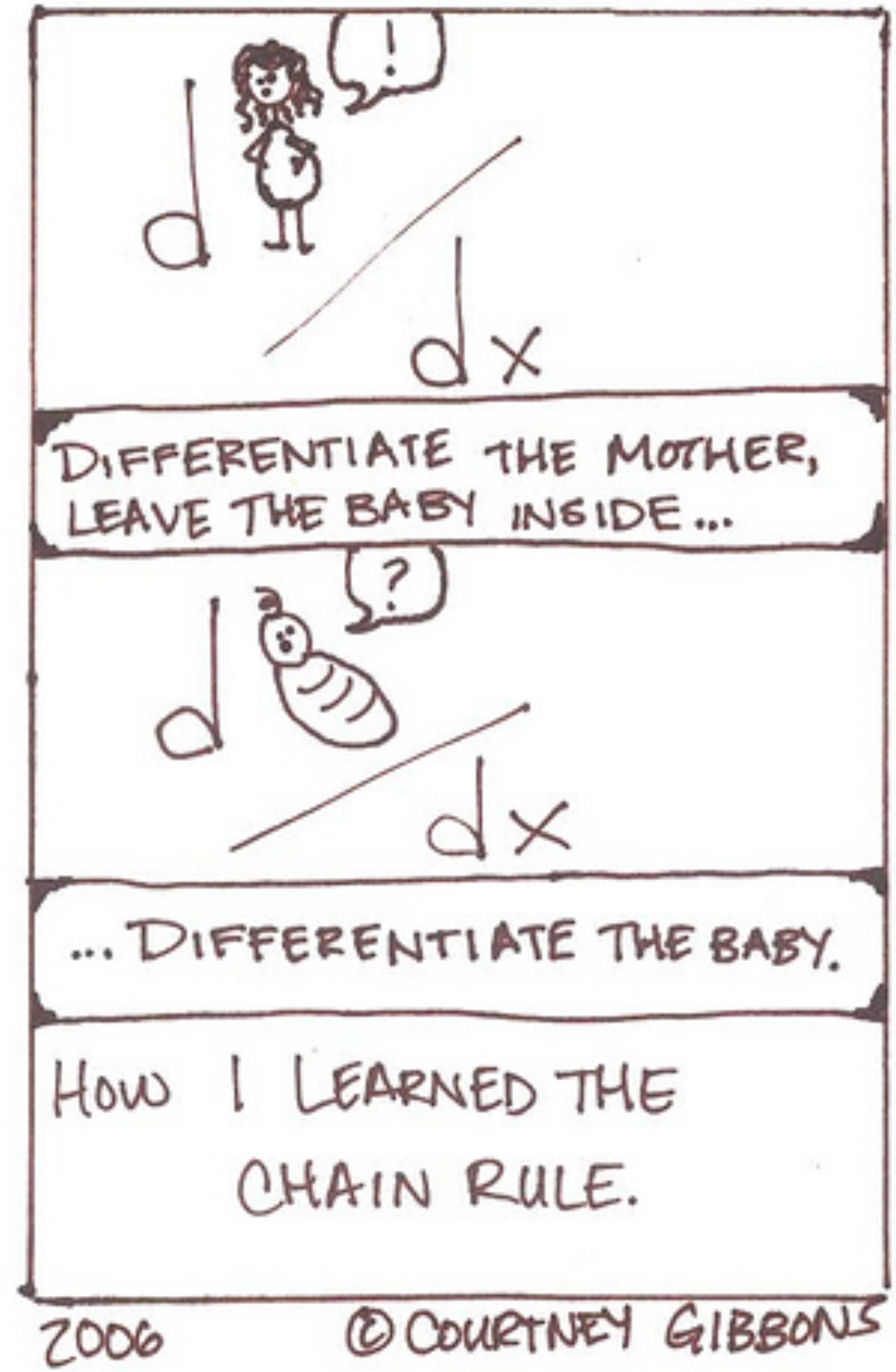
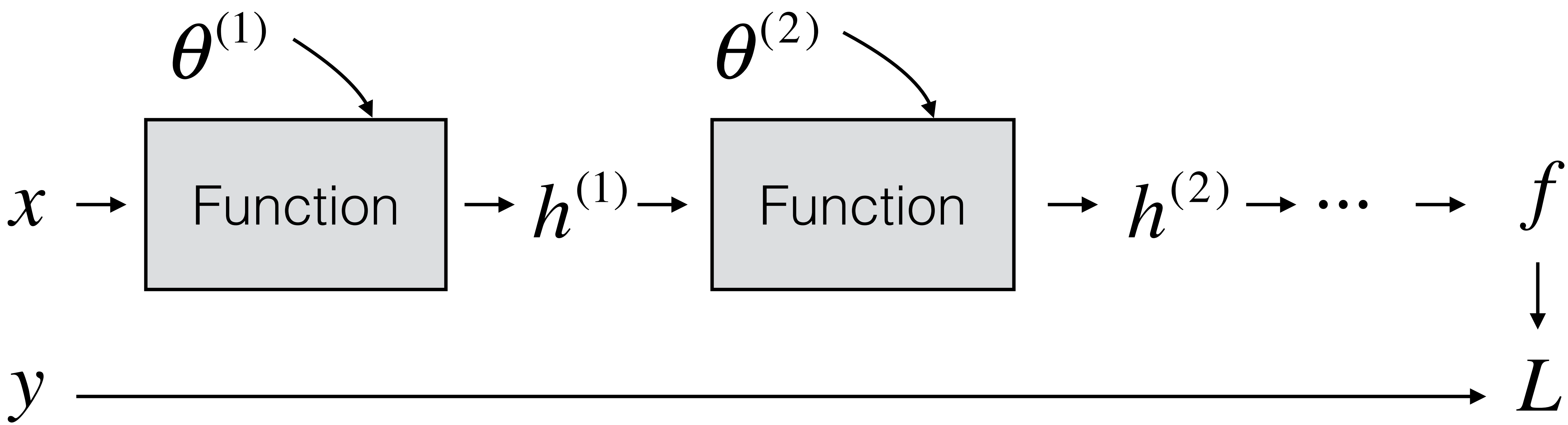
Convolutional Neural Networks

