Karpathy and Fei-Fei Li
http://vision.stanford.edu/teaching/cs231n/

# Announcements

- Final project (P5), due **Wednesday, 5/10**, by 11:59pm
- Final exam will be handed out at the end of class today, due Friday, 5/12, by 5pm to Christina Ko's desk on 12th floor

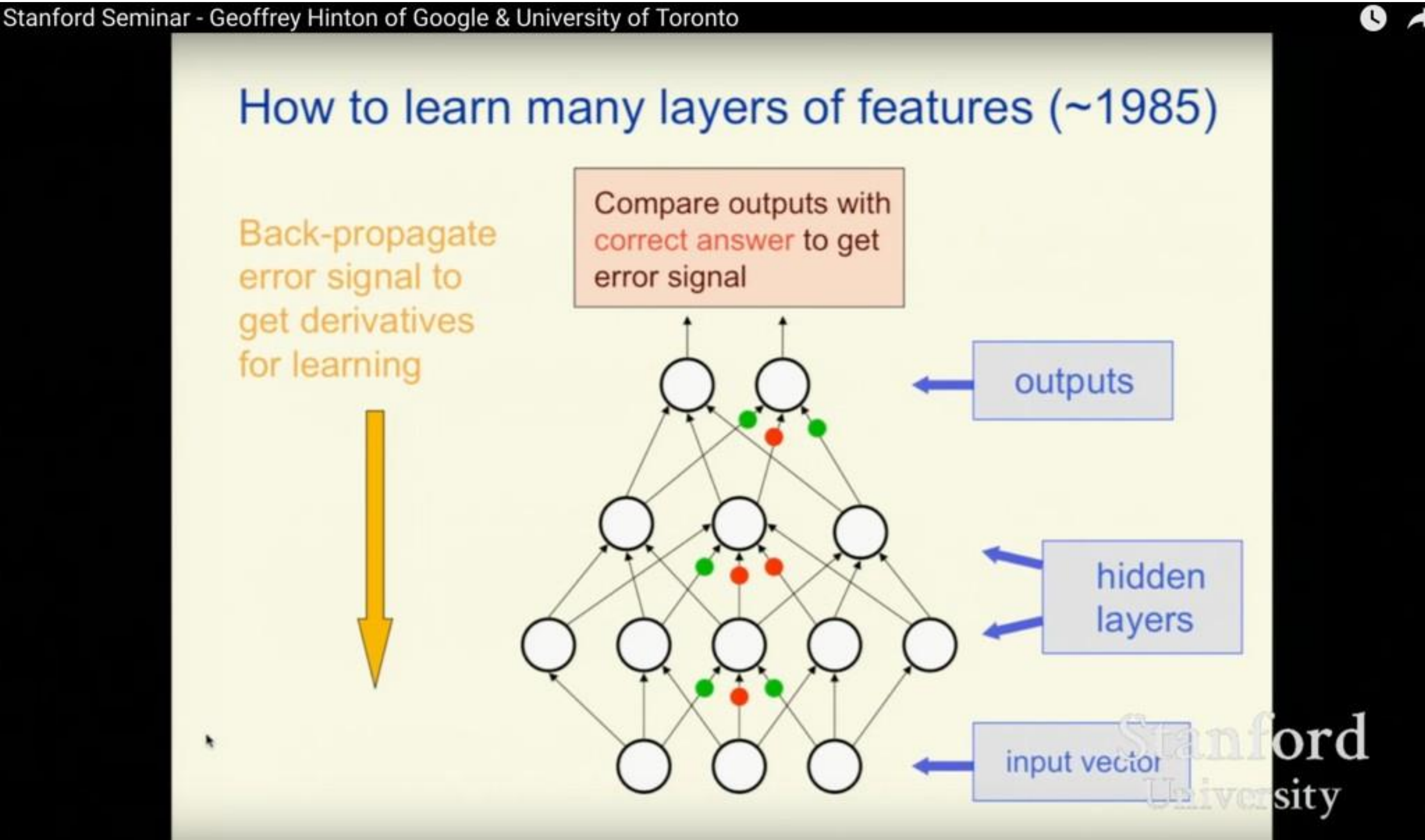


# Today

- Finishing up backpropagation
- ConvNet architectures
- How to train ConvNets

# (Recap) Backprop

From Geoff Hinton seminar at Stanford



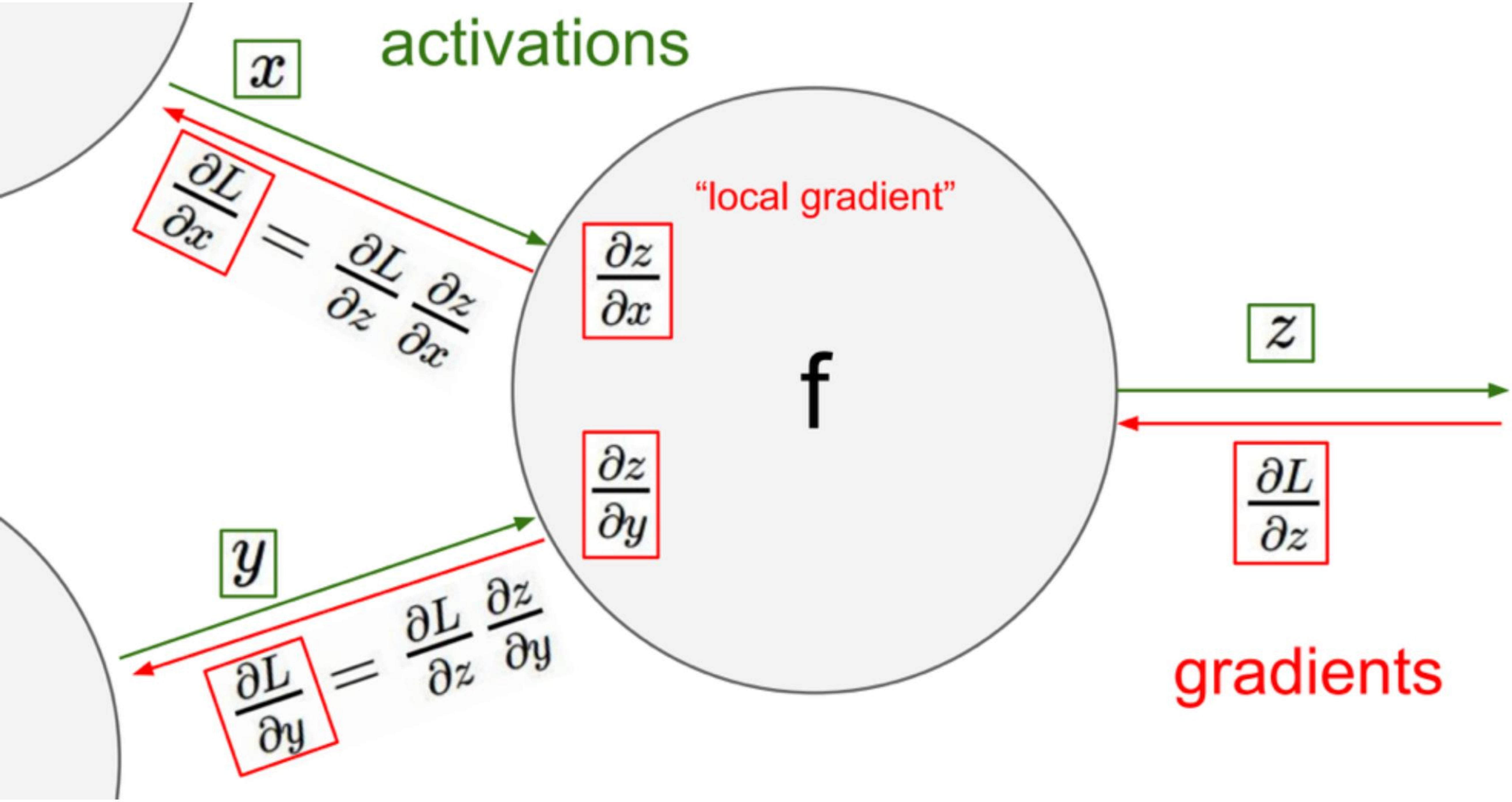
# (Recap) Backprop

**Parameters:** $$\theta = \begin{bmatrix} \theta_1 & \theta_2 & \cdots \end{bmatrix}$$

*All of the weights and biases in the network, stacked together*

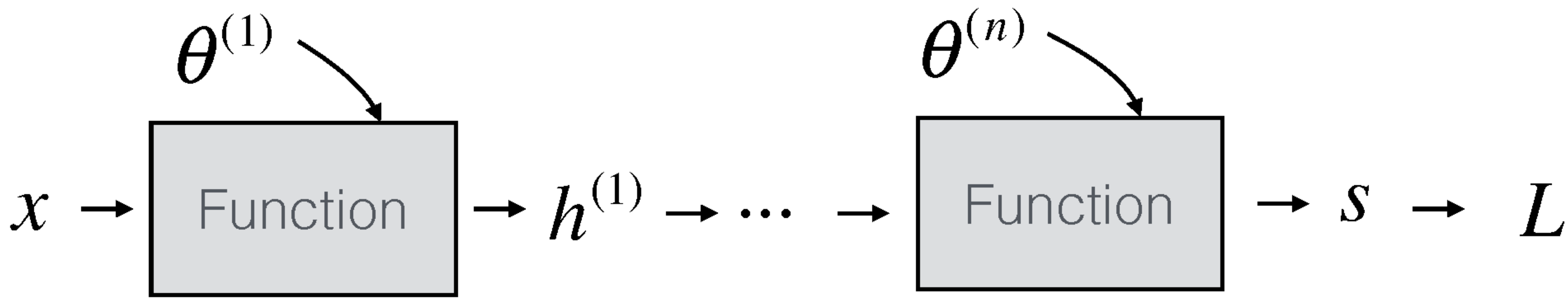
**Gradient:** $$\frac{\partial L}{\partial \theta} = \begin{bmatrix} \frac{\partial L}{\partial \theta_1} & \frac{\partial L}{\partial \theta_2} & \cdots \end{bmatrix}$$

*Intuition: "How fast would the error change if I change myself by a little bit"*

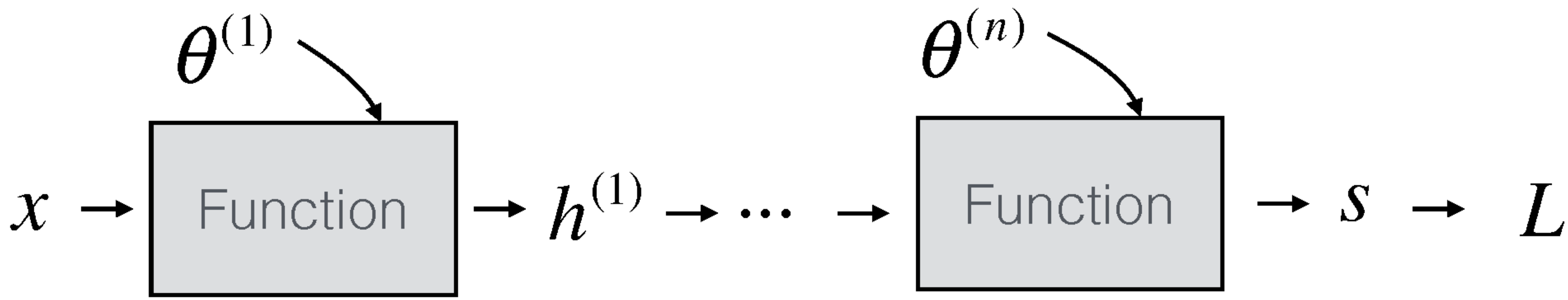



Slide from Karpathy 2016

# Forward Propagation:

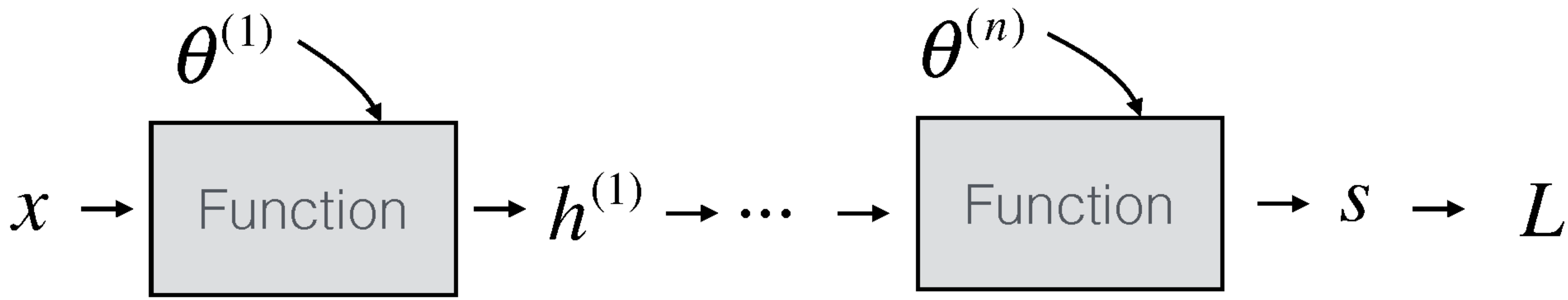


**Forward Propagation:**




**Backward Propagation:**

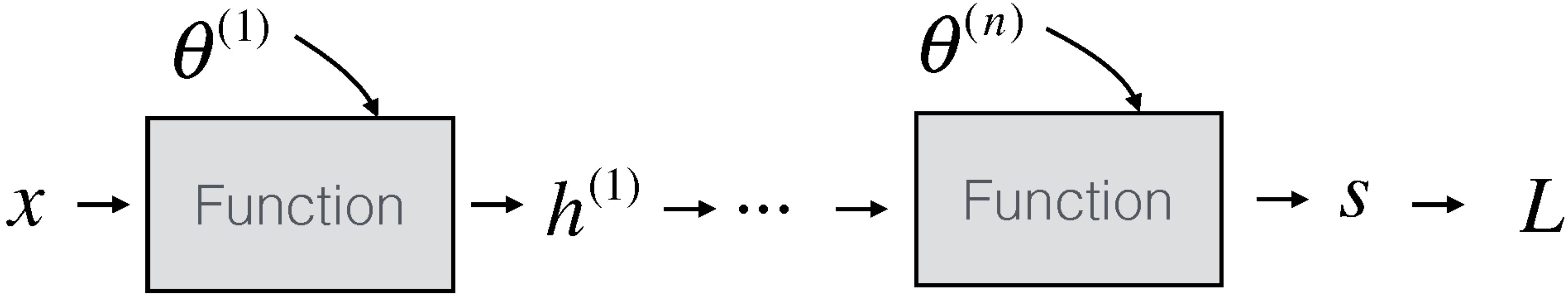
**Forward Propagation:**




**Backward Propagation:**

$L$

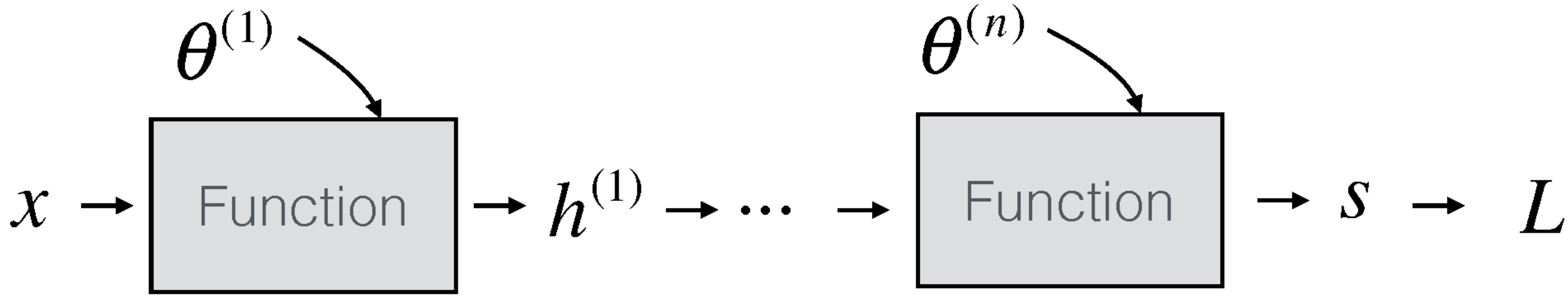
**Forward Propagation:**



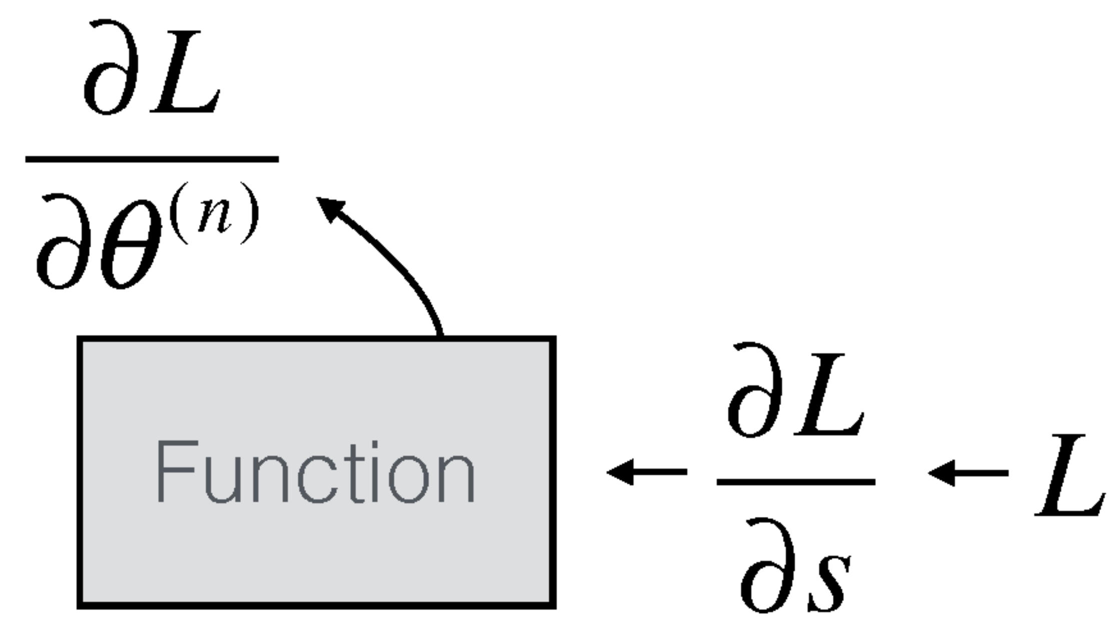

**Backward Propagation:**

$$\frac{\partial L}{\partial s} \leftarrow L$$

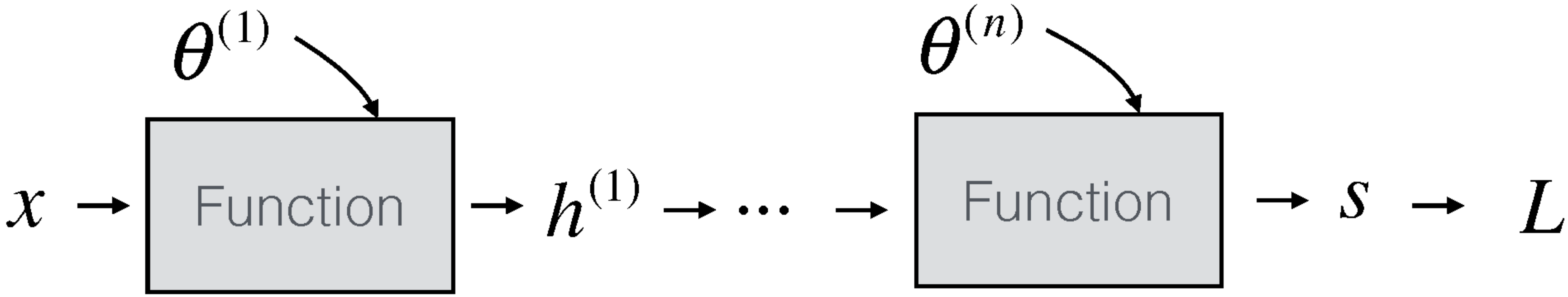
**Forward Propagation:**



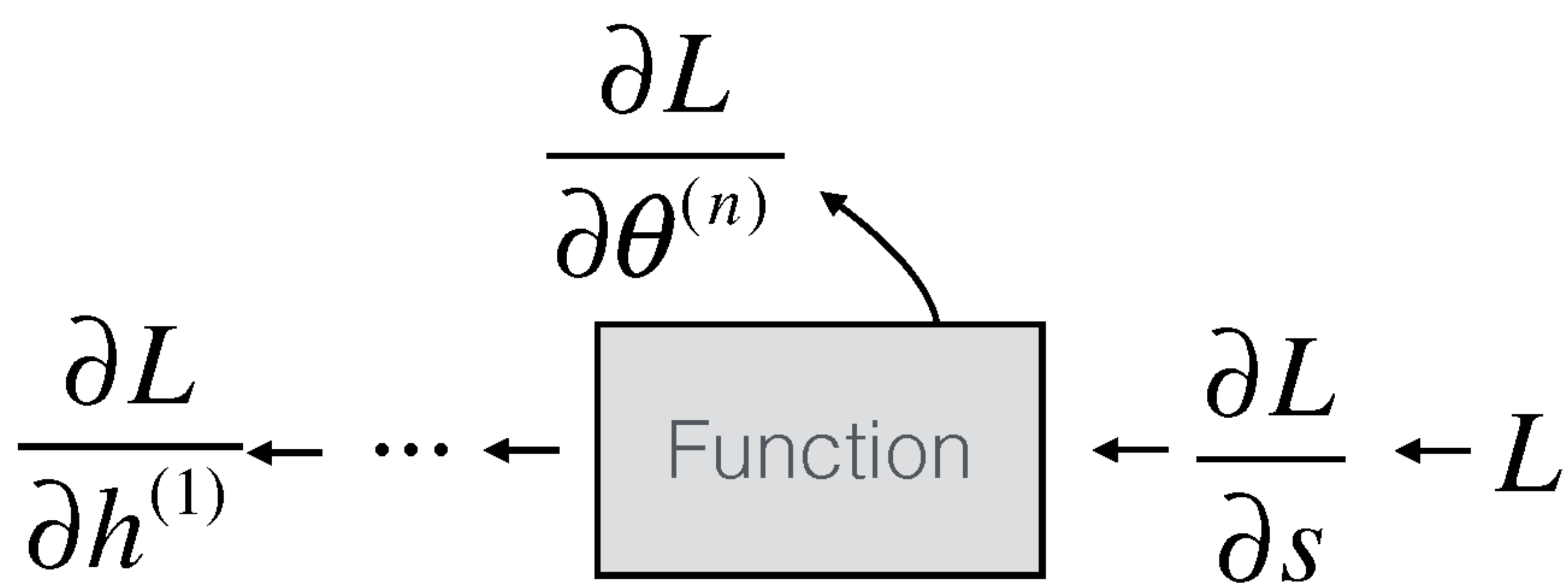

**Backward Propagation:**



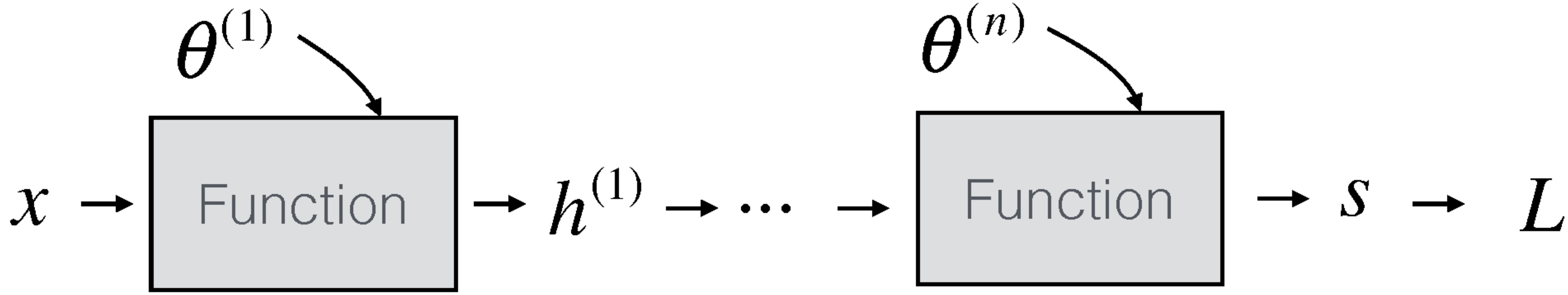
## Forward Propagation:



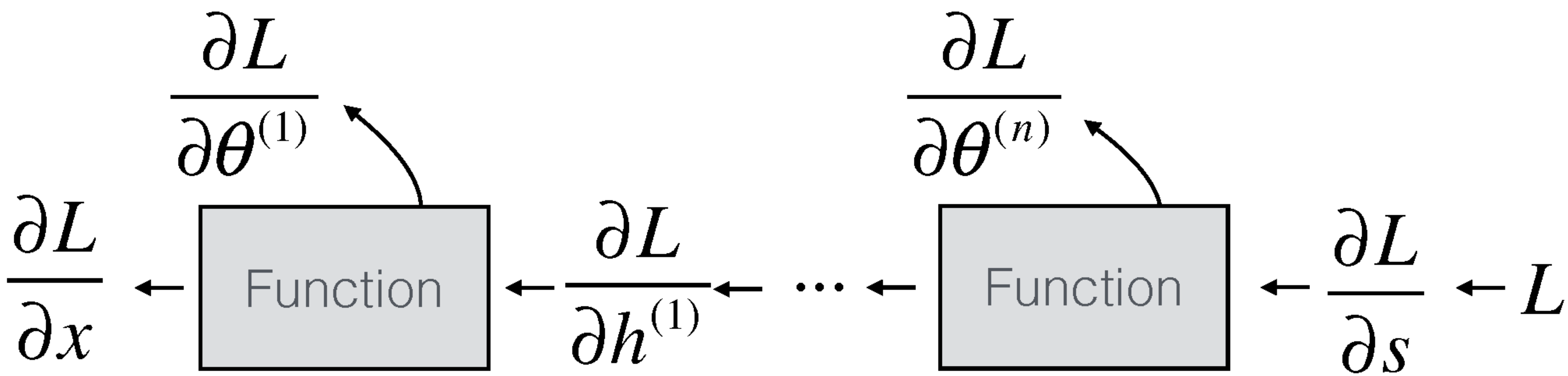

## Backward Propagation:



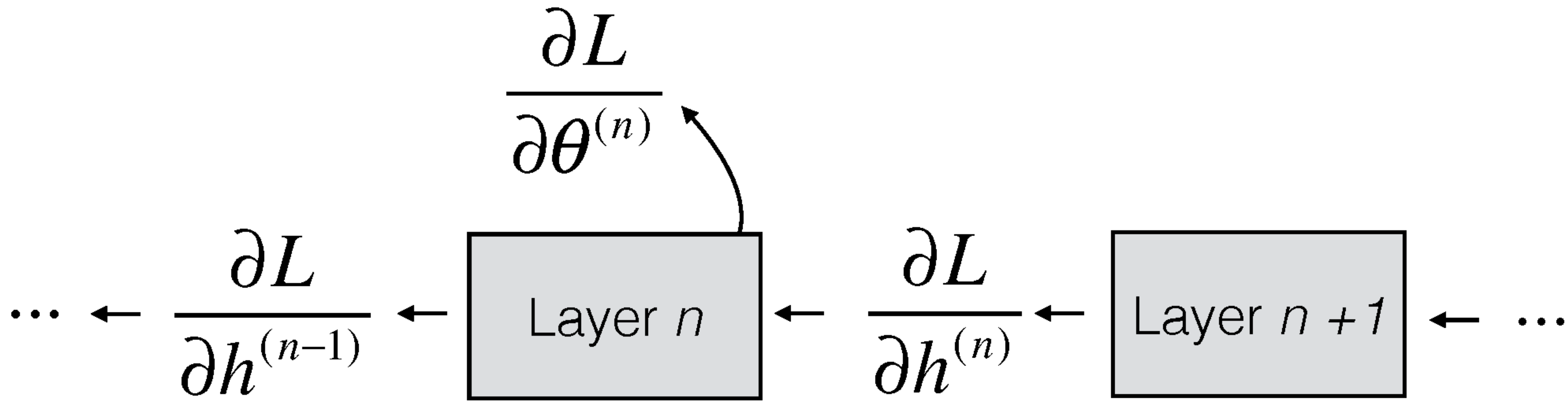
## Forward Propagation:



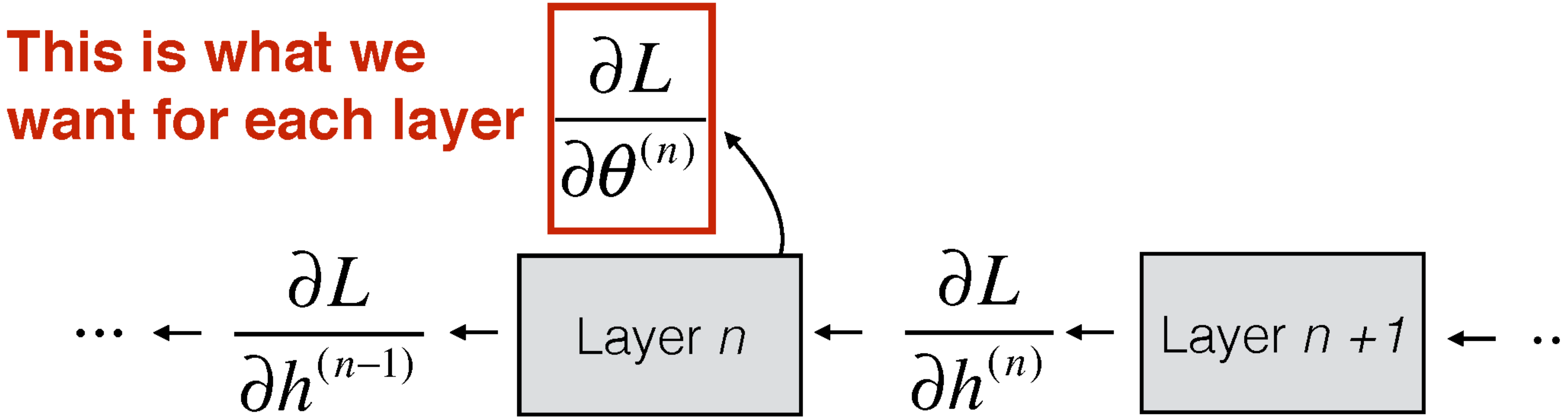

## Backward Propagation:

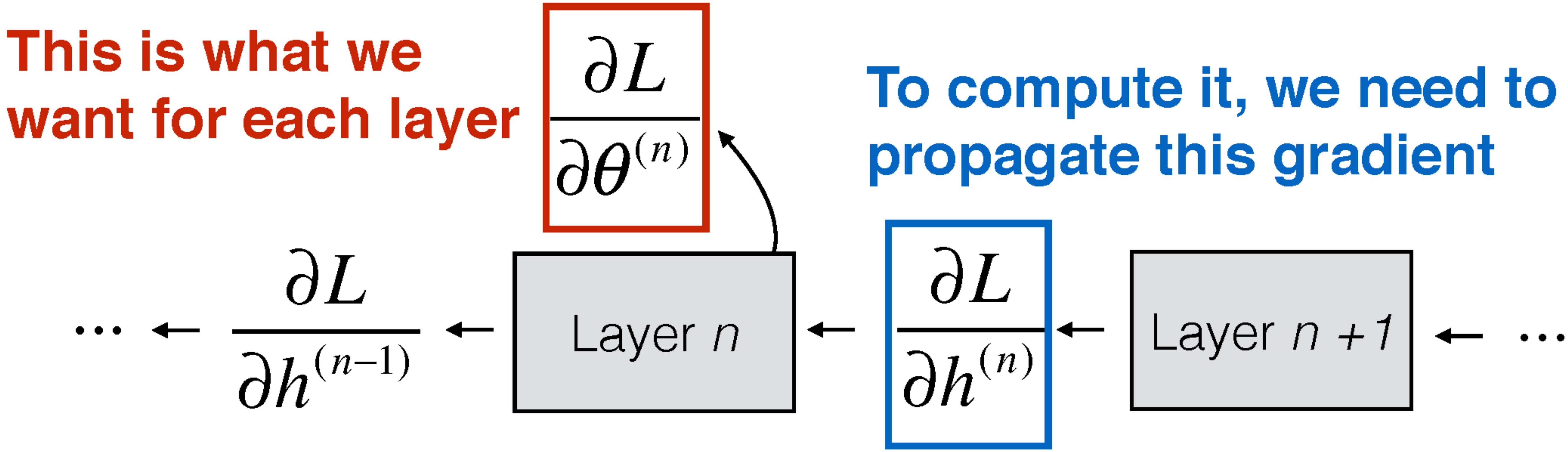


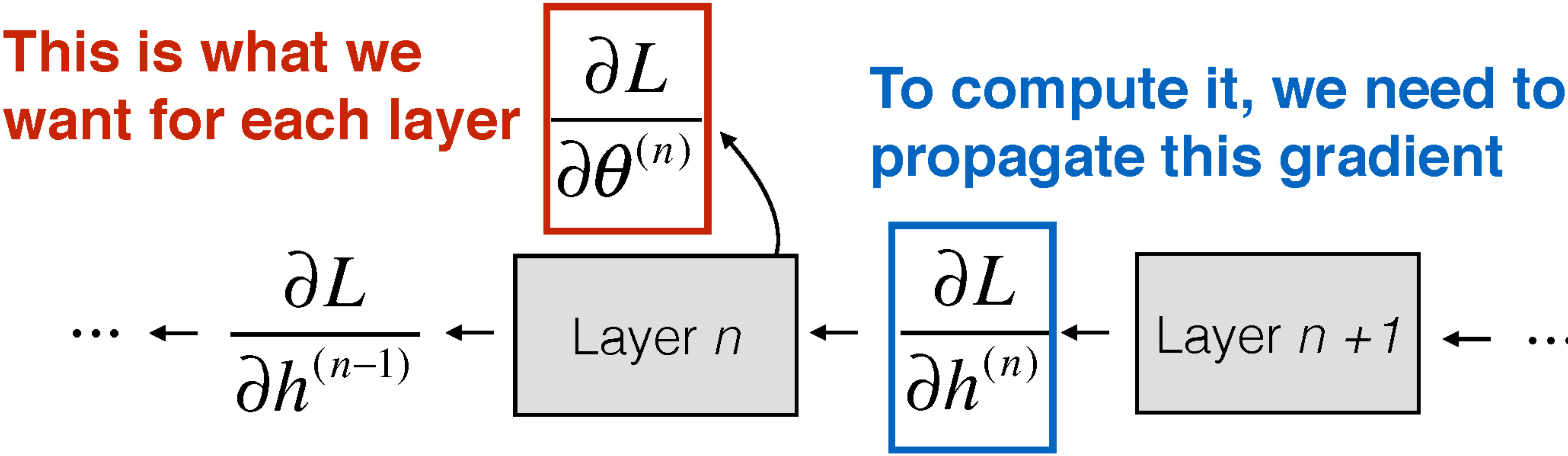
# What to do for each layer



$\frac{\partial L}{\partial \theta^{(n)}}$
$\cdots$
$\frac{\partial L}{\partial h^{(n-1)}}$
Layer $n$
$\frac{\partial L}{\partial h^{(n)}}$
Layer $n+1$
$\cdots$


This is what we want for each layer
$\frac{\partial L}{\partial \theta^{(n)}}$
$\cdots$
$\frac{\partial L}{\partial h^{(n-1)}}$
Layer $n$
$\frac{\partial L}{\partial h^{(n)}}$
Layer $n+1$
$\cdots$


This is what we want for each layer
$\frac{\partial L}{\partial \theta^{(n)}}$
To compute it, we need to propagate this gradient
$\frac{\partial L}{\partial h^{(n-1)}}$
Layer $n$
$\frac{\partial L}{\partial h^{(n)}}$
Layer $n+1$




**For each layer:**

This is what we want for each layer

$$\boxed{\frac{\partial L}{\partial \theta^{(n)}}}$$

To compute it, we need to propagate this gradient

$$\cdots \leftarrow \frac{\partial L}{\partial h^{(n-1)}} \leftarrow \boxed{\text{Layer } n} \leftarrow \boxed{\frac{\partial L}{\partial h^{(n)}}} \leftarrow \boxed{\text{Layer } n+1} \leftarrow \cdots$$

**For each layer:**

$$\boxed{\frac{\partial L}{\partial \theta^{(n)}}} = \frac{\partial L}{\partial h^{(n)}} \cdot \frac{\partial h^{(n)}}{\partial \theta^{(n)}}$$

What we want

This is what we want for each layer

$$\boxed{\frac{\partial L}{\partial \theta^{(n)}}}$$

To compute it, we need to propagate this gradient

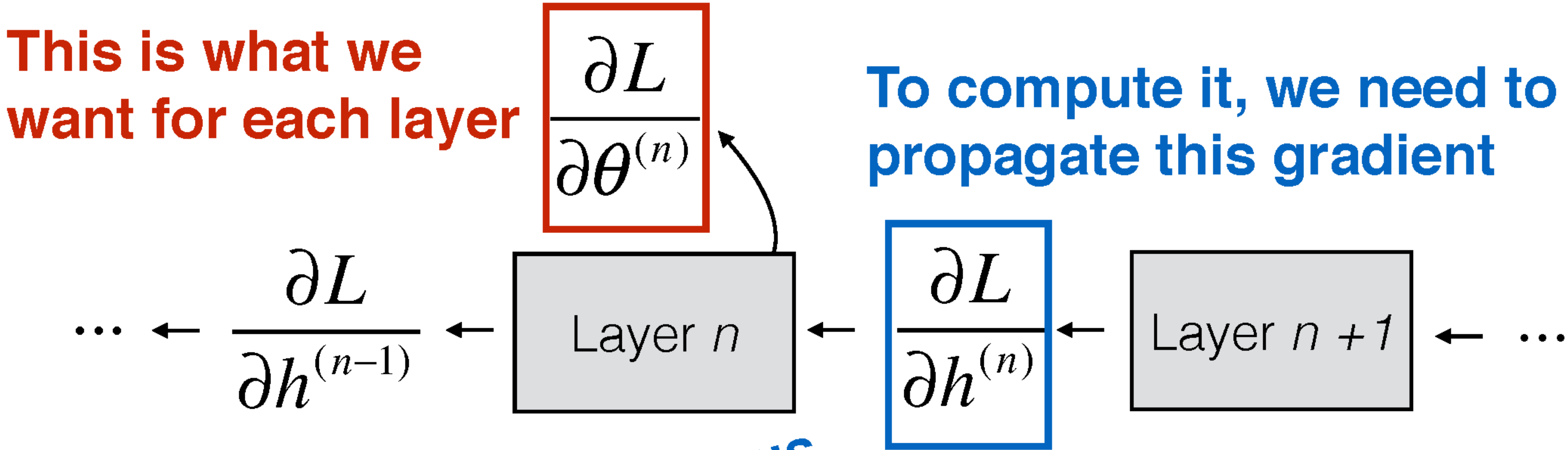
$$\cdots \leftarrow \frac{\partial L}{\partial h^{(n-1)}} \leftarrow \text{Layer } n \leftarrow \boxed{\frac{\partial L}{\partial h^{(n)}}} \leftarrow \text{Layer } n+1 \leftarrow \cdots$$

given to us

**For each layer:**

$$\boxed{\frac{\partial L}{\partial \theta^{(n)}}} = \boxed{\frac{\partial L}{\partial h^{(n)}}} \cdot \frac{\partial h^{(n)}}{\partial \theta^{(n)}}$$

What we want




given to us

**For each layer:**

$$\boxed{\frac{\partial L}{\partial \theta^{(n)}}} = \boxed{\frac{\partial L}{\partial h^{(n)}}} \cdot \boxed{\frac{\partial h^{(n)}}{\partial \theta^{(n)}}}$$

What we want

This is just the local gradient of layer *n*

**This is what we want for each layer** $\boxed{\dfrac{\partial L}{\partial \theta^{(n)}}}$

**To compute it, we need to propagate this gradient**

$$\cdots \leftarrow \frac{\partial L}{\partial h^{(n-1)}} \leftarrow \boxed{\text{Layer } n} \leftarrow \boxed{\frac{\partial L}{\partial h^{(n)}}} \leftarrow \boxed{\text{Layer } n+1} \leftarrow \cdots$$

**given to us**

**For each layer:**

$$\boxed{\frac{\partial L}{\partial \theta^{(n)}}} = \boxed{\frac{\partial L}{\partial h^{(n)}}} \cdot \boxed{\frac{\partial h^{(n)}}{\partial \theta^{(n)}}}$$

$$\frac{\partial L}{\partial h^{(n-1)}} = \frac{\partial L}{\partial h^{(n)}} \cdot \frac{\partial h^{(n)}}{\partial h^{(n-1)}}$$

**What we want**

**This is just the local gradient of layer *n***

**This is what we want for each layer** $\boxed{\dfrac{\partial L}{\partial \theta^{(n)}}}$

**To compute it, we need to propagate this gradient**

$$\cdots \leftarrow \frac{\partial L}{\partial h^{(n-1)}} \leftarrow \boxed{\text{Layer } n} \leftarrow \boxed{\frac{\partial L}{\partial h^{(n)}}} \leftarrow \boxed{\text{Layer } n+1} \leftarrow \cdots$$

**given to us**

**For each layer:**

$$\boxed{\frac{\partial L}{\partial \theta^{(n)}}} = \boxed{\frac{\partial L}{\partial h^{(n)}}} \cdot \boxed{\frac{\partial h^{(n)}}{\partial \theta^{(n)}}}$$

$$\frac{\partial L}{\partial h^{(n-1)}} = \boxed{\frac{\partial L}{\partial h^{(n)}}} \cdot \frac{\partial h^{(n)}}{\partial h^{(n-1)}}$$

**What we want**

**This is just the local gradient of layer *n***

This is what we want for each layer

$$\boxed{\frac{\partial L}{\partial \theta^{(n)}}}$$

To compute it, we need to propagate this gradient

$$\cdots \leftarrow \frac{\partial L}{\partial h^{(n-1)}} \leftarrow \text{Layer } n \leftarrow \boxed{\frac{\partial L}{\partial h^{(n)}}} \leftarrow \text{Layer } n+1 \leftarrow \cdots$$

given to us

For each layer:

$$\boxed{\frac{\partial L}{\partial \theta^{(n)}}} = \boxed{\frac{\partial L}{\partial h^{(n)}}} \cdot \boxed{\frac{\partial h^{(n)}}{\partial \theta^{(n)}}} \qquad \frac{\partial L}{\partial h^{(n-1)}} = \boxed{\frac{\partial L}{\partial h^{(n)}}} \cdot \boxed{\frac{\partial h^{(n)}}{\partial h^{(n-1)}}}$$

What we want

This is just the local gradient of layer *n*

# Summary

**For each layer, we compute:**

$$\left[\text{Propagated gradient to the left}\right] = \left[\text{Propagated gradient from right}\right] \cdot \left[\text{Local gradient}\right]$$

# Summary

**For each layer, we compute:**

$$[\text{Propagated gradient to the left}] = [\text{Propagated gradient from right}] \cdot \underbrace{[\text{Local gradient}]}_{\text{(Can compute immediately)}}$$

# Summary

**For each layer, we compute:**

$$\left[\text{Propagated gradient to the left}\right] = \left[\text{Propagated gradient from right}\right] \cdot \left[\text{Local gradient}\right]$$

(Received during backprop) (Can compute immediately)

# Backprop in N-dimensions

*just add more subscripts and more summations*

# Backprop in N-dimensions

*just add more subscripts and more summations*

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial h}\frac{\partial h}{\partial x}$$

$x, h$ scalars
($L$ is always scalar)

# Backprop in N-dimensions

*just add more subscripts and more summations*

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial h}\frac{\partial h}{\partial x}$$

$x, h$ scalars
($L$ is always scalar)

$$\frac{\partial L}{\partial x_j} = \sum_i \frac{\partial L}{\partial h_i}\frac{\partial h_i}{\partial x_j}$$

$x, h$ 1D arrays (vectors)

# Backprop in N-dimensions

*just add more subscripts and more summations*

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial h}\frac{\partial h}{\partial x}$$

$x,h$ scalars
($L$ is always scalar)

$$\frac{\partial L}{\partial x_j} = \sum_i \frac{\partial L}{\partial h_i}\frac{\partial h_i}{\partial x_j}$$

$x,h$ 1D arrays (vectors)

$$\frac{\partial L}{\partial x_{ab}} = \sum_i \sum_j \frac{\partial L}{\partial h_{ij}}\frac{\partial h_{ij}}{\partial x_{ab}}$$

$x,h$ 2D arrays

# Backprop in N-dimensions

*just add more subscripts and more summations*

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial h}\frac{\partial h}{\partial x}$$

$x, h$ scalars
($L$ is always scalar)

$$\frac{\partial L}{\partial x_j} = \sum_i \frac{\partial L}{\partial h_i}\frac{\partial h_i}{\partial x_j}$$

$x, h$ 1D arrays (vectors)

$$\frac{\partial L}{\partial x_{ab}} = \sum_i \sum_j \frac{\partial L}{\partial h_{ij}}\frac{\partial h_{ij}}{\partial x_{ab}}$$

$x, h$ 2D arrays

$$\frac{\partial L}{\partial x_{abc}} = \sum_i \sum_j \sum_k \frac{\partial L}{\partial h_{ijk}}\frac{\partial h_{ijk}}{\partial x_{abc}}$$

$x, h$ 3D arrays

# Examples

# Example: Mean Subtraction

(for a single input)

# Example: Mean Subtraction

## (for a single input)

- Example layer: mean subtraction:

# Example: Mean Subtraction
## (for a single input)

- Example layer: mean subtraction:

$$h_i = x_i - \frac{1}{D}\sum_k x_k$$

# Example: Mean Subtraction
## (for a single input)

- Example layer: mean subtraction:

$$h_i = x_i - \frac{1}{D}\sum_k x_k$$

(here, “i” and “k” are channels)

# Example: Mean Subtraction
## (for a single input)

- Example layer: mean subtraction:

$$h_i = x_i - \frac{1}{D}\sum_k x_k$$

(here, “i” and “k” are channels)

- Always start with the chain rule (this one is for 1D):

$$\frac{\partial L}{\partial x_j} = \sum_i \frac{\partial L}{\partial h_i}\frac{\partial h_i}{\partial x_j}$$

# Example: Mean Subtraction
## (for a single input)

- Example layer: mean subtraction:

$$h_i = x_i - \frac{1}{D}\sum_k x_k$$

(here, "i" and "k" are channels)

- Always start with the chain rule (this one is for 1D):

$$\frac{\partial L}{\partial x_j} = \sum_i \frac{\partial L}{\partial h_i}\frac{\partial h_i}{\partial x_j}$$

- **Note:** Be very careful with your subscripts! Introduce new variables and don't re-use letters.

# Example: Mean Subtraction

(for a single input)

# Example: Mean Subtraction
## (for a single input)

- Forward: $h_i = x_i - \frac{1}{D}\sum_k x_k$

# Example: Mean Subtraction

## (for a single input)

- Forward: $h_i = x_i - \frac{1}{D}\sum_k x_k$
- Taking the derivative of the layer:

# Example: Mean Subtraction

(for a single input)

- Forward: $h_i = x_i - \frac{1}{D}\sum_k x_k$
- Taking the derivative of the layer: $\frac{\partial h_i}{\partial x_j} = \delta_{ij} - \frac{1}{D}$

# Example: Mean Subtraction
## (for a single input)

- Forward: $h_i = x_i - \frac{1}{D}\sum_k x_k$
- Taking the derivative of the layer: $\frac{\partial h_i}{\partial x_j} = \delta_{ij} - \frac{1}{D}$

$$\left( \delta_{ij} = \begin{cases} 1 & i = j \\ 0 & \text{else} \end{cases} \right)$$

# Example: Mean Subtraction
## (for a single input)

- Forward: $h_i = x_i - \frac{1}{D}\sum_k x_k$
- Taking the derivative of the layer: $\frac{\partial h_i}{\partial x_j} = \delta_{ij} - \frac{1}{D}$

$$\frac{\partial L}{\partial x_j} = \sum_i \frac{\partial L}{\partial h_i}\frac{\partial h_i}{\partial x_j}$$

(backprop aka chain rule)

$$\left( \delta_{ij} = \begin{cases} 1 & i = j \\ 0 & \text{else} \end{cases} \right)$$

# Example: Mean Subtraction
## (for a single input)

- Forward: $$h_i = x_i - \frac{1}{D}\sum_k x_k$$
- Taking the derivative of the layer: $$\frac{\partial h_i}{\partial x_j} = \delta_{ij} - \frac{1}{D}$$

$$\frac{\partial L}{\partial x_j} = \sum_i \frac{\partial L}{\partial h_i}\frac{\partial h_i}{\partial x_j}$$ (backprop aka chain rule)

$$= \sum_i \frac{\partial L}{\partial h_i}\left(\delta_{ij} - \frac{1}{D}\right)$$

$$\left(\delta_{ij} = \begin{cases} 1 & i = j \\ 0 & \text{else} \end{cases}\right)$$

# Example: Mean Subtraction
## (for a single input)

- Forward: $h_i = x_i - \frac{1}{D}\sum_k x_k$
- Taking the derivative of the layer: $\frac{\partial h_i}{\partial x_j} = \delta_{ij} - \frac{1}{D}$

$$\left( \delta_{ij} = \begin{cases} 1 & i = j \\ 0 & \text{else} \end{cases} \right)$$

$$\frac{\partial L}{\partial x_j} = \sum_i \frac{\partial L}{\partial h_i} \frac{\partial h_i}{\partial x_j}$$

(backprop aka chain rule)

$$= \sum_i \frac{\partial L}{\partial h_i}\left( \delta_{ij} - \frac{1}{D} \right)$$

$$= \sum_i \frac{\partial L}{\partial h_i}\delta_{ij} - \frac{1}{D}\sum_i \frac{\partial L}{\partial h_i}$$

# Example: Mean Subtraction
(for a single input)

- Forward: $h_i = x_i - \frac{1}{D}\sum_k x_k$
- Taking the derivative of the layer: $\frac{\partial h_i}{\partial x_j} = \delta_{ij} - \frac{1}{D}$

$$\left( \delta_{ij} = \begin{cases} 1 & i = j \\ 0 & \text{else} \end{cases} \right)$$

$$\frac{\partial L}{\partial x_j} = \sum_i \frac{\partial L}{\partial h_i}\frac{\partial h_i}{\partial x_j}$$ (backprop aka chain rule)

$$= \sum_i \frac{\partial L}{\partial h_i}\left( \delta_{ij} - \frac{1}{D} \right)$$

$$= \sum_i \frac{\partial L}{\partial h_i}\delta_{ij} - \frac{1}{D}\sum_i \frac{\partial L}{\partial h_i}$$

$$= \frac{\partial L}{\partial h_j} - \frac{1}{D}\sum_i \frac{\partial L}{\partial h_i}$$

# Example: Mean Subtraction

(for a single input)

- Forward: $h_i = x_i - \frac{1}{D}\sum_k x_k$
- Taking the derivative of the layer: $\frac{\partial h_i}{\partial x_j} = \delta_{ij} - \frac{1}{D}$

$$\left( \delta_{ij} = \begin{cases} 1 & i = j \\ 0 & \text{else} \end{cases} \right)$$

$$\frac{\partial L}{\partial x_j} = \sum_i \frac{\partial L}{\partial h_i}\frac{\partial h_i}{\partial x_j}$$

(backprop aka chain rule)

$$= \sum_i \frac{\partial L}{\partial h_i}\left(\delta_{ij} - \frac{1}{D}\right)$$

$$= \sum_i \frac{\partial L}{\partial h_i}\delta_{ij} - \frac{1}{D}\sum_i \frac{\partial L}{\partial h_i}$$

$$= \frac{\partial L}{\partial h_j} - \frac{1}{D}\sum_i \frac{\partial L}{\partial h_i}$$

**Done!**

# Example: Mean Subtraction

## (for a single input)

$$h_i = x_i - \frac{1}{D}\sum_k x_k$$

$$\frac{\partial L}{\partial x_i} = \frac{\partial L}{\partial h_i} - \frac{1}{D}\sum_k \frac{\partial L}{\partial h_k}$$

# Example: Mean Subtraction

(for a single input)

- Forward: $$h_i = x_i - \frac{1}{D}\sum_k x_k$$
- Backward: $$\frac{\partial L}{\partial x_i} = \frac{\partial L}{\partial h_i} - \frac{1}{D}\sum_k \frac{\partial L}{\partial h_k}$$

# Example: Mean Subtraction

## (for a single input)

- Forward: $$h_i = x_i - \frac{1}{D}\sum_k x_k$$
- Backward: $$\frac{\partial L}{\partial x_i} = \frac{\partial L}{\partial h_i} - \frac{1}{D}\sum_k \frac{\partial L}{\partial h_k}$$
- In this case, they're identical operations!

# Example: Mean Subtraction

## (for a single input)

- Forward: $$h_i = x_i - \frac{1}{D}\sum_k x_k$$
- Backward: $$\frac{\partial L}{\partial x_i} = \frac{\partial L}{\partial h_i} - \frac{1}{D}\sum_k \frac{\partial L}{\partial h_k}$$

- In this case, they're identical operations!

- Usually the forwards pass and backwards pass are similar **but not the same**.

# Example: Mean Subtraction

(for a single input)

- Forward: $$h_i = x_i - \frac{1}{D}\sum_k x_k$$
- Backward: $$\frac{\partial L}{\partial x_i} = \frac{\partial L}{\partial h_i} - \frac{1}{D}\sum_k \frac{\partial L}{\partial h_k}$$

- In this case, they're identical operations!

- Usually the forwards pass and backwards pass are similar **but not the same**.

- Derive it by hand, and check it numerically

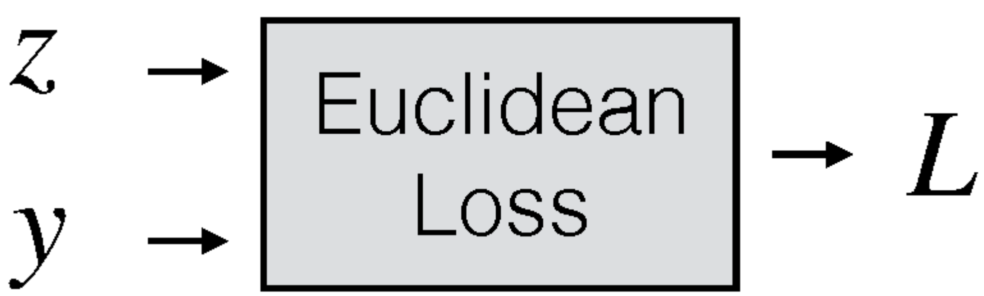
# Example: Euclidean Loss

# Example: Euclidean Loss

- Euclidean loss layer:

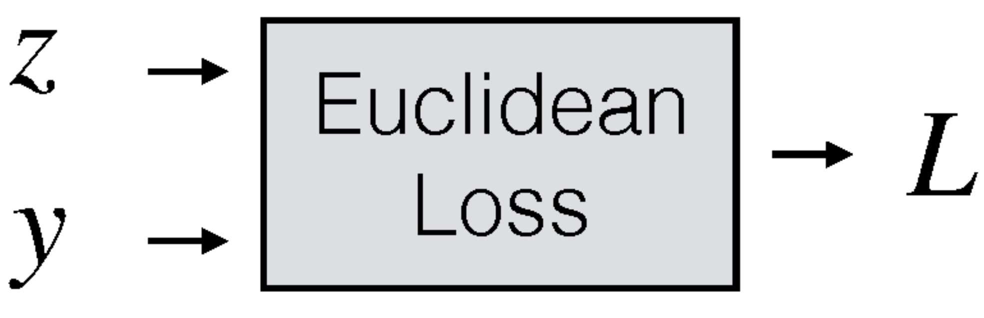
# Example: Euclidean Loss

- Euclidean loss layer:



# Example: Euclidean Loss

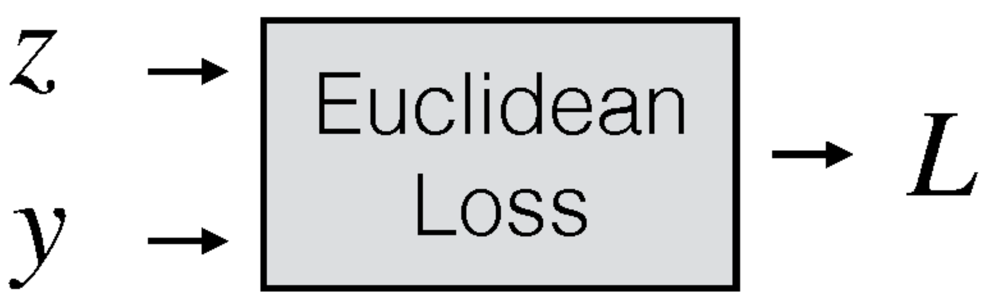
- Euclidean loss layer:




$$L_i = \frac{1}{2}\sum_j (z_{i,j} - y_{i,j})^2$$

# Example: Euclidean Loss

- Euclidean loss layer:

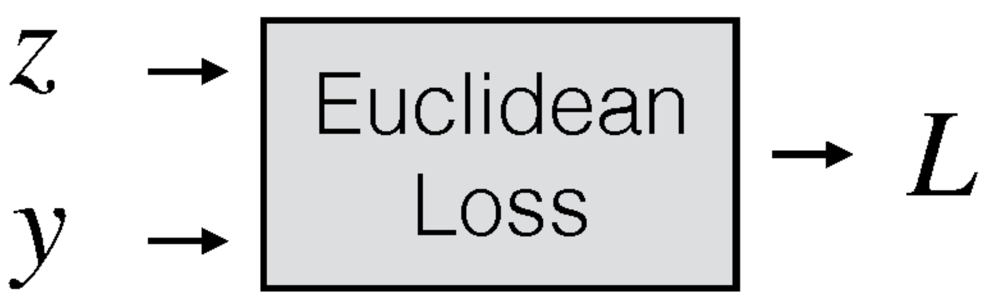



$$L_i = \frac{1}{2}\sum_j (z_{i,j} - y_{i,j})^2$$

(“i” is the batch index,
“j” is the channel)

# Example: Euclidean Loss

- Euclidean loss layer:



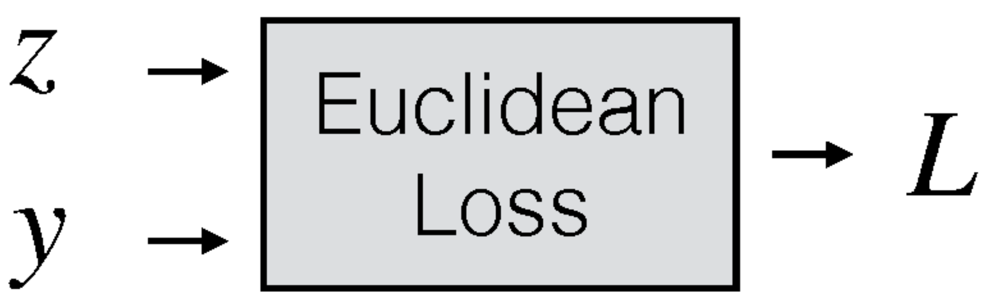

$$L_i = \frac{1}{2} \sum_j (z_{i,j} - y_{i,j})^2$$

("i" is the batch index,
"j" is the channel)

- The total loss is the average over N examples:

# Example: Euclidean Loss

- Euclidean loss layer:




$$L_i = \frac{1}{2}\sum_j (z_{i,j} - y_{i,j})^2$$

("i" is the batch index, "j" is the channel)

- The total loss is the average over N examples:

$$L = \frac{1}{N}\sum_i L_i$$

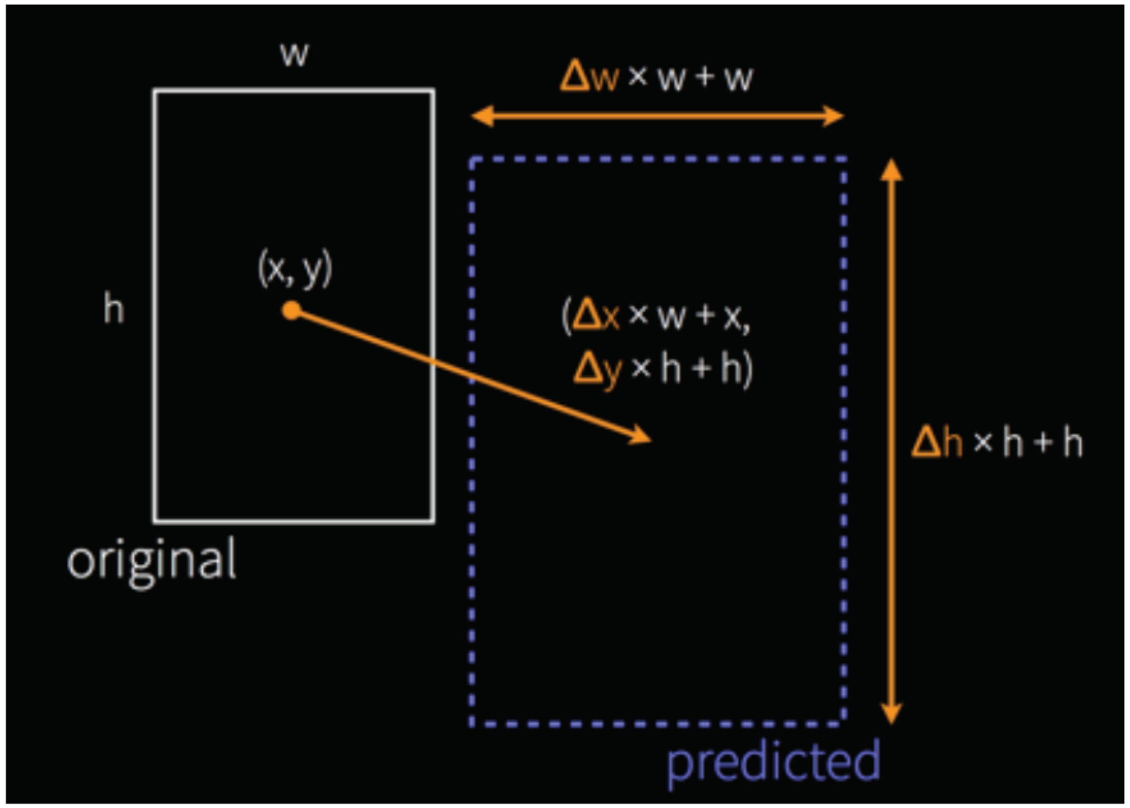
# Example: Euclidean Loss

# Example: Euclidean Loss

- Used for **regression**, e.g. predicting an adjustment to box coordinates when detecting objects:

# Example: Euclidean Loss

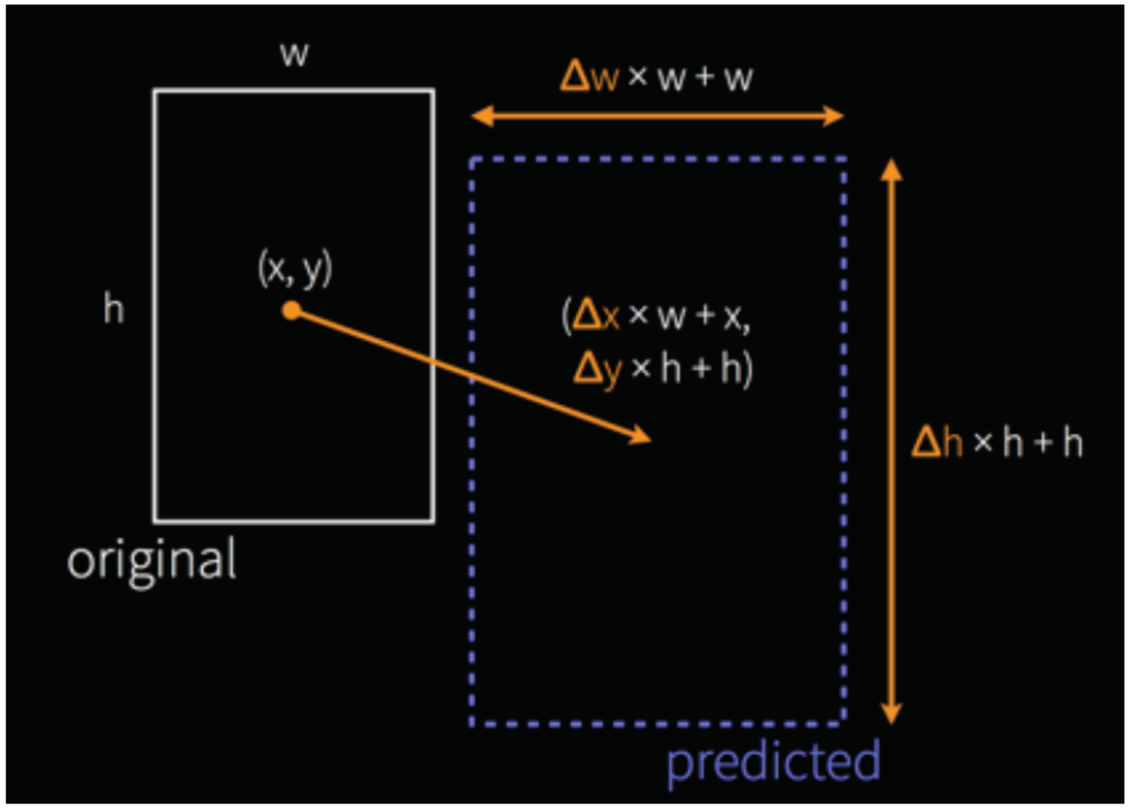
- Used for **regression**, e.g. predicting an adjustment to box coordinates when detecting objects:




Bounding box regression from the R-CNN object detector [Girshick 2014]

# Example: Euclidean Loss

- Used for **regression**, e.g. predicting an adjustment to box coordinates when detecting objects:




Bounding box regression from the R-CNN object detector [Girshick 2014]

- **Note:** Can be unstable and other losses often work better. Alternatives: L1 distance (instead of L2), discretizing into category bins and using softmax

# Example: Euclidean Loss

# Example: Euclidean Loss

- Forward: $$L_i = \frac{1}{2}\sum_j (z_{i,j} - y_{i,j})^2$$

# Example: Euclidean Loss

- Forward: $$L_i = \frac{1}{2}\sum_j (z_{i,j} - y_{i,j})^2$$

- Backward:

# Example: Euclidean Loss

- Forward: $$L_i = \frac{1}{2}\sum_j (z_{i,j} - y_{i,j})^2$$
- Backward: $$\frac{\partial L_i}{\partial z_{i,j}} = z_{i,j} - y_{i,j}$$

# Example: Euclidean Loss

- Forward: $$L_i = \frac{1}{2}\sum_j (z_{i,j} - y_{i,j})^2$$
- Backward: $$\frac{\partial L_i}{\partial z_{i,j}} = z_{i,j} - y_{i,j}$$

$$\frac{\partial L_i}{\partial y_{i,j}} = y_{i,j} - z_{i,j}$$

# Example: Euclidean Loss

- Forward: $$L_i = \frac{1}{2}\sum_j (z_{i,j} - y_{i,j})^2$$

- Backward: $$\frac{\partial L_i}{\partial z_{i,j}} = z_{i,j} - y_{i,j}$$

$$\frac{\partial L_i}{\partial y_{i,j}} = y_{i,j} - z_{i,j}$$

- **Q:** If you scale the loss by *C*, what happens to gradient computed in the backwards pass?

# Example: Euclidean Loss

- Forward: $$L_i = \frac{1}{2}\sum_j (z_{i,j} - y_{i,j})^2$$

- Backward: $$\frac{\partial \boxed{L_i}}{\partial z_{i,j}} = z_{i,j} - y_{i,j}$$

$$\frac{\partial L_i}{\partial y_{i,j}} = y_{i,j} - z_{i,j}$$

(note that this is with respect to Li, not L)

- **Q:** If you scale the loss by *C*, what happens to gradient computed in the backwards pass?

# Example: Euclidean Loss

# Example: Euclidean Loss

- Forward pass, for a batch of N inputs:

# Example: Euclidean Loss

- Forward pass, for a batch of N inputs:

$$L = \frac{1}{N} \sum_i L_i \qquad\qquad L_i = \frac{1}{2} \sum_j (z_{i,j} - y_{i,j})^2$$

# Example: Euclidean Loss

- Forward pass, for a batch of N inputs:

$$L = \frac{1}{N}\sum_i L_i \qquad\qquad L_i = \frac{1}{2}\sum_j (z_{i,j} - y_{i,j})^2$$

- Backward pass:

# Example: Euclidean Loss

- Forward pass, for a batch of N inputs:

$$L = \frac{1}{N} \sum_i L_i \qquad L_i = \frac{1}{2} \sum_j (z_{i,j} - y_{i,j})^2$$

- Backward pass:

$$\frac{\partial L}{\partial x_{i,j}} = \frac{z_{i,j} - y_{i,j}}{N} \qquad \frac{\partial L}{\partial y_{i,j}} = \frac{y_{i,j} - z_{i,j}}{N}$$

# Example: Euclidean Loss

- Forward pass, for a batch of N inputs:

$$L = \frac{1}{N}\sum_i L_i \qquad L_i = \frac{1}{2}\sum_j (z_{i,j} - y_{i,j})^2$$

- Backward pass:

$$\frac{\partial L}{\partial x_{i,j}} = \frac{z_{i,j} - y_{i,j}}{N} \qquad \frac{\partial L}{\partial y_{i,j}} = \frac{y_{i,j} - z_{i,j}}{N}$$

*(You should be able to derive this)*

# What about the weights?

To get the derivative of the weights, use the chain rule again!

# What about the weights?

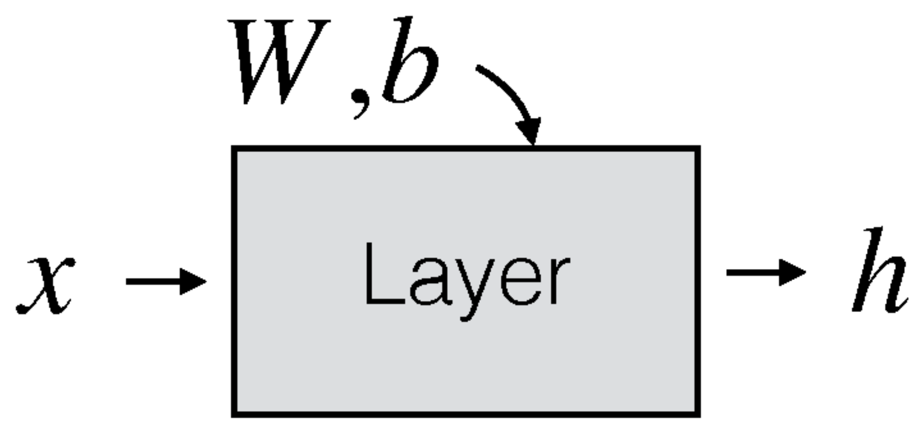
To get the derivative of the weights, use the chain rule again!

**Example:** 2D weights, 1D bias, 1D hidden activations:

# What about the weights?

To get the derivative of the weights, use the chain rule again!

**Example:** 2D weights, 1D bias, 1D hidden activations:

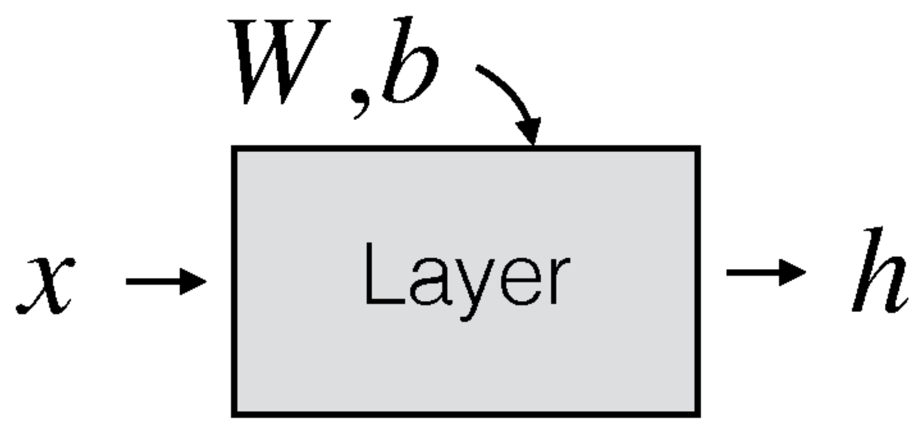



$$h = h(x;W)$$

# What about the weights?

To get the derivative of the weights, use the chain rule again!

**Example:** 2D weights, 1D bias, 1D hidden activations:



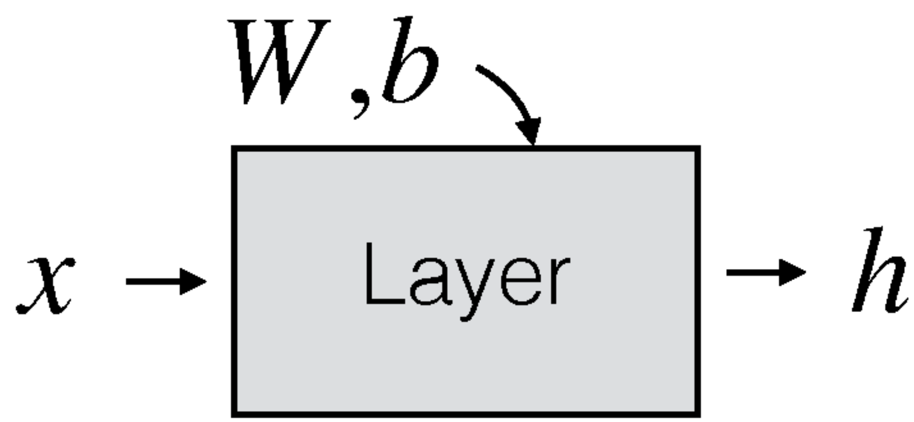

$$h = h(x;W)$$

$$\frac{\partial L}{\partial W_{ij}} = \sum_k \frac{\partial L}{\partial h_k} \frac{\partial h_k}{\partial W_{ij}}$$

# What about the weights?

To get the derivative of the weights, use the chain rule again!

**Example:** 2D weights, 1D bias, 1D hidden activations:




$$h = h(x; W)$$

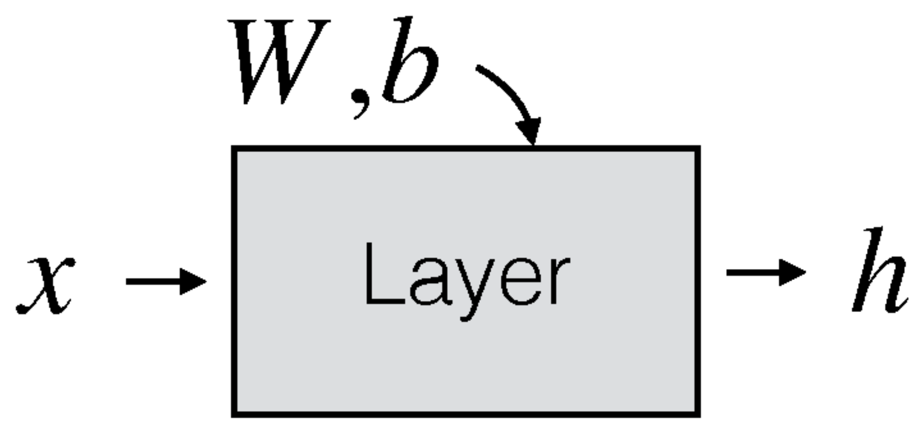
$$\frac{\partial L}{\partial W_{ij}} = \sum_k \frac{\partial L}{\partial h_k} \frac{\partial h_k}{\partial W_{ij}}$$

$$\frac{\partial L}{\partial b_i} = \sum_k \frac{\partial L}{\partial h_k} \frac{\partial h_k}{\partial b_i}$$

# What about the weights?

To get the derivative of the weights, use the chain rule again!

**Example:** 2D weights, 1D bias, 1D hidden activations:




$$h = h(x; W)$$

$$\frac{\partial L}{\partial W_{ij}} = \sum_k \frac{\partial L}{\partial h_k} \frac{\partial h_k}{\partial W_{ij}} \qquad \frac{\partial L}{\partial b_i} = \sum_k \frac{\partial L}{\partial h_k} \frac{\partial h_k}{\partial b_i}$$

*(the number of subscripts and summations changes depending on your layer and parameter sizes)*

# 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)

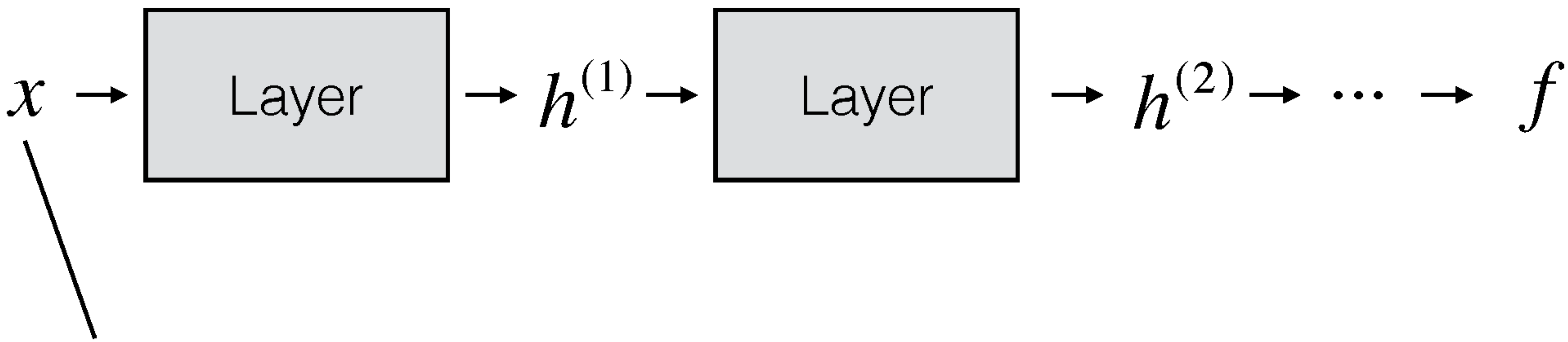
# What shape should the activations have?

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

- 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?



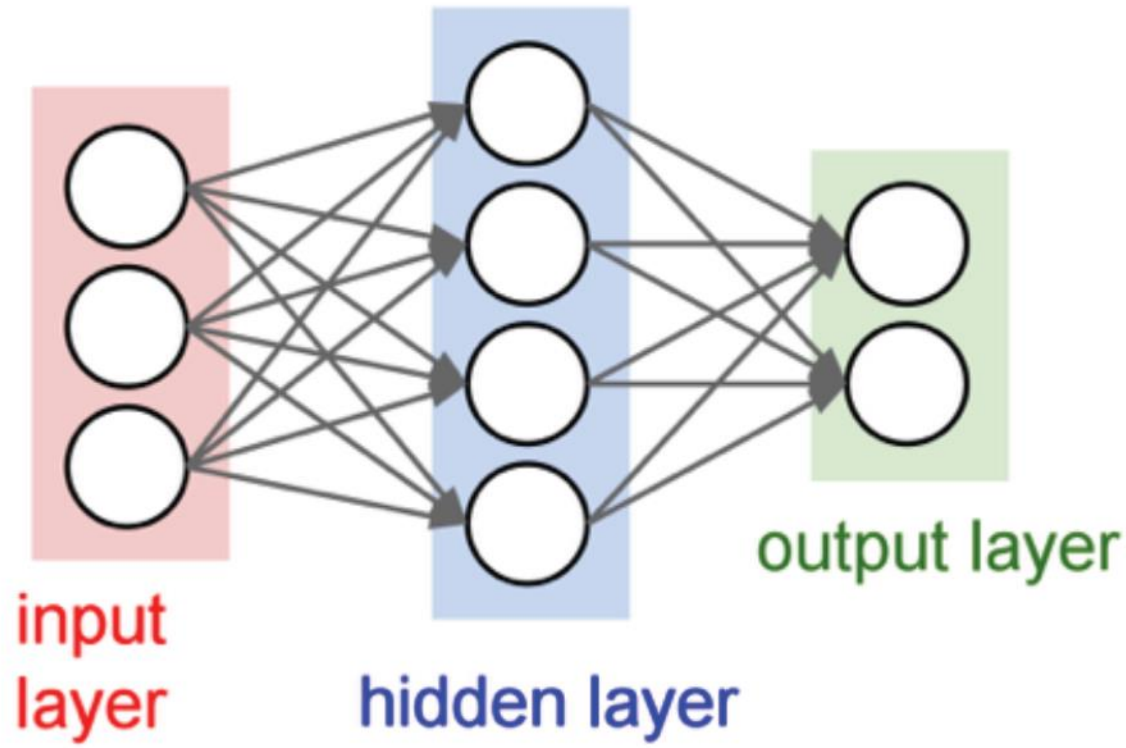

- The input is an image, which is 3D (RGB channel, height, width)

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

- What about keeping everything in 3D?

# 3D Activations

before:

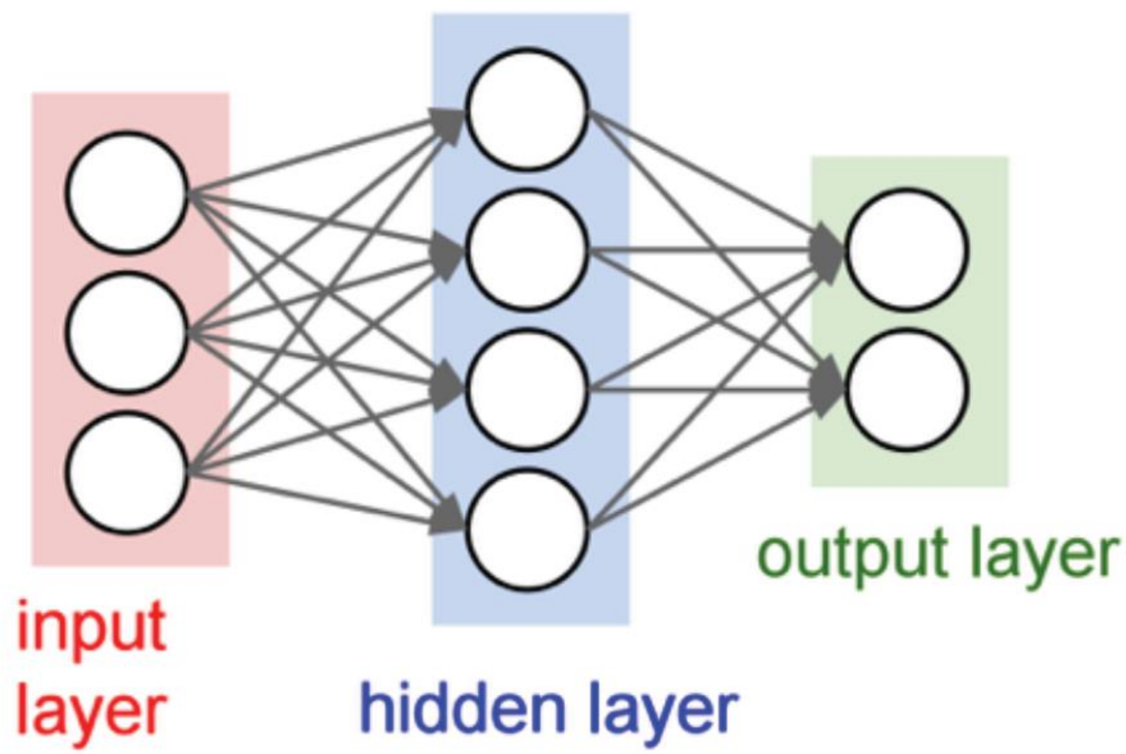



**(1D vectors)**

*Figure: Andrej Karpathy*

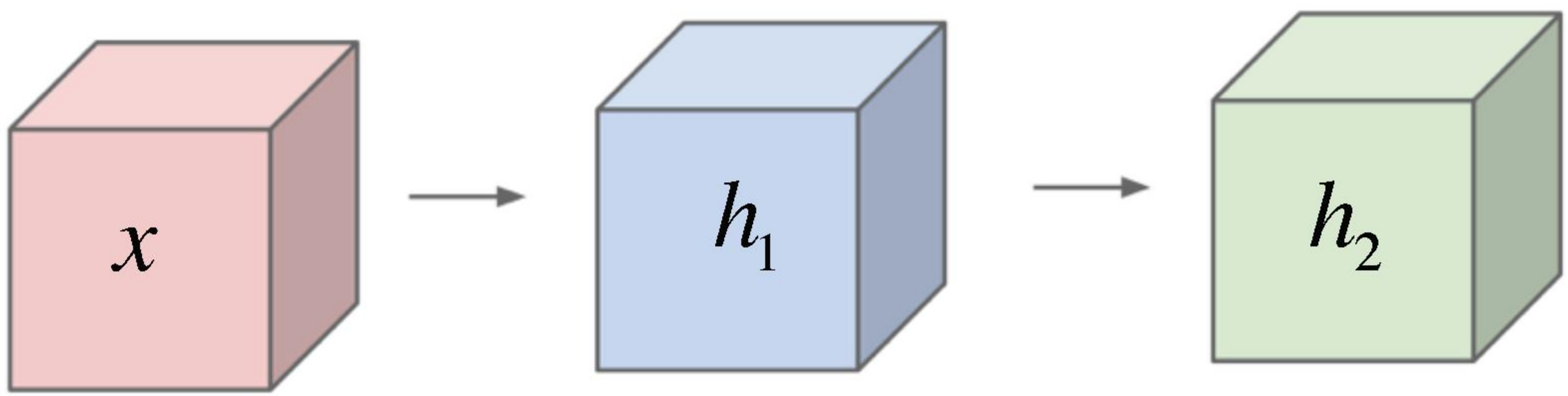
# 3D Activations

before:




**(1D vectors)**

now:




**(3D arrays)**

*Figure: Andrej Karpathy*

# 3D Activations

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

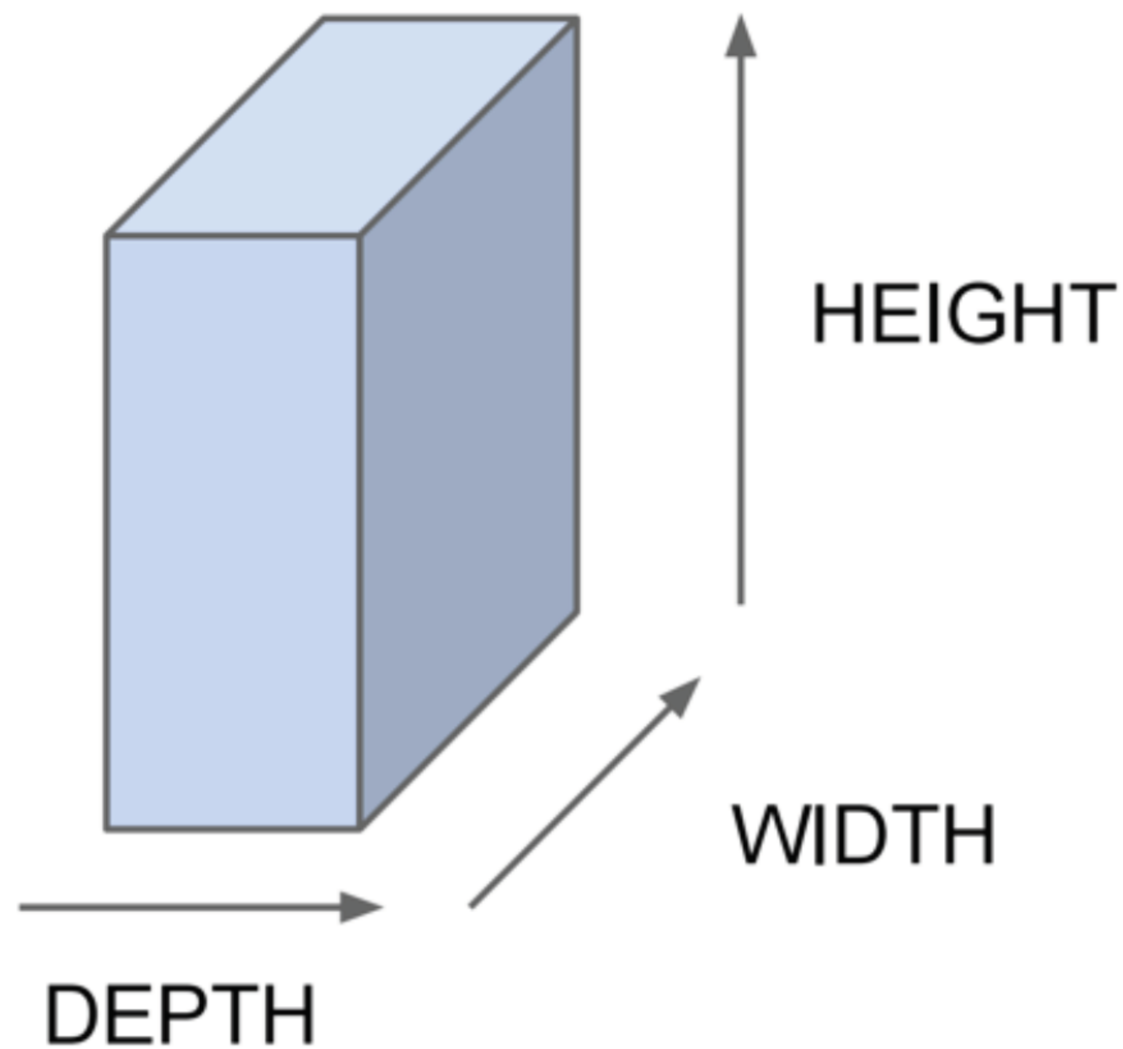
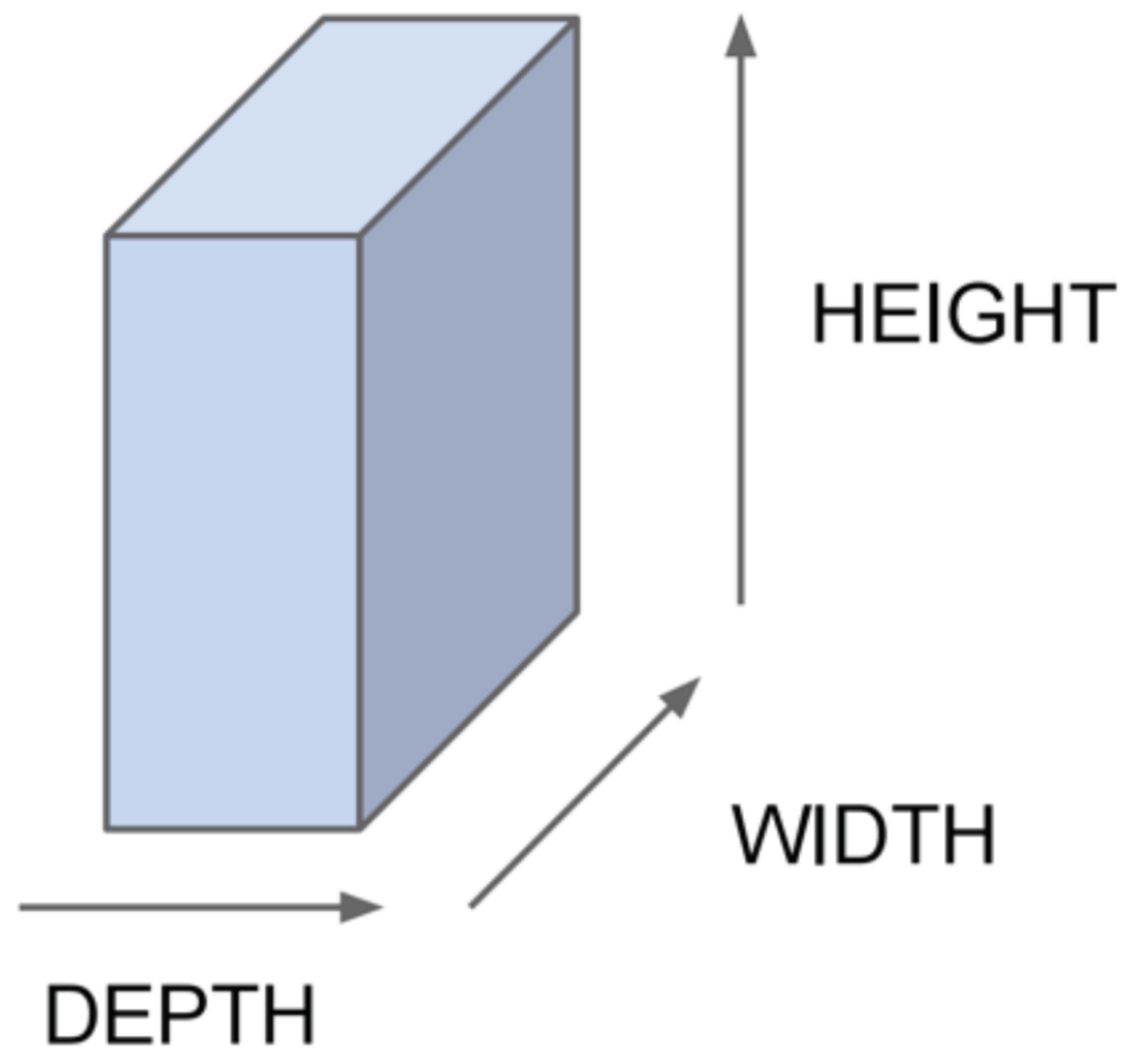



*Figure: Andrej Karpathy*

# 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:**

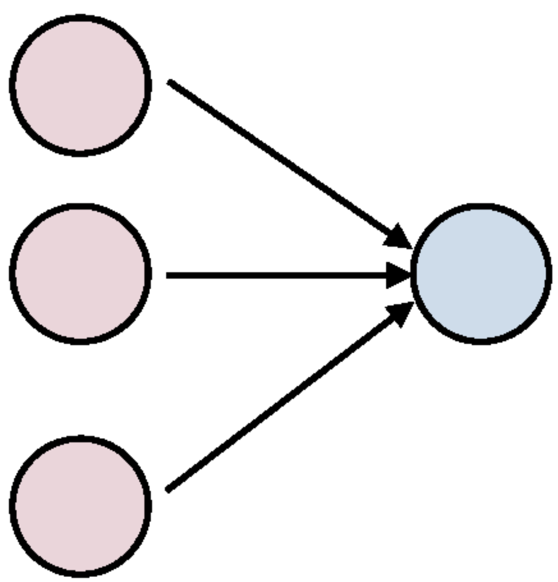
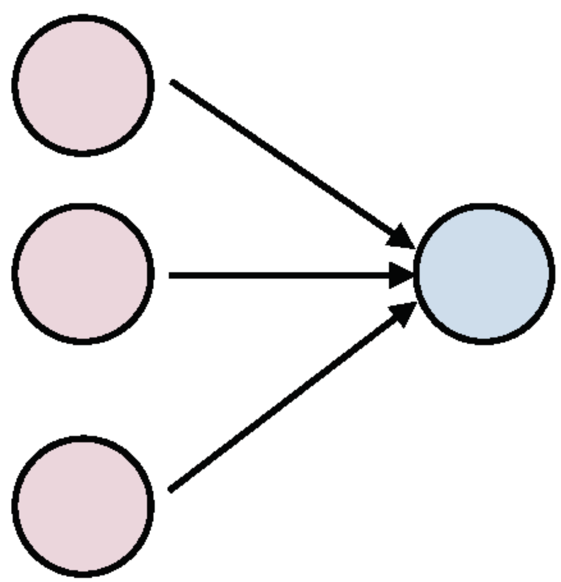


*Figure: Andrej Karpathy*

# 3D Activations

**1D Activations:**



**3D Activations:**

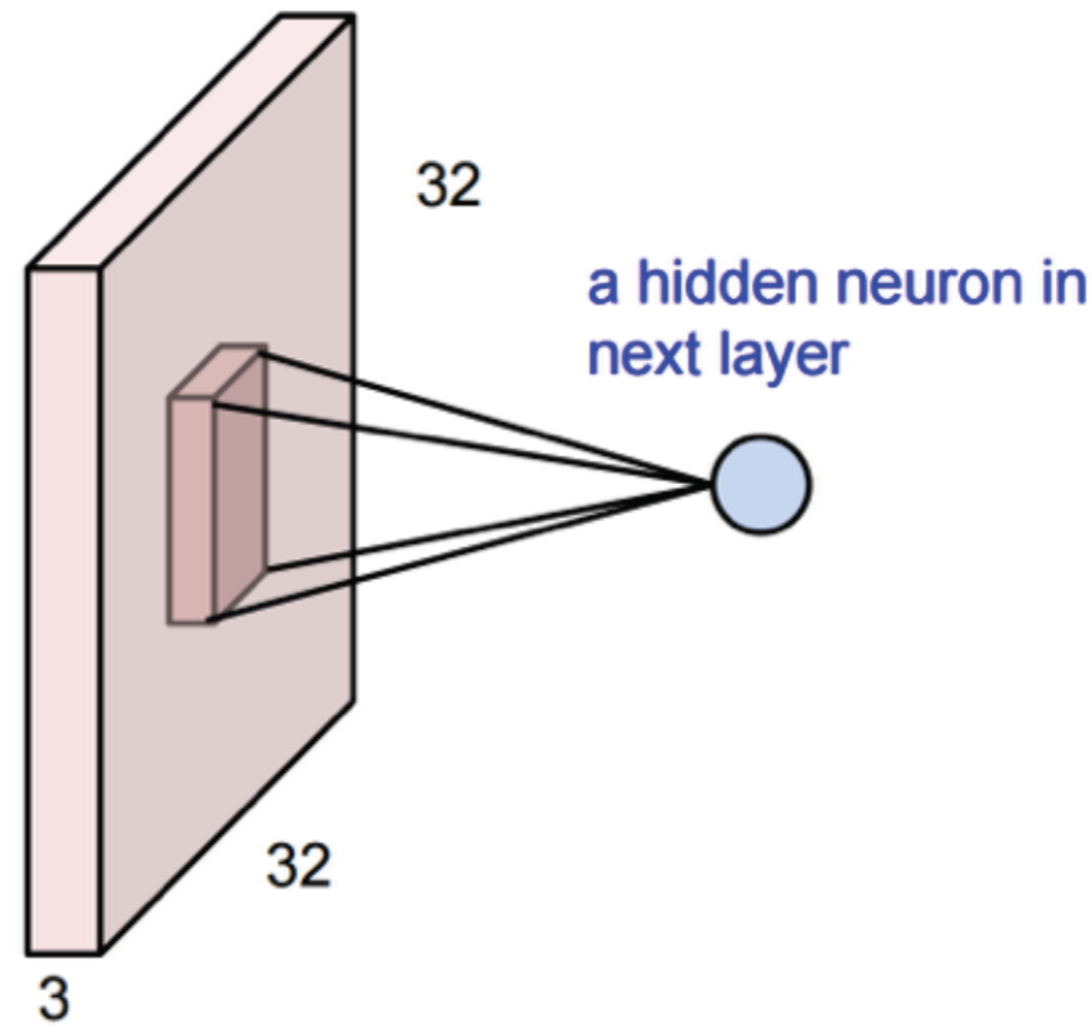
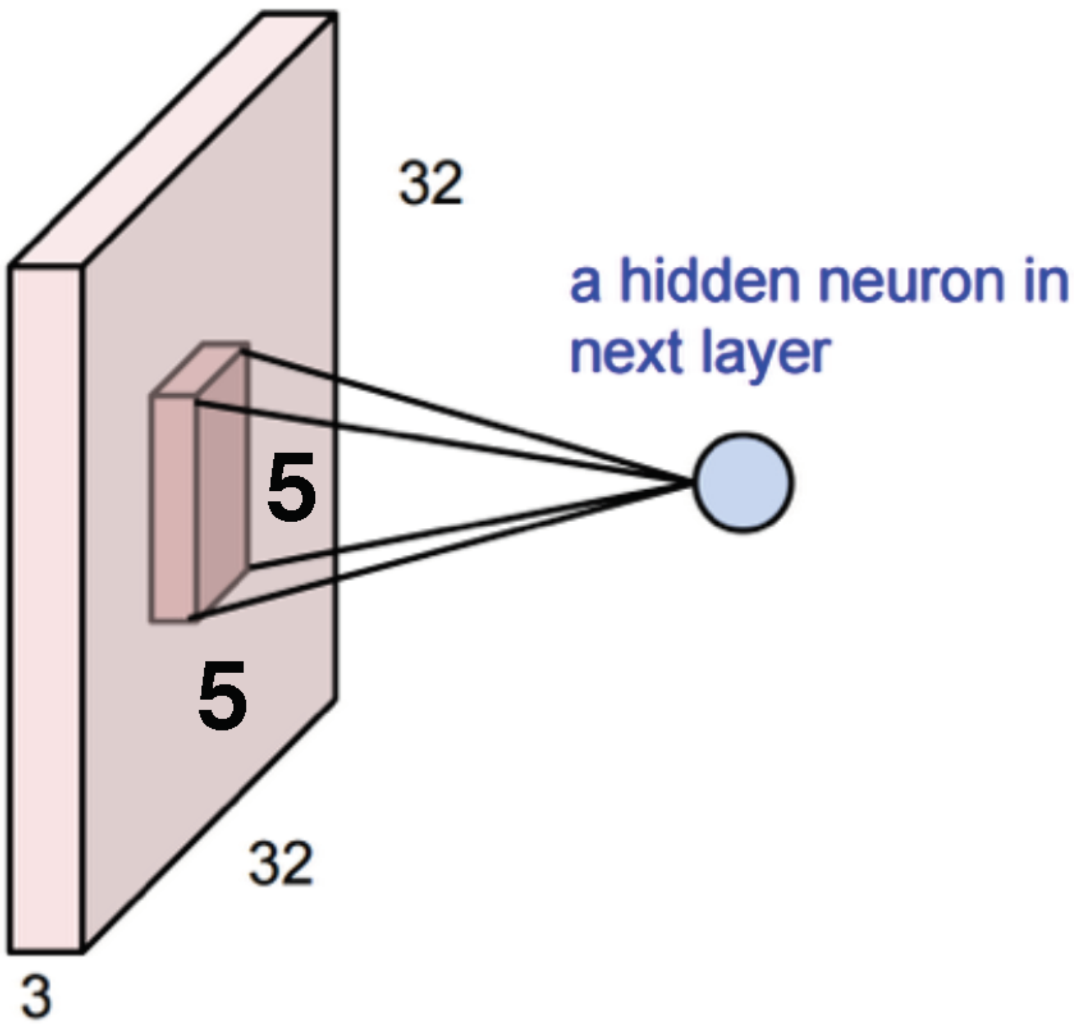



*Figure: Andrej Karpathy*

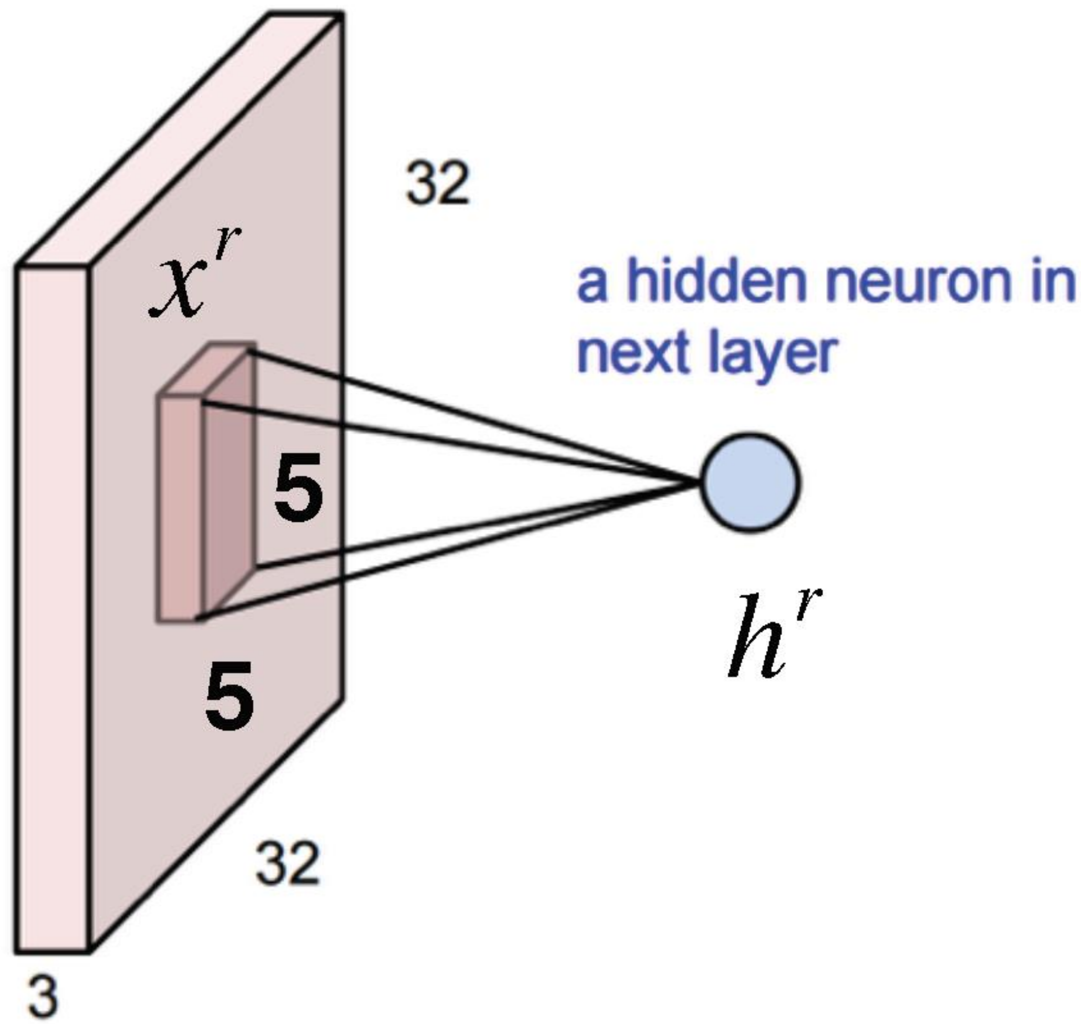
# 3D Activations




- 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

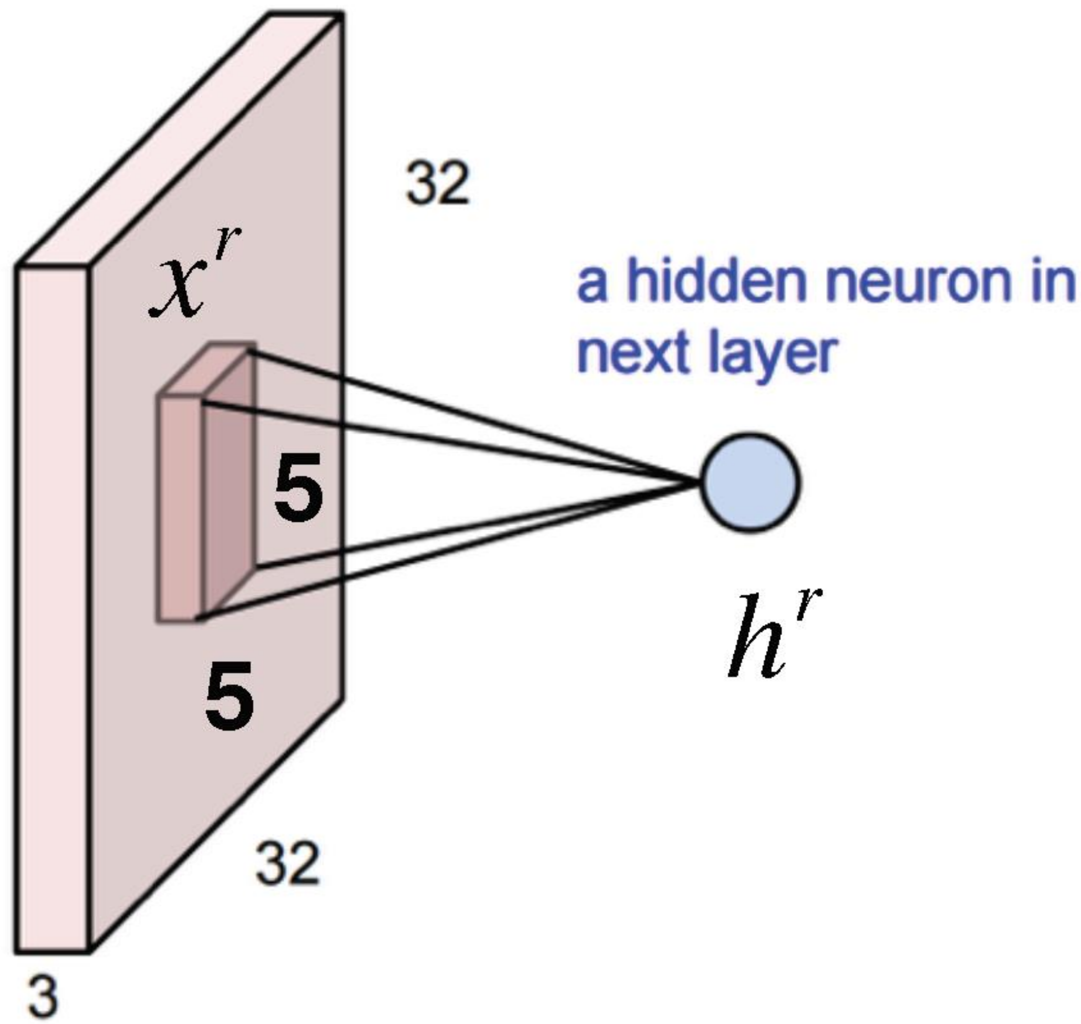



Example: consider the region of the input "$x^r$"

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

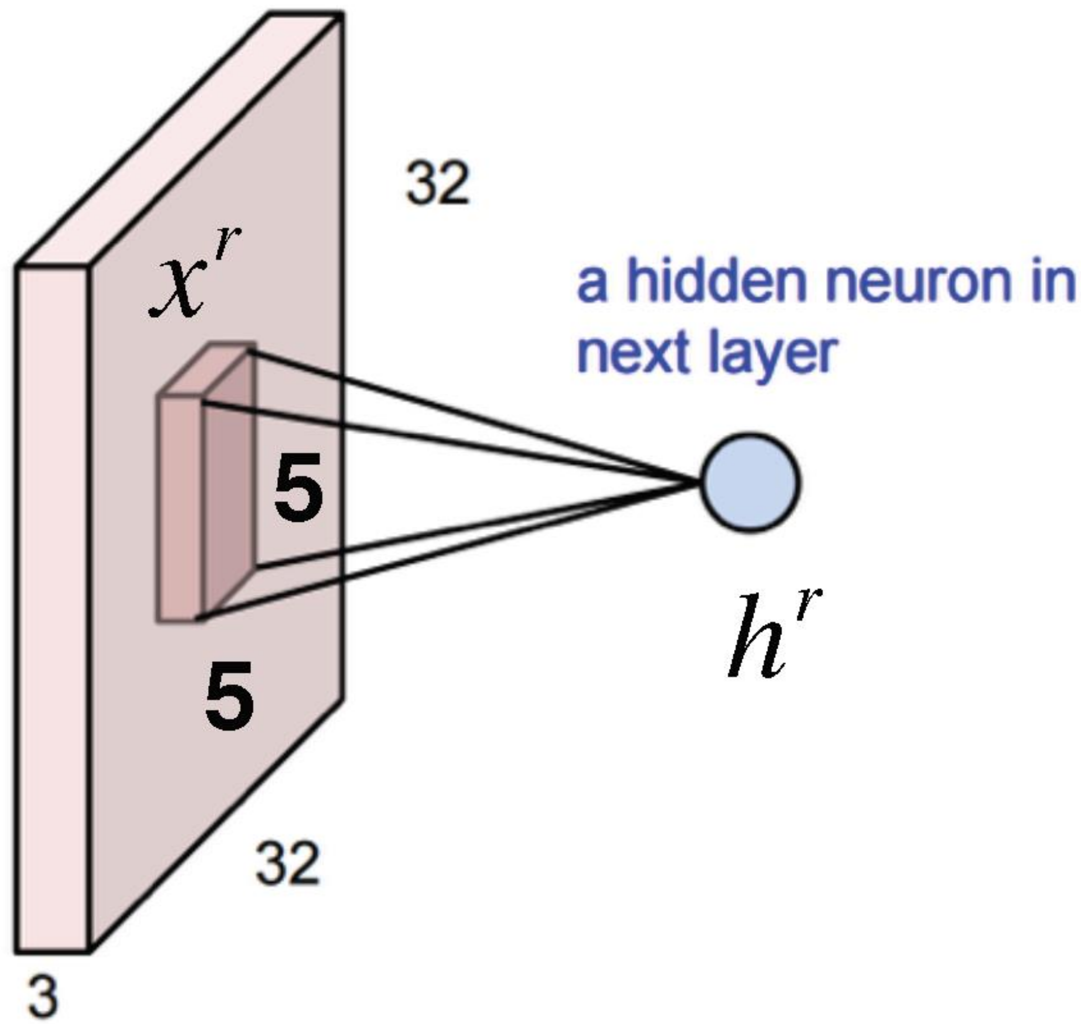



Figure: Andrej Karpathy

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

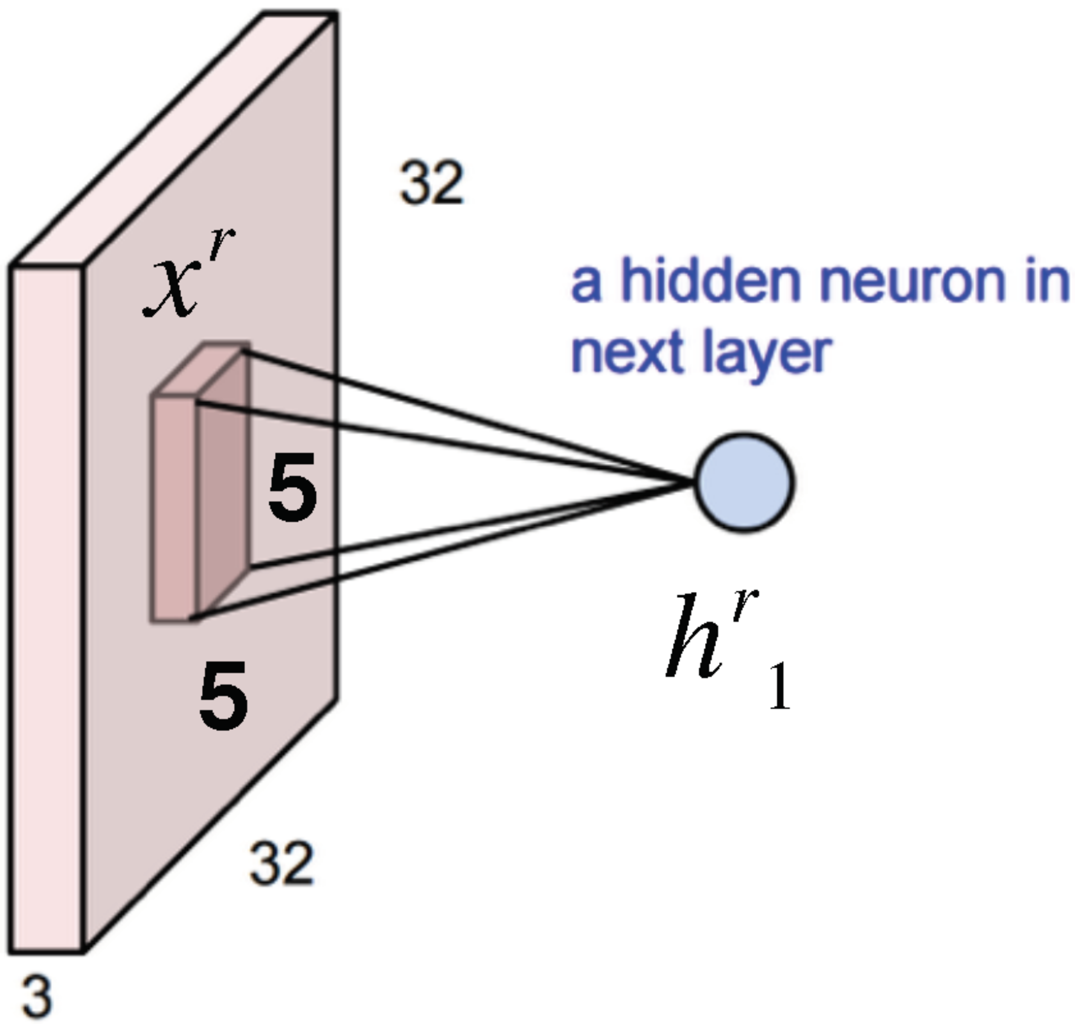



*Figure: Andrej Karpathy*

# 3D Activations

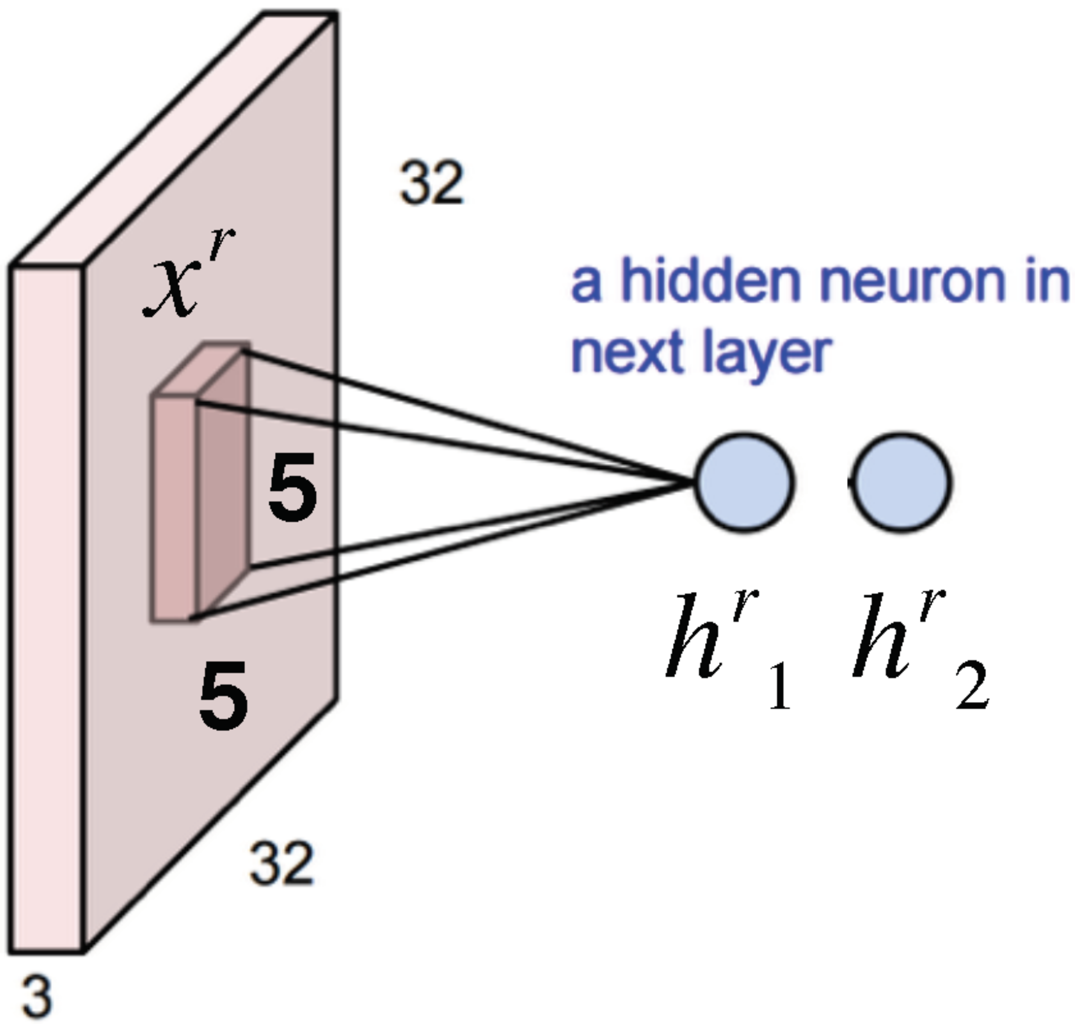
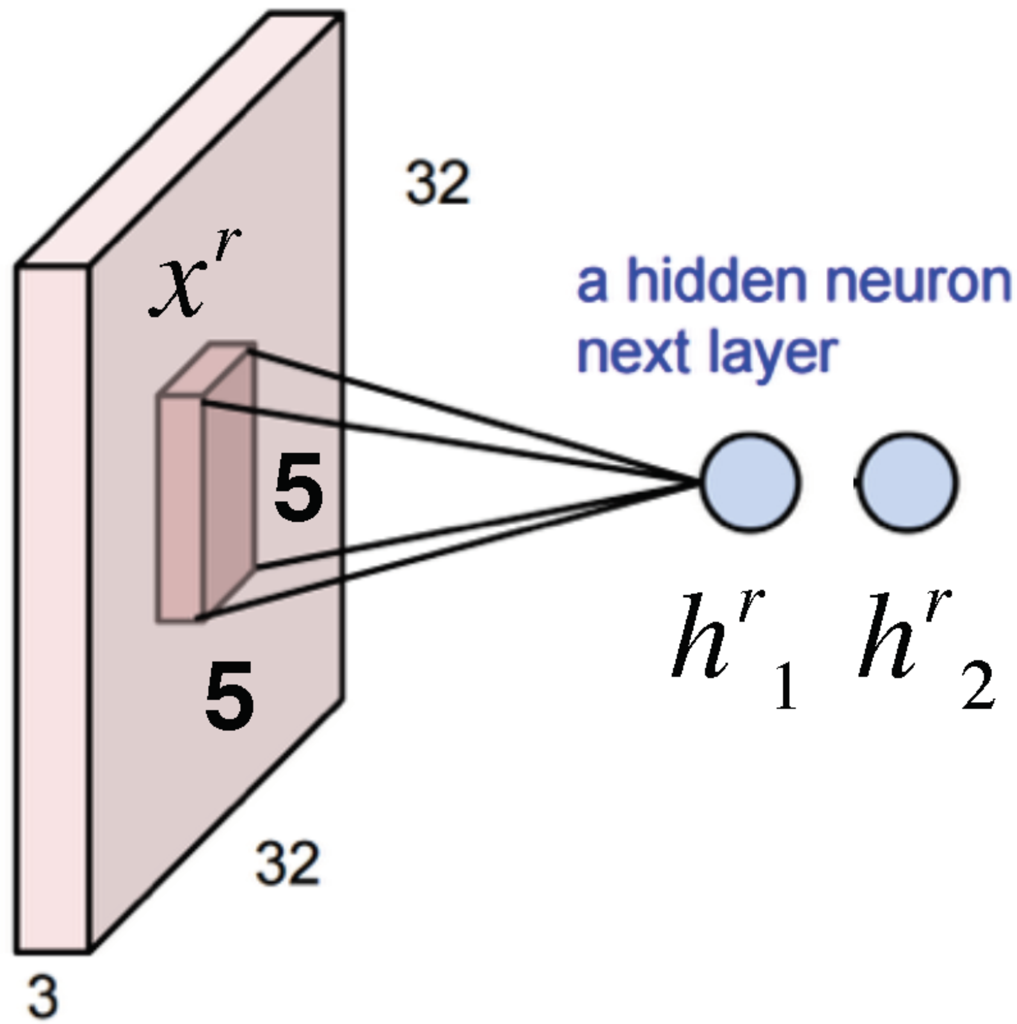



*Figure: Andrej Karpathy*

# 3D Activations



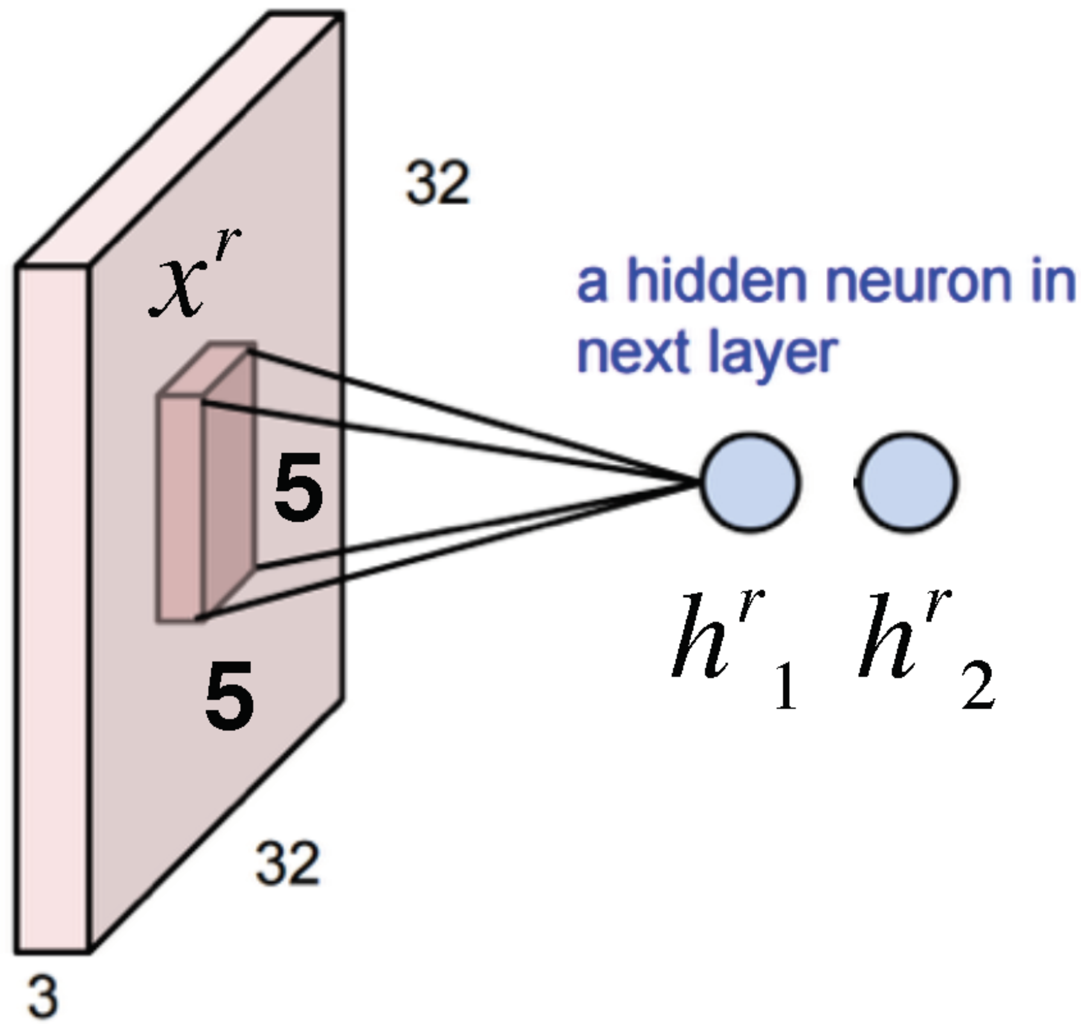

With **2** output neurons

$$h^r{}_1 = \sum_{ijk} x^r{}_{ijk} W_{1ijk} + b_1$$

$$h^r{}_2 = \sum_{ijk} x^r{}_{ijk} W_{2ijk} + b_2$$

*Figure: Andrej Karpathy*

# 3D Activations




With **2** output neurons

$$h^r{}_1 = \sum_{ijk} x^r{}_{ijk} W_{\boxed{1}ijk} + b_{\boxed{1}}$$

$$h^r{}_2 = \sum_{ijk} x^r{}_{ijk} W_{\boxed{2}ijk} + b_{\boxed{2}}$$

*Figure: Andrej Karpathy*

# 3D Activations

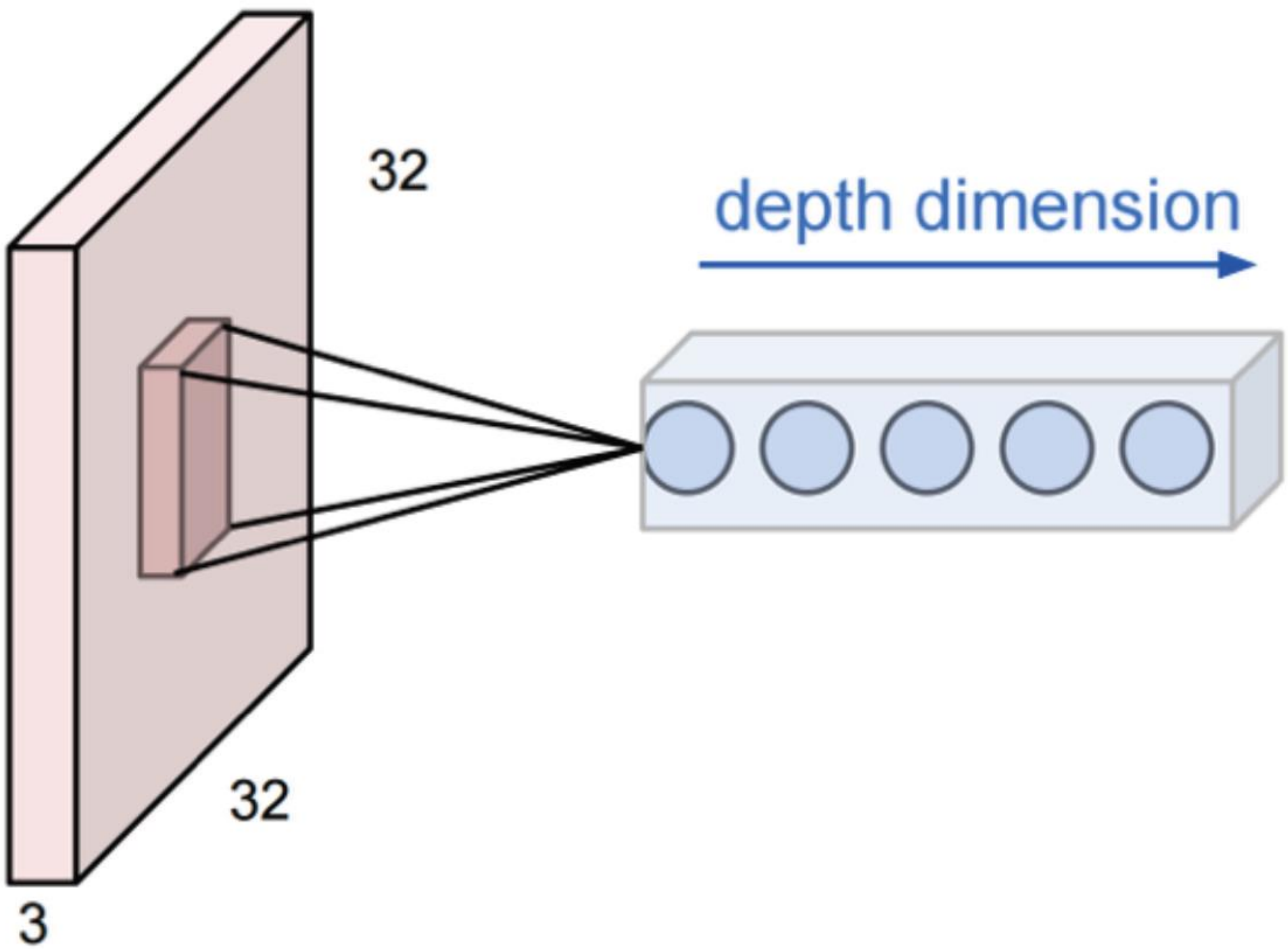
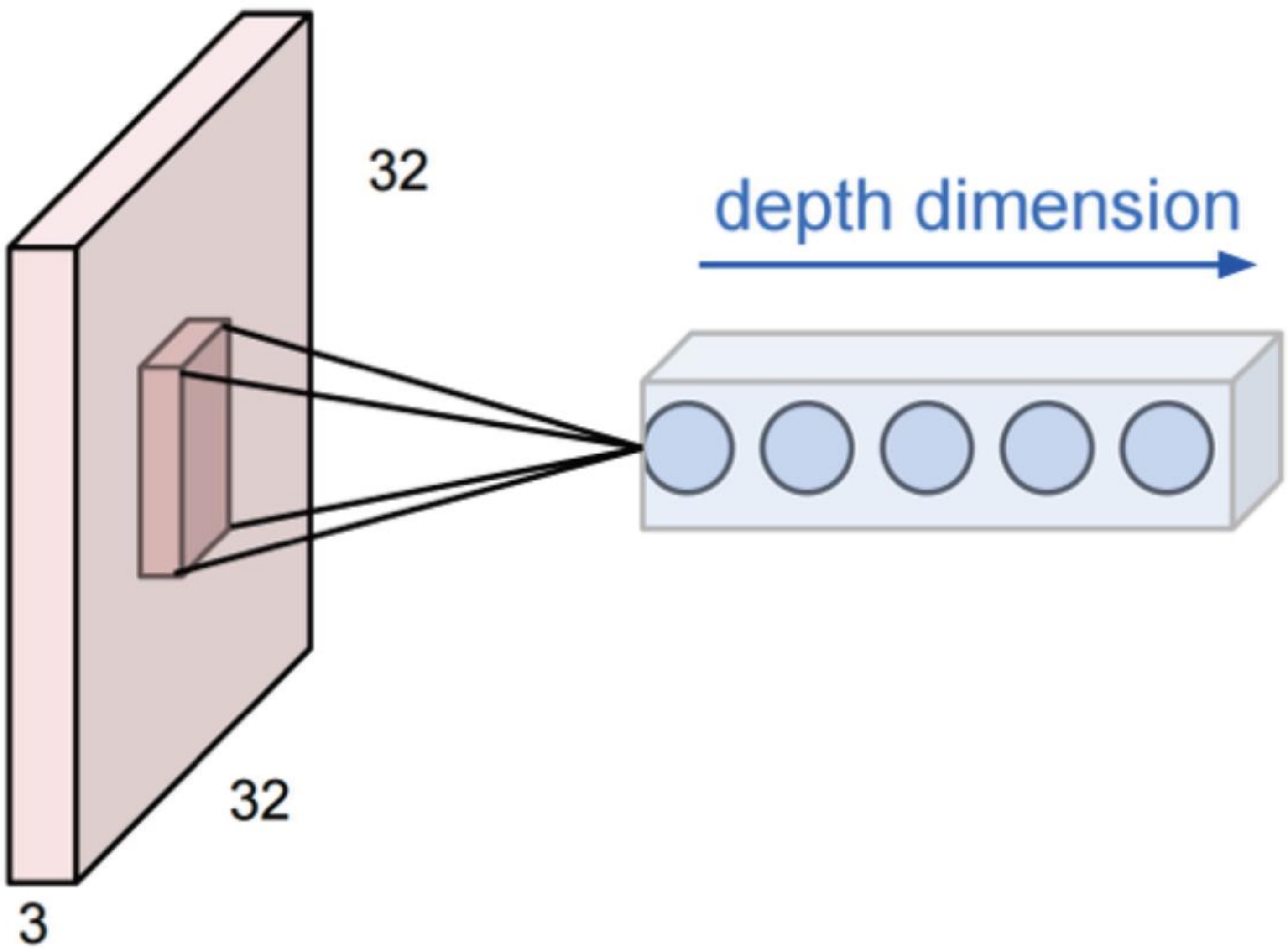



*Figure: Andrej Karpathy*

# 3D Activations

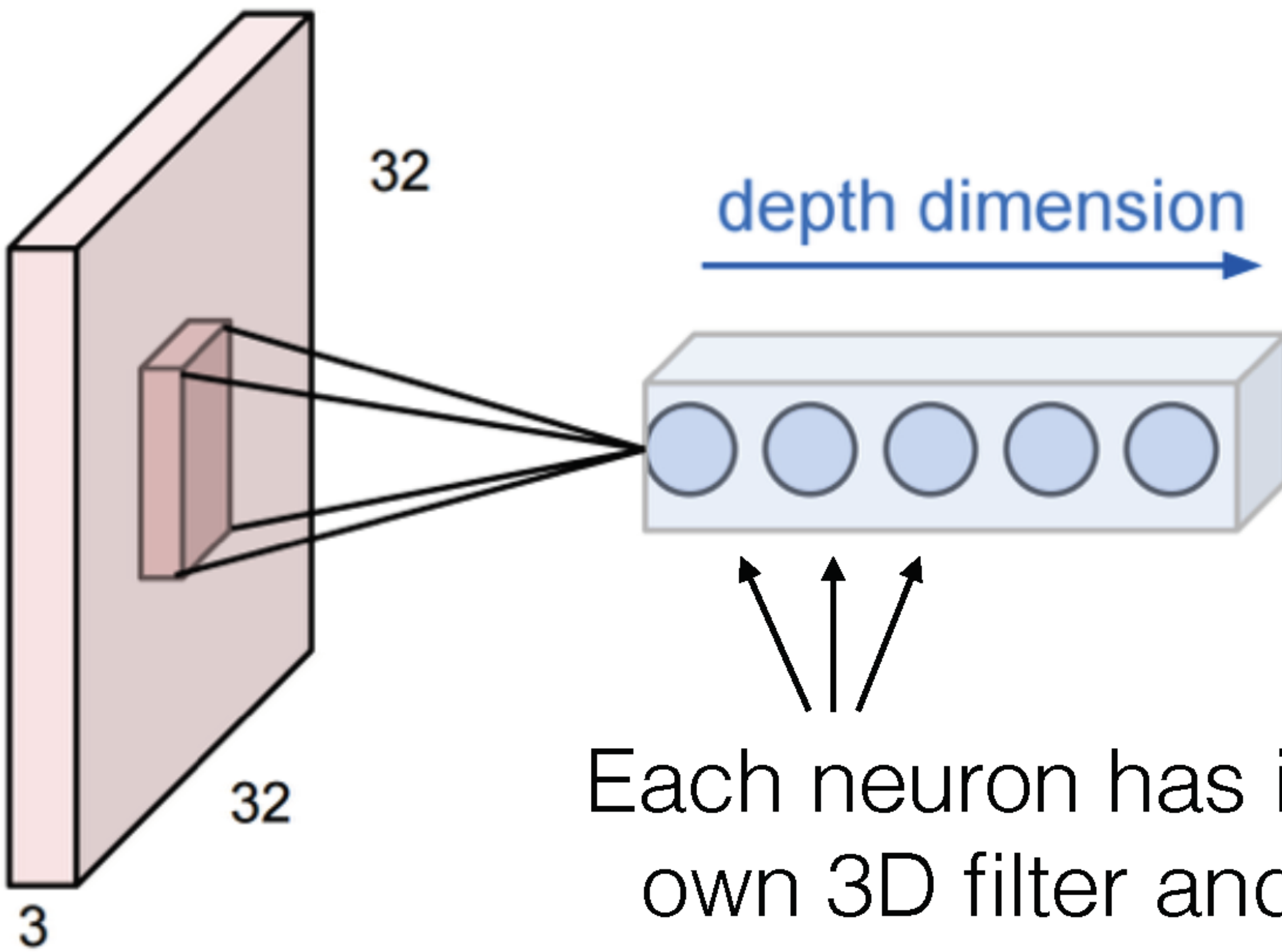



We can keep adding more outputs

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

*Figure: Andrej Karpathy*

# 3D Activations



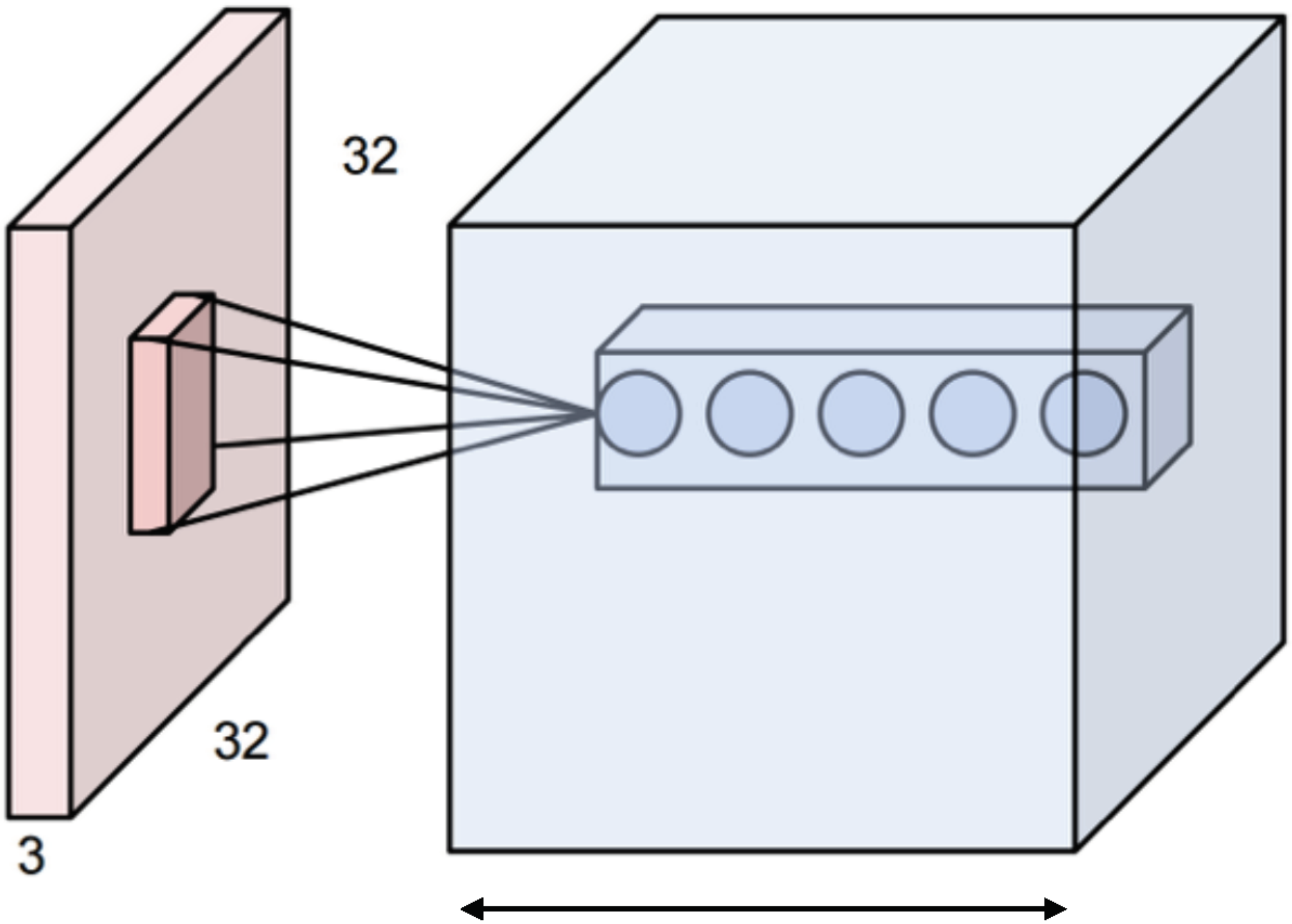

Each neuron has its own 3D filter and own (scalar) bias

We can keep adding more outputs

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

*Figure: Andrej Karpathy*

# 3D Activations

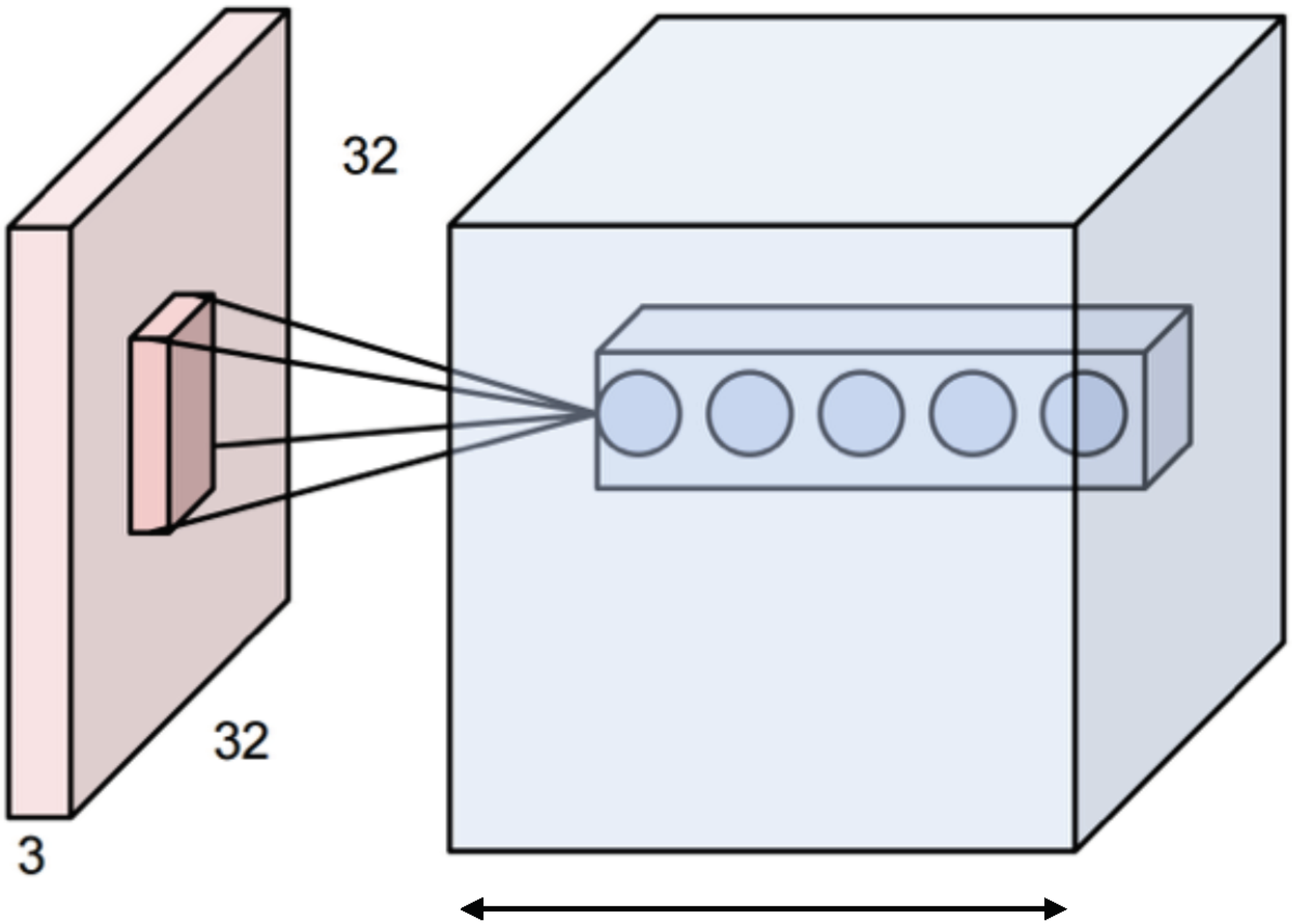



*D* sets of weights
(also called filters)

Now repeat this across the input

*Figure: Andrej Karpathy*

# 3D Activations



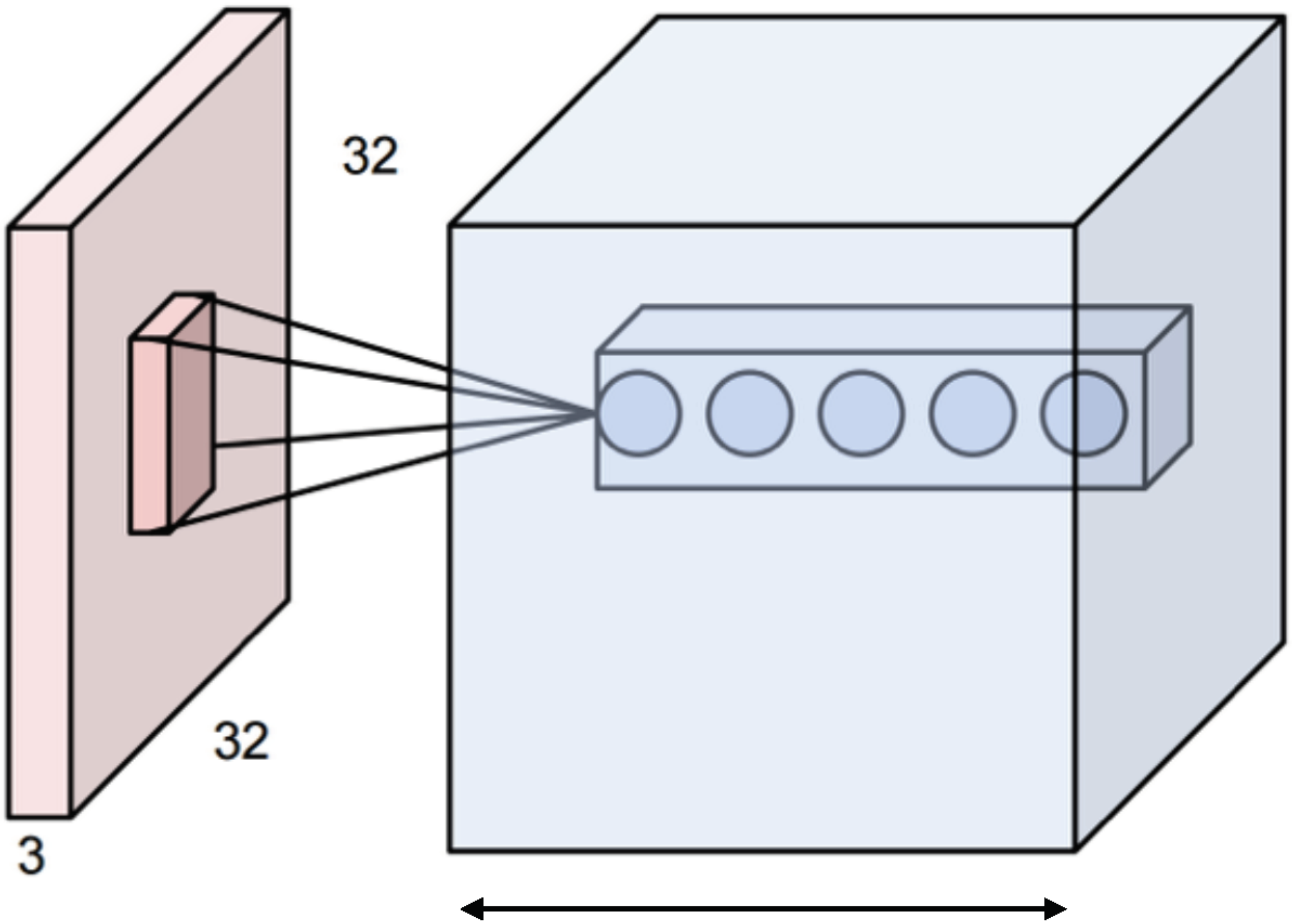

*D* sets of weights
(also called filters)

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*

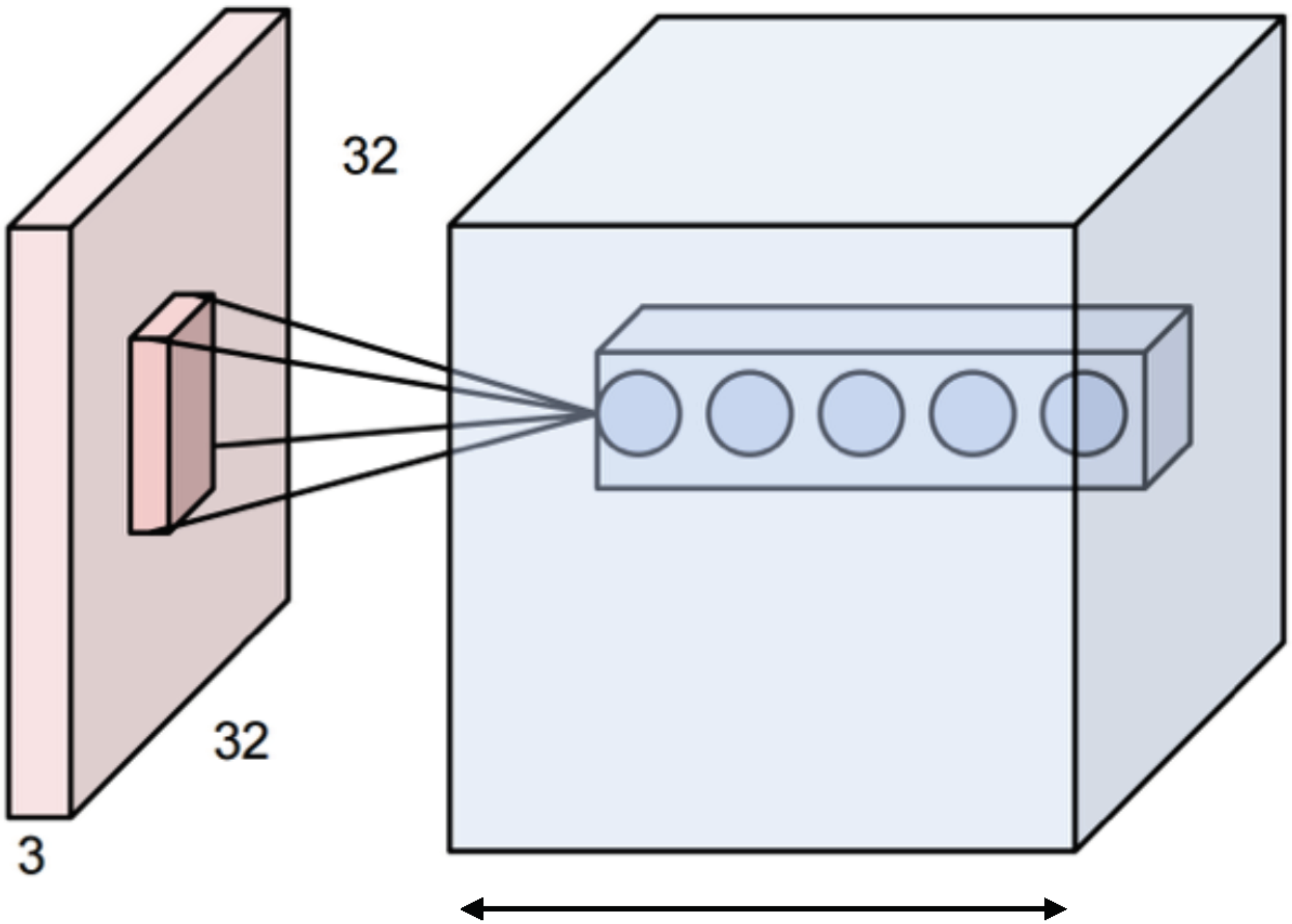
# 3D Activations




*D* sets of weights
(also called filters)

*Figure: Andrej Karpathy*

# 3D Activations

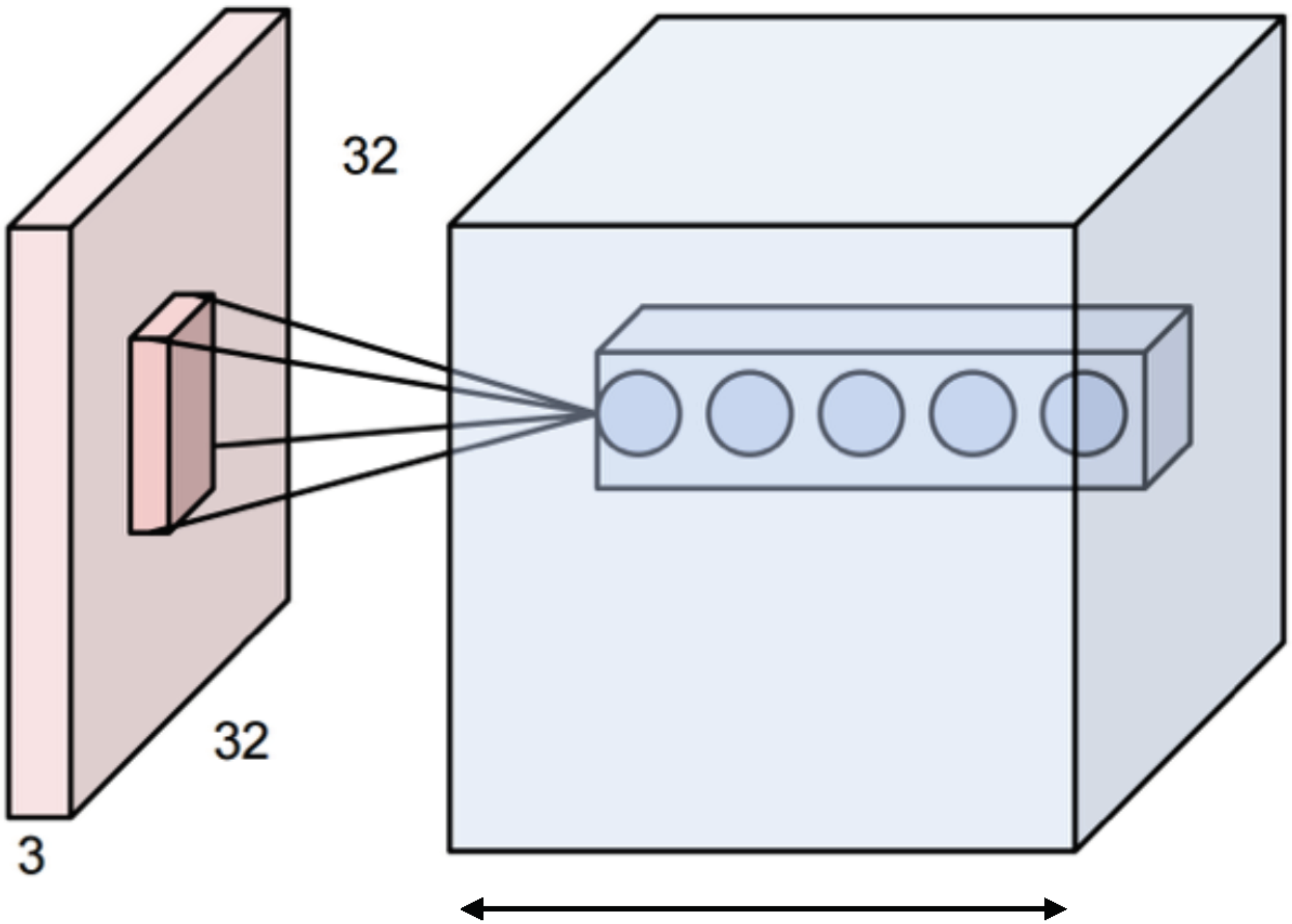



With weight sharing, this is called **convolution**

*D* sets of weights (also called filters)

*Figure: Andrej Karpathy*

# 3D Activations



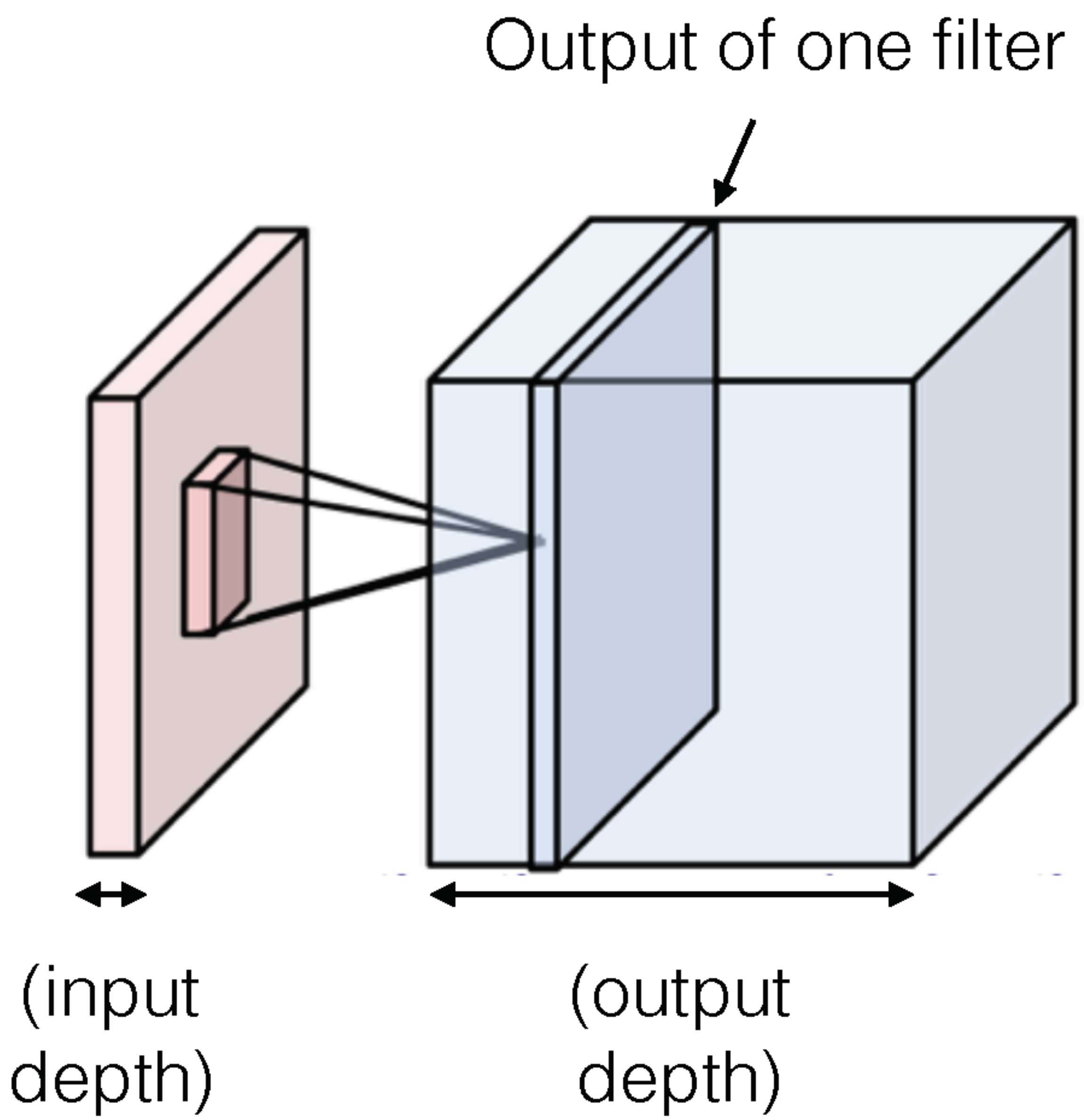

*D* sets of weights
(also called filters)

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

Without weight
sharing,
this is called a
**locally**
**connected layer**

*Figure: Andrej Karpathy*

# 3D Activations

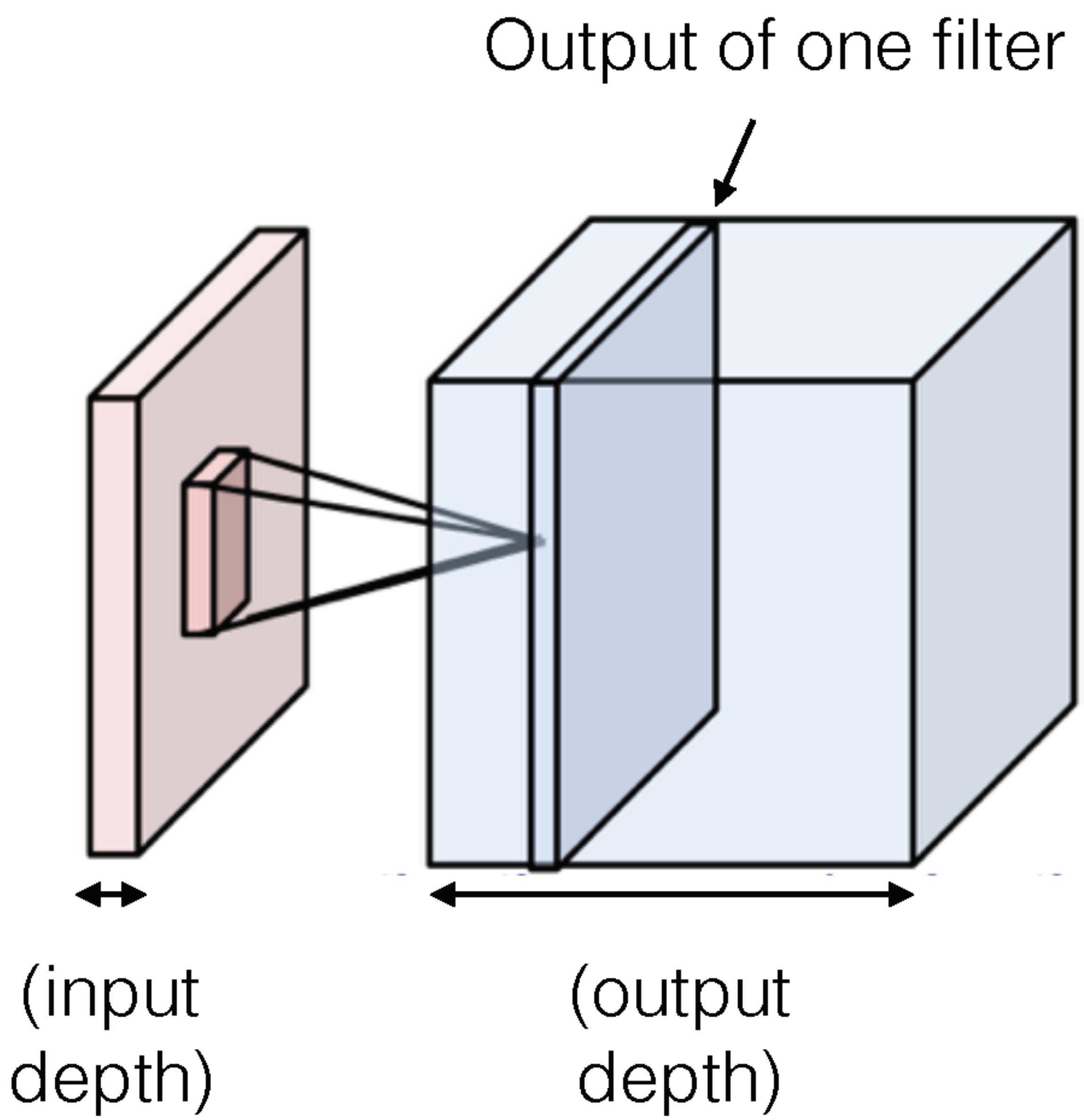



One set of weights gives one slice in the output

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:**

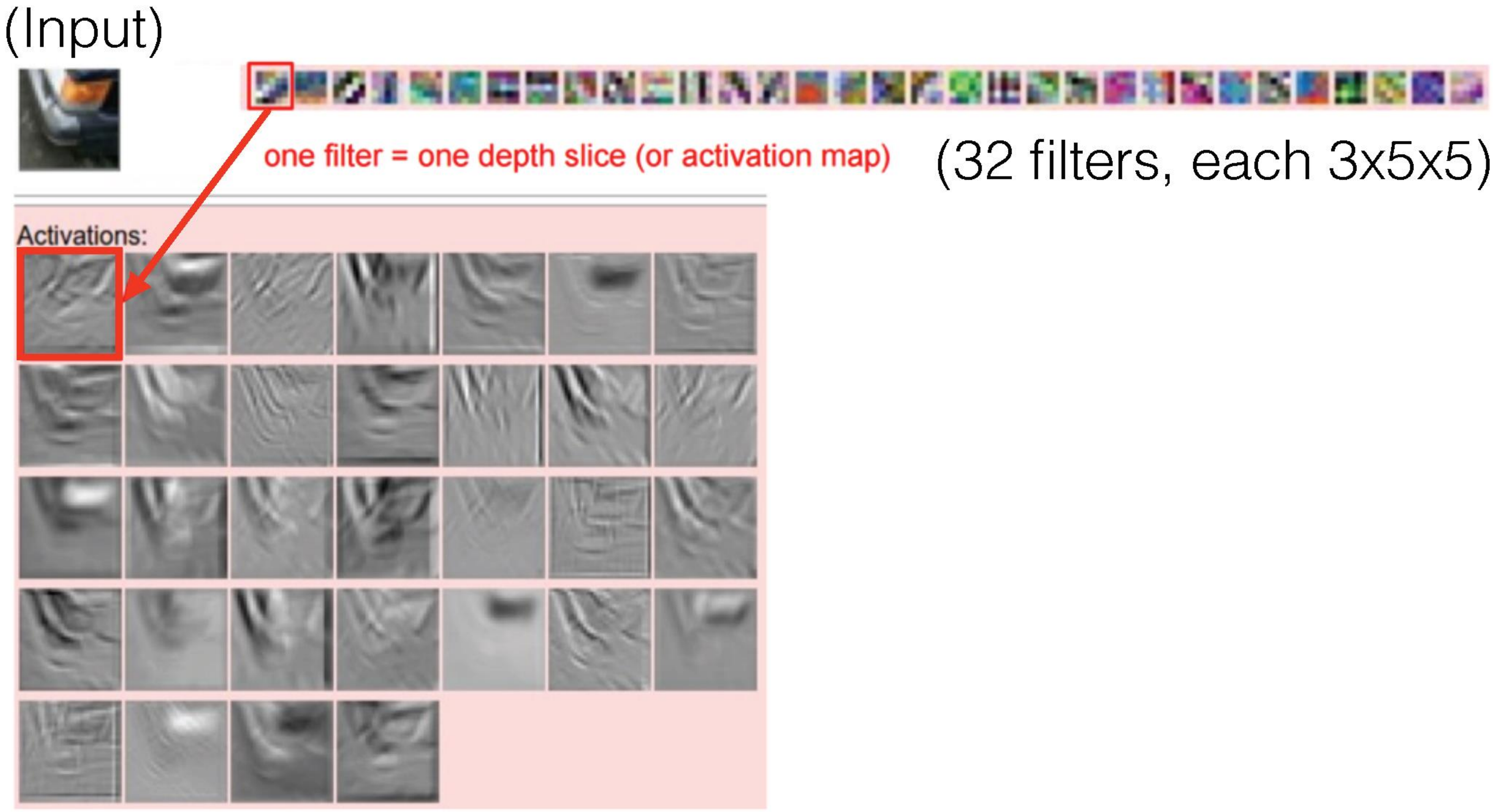



*Figure: Andrej Karpathy*

# 3D Activations

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

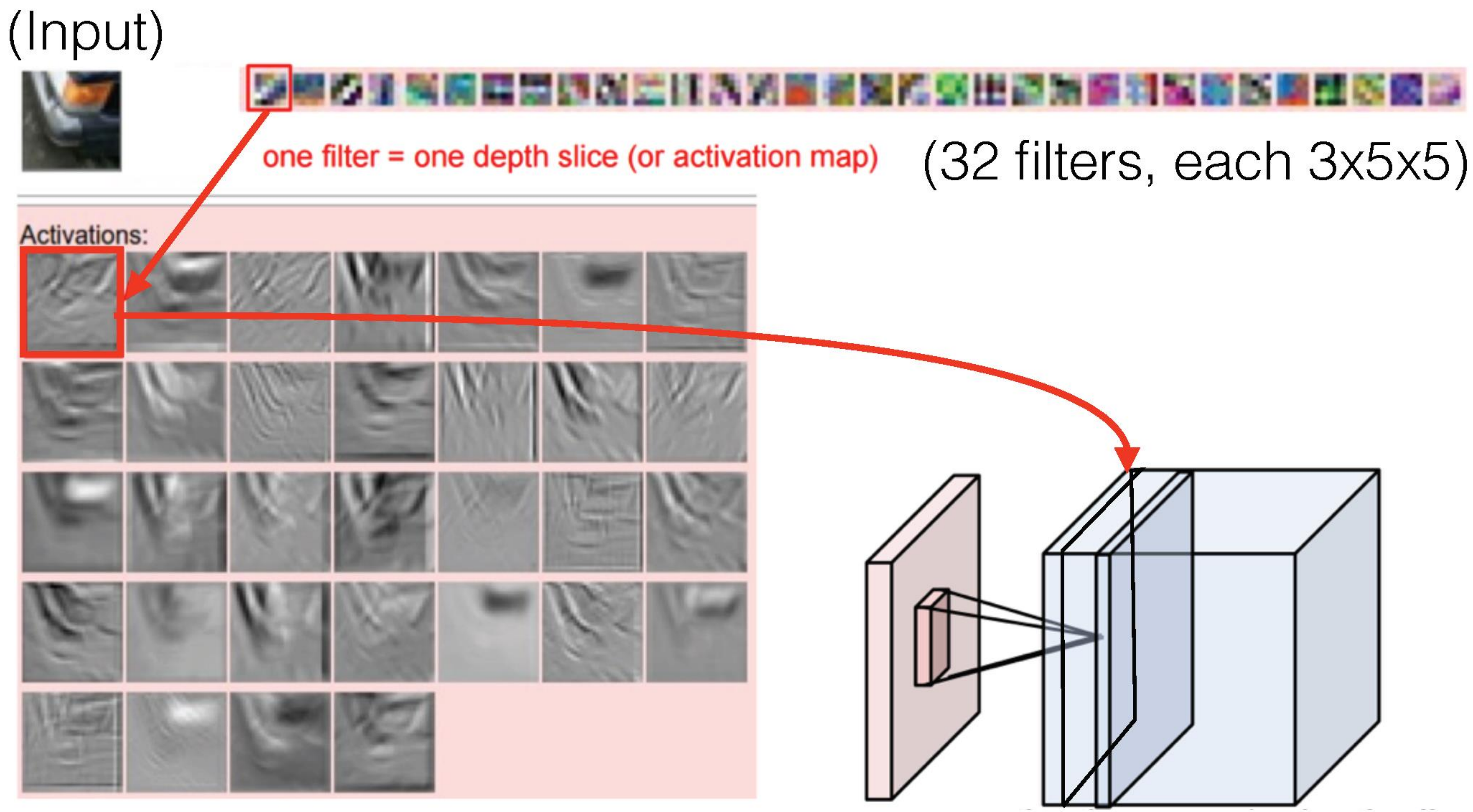



*Figure: Andrej Karpathy*

# 3D Activations

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

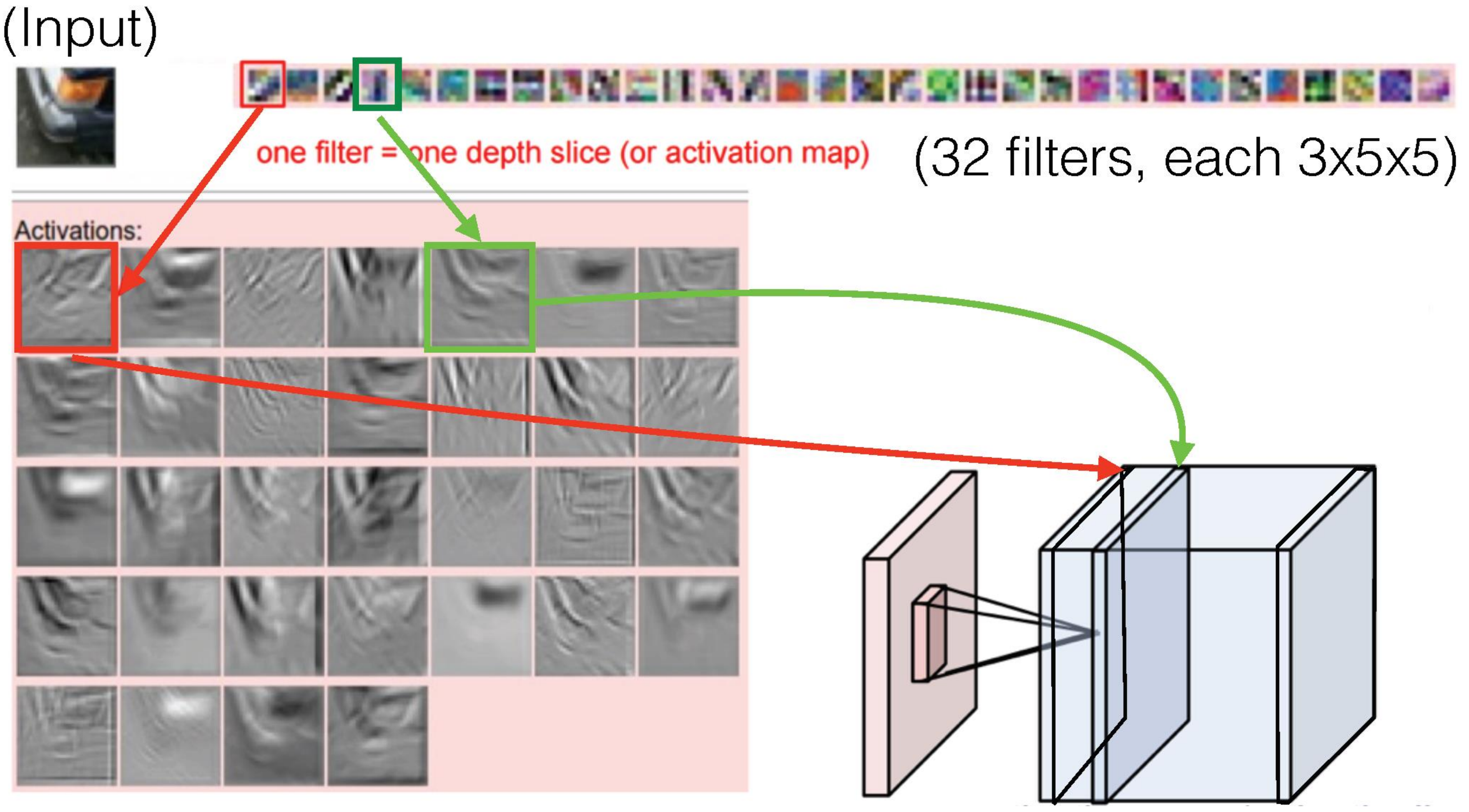



*Figure: Andrej Karpathy*

# 3D Activations

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

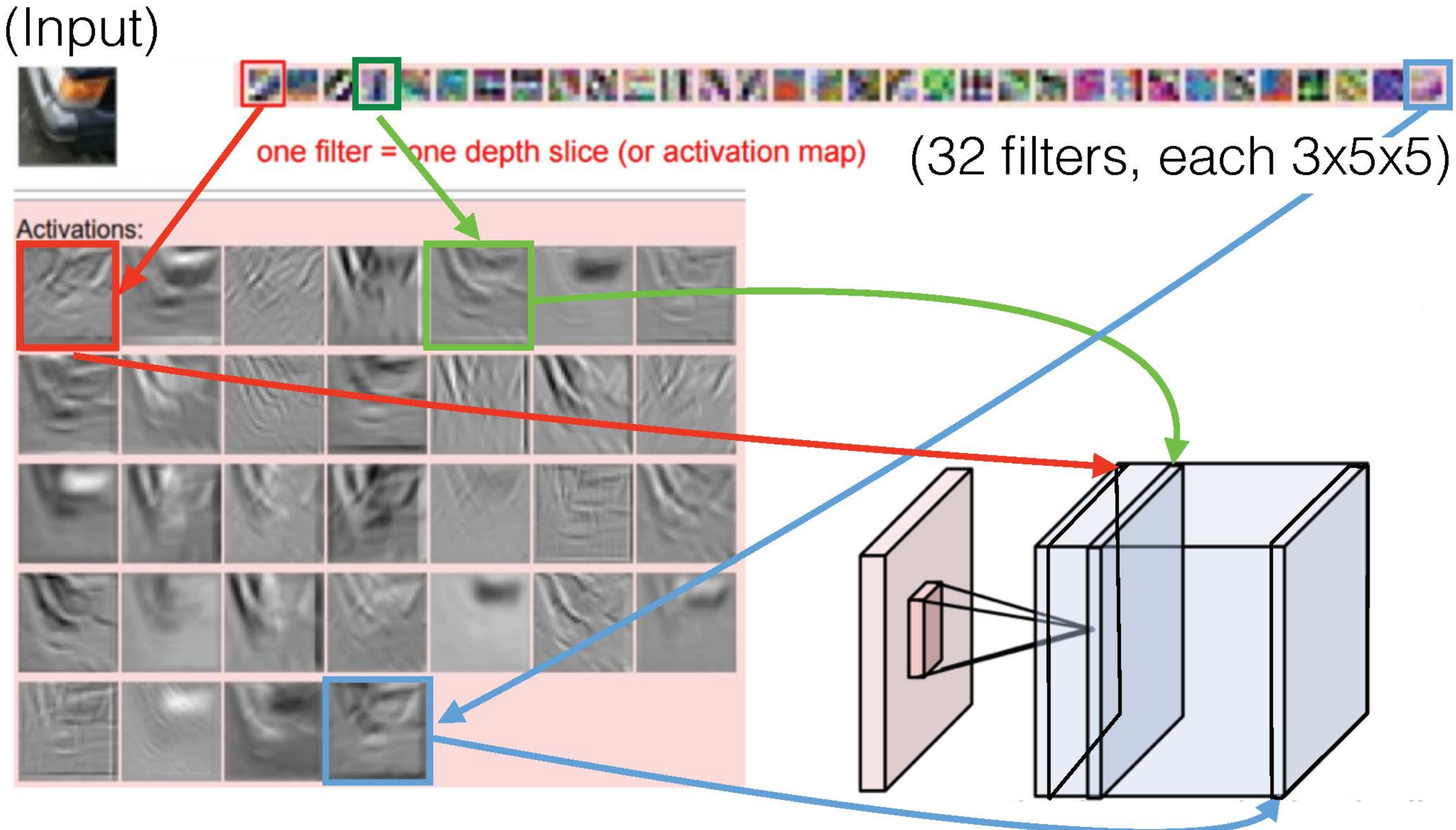
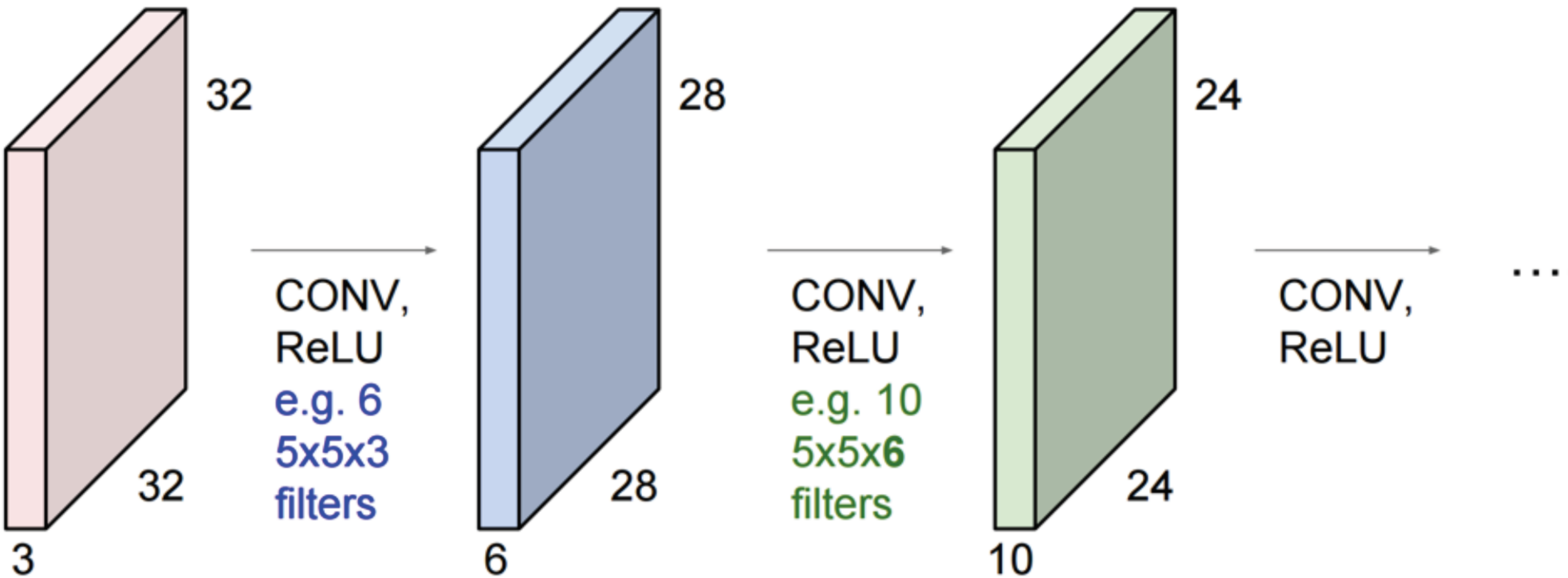



*Figure: Andrej Karpathy*

# Questions?

# (Recap)

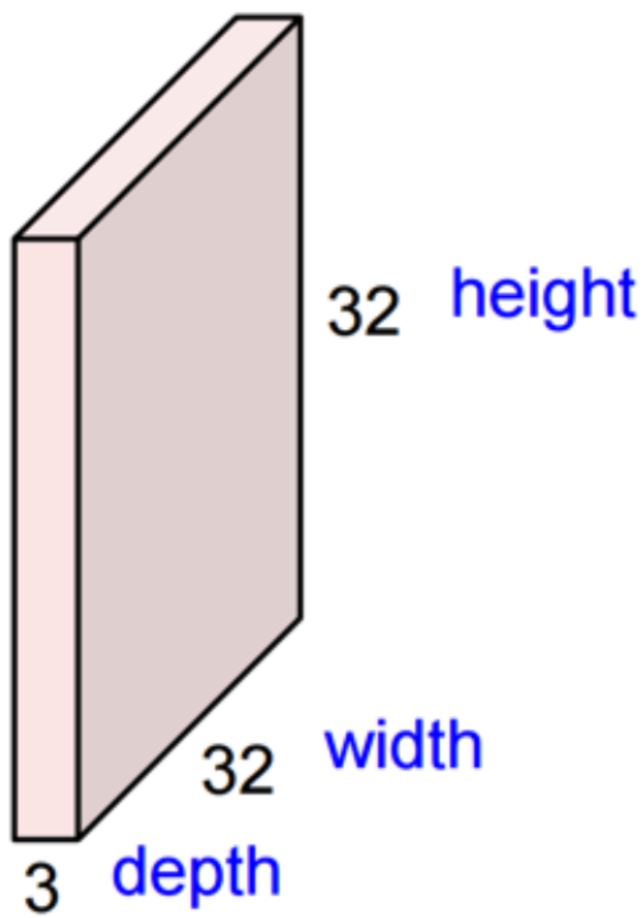
A **ConvNet** is a sequence of convolutional layers, interspersed with activation functions (and possibly other layer types)



# (Recap)

## Convolution Layer

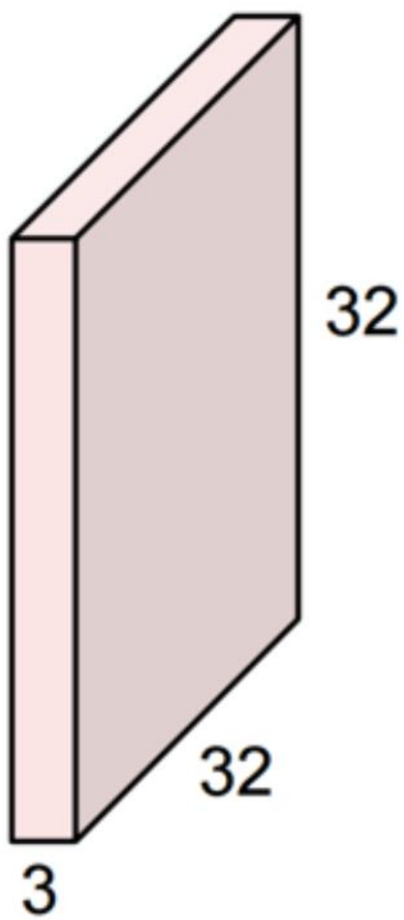
32x32x3 image



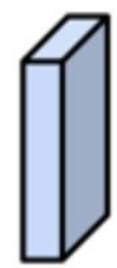
# (Recap)

## Convolution Layer

32x32x3 image




5x5x3 filter



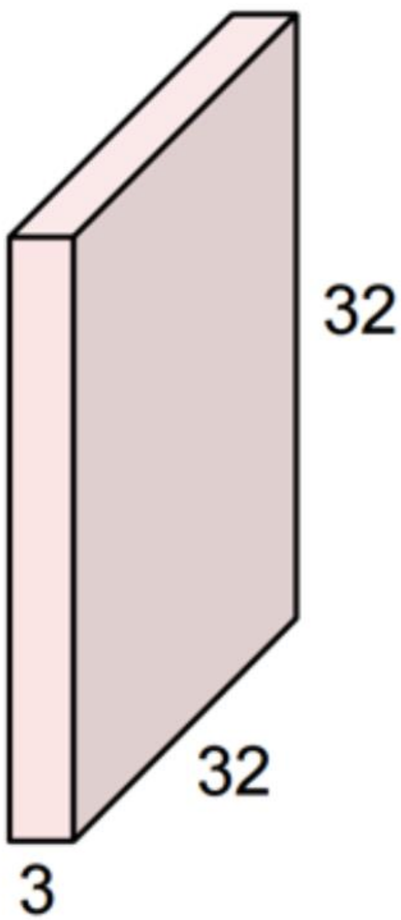
**Convolve** the filter with the image i.e. “slide over the image spatially, computing dot products”

# (Recap)

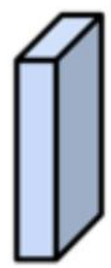
## Convolution Layer

Filters always extend the full depth of the input volume

32x32x3 image



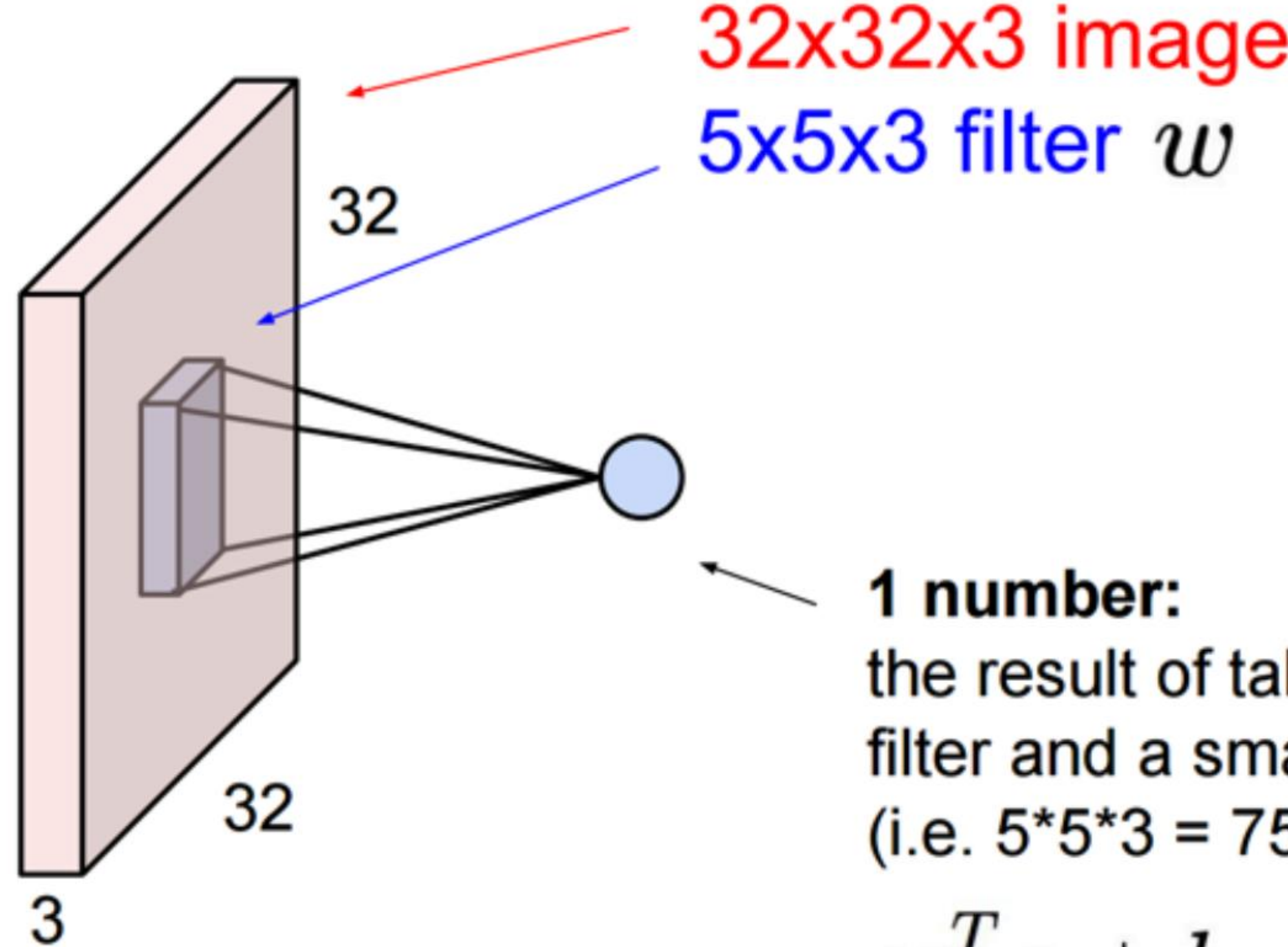

5x5x3 filter



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

# (Recap)

## Convolution Layer

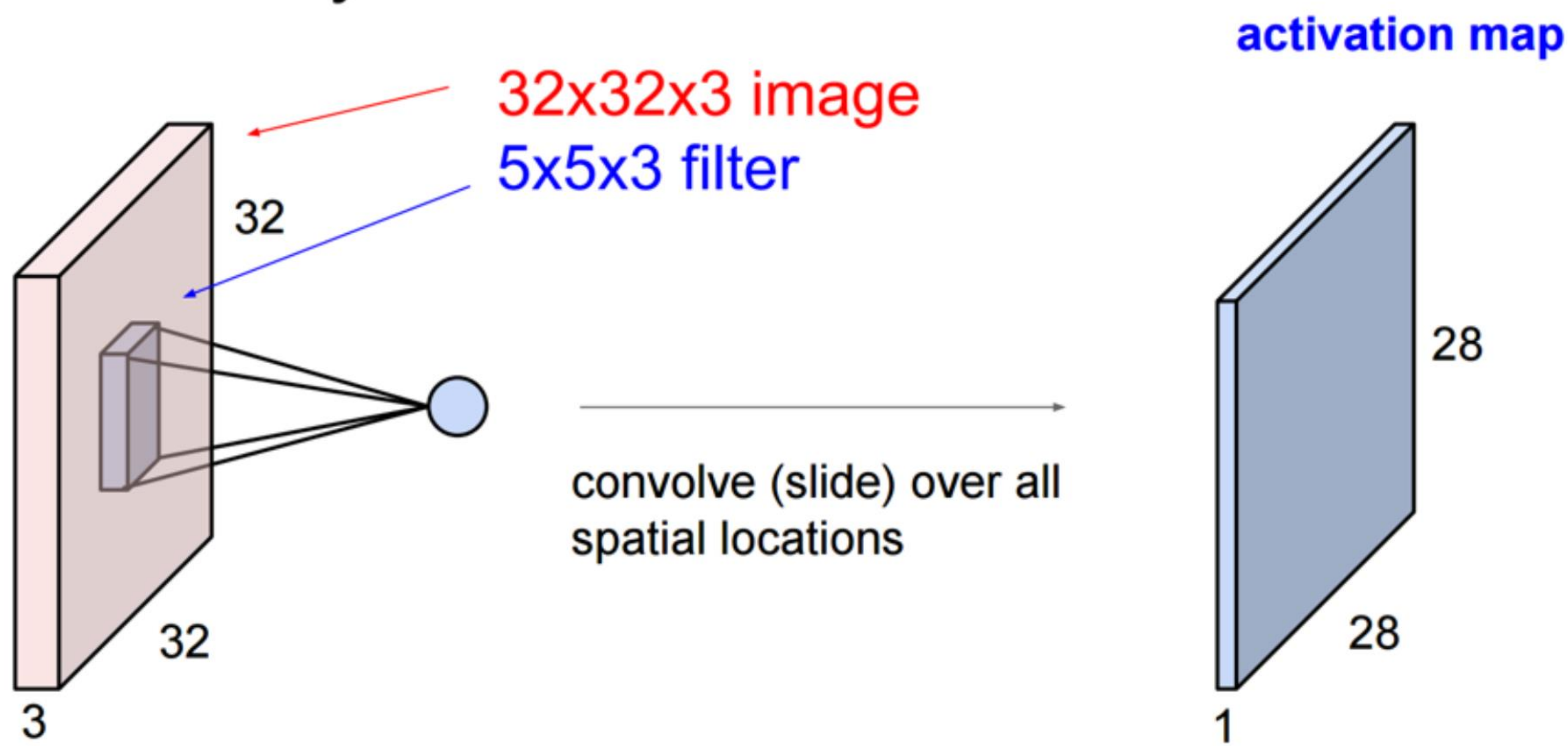



**1 number:**
the result of taking a dot product between the filter and a small 5x5x3 chunk of the image
(i.e. 5*5*3 = 75-dimensional dot product + bias)

$$w^T x + b$$

# (Recap)

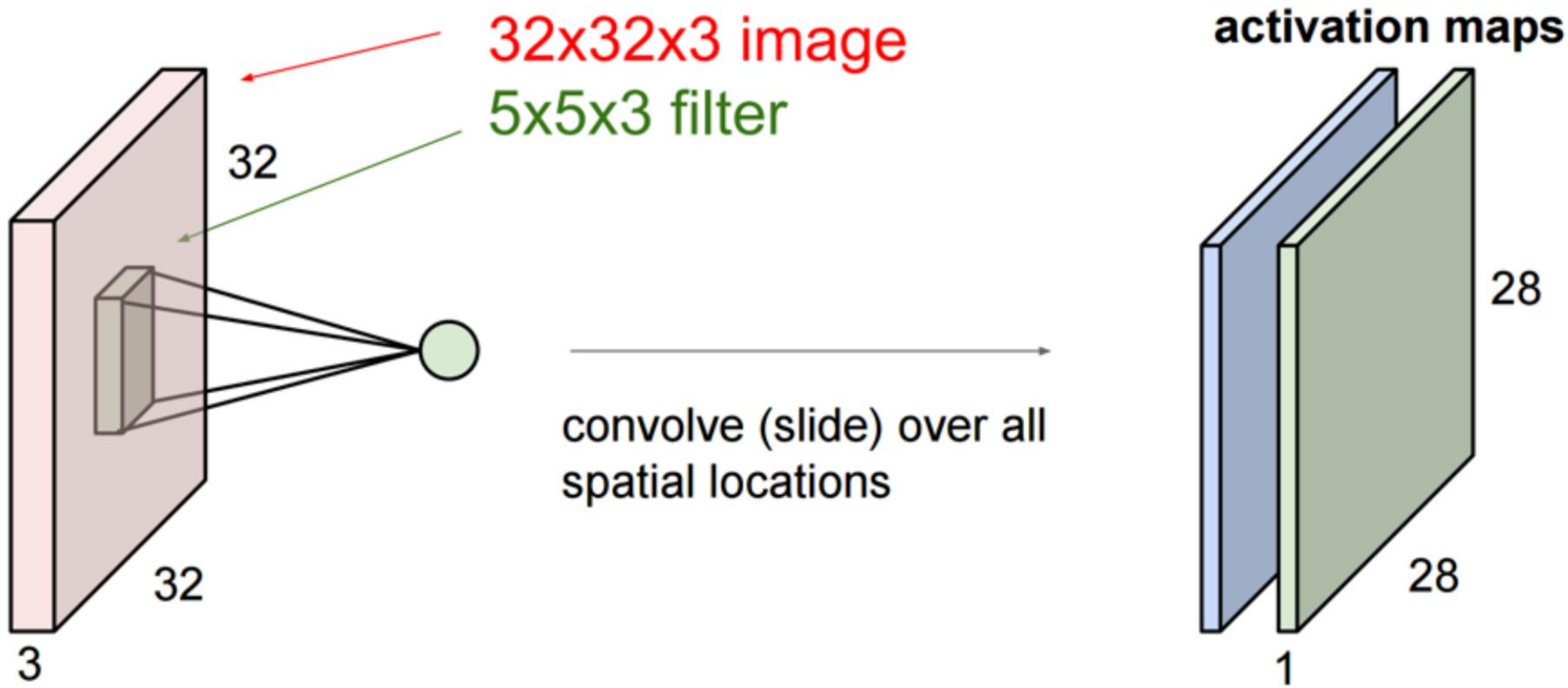
## Convolution Layer



# (Recap)

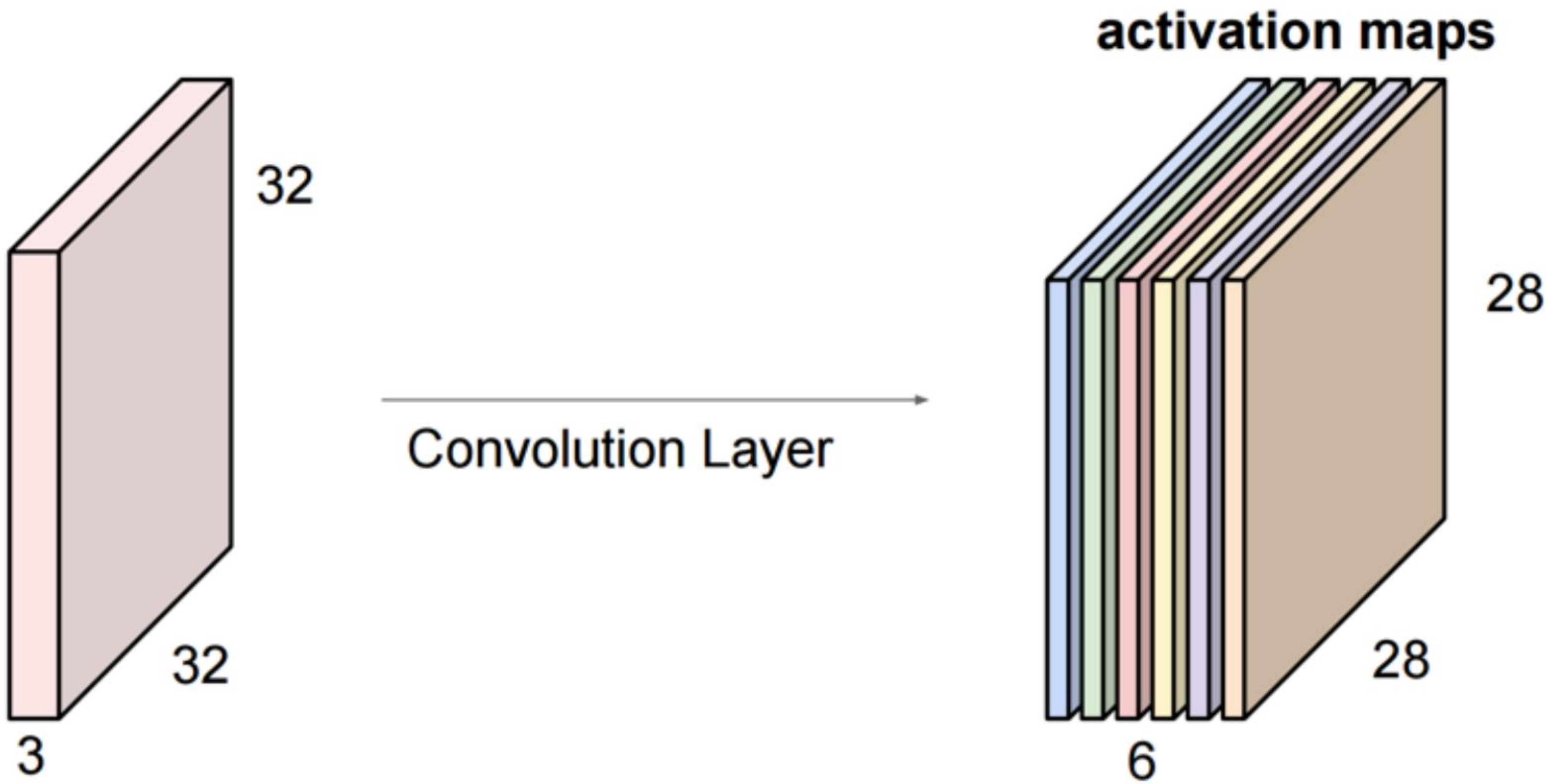
## Convolution Layer

consider a second, green filter



# (Recap)

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



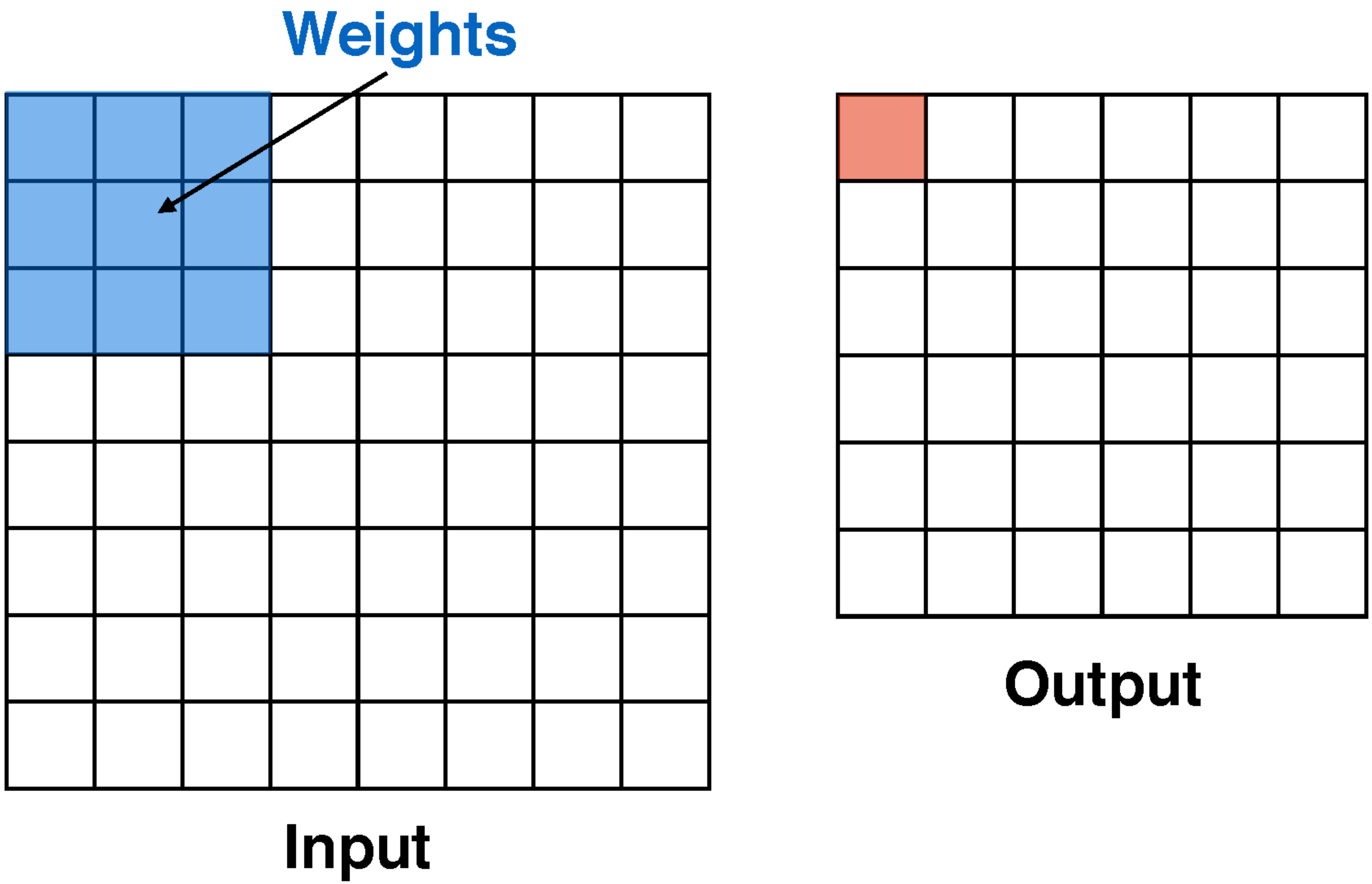

We stack these up to get a "new image" of size 28x28x6!

# Demos

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

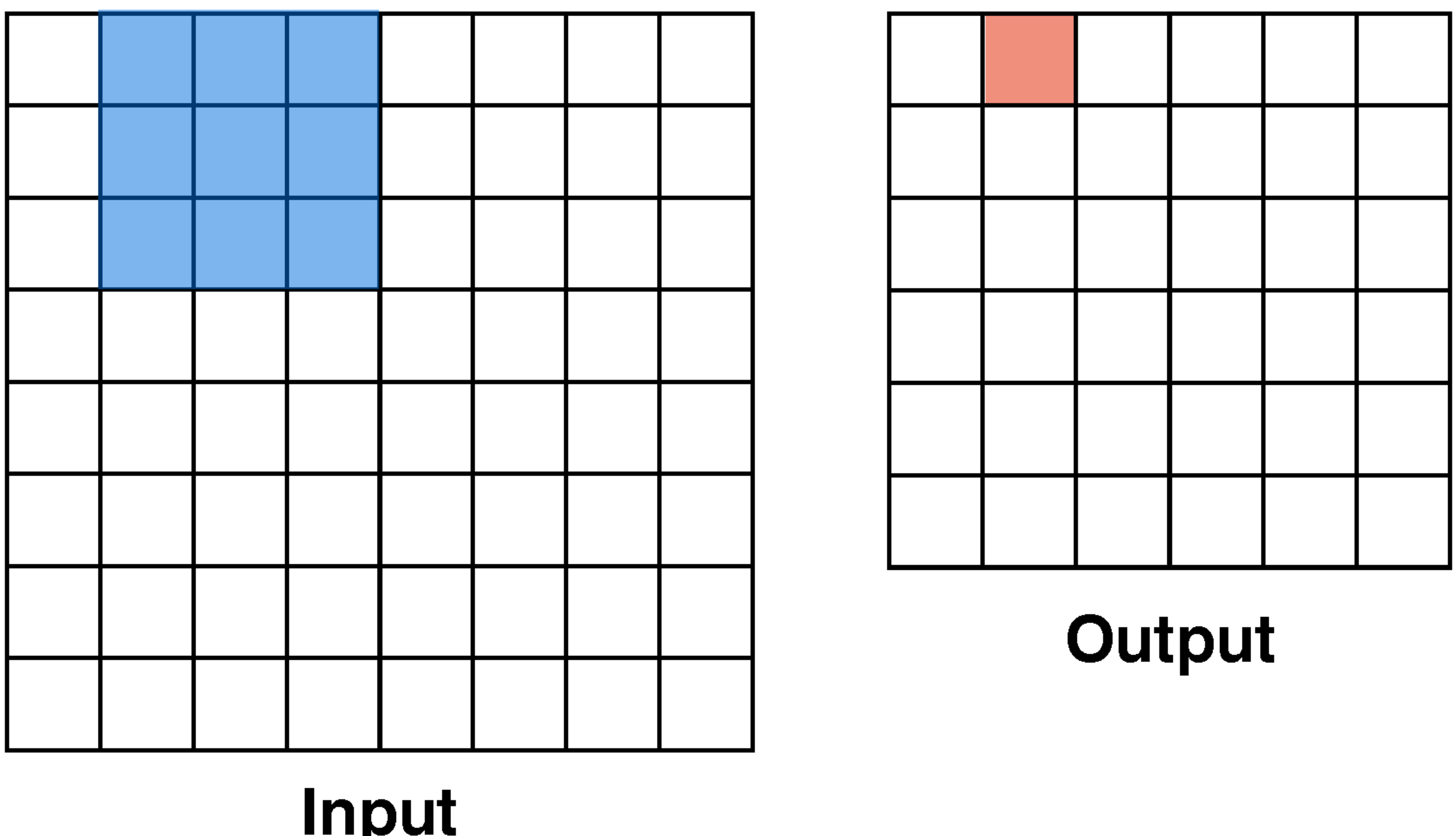
# Convolution: Stride

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



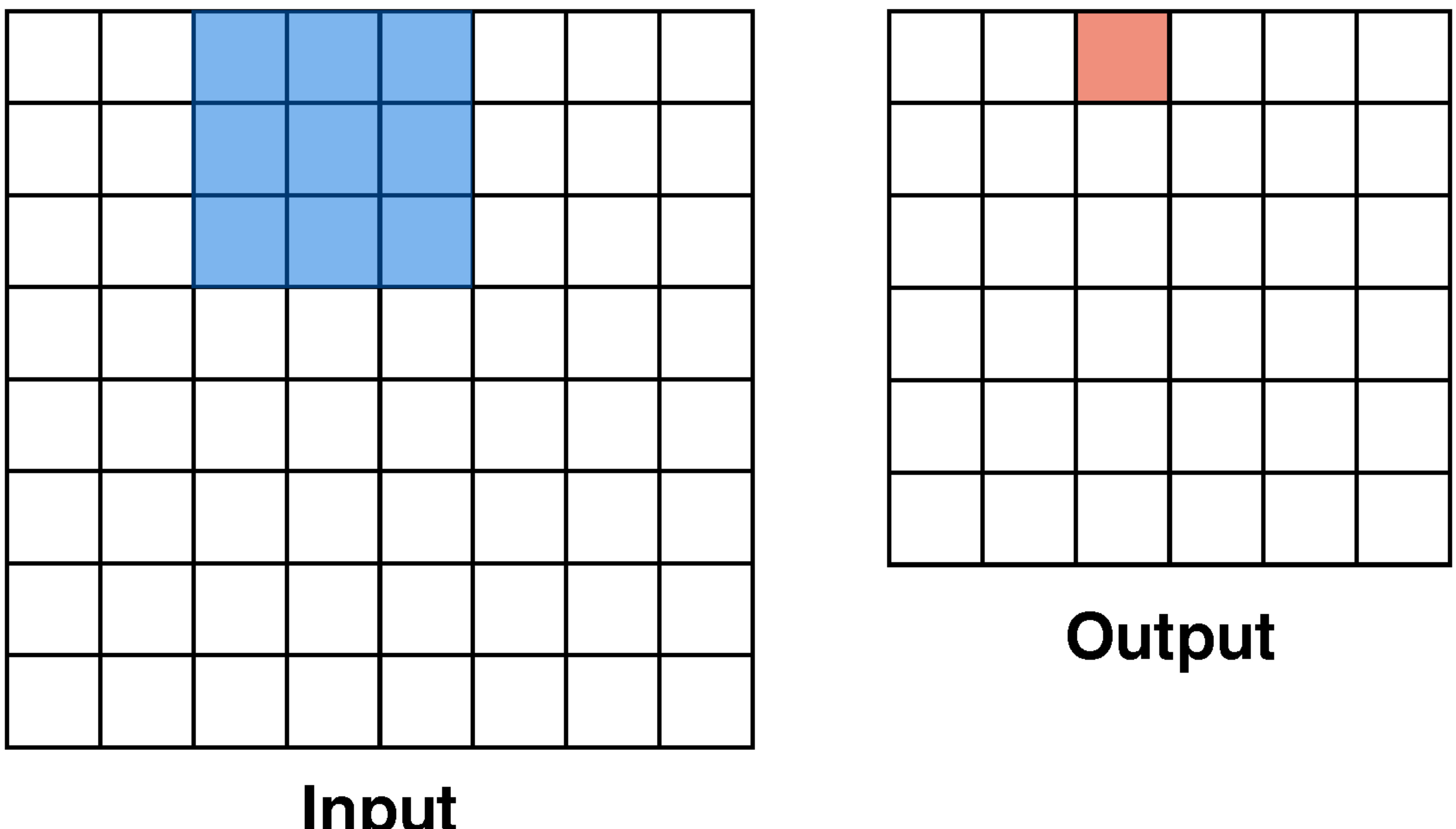
# Convolution: Stride

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



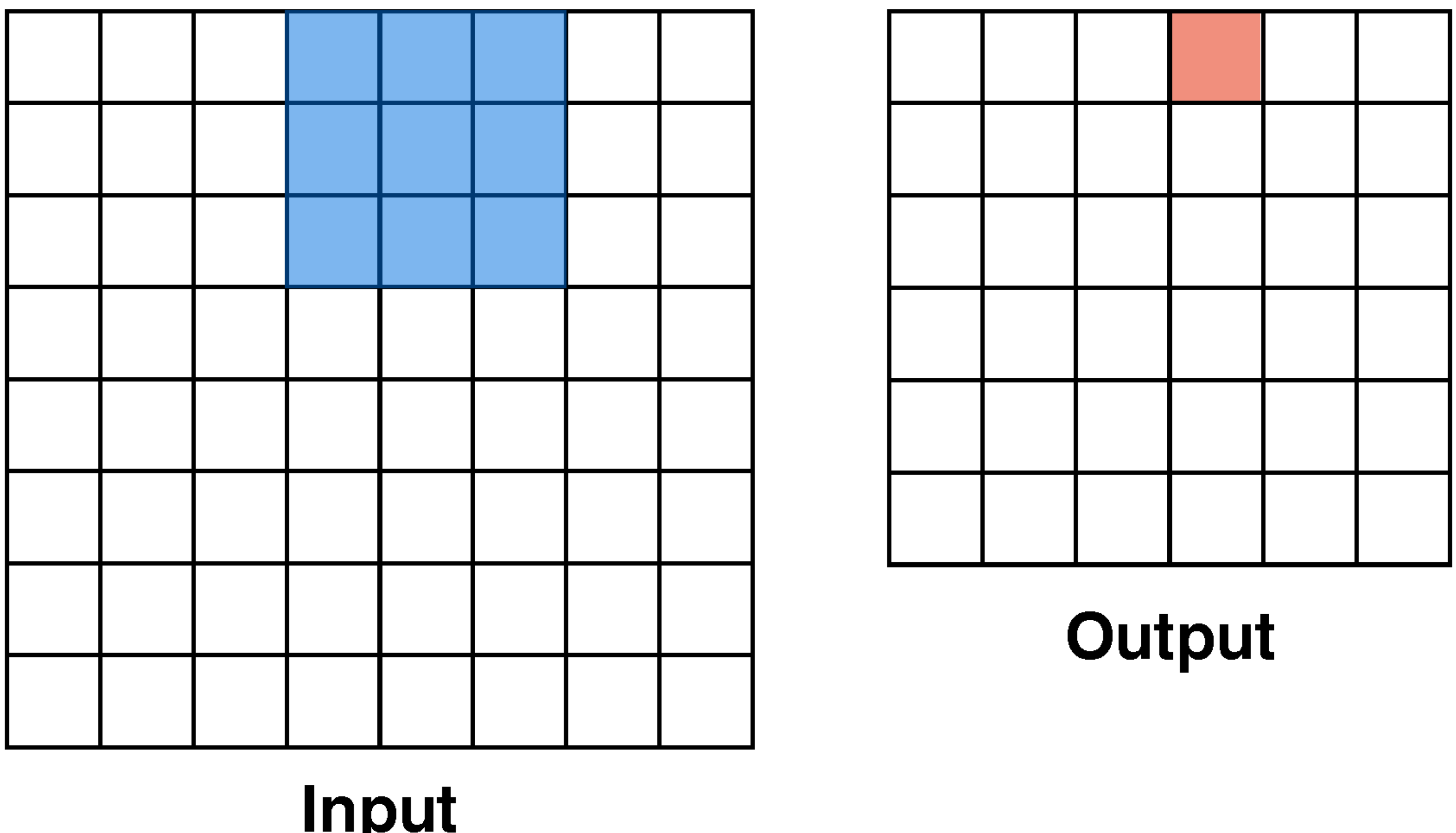
# Convolution: Stride

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



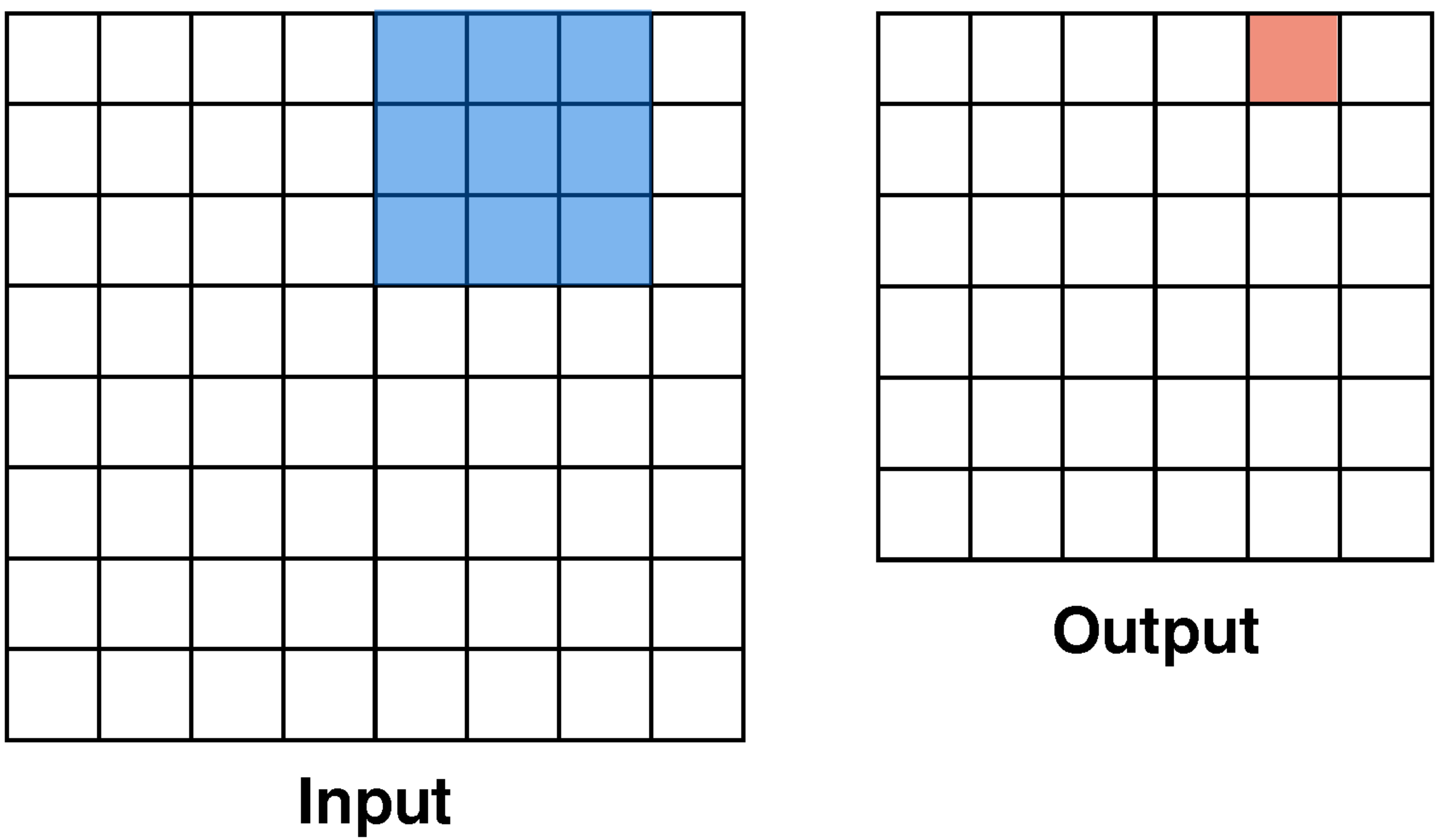
# Convolution: Stride

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



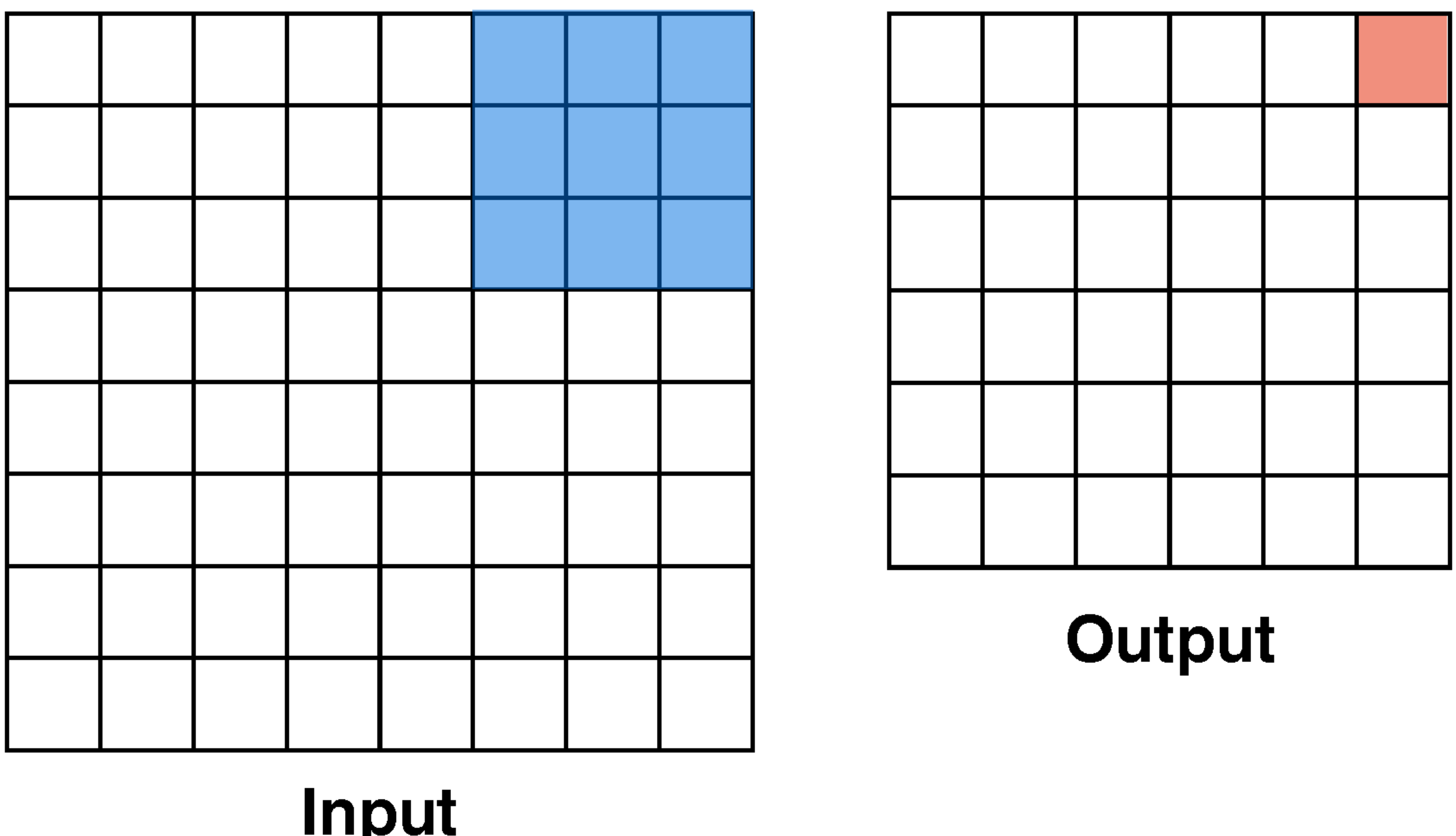
# Convolution: Stride

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



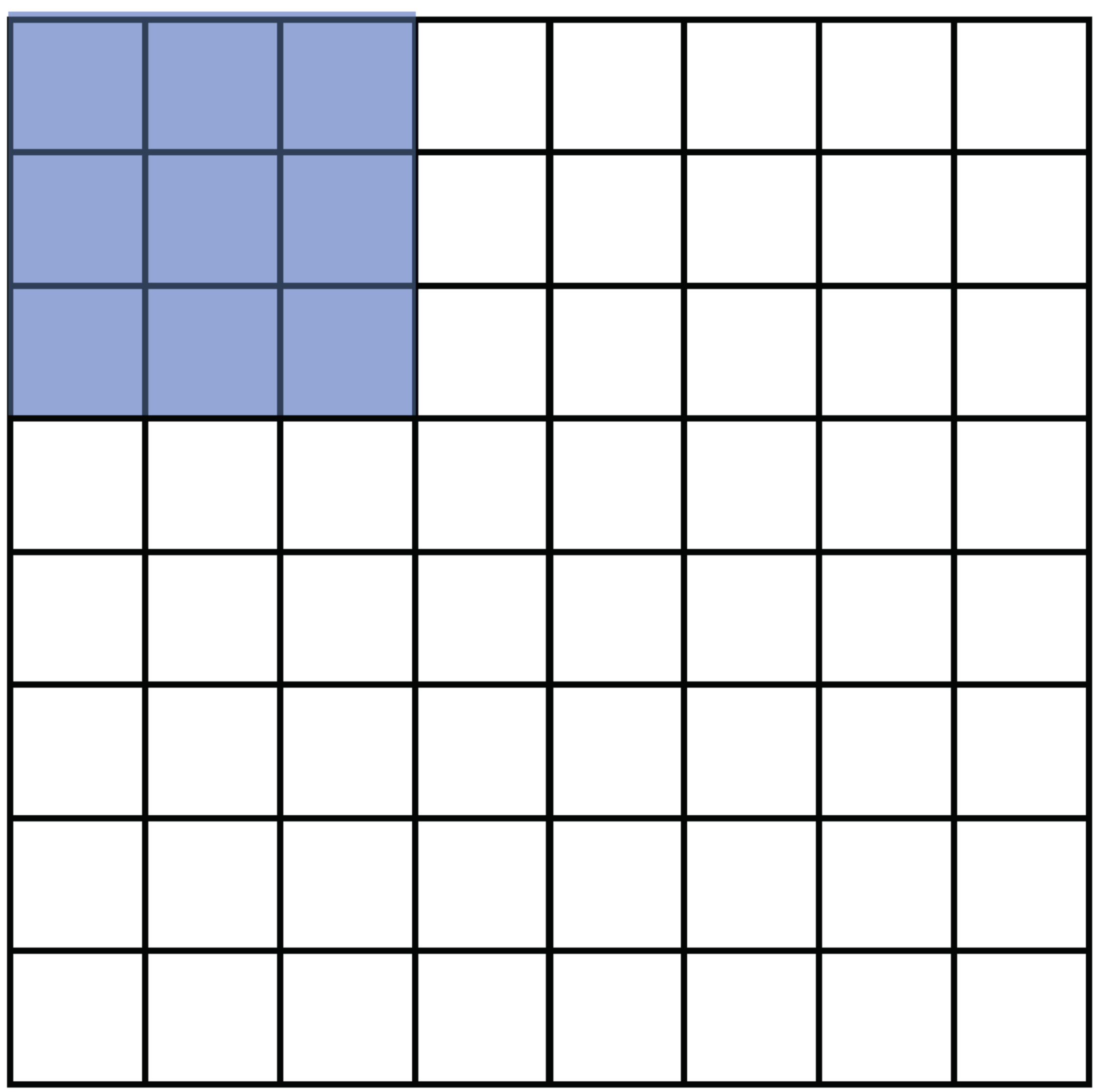
# Convolution: Stride

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



# Convolution: Stride

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



**Input**

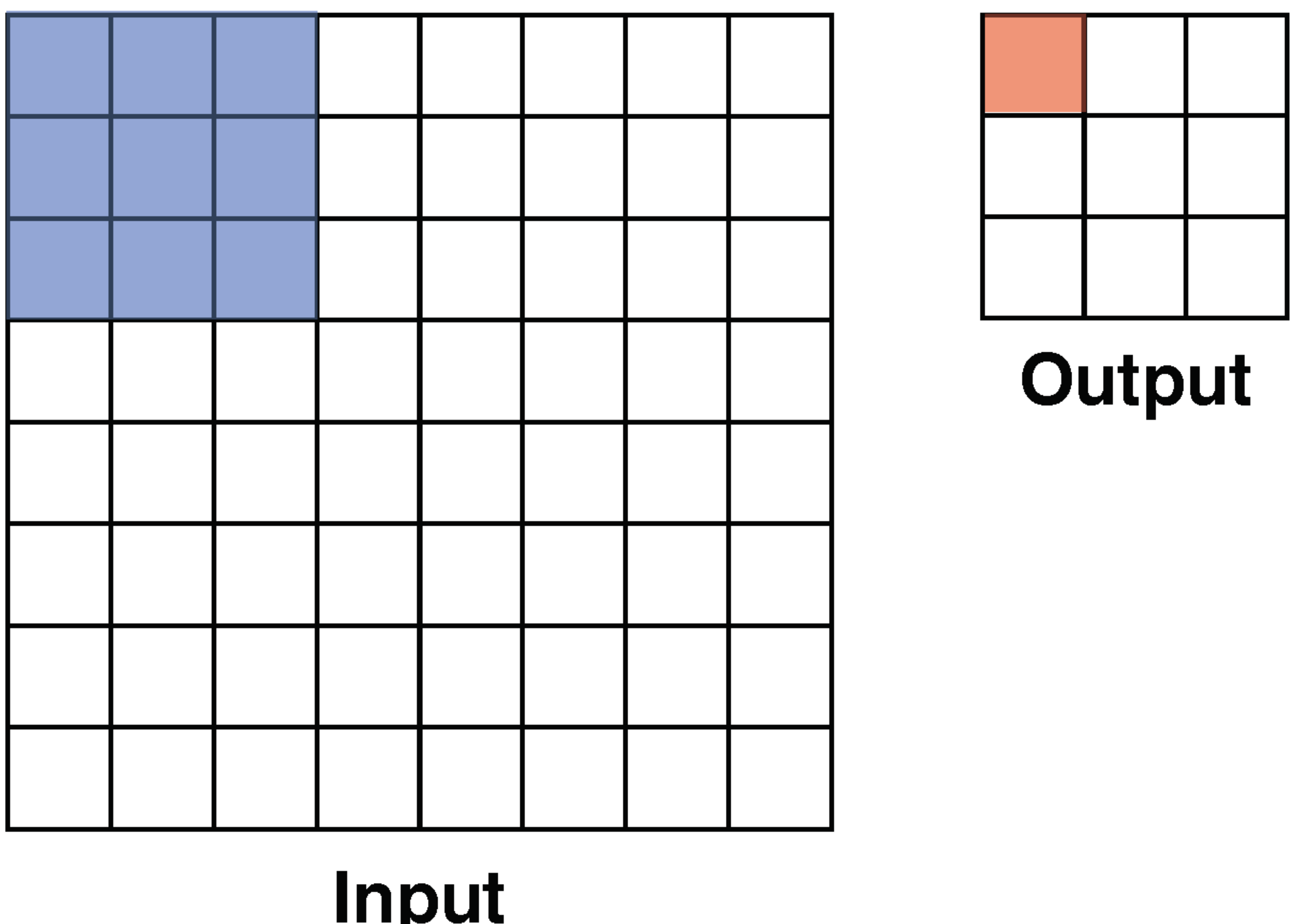
Recall that at each position, we are doing a **3D** sum:

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

*(channel, row, column)*

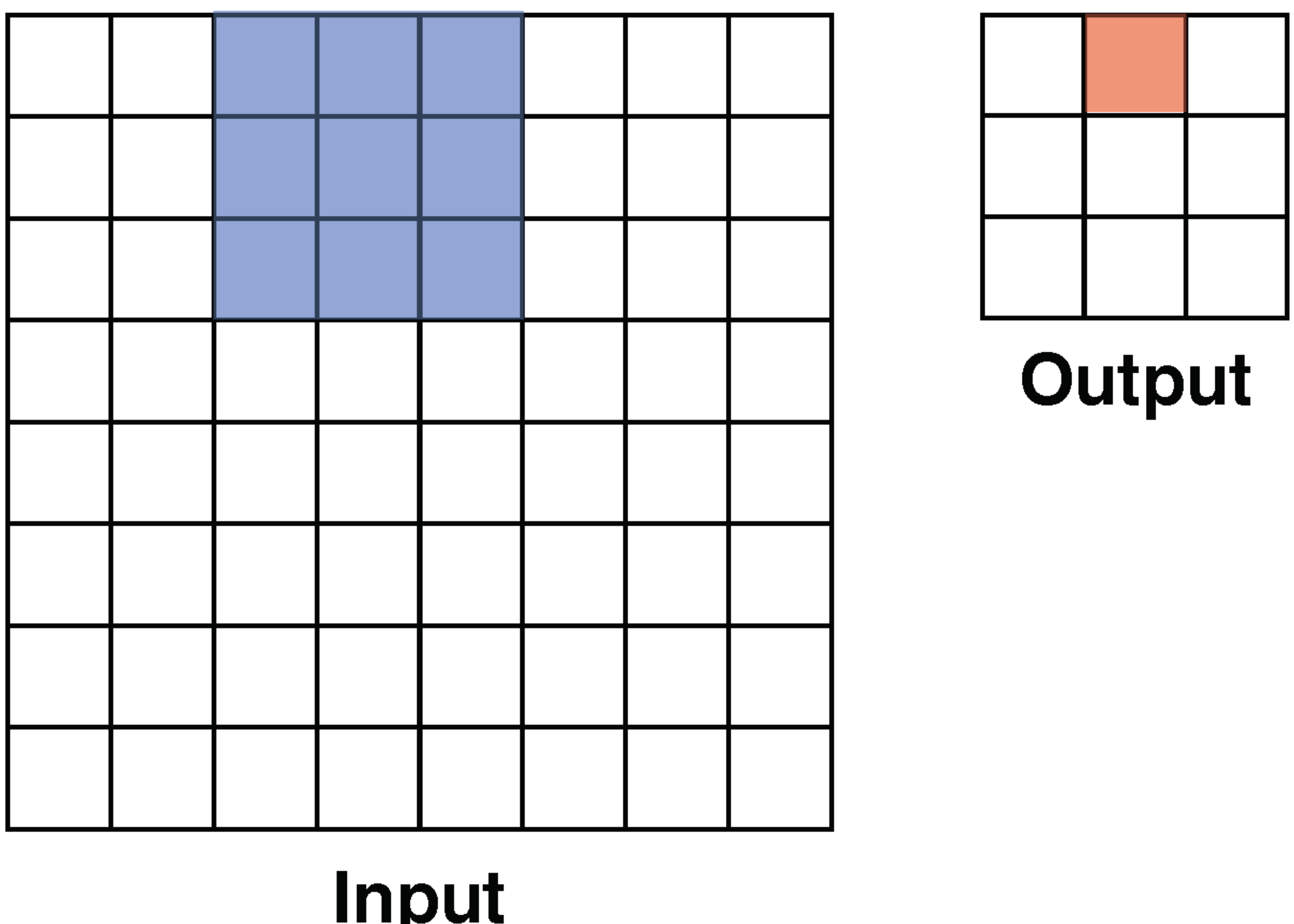
# Convolution: Stride

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



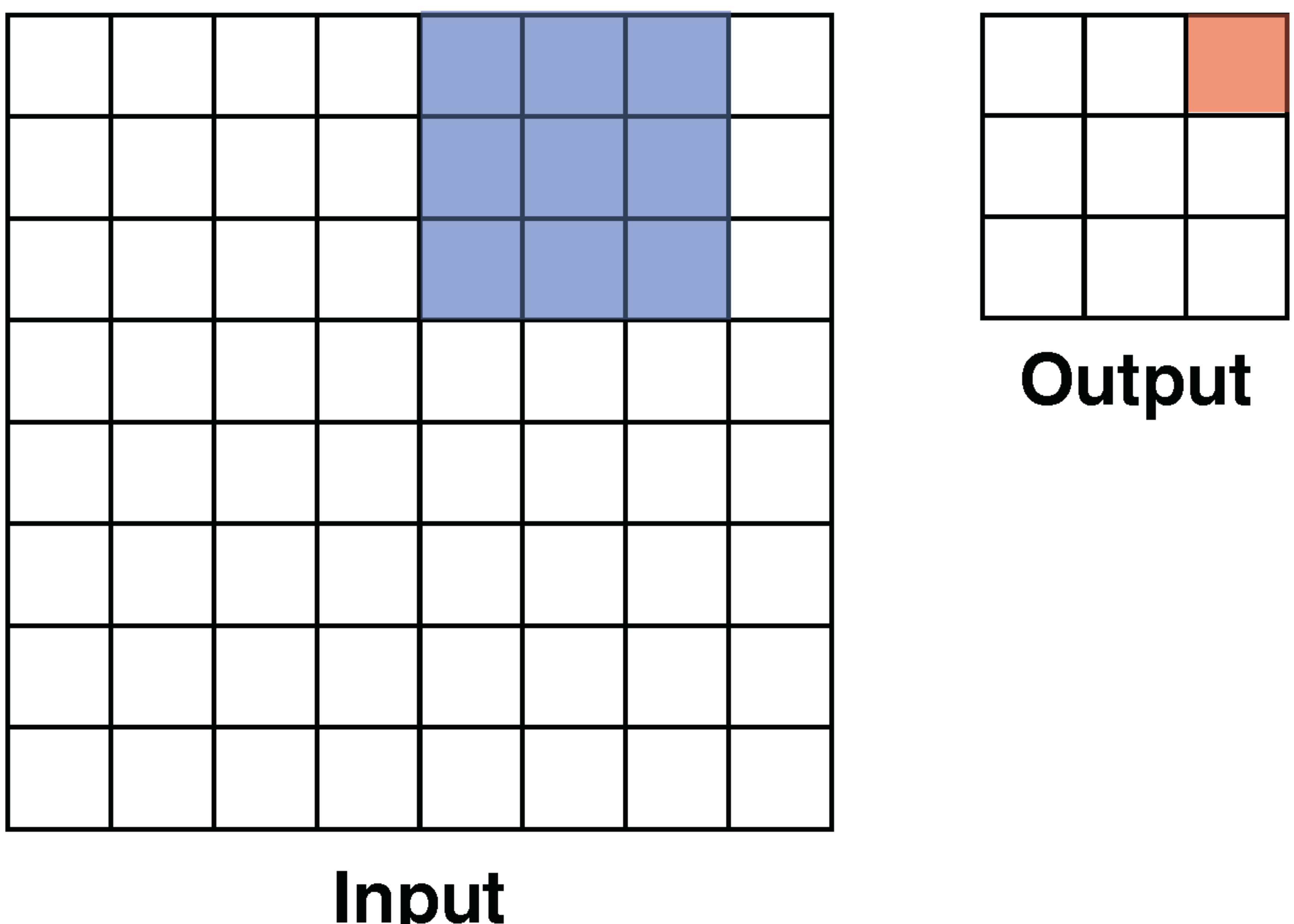
# Convolution: Stride

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



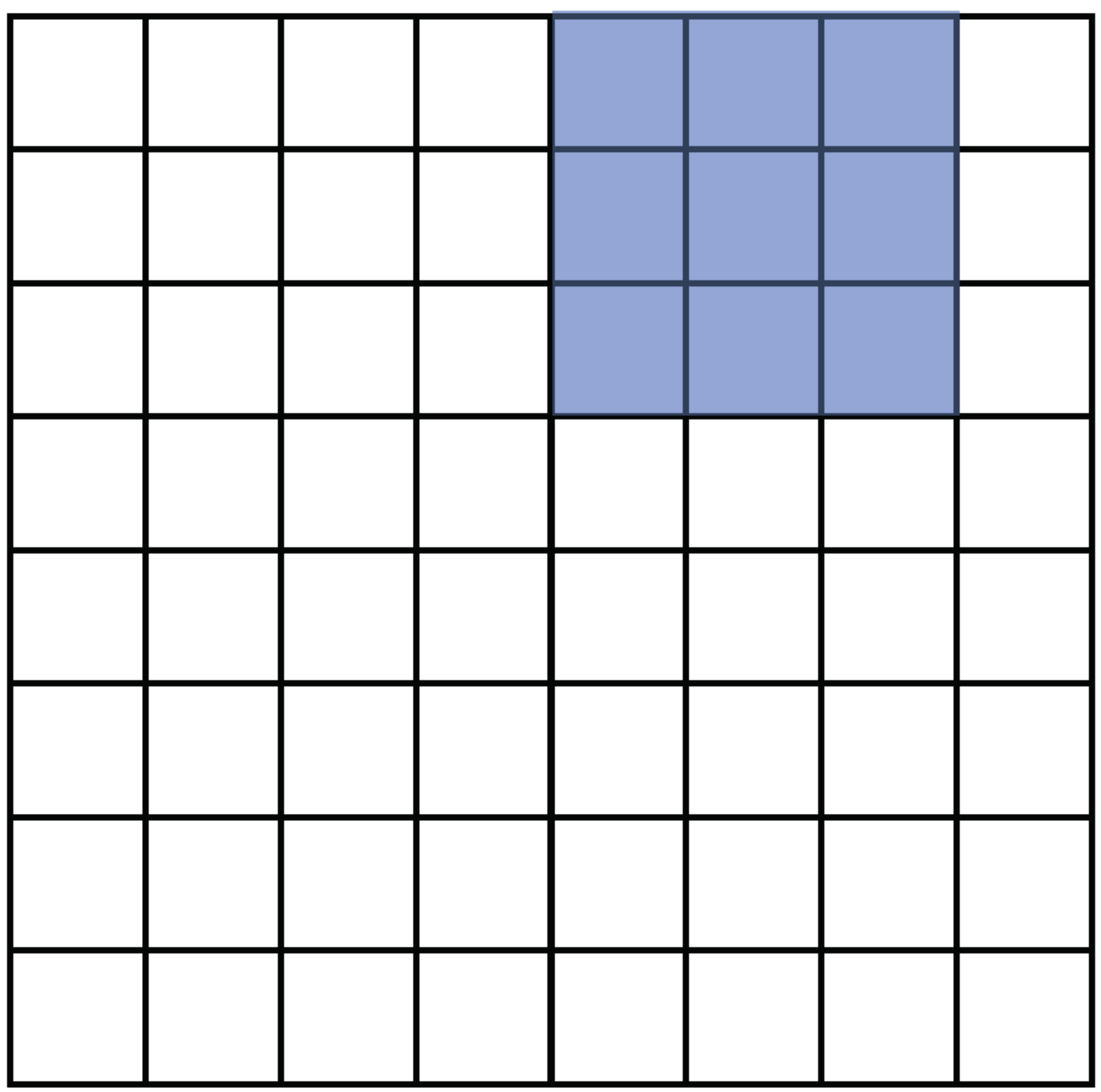
# Convolution: Stride

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

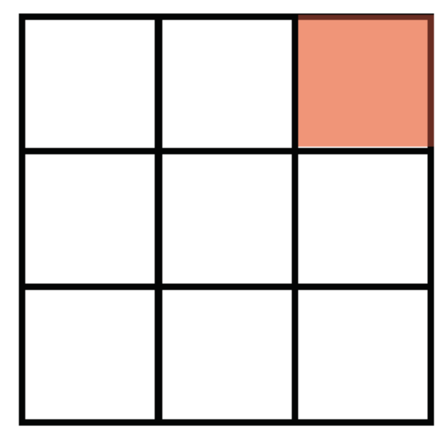


# Convolution: Stride

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



Input



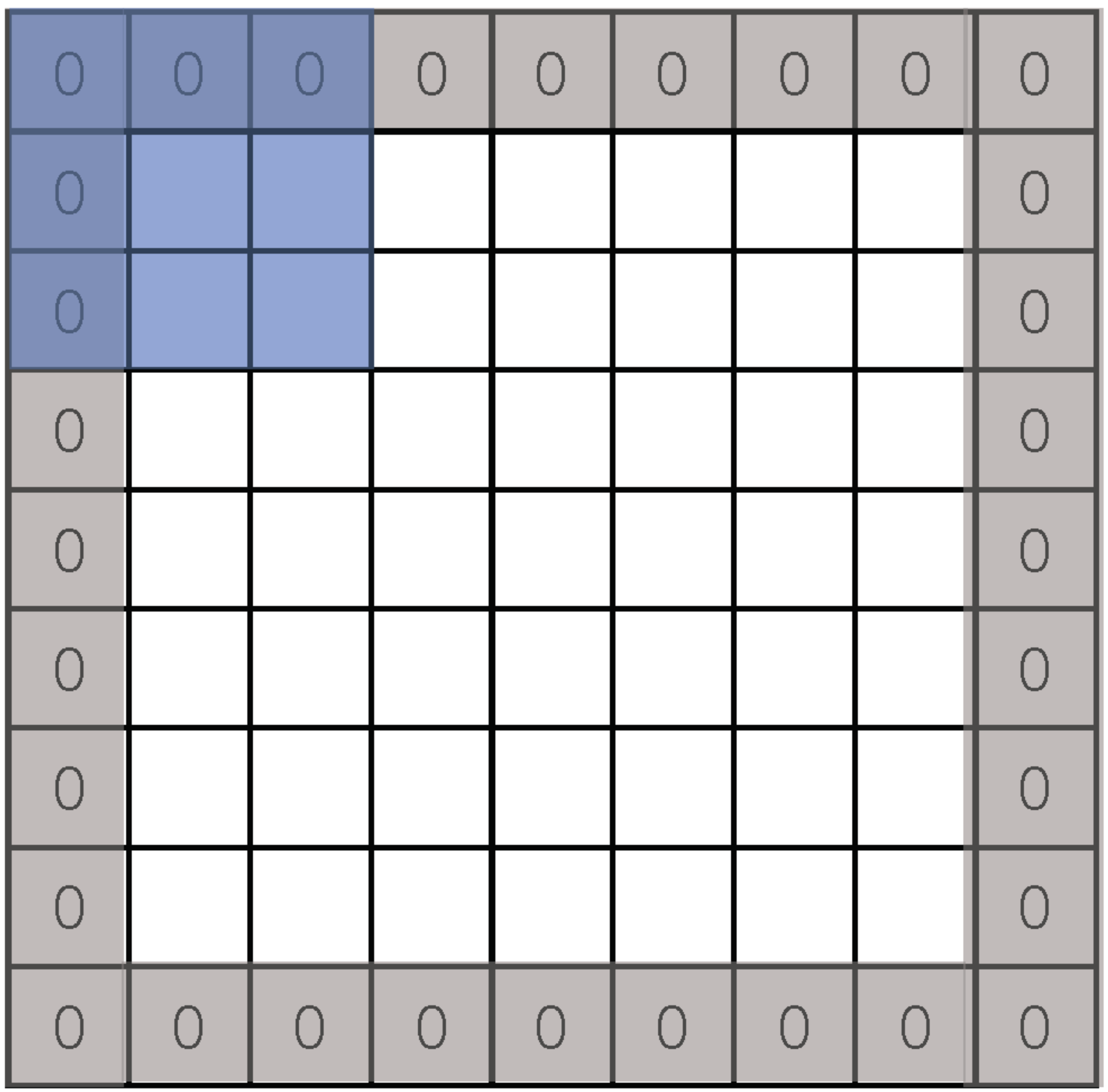
Output

*- Notice that with certain strides, we may not be able to cover all of the input*

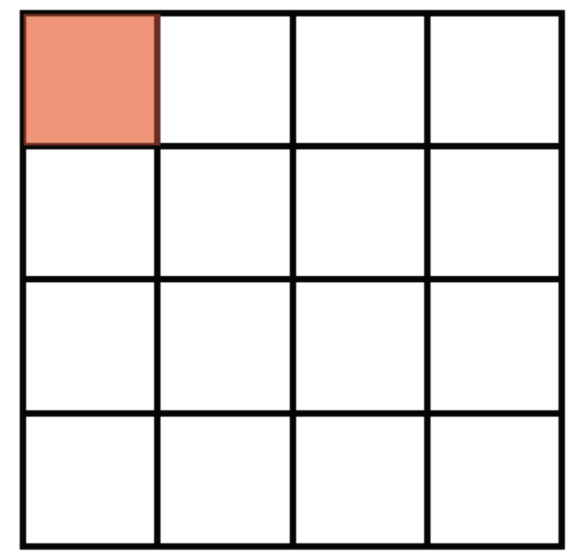
*- The output is also half the size of the input*

# Convolution: Padding

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



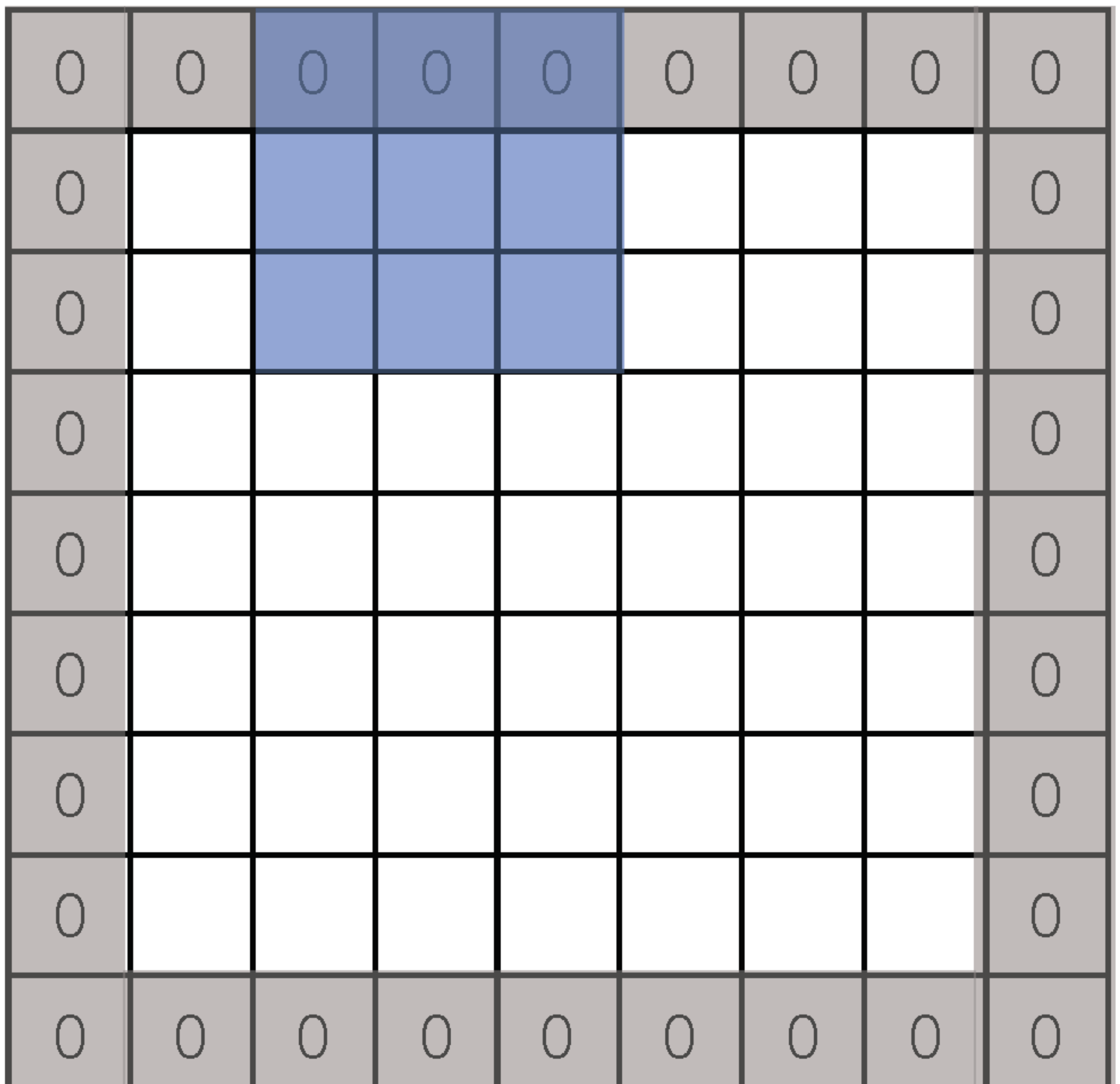

**Input**



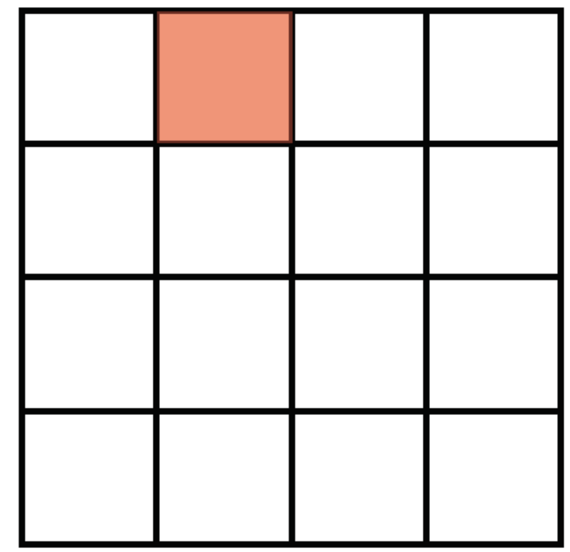
**Output**

# Convolution: Padding

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



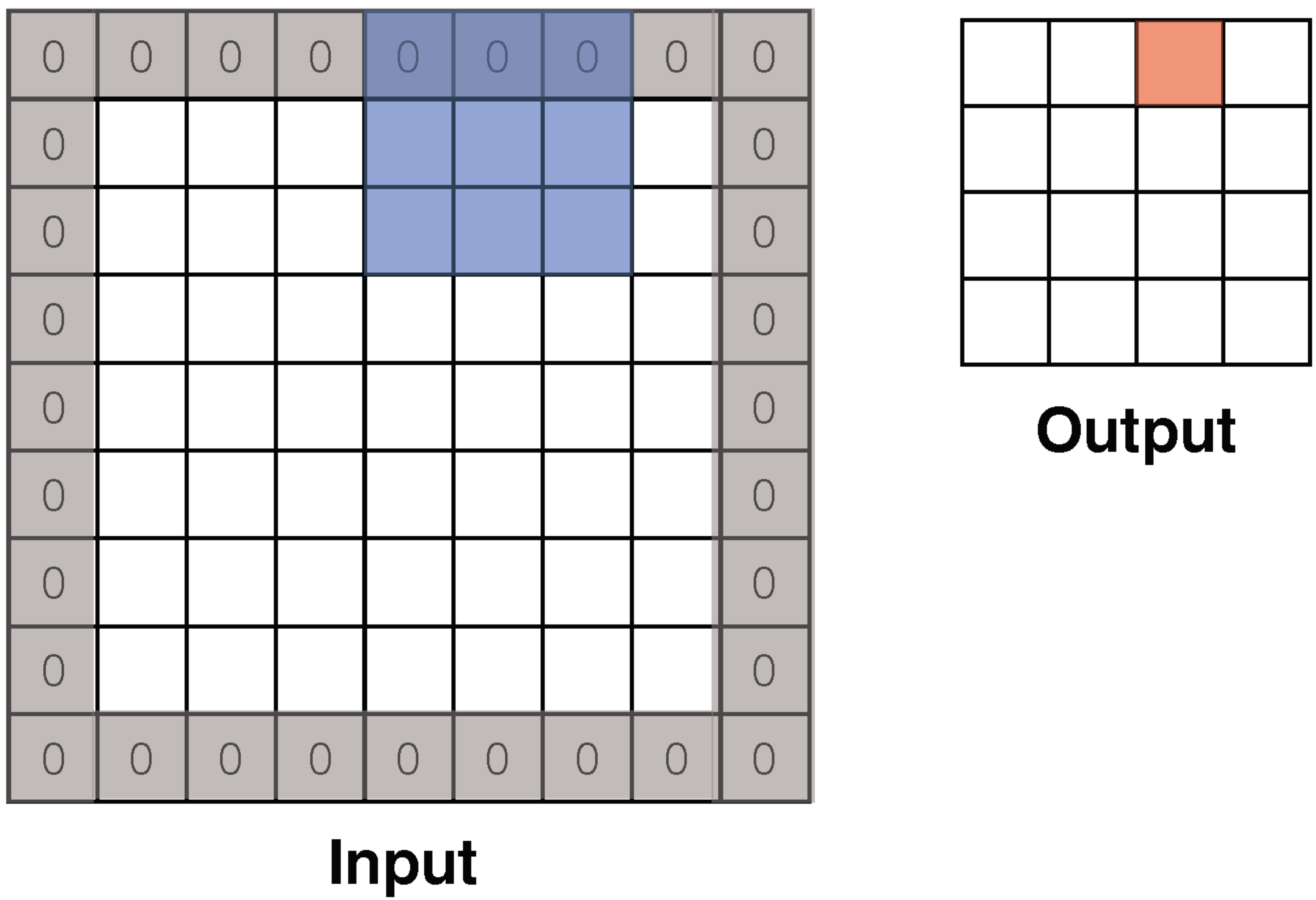

**Input**



**Output**

# Convolution: Padding

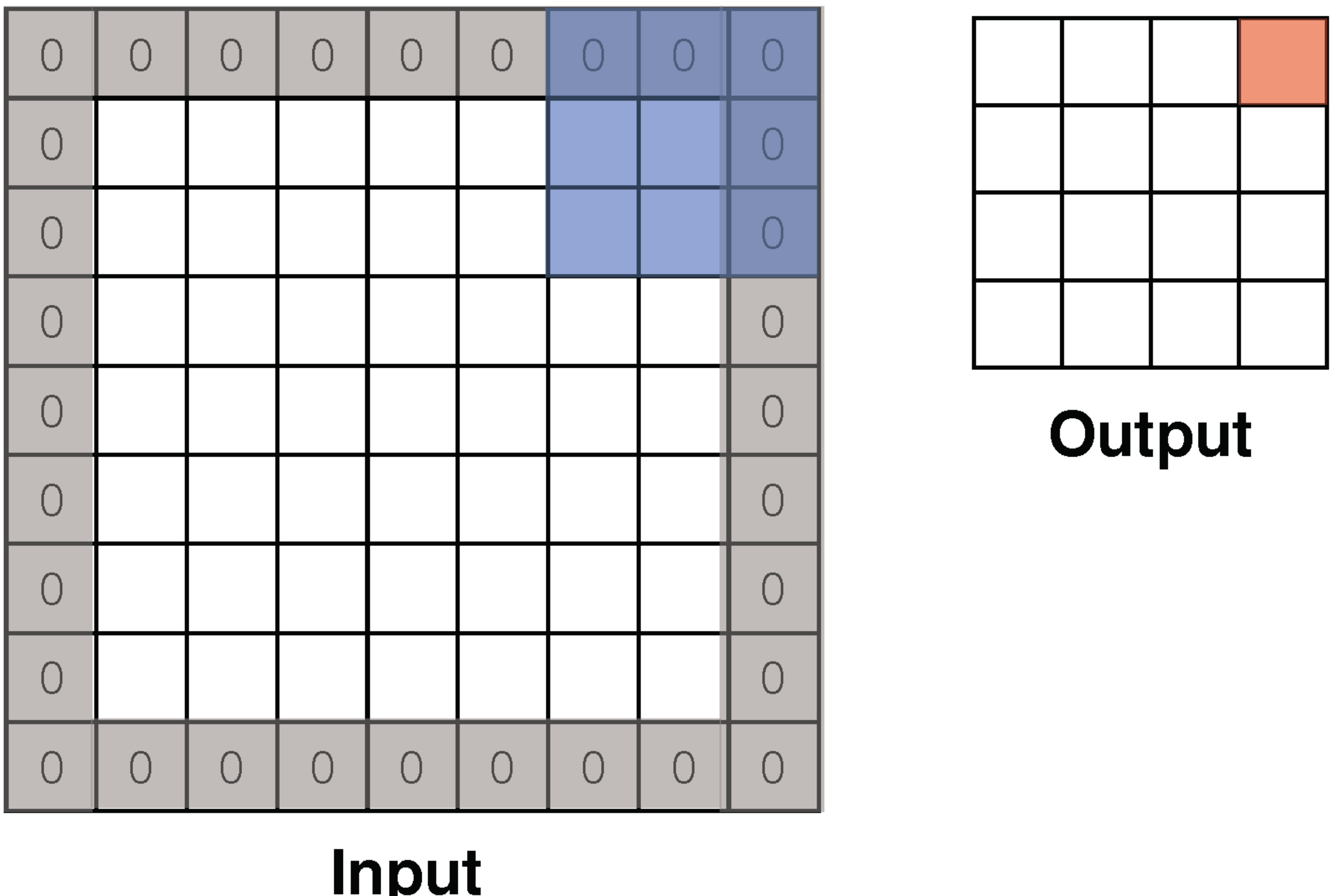
We can also pad the input with zeros.
Here, **pad = 1, stride = 2**



# Convolution: Padding

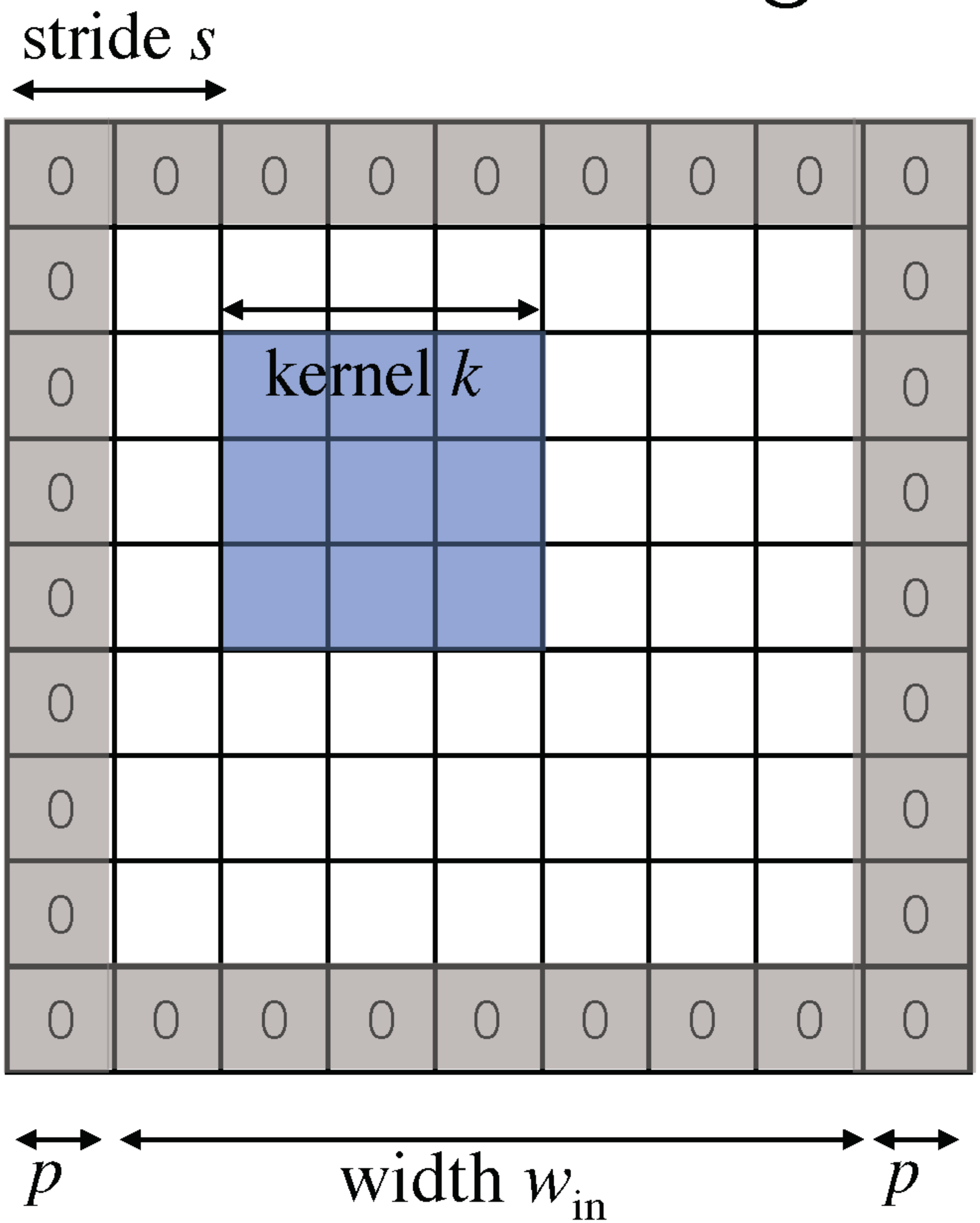
We can also pad the input with zeros.

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



# Convolution:

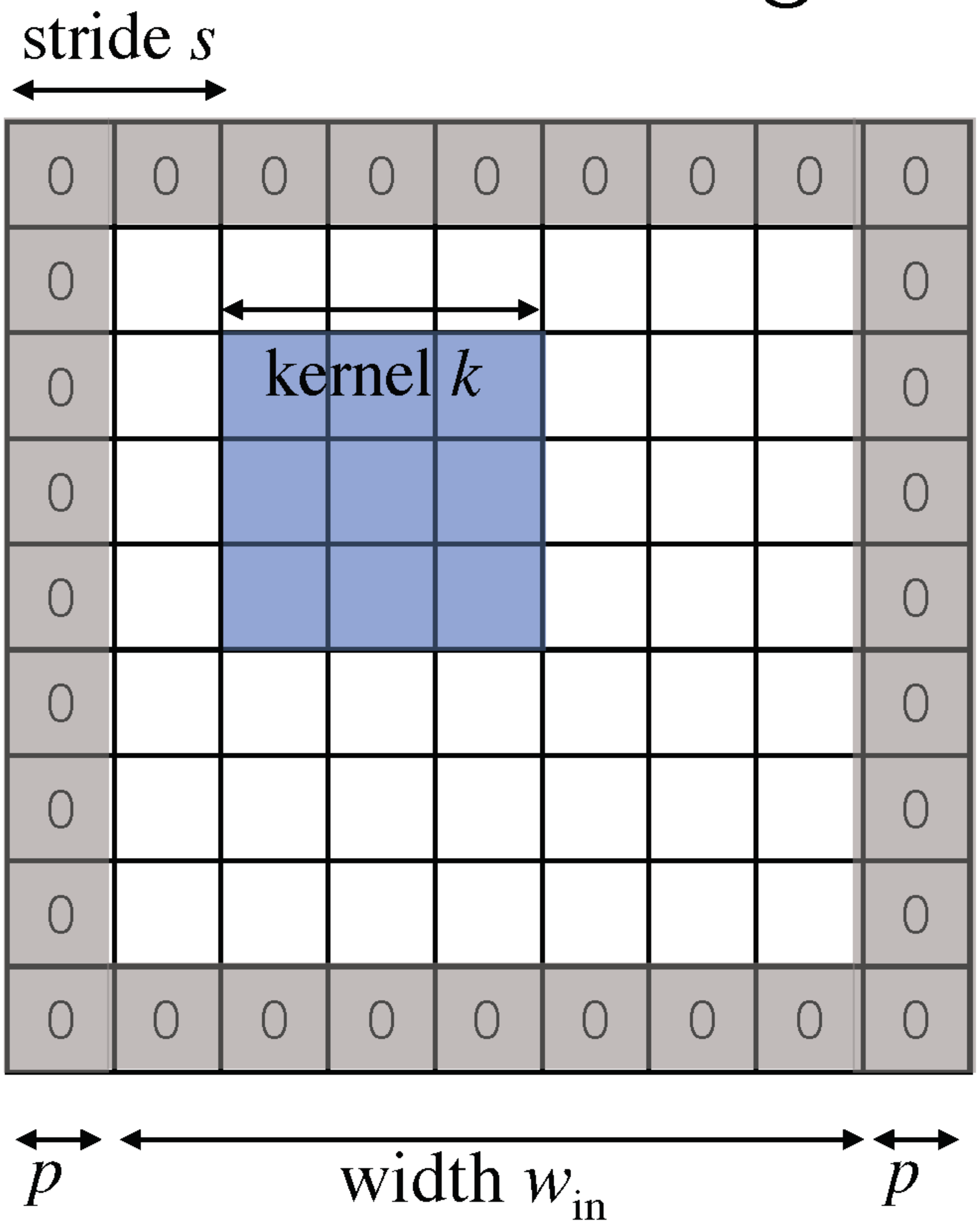
## How big is the output?




In general, the output has size:

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

# Convolution:
## How big is the output?




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

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

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

$$= w_{\text{in}}$$

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?

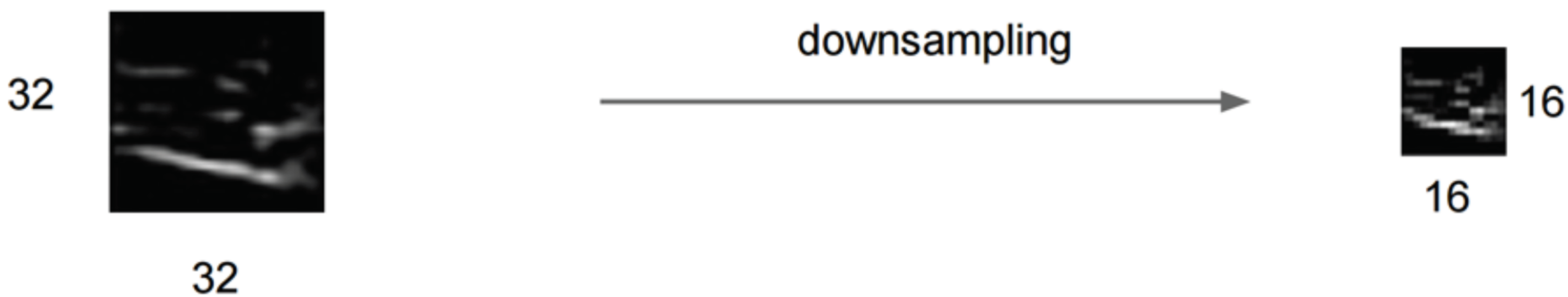
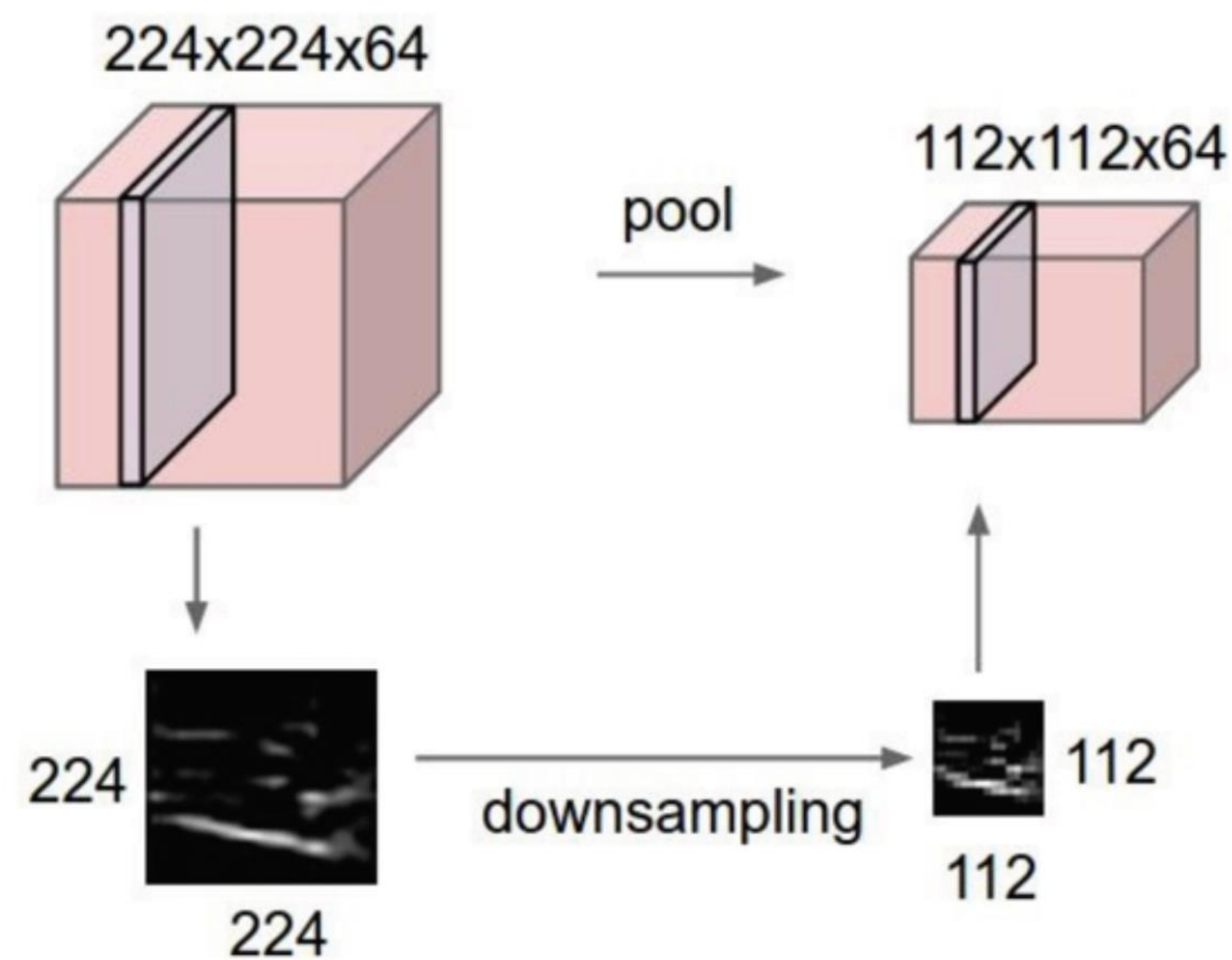


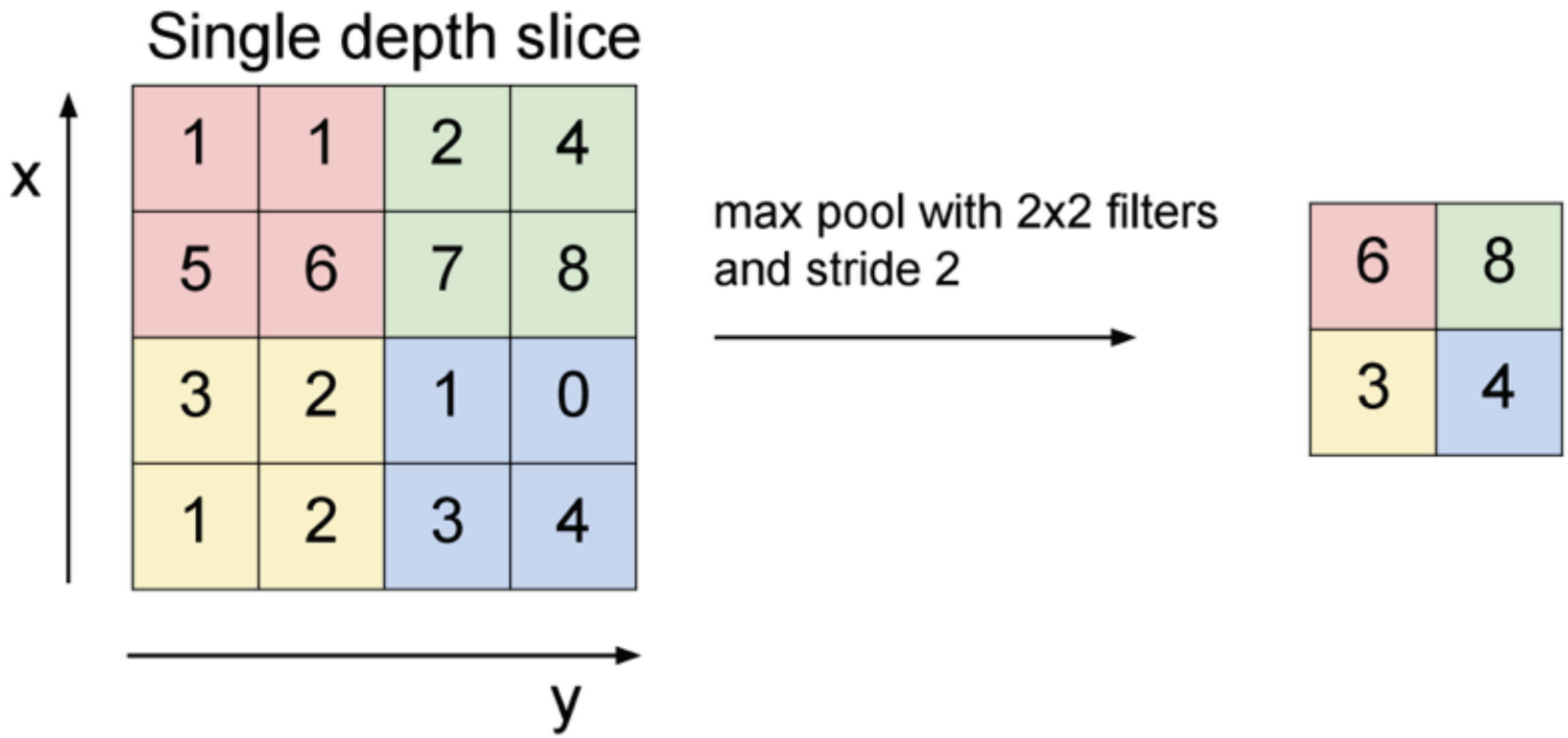

*Figure: Andrej Karpathy*

# Pooling

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



# Max Pooling




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


*Figure: Andrej Karpathy*

# Example ConvNet

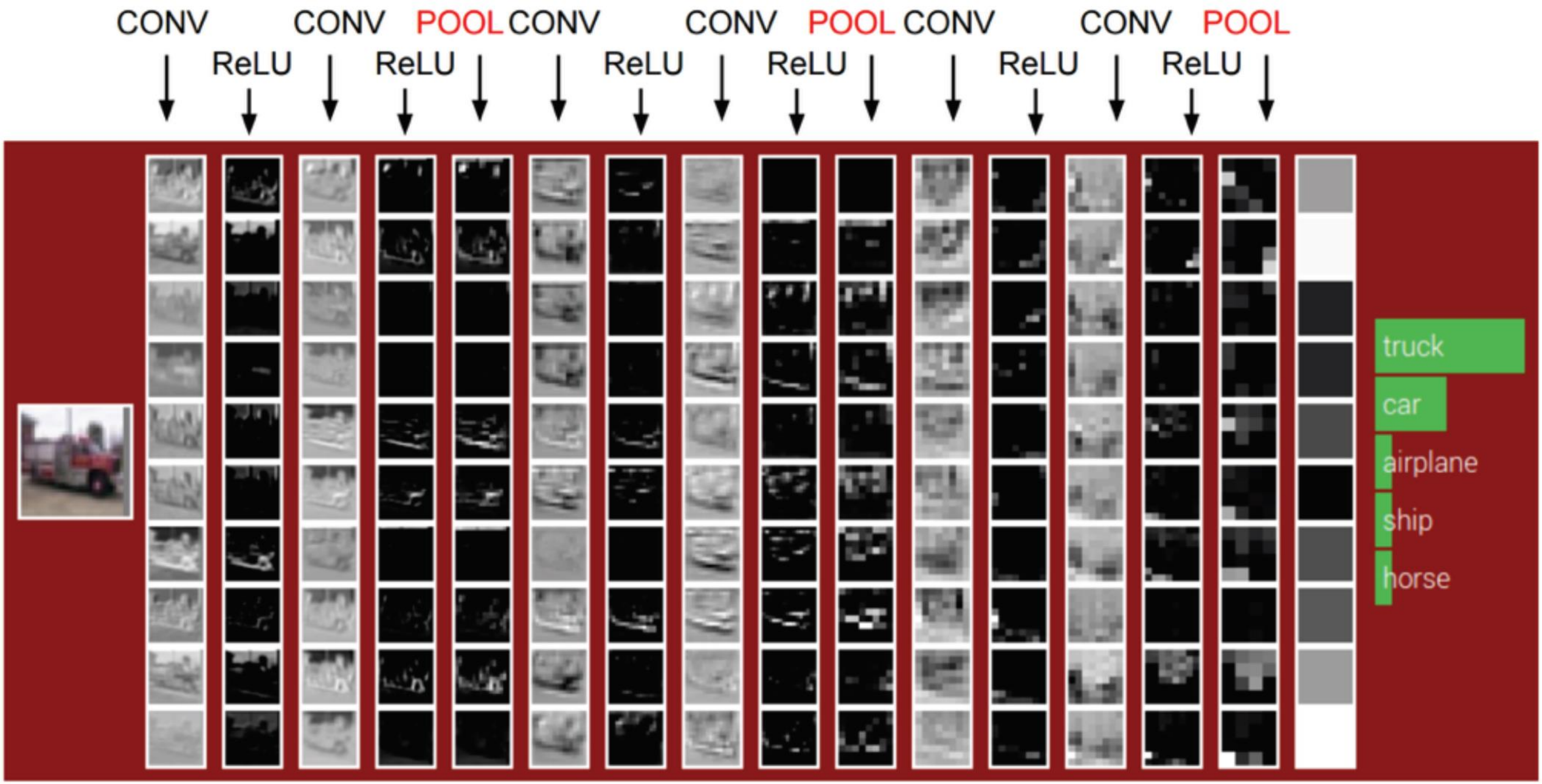



*Figure: Andrej Karpathy*

# Example ConvNet

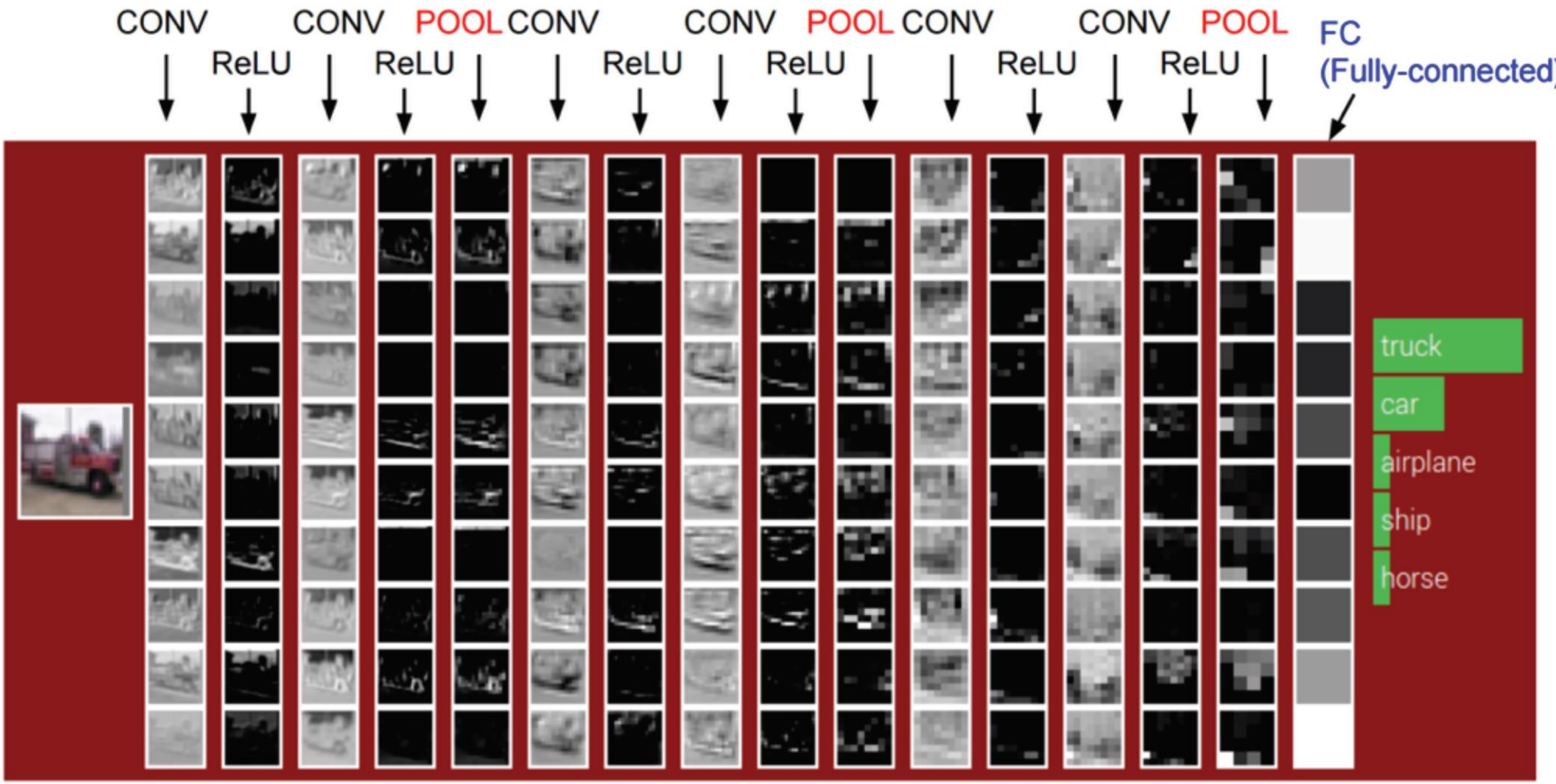
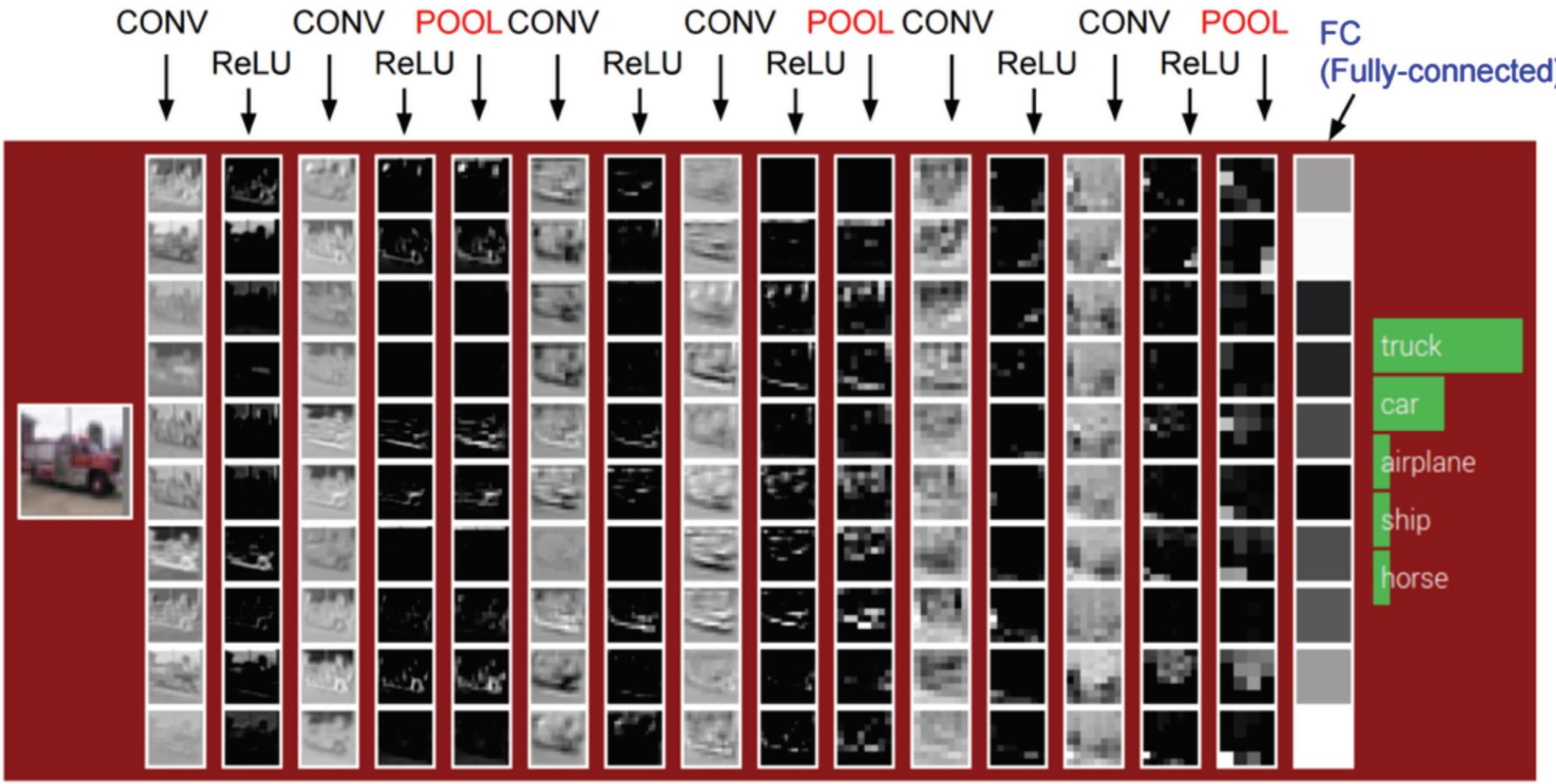



*Figure: Andrej Karpathy*

# Example ConvNet

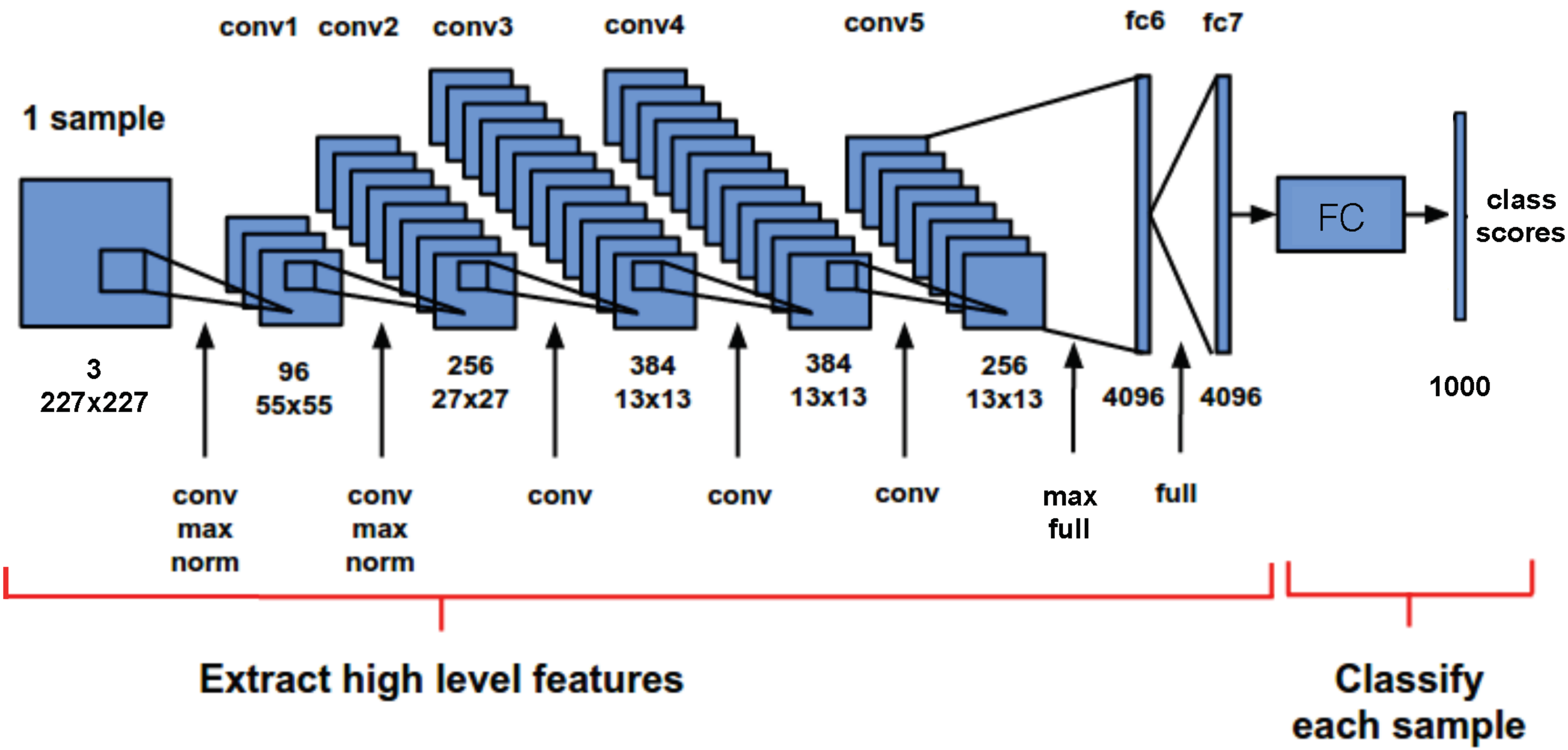



10x3x3 conv filters, stride 1, pad 1

2x2 pool filters, stride 2

*Figure: Andrej Karpathy*

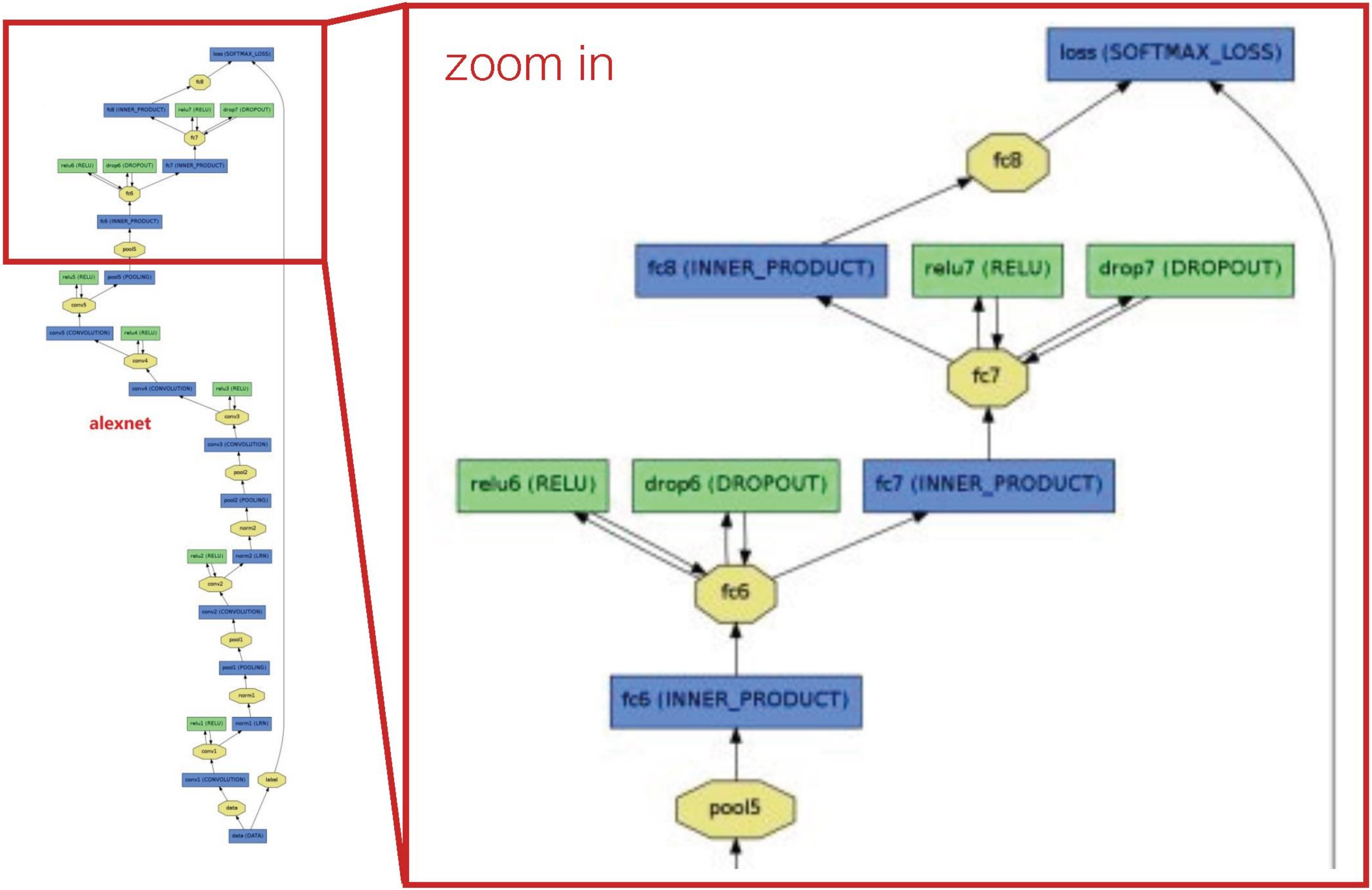
# Example: AlexNet [Krizhevsky 2012]




"max": max pooling
"norm": local response normalization
"full": fully connected

Figure: [Karnowski 2015] *(with corrections)*

# Example: AlexNet [Krizhevsky 2012]

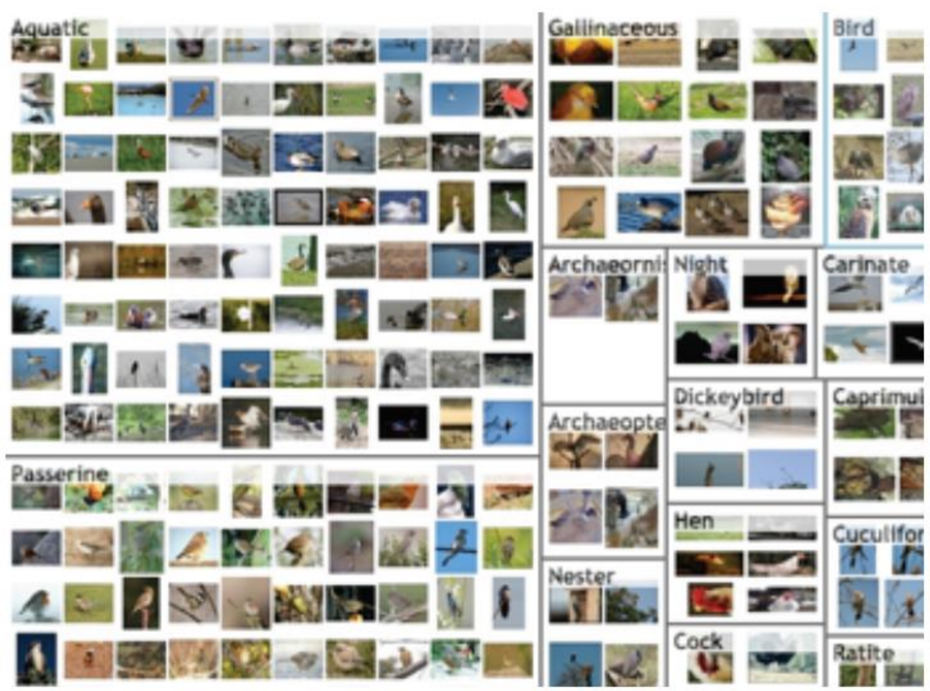


Questions?

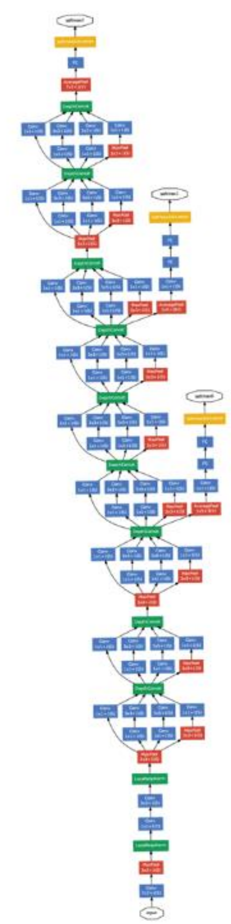
# How do you actually train these things?

**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:**

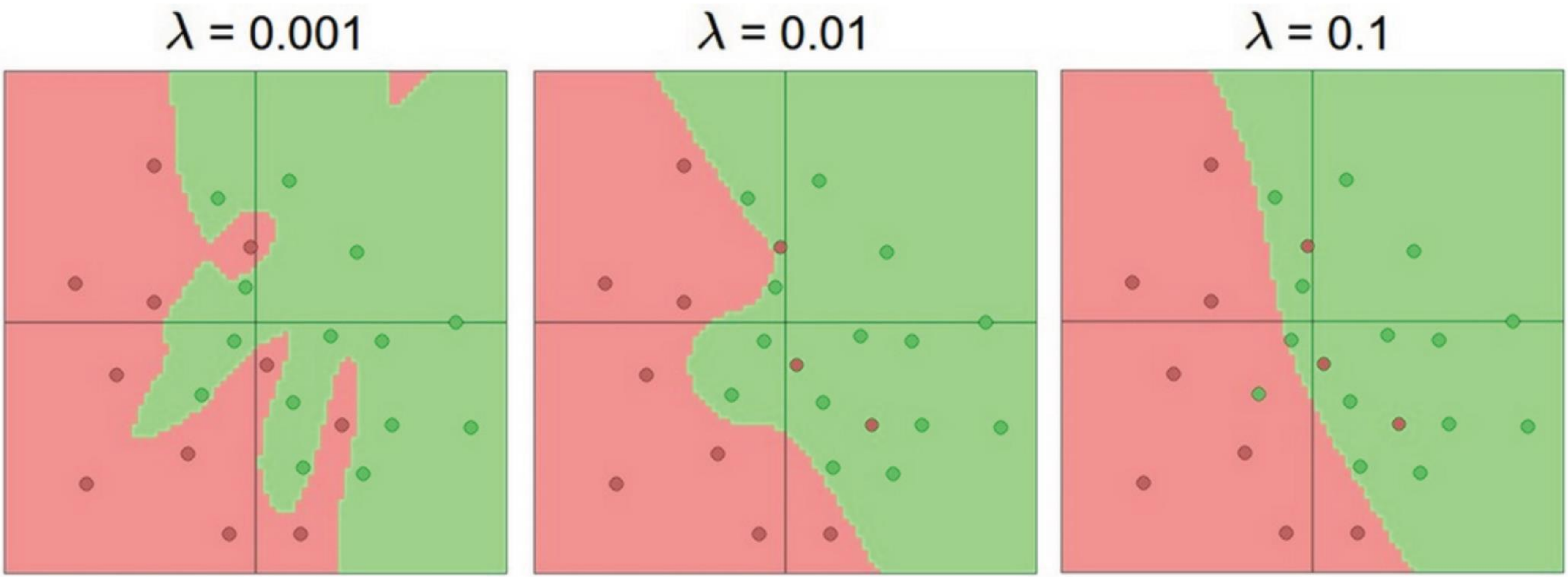
1. Sample a batch of training data (~100 images)
2. Forwards pass: compute loss (avg. over batch)
3. Backwards pass: compute gradient
4. Update all parameters

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

# Regularization

**Regularization reduces overfitting:**

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



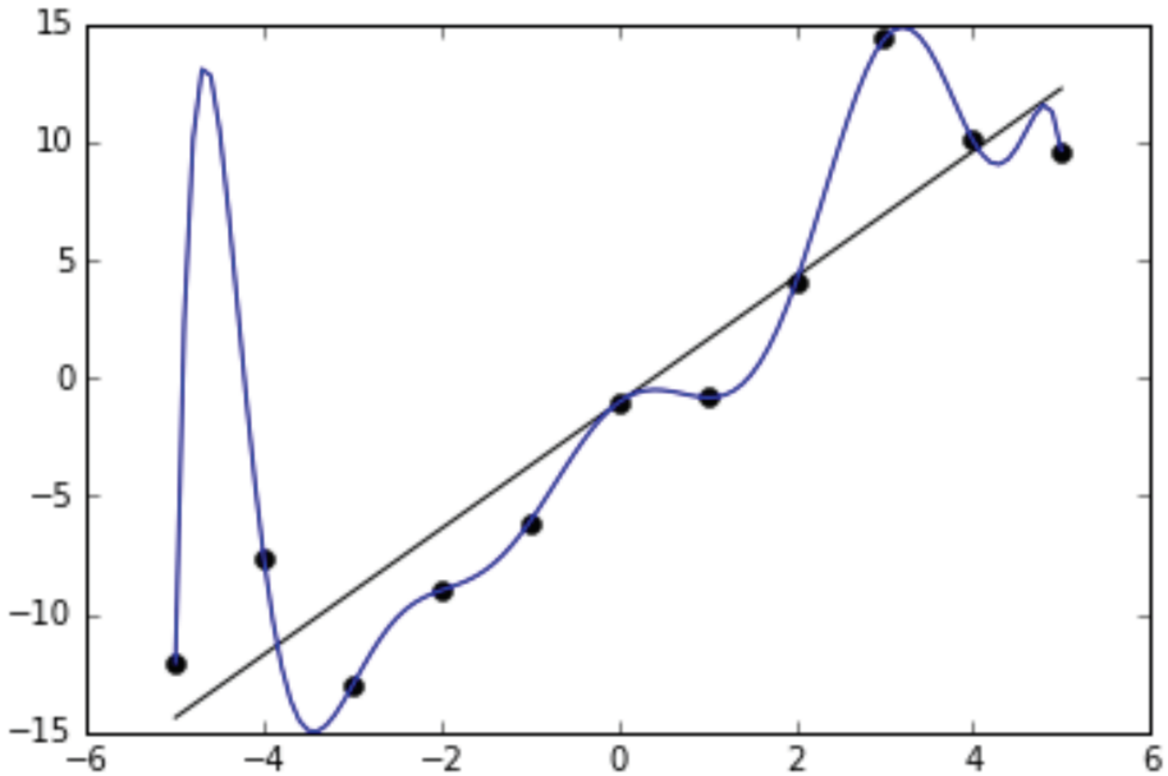

[Andrej Karpathy http://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html]

# Overfitting

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

**Underfitting:** insufficiently modeling the relationship in the training set

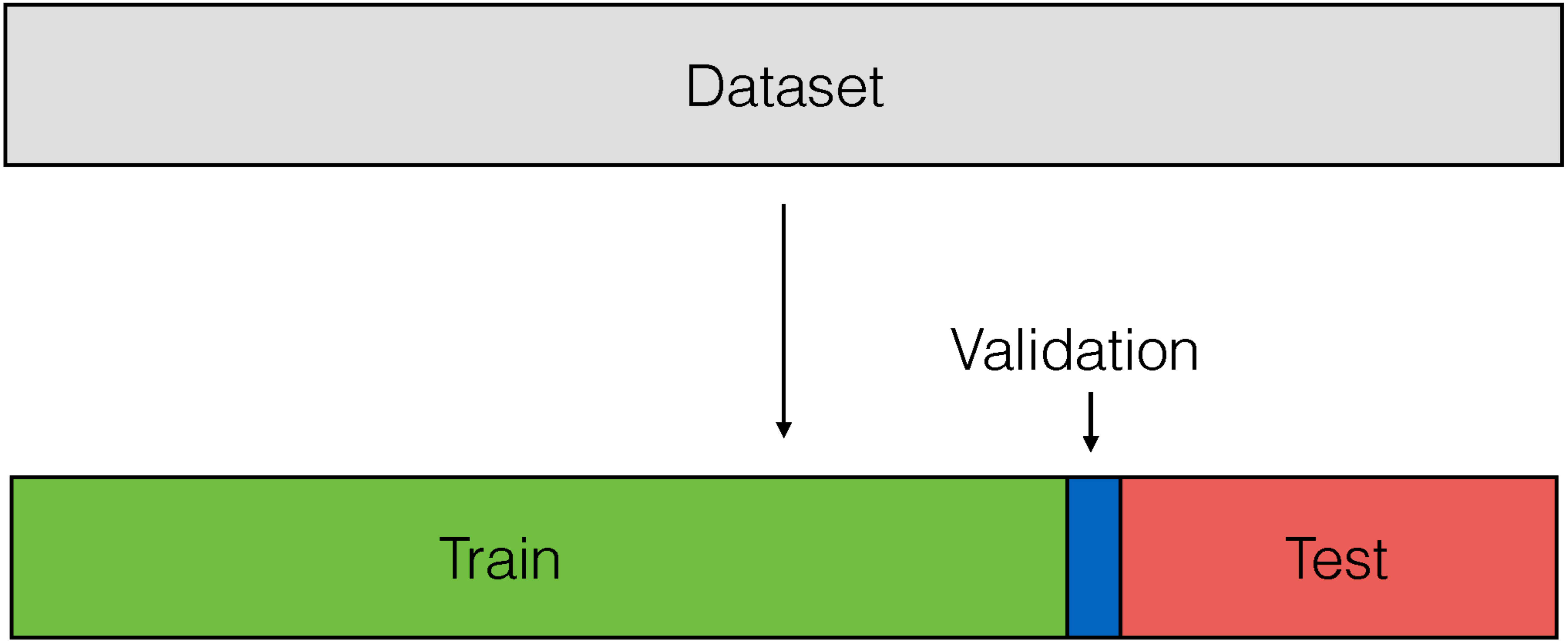
**General rule:** models that are "bigger" or have more capacity are more likely to overfit




[Image: https://en.wikipedia.org/wiki/File:Overfitted_Data.png]

# (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

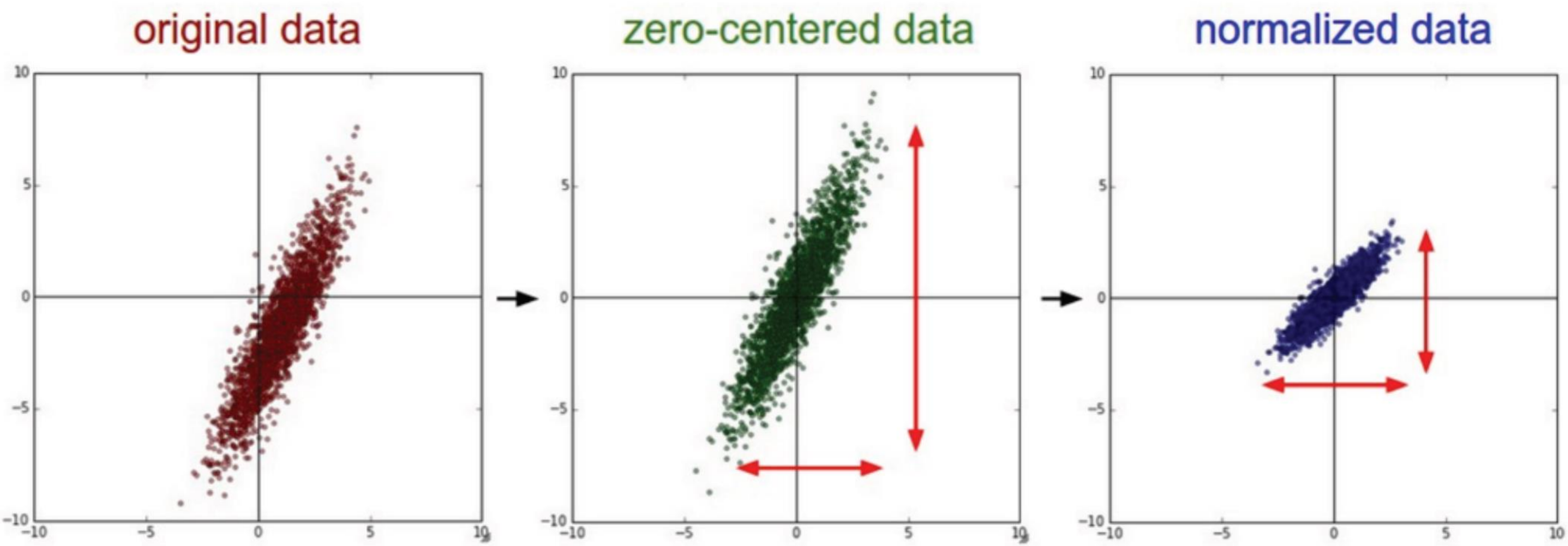

**Be careful with false discovery:**

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:**



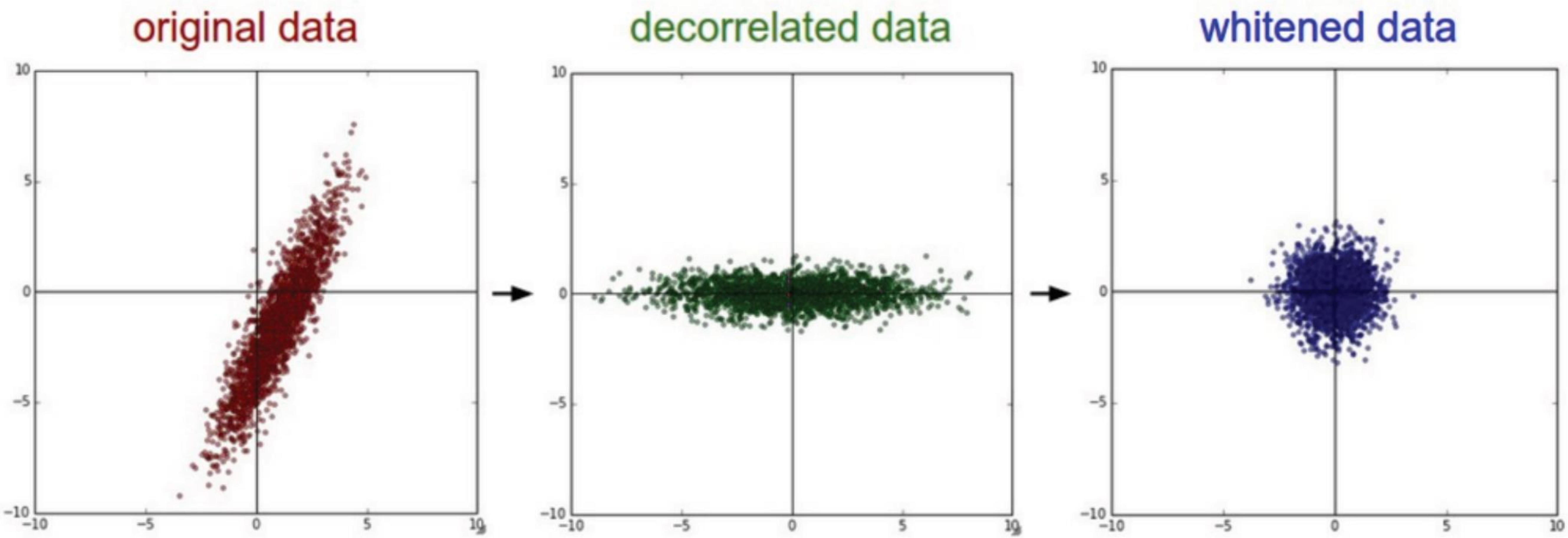

```
X -= np.mean(axis=0, keepdims=True)
```

```
X /= np.std(axis=0, keepdims=True)
```

*Figure: Andrej Karpathy*

# (1) Data preprocessing

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



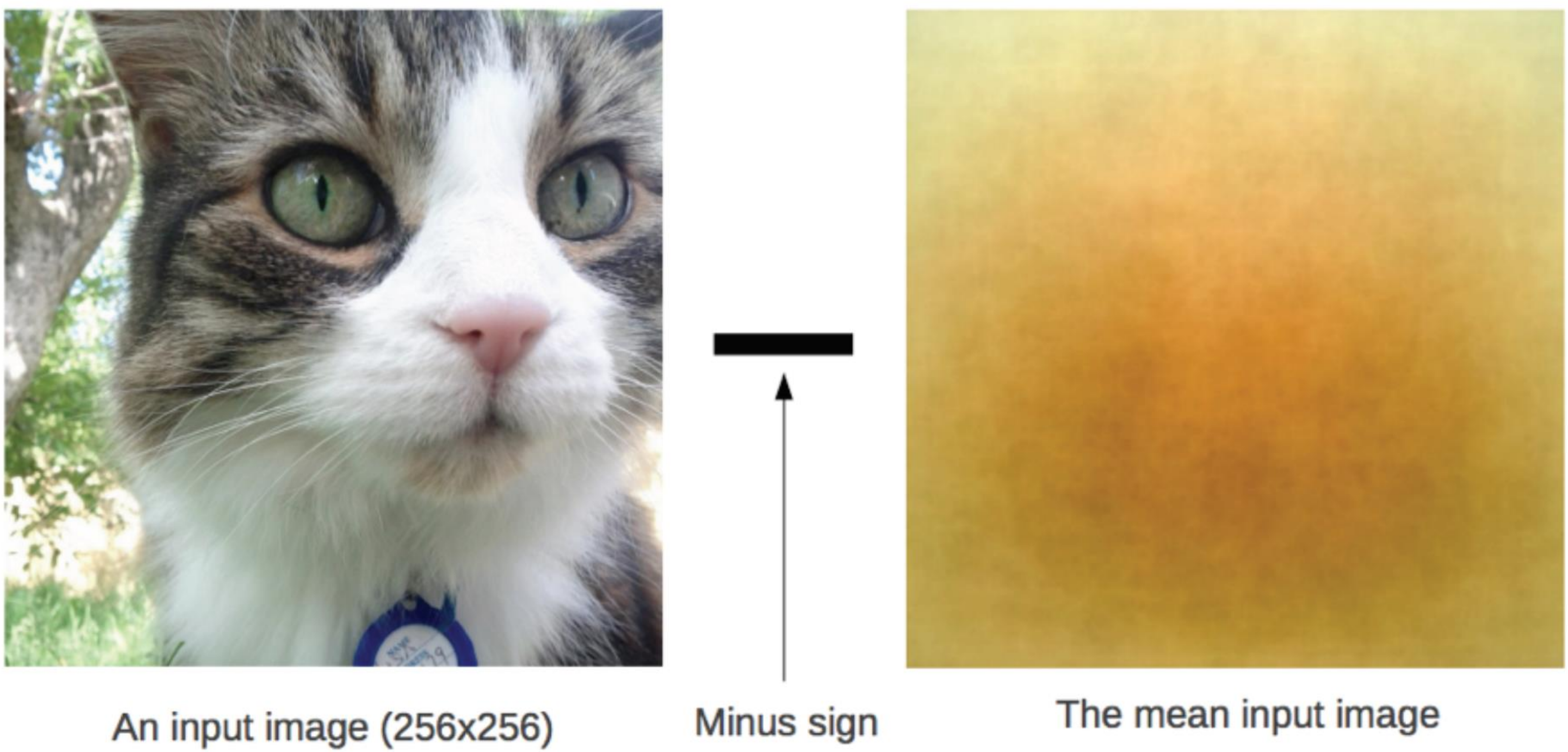

(data has diagonal covariance matrix)

(covariance matrix is the identity matrix)

*Slide: Andrej Karpathy*

# (1) Data preprocessing

For ConvNets, typically only the mean is subtracted.

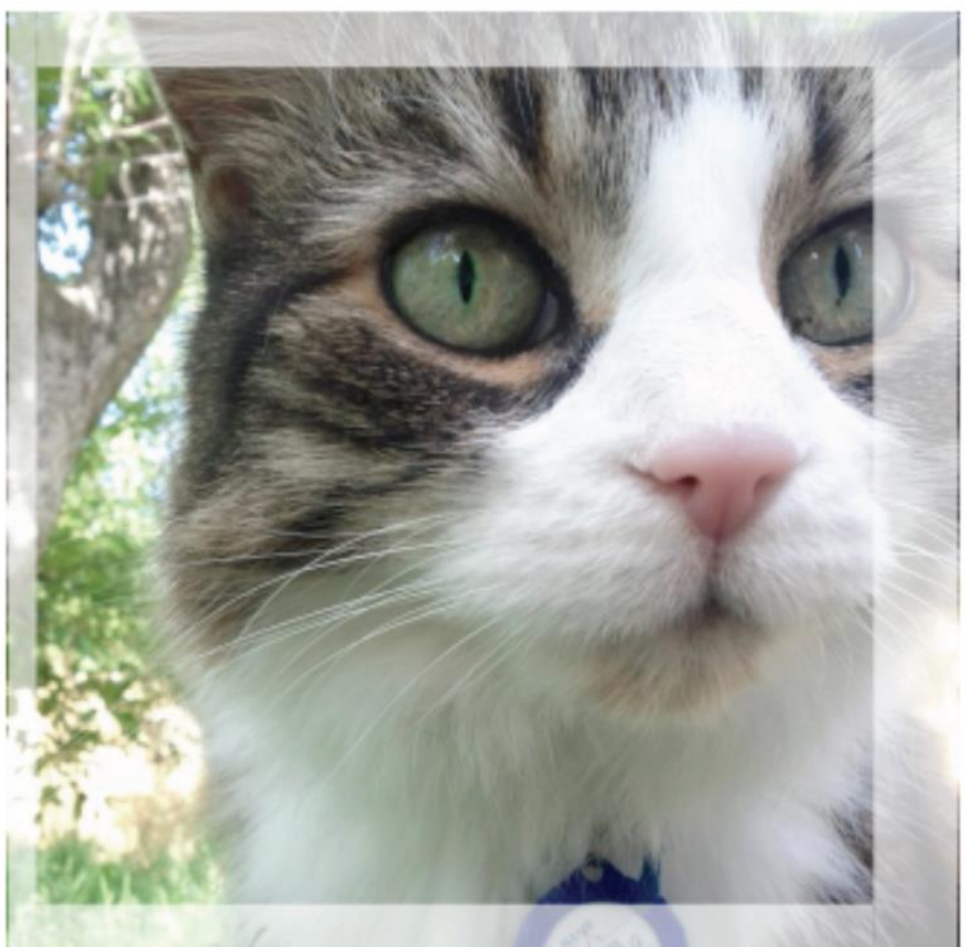



A per-channel mean also works (one value per R,G,B).

*Figure: Alex Krizhevsky*

# (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

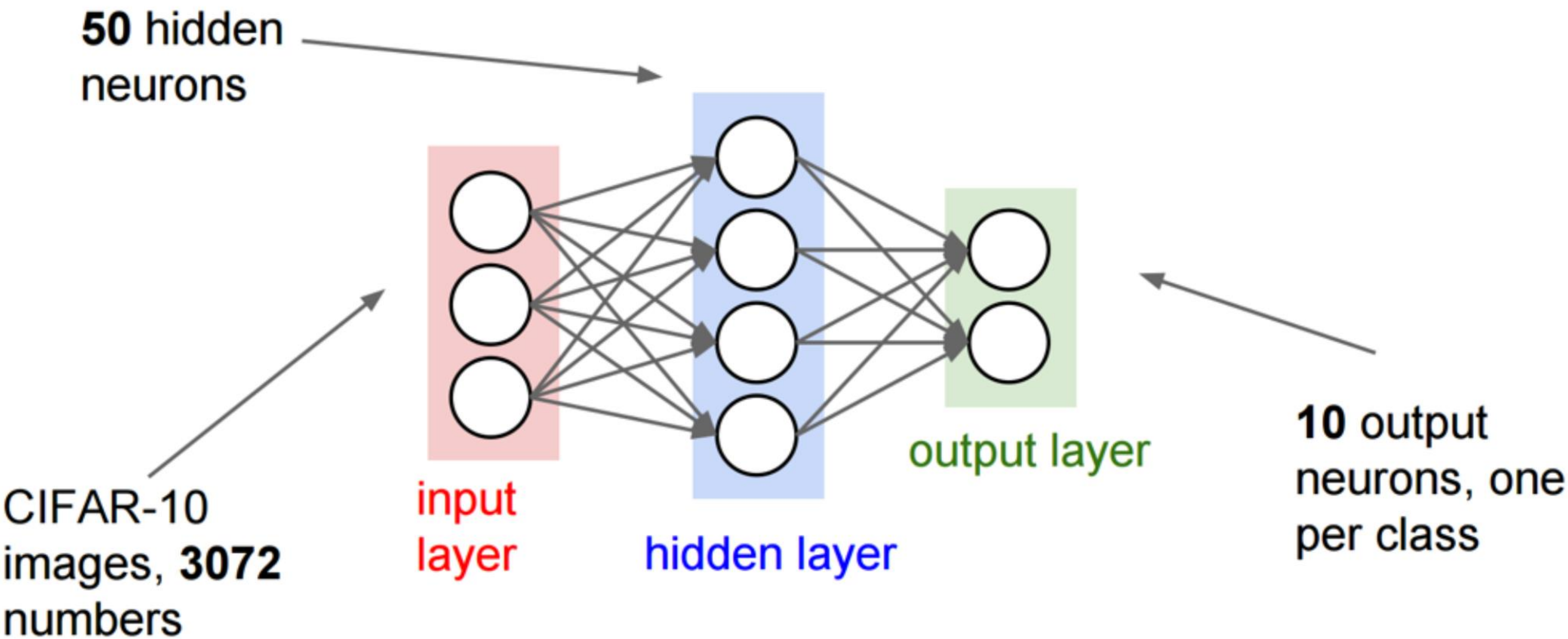
Randomly reflect horizontally

Perform the augmentation live during training

*Figure: Alex Krizhevsky*

# (2) Choose your architecture

**Toy example: one hidden layer of size 50**




*Slide: Andrej Karpathy*

# (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

```
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)
print loss
```

disable regularization

returns the loss and the gradient for all parameters

Slide: Andrej Karpathy

# (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)
print loss
```

crank up regularization

loss went up, good. (sanity check)

*Slide: Andrej Karpathy*

# (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:

```
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)
```

# (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?

```
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=1e6, verbose=True)
```

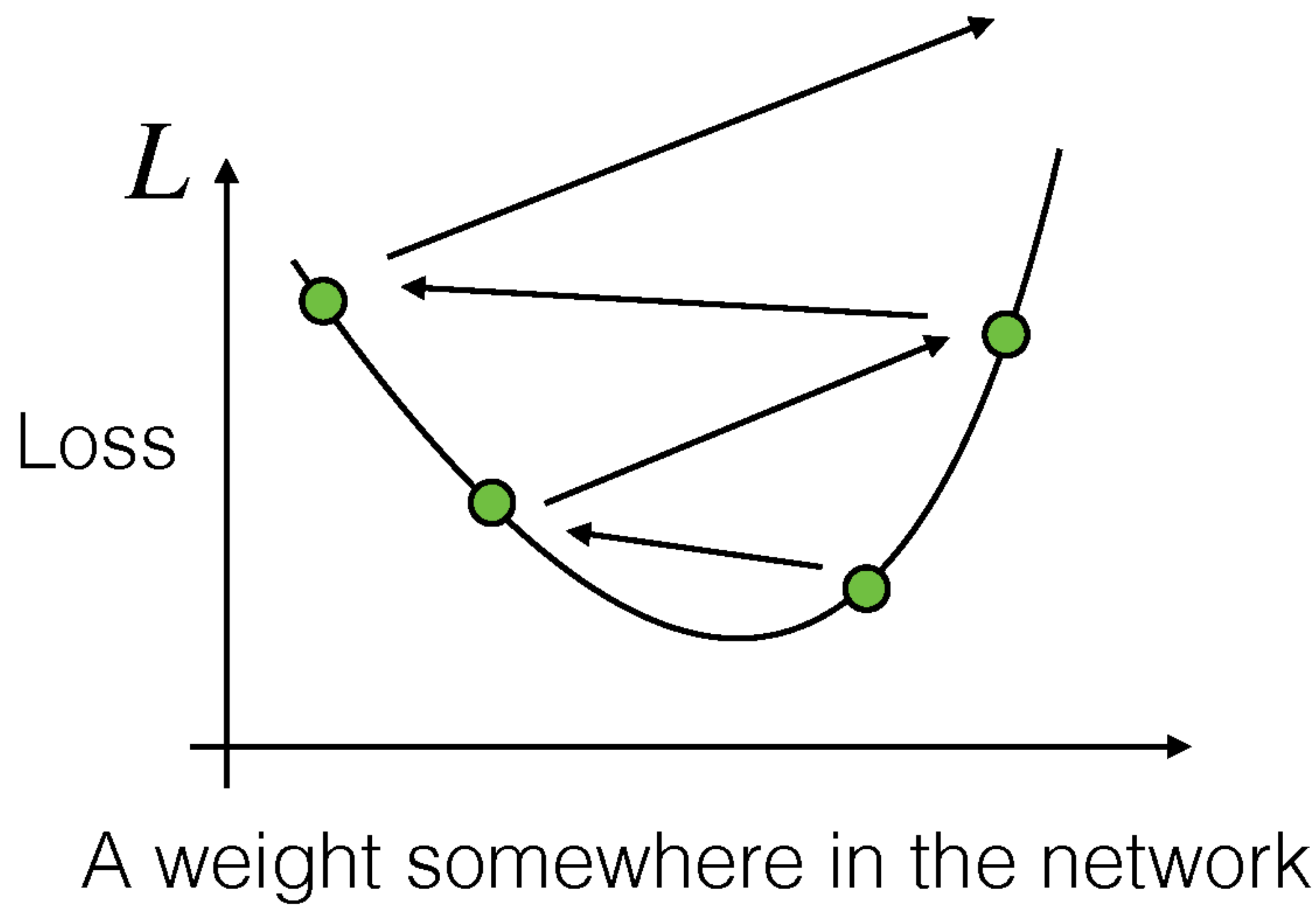
```
/home/karpathy/cs231n/code/cs231n/classifiers/neural_net.py:50: RuntimeWarning: divide by zero en
countered in log
```

**Loss is NaN —> learning rate is too high**

*Slide: Andrej Karpathy*

# (4) Find a learning rate

Learning rate: 1e6 — what could go wrong?



# (4) Find a learning rate

Learning rate: 3e-3

```
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=3e-3, verbose=True)
```

```
Finished epoch 1 / 10: cost 2.186654, train: 0.308000, val 0.306000, lr 3.000000e-03
Finished epoch 2 / 10: cost 2.176230, train: 0.330000, val 0.350000, lr 3.000000e-03
Finished epoch 3 / 10: cost 1.942257, train: 0.376000, val 0.352000, lr 3.000000e-03
Finished epoch 4 / 10: cost 1.827868, train: 0.329000, val 0.310000, lr 3.000000e-03
Finished epoch 5 / 10: cost inf, train: 0.128000, val 0.128000, lr 3.000000e-03
Finished epoch 6 / 10: cost inf, train: 0.144000, val 0.147000, lr 3.000000e-03
```

**Loss is inf —> still too high**

But now we know we should be searching the range [1e-5 … 1e-3]

*Slide: Andrej Karpathy*

# (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)

*Slide: Andrej Karpathy*

# (4) Find a learning rate

## Coarse to fine search

```
max_count = 100
for count in xrange(max_count):
      reg = 10**uniform(-5, 5)
      lr = 10**uniform(-3, -6)

      trainer = ClassifierTrainer()
      model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
      trainer = ClassifierTrainer()
      best_model_local, stats = trainer.train(X_train, y_train, X_val, y_val,
                                   model, two_layer_net,
                                   num_epochs=5, reg=reg,
                                   update='momentum', learning_rate_decay=0.9,
                                   sample_batches = True, batch_size = 100,
                                   learning_rate=lr, verbose=False)
```

note it's best to optimize in log space

```
val_acc: 0.412000, lr: 1.405206e-04, reg: 4.793564e-01, (1 / 100)
val_acc: 0.214000, lr: 7.231888e-06, reg: 2.321281e-04, (2 / 100)
val_acc: 0.208000, lr: 2.119571e-06, reg: 8.011857e+01, (3 / 100)
val_acc: 0.196000, lr: 1.551131e-05, reg: 4.374936e-05, (4 / 100)
val_acc: 0.079000, lr: 1.753300e-05, reg: 1.200424e+03, (5 / 100)
val_acc: 0.223000, lr: 4.215128e-05, reg: 4.196174e+01, (6 / 100)
val_acc: 0.441000, lr: 1.750259e-04, reg: 2.110807e-04, (7 / 100)
val_acc: 0.241000, lr: 6.749231e-05, reg: 4.226413e+01, (8 / 100)
val_acc: 0.482000, lr: 4.296863e-04, reg: 6.642555e-01, (9 / 100)
val_acc: 0.079000, lr: 5.401602e-06, reg: 1.599828e+04, (10 / 100)
val_acc: 0.154000, lr: 1.618508e-06, reg: 4.925252e-01, (11 / 100)
```

*Slide: Andrej Karpathy*

# (4) Find a learning rate

## Coarse to fine search

```
max_count = 100
for count in xrange(max_count):
      reg = 10**uniform(-5, 5)
      lr = 10**uniform(-3, -6)
```

```
val_acc: 0.527000, lr: 5.340517e-04, reg: 4.097824e-01, (0 / 100)
val_acc: 0.492000, lr: 2.279484e-04, reg: 9.991345e-04, (1 / 100)
val_acc: 0.512000, lr: 8.680827e-04, reg: 1.349727e-02, (2 / 100)
val_acc: 0.461000, lr: 1.028377e-04, reg: 1.220193e-02, (3 / 100)
val_acc: 0.460000, lr: 1.113730e-04, reg: 5.244309e-02, (4 / 100)
val_acc: 0.498000, lr: 9.477776e-04, reg: 2.001293e-03, (5 / 100)
val_acc: 0.469000, lr: 1.484369e-04, reg: 4.328313e-01, (6 / 100)
val_acc: 0.522000, lr: 5.586261e-04, reg: 2.312685e-04, (7 / 100)
val_acc: 0.530000, lr: 5.808183e-04, reg: 8.259964e-02, (8 / 100)
val_acc: 0.489000, lr: 1.979168e-04, reg: 1.010889e-04, (9 / 100)
val_acc: 0.490000, lr: 2.036031e-04, reg: 2.406271e-03, (10 / 100)
val_acc: 0.475000, lr: 2.021162e-04, reg: 2.287807e-01, (11 / 100)
val_acc: 0.460000, lr: 1.135527e-04, reg: 3.905040e-02, (12 / 100)
val_acc: 0.515000, lr: 6.947668e-04, reg: 1.562808e-02, (13 / 100)
val_acc: 0.531000, lr: 9.471549e-04, reg: 1.433895e-03, (14 / 100)
val_acc: 0.509000, lr: 3.140888e-04, reg: 2.857518e-01, (15 / 100)
val_acc: 0.514000, lr: 6.438349e-04, reg: 3.033781e-01, (16 / 100)
val_acc: 0.502000, lr: 3.921784e-04, reg: 2.707126e-04, (17 / 100)
val_acc: 0.509000, lr: 9.752279e-04, reg: 2.850865e-03, (18 / 100)
val_acc: 0.500000, lr: 2.412048e-04, reg: 4.997821e-04, (19 / 100)
val_acc: 0.466000, lr: 1.319314e-04, reg: 1.189915e-02, (20 / 100)
val_acc: 0.516000, lr: 8.039527e-04, reg: 1.528291e-02, (21 / 100)
```

**Remember this is just a 2 layer neural net with 50 neurons**

**53%**

*Slide: Andrej Karpathy*

# (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

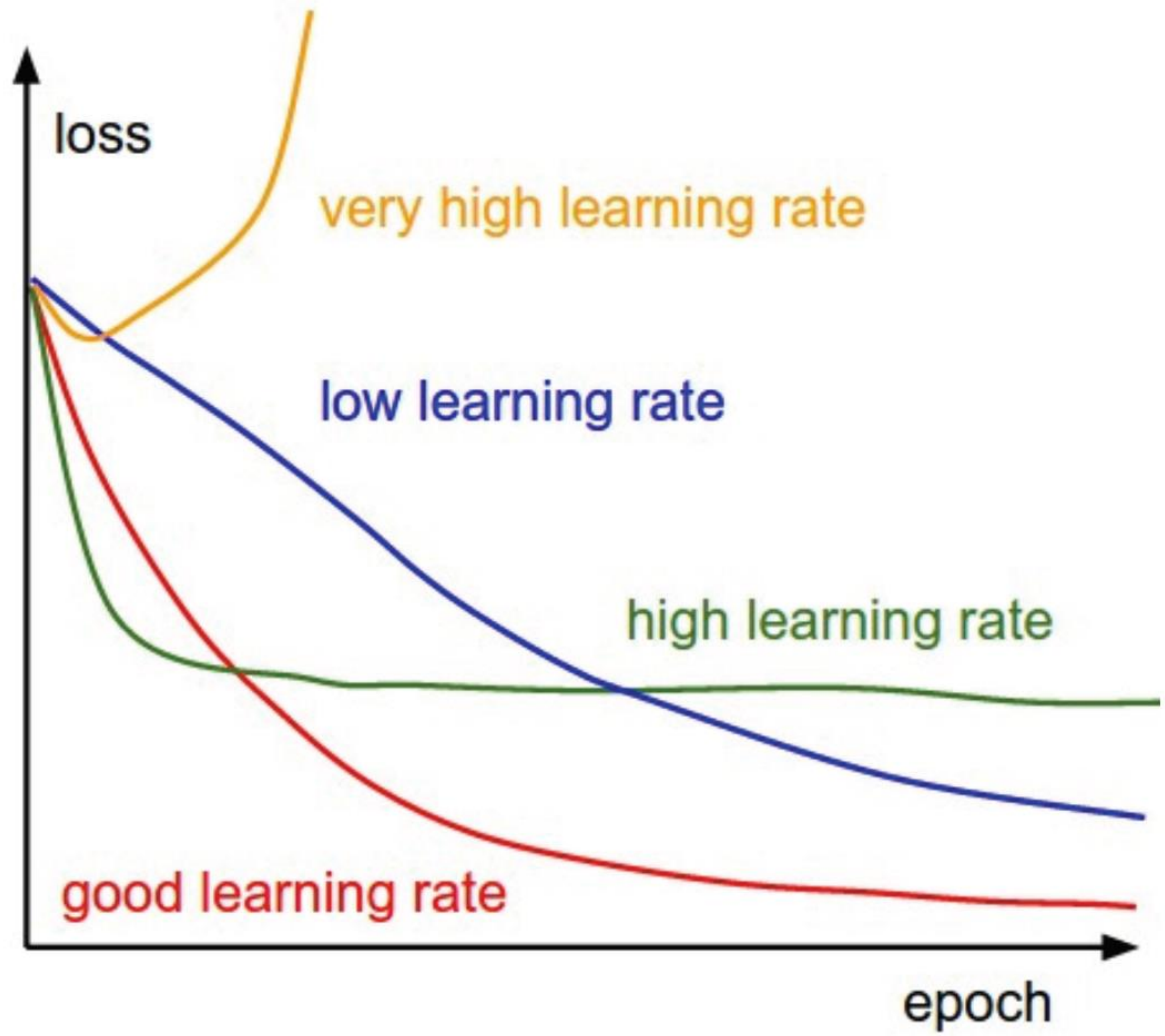



*Figure: Andrej Karpathy*

# (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)

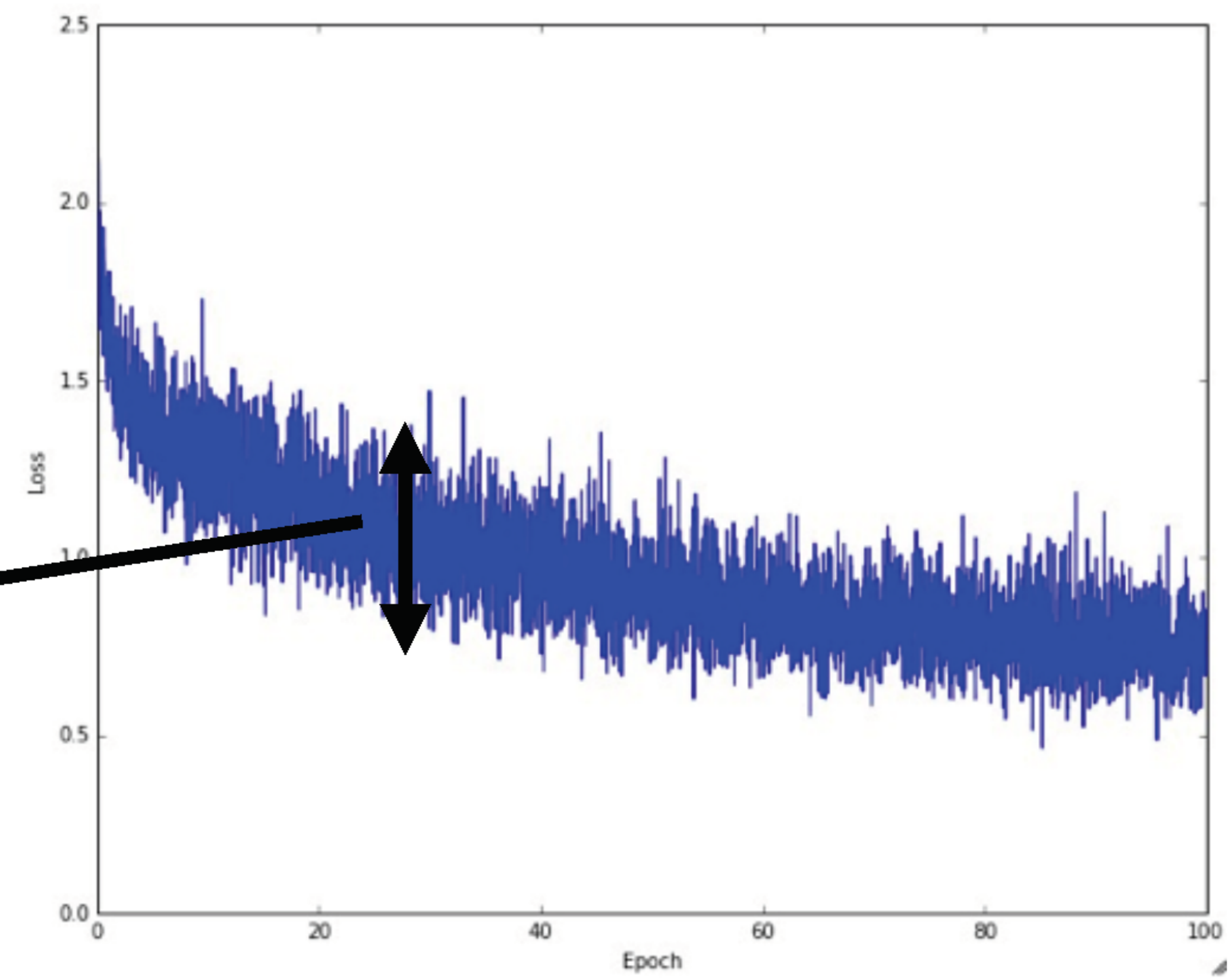
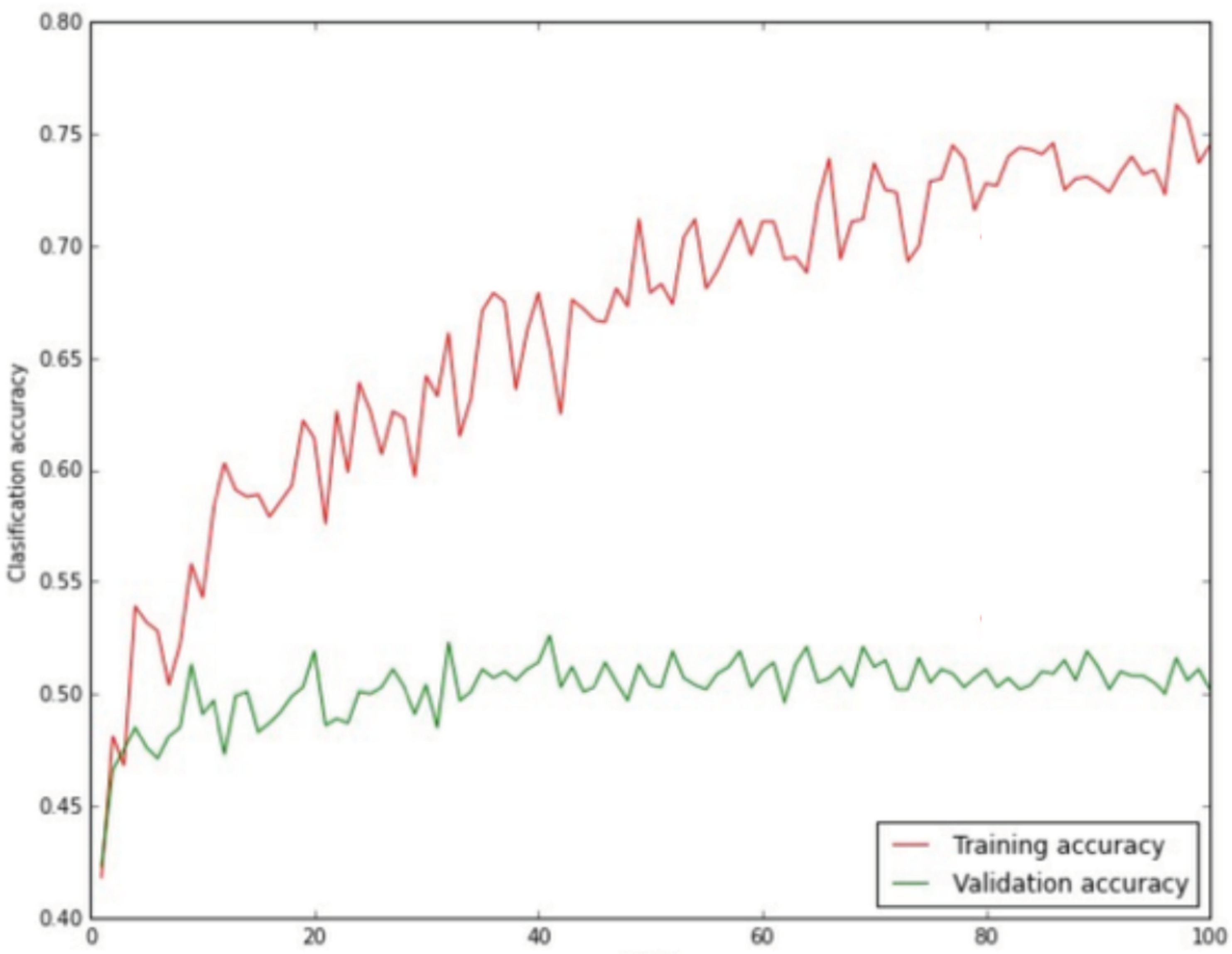



*Figure: Andrej Karpathy*

# (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

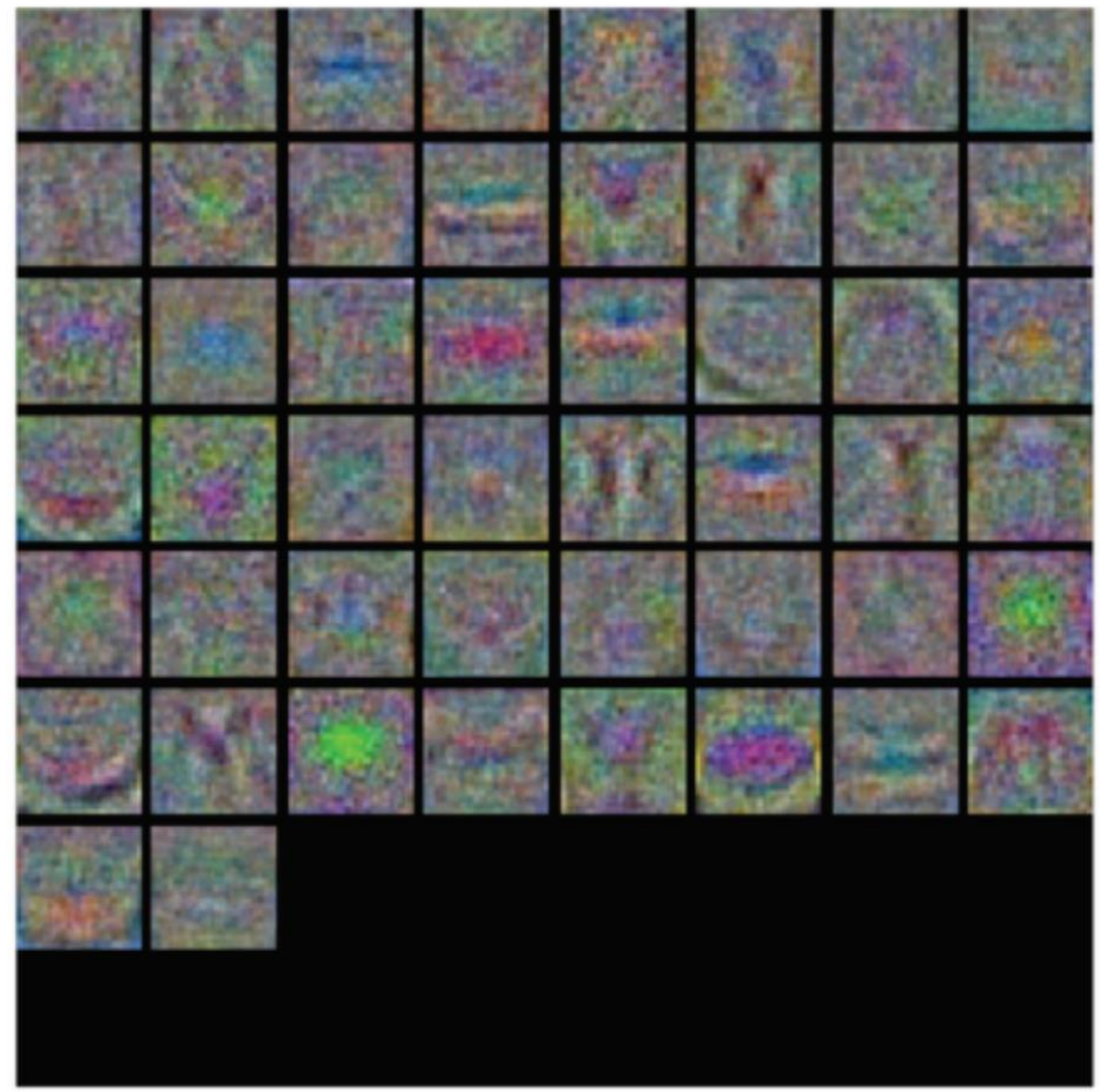
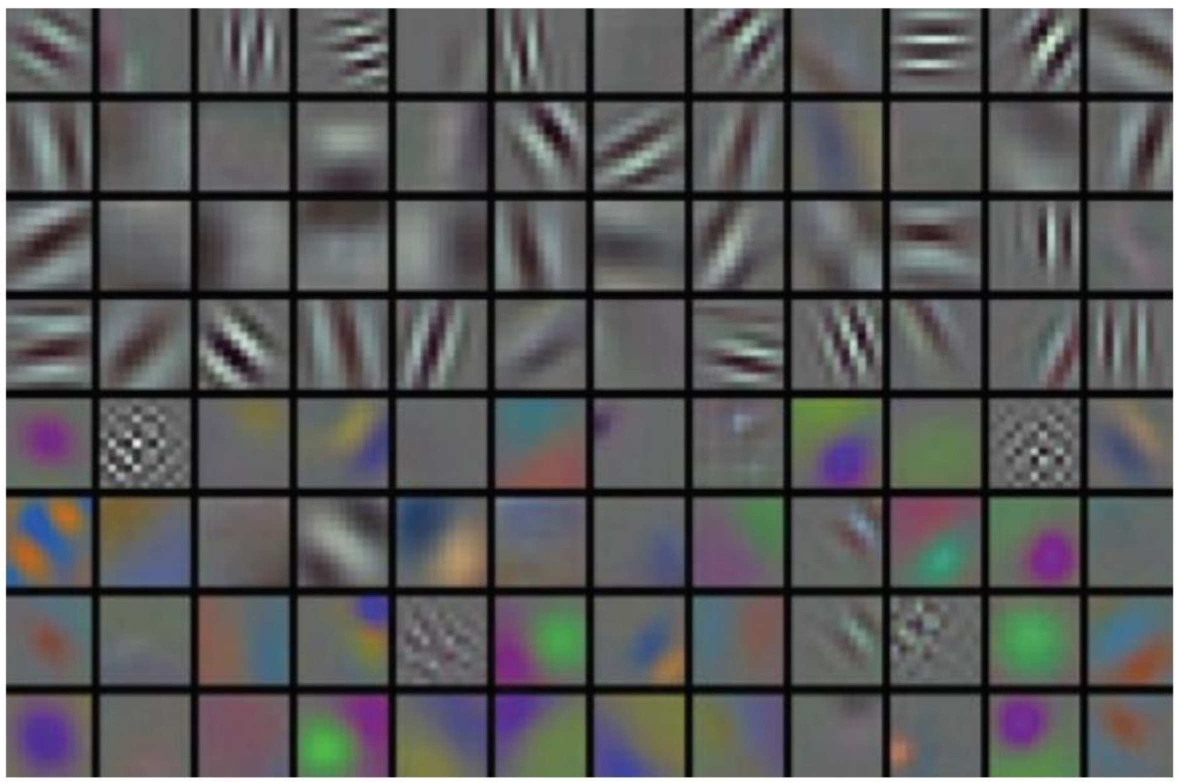


*Figure: Andrej Karpathy*

# (4) Find a learning rate

**Visualize the weights**



Nice clean weights:
training is proceeding well

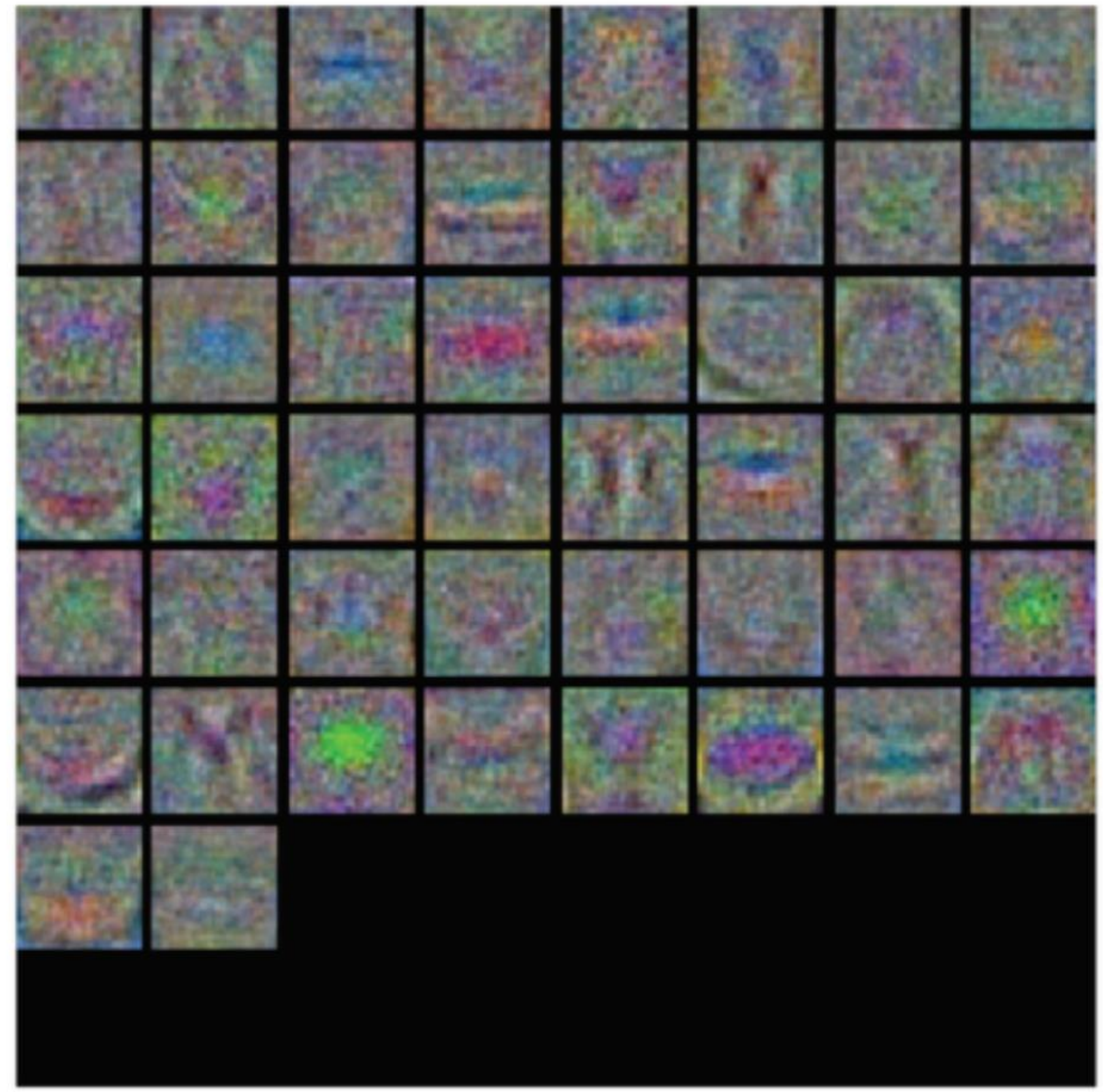


*Figure: Alex Krizhevsky , Andrej Karpathy*

# 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

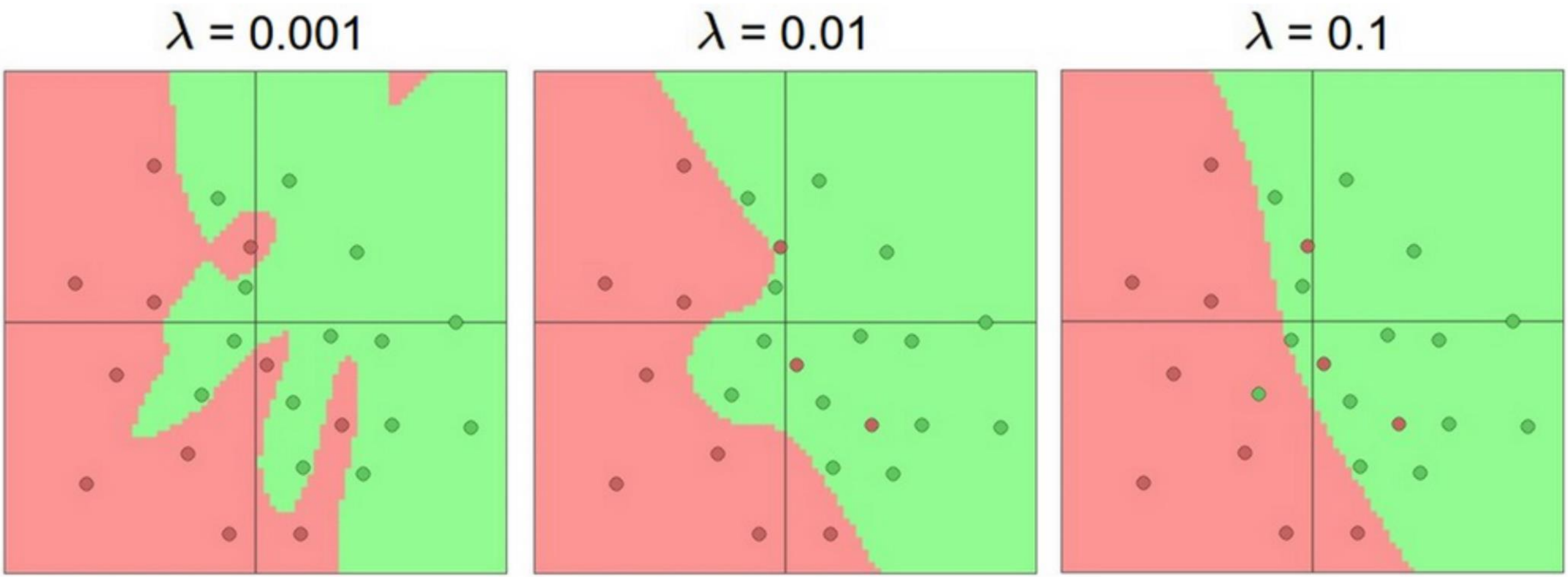
- Network architecture
- Learning rate, decay schedule, update type
- Regularization (L2, L1, maxnorm, dropout, …)
- Loss function (softmax, SVM, …)
- Weight initialization

Neural network parameters

# (Recall) Regularization reduces overfitting

$$L = L_{\text{data}} + L_{\text{reg}}$$

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




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

# Example Regularizers

**L2 regularization**

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

(L2 regularization encourages small weights)

**L1 regularization**

$$L_{\text{reg}} = \lambda \left\| W \right\|_1 = \lambda \sum_{ij} \left| W_{ij} \right|$$

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

**"Elastic net"**

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

(combine L1 and L2 regularization)

**Max norm**

Clamp weights to some max norm

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

# “Weight decay”

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

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

Gradient descent step:

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

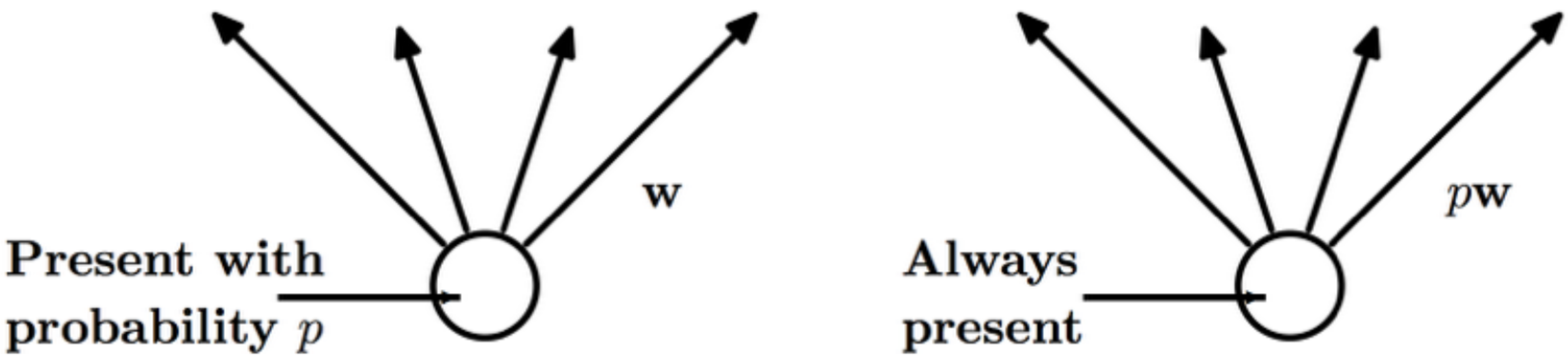
Weight decay: $\alpha\lambda$ (weights always decay by this amount)

**Note:** biases are sometimes excluded from regularization

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

# Dropout

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



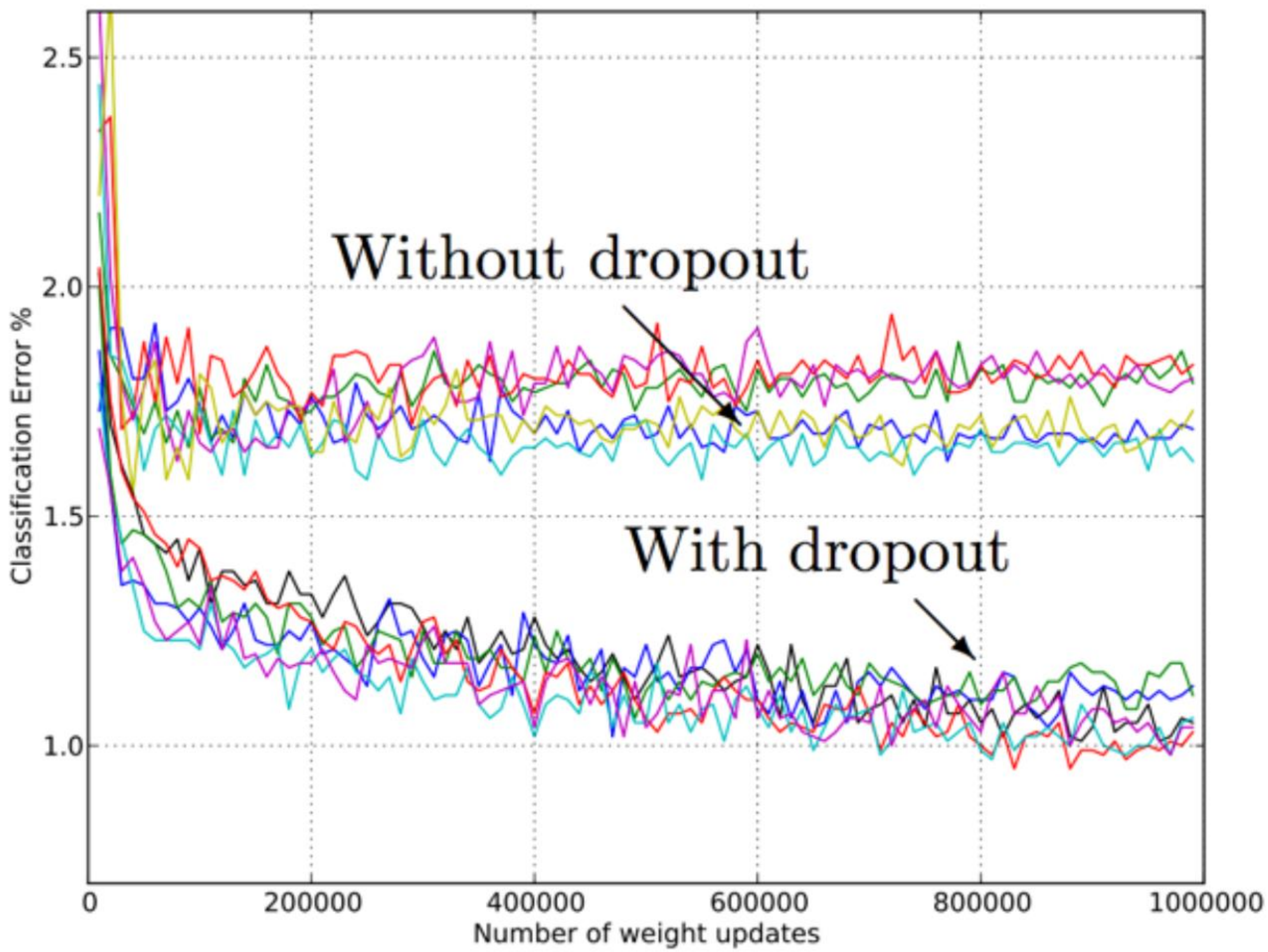

(a) At training time

(b) At test time

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

# Dropout

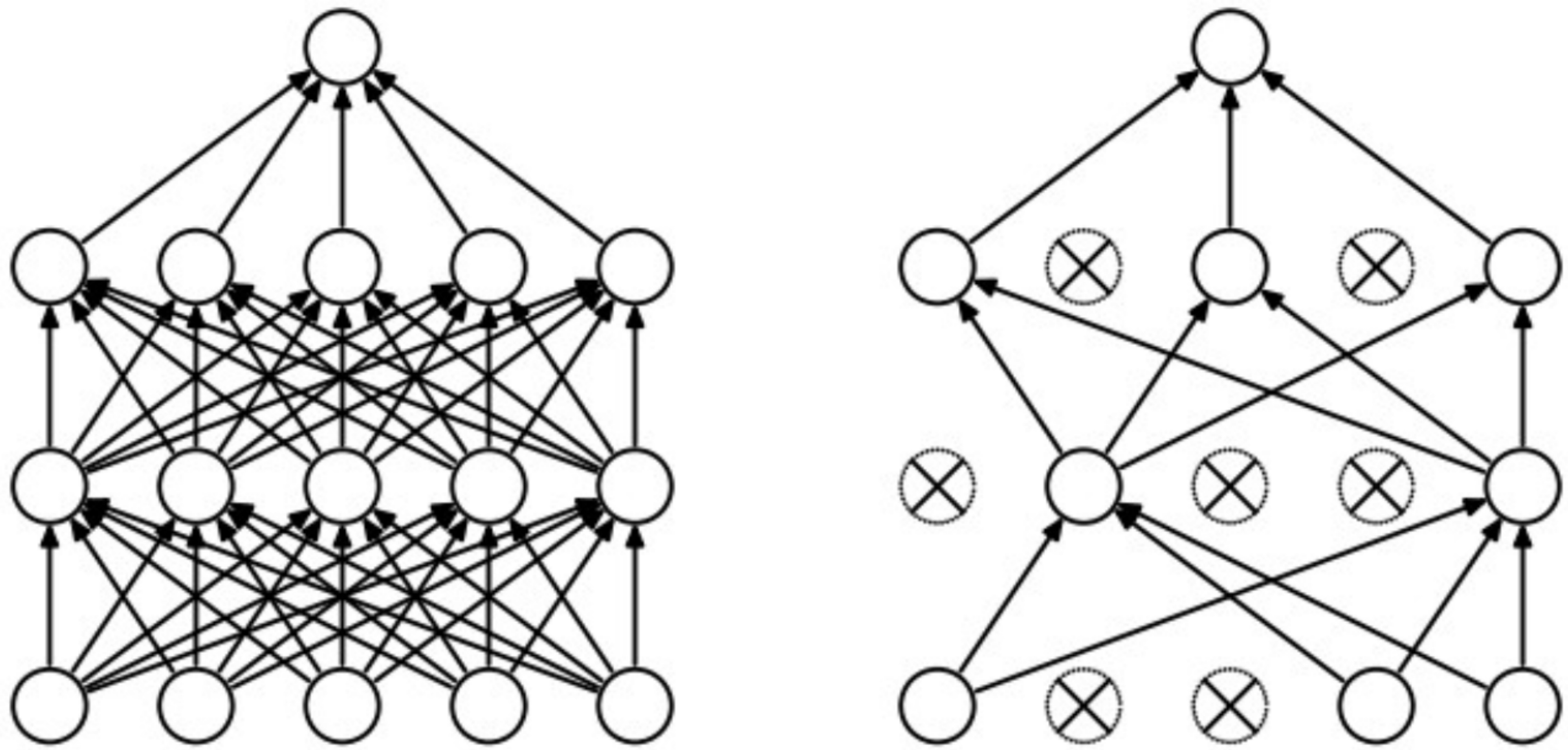
**Simple but powerful technique to reduce overfitting:**




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

# Dropout

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



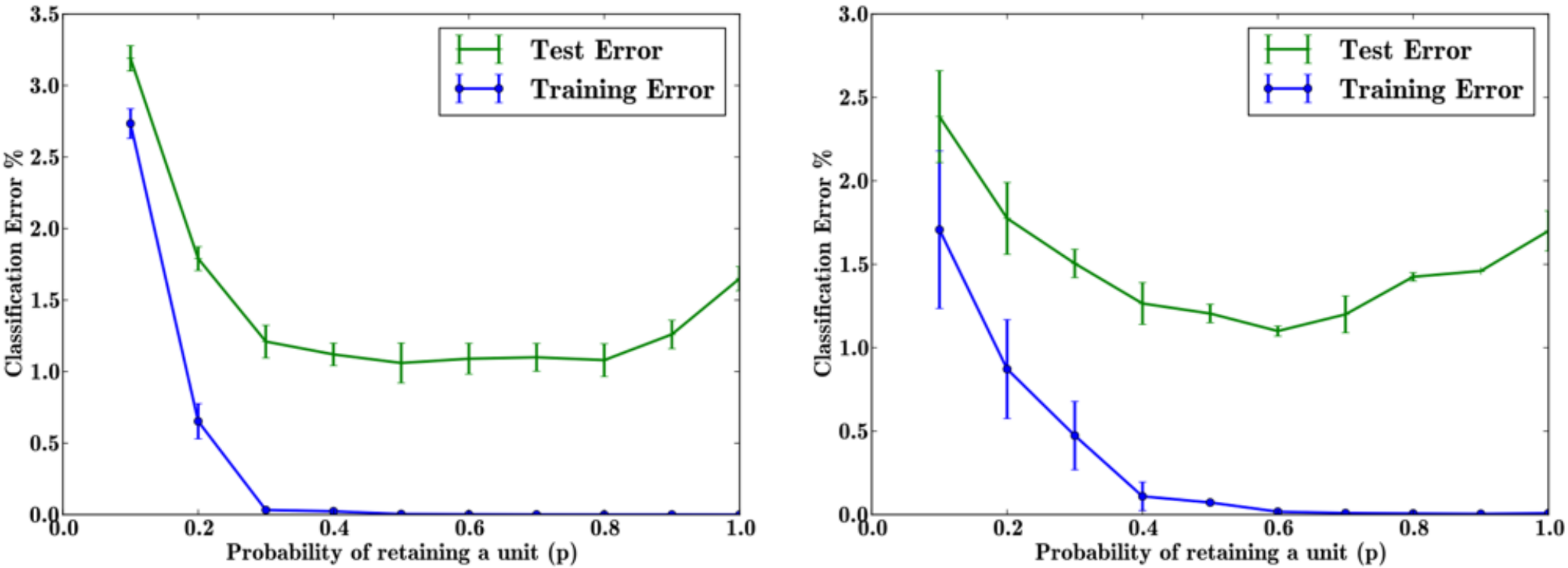
(a) Standard Neural Net (b) After applying dropout.

**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.

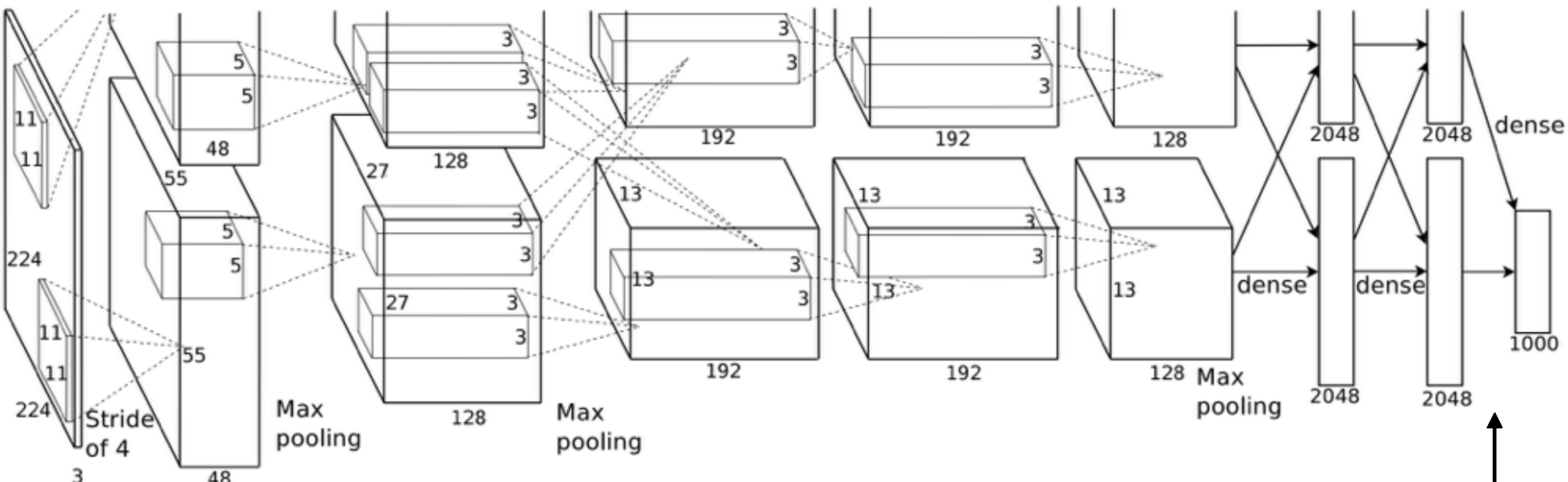
[Srivasta et al, "Dropout: A Simple Way to Prevent Neural Networks from Overfitting", JMLR 2014]

# 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

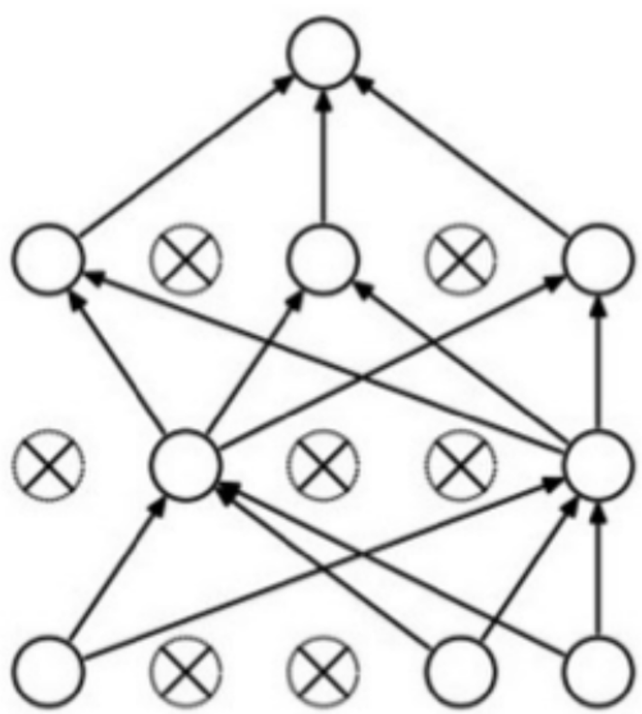
```python
p = 0.5 # probability of keeping a unit active. higher = less dropout

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:

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

```
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

# Questions?