CS 4670
Sean Bell

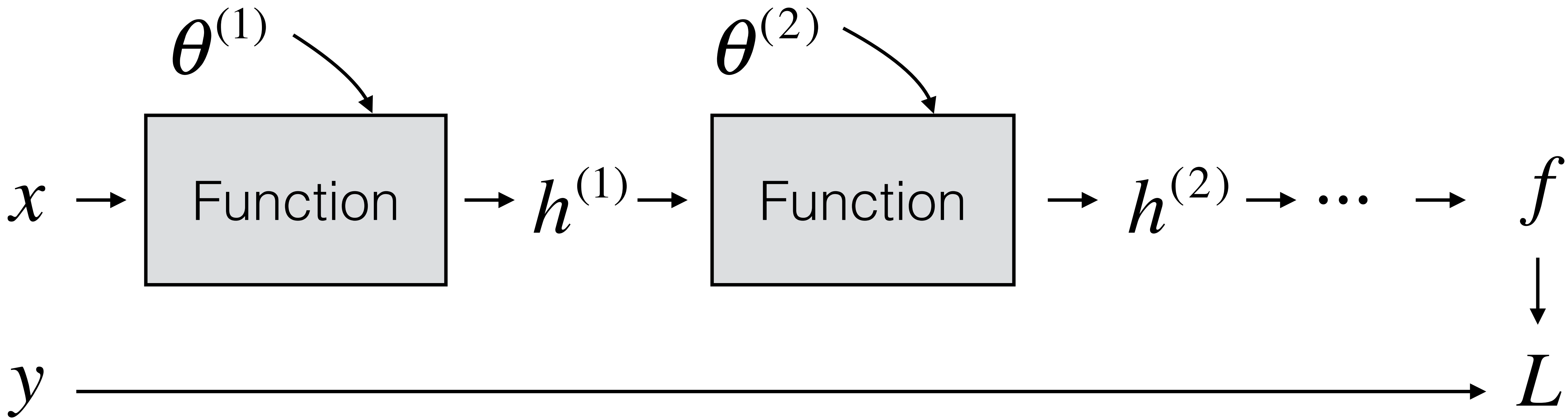


http://brownsharpie.courtneygibbons.org/?p=90

Review: Setup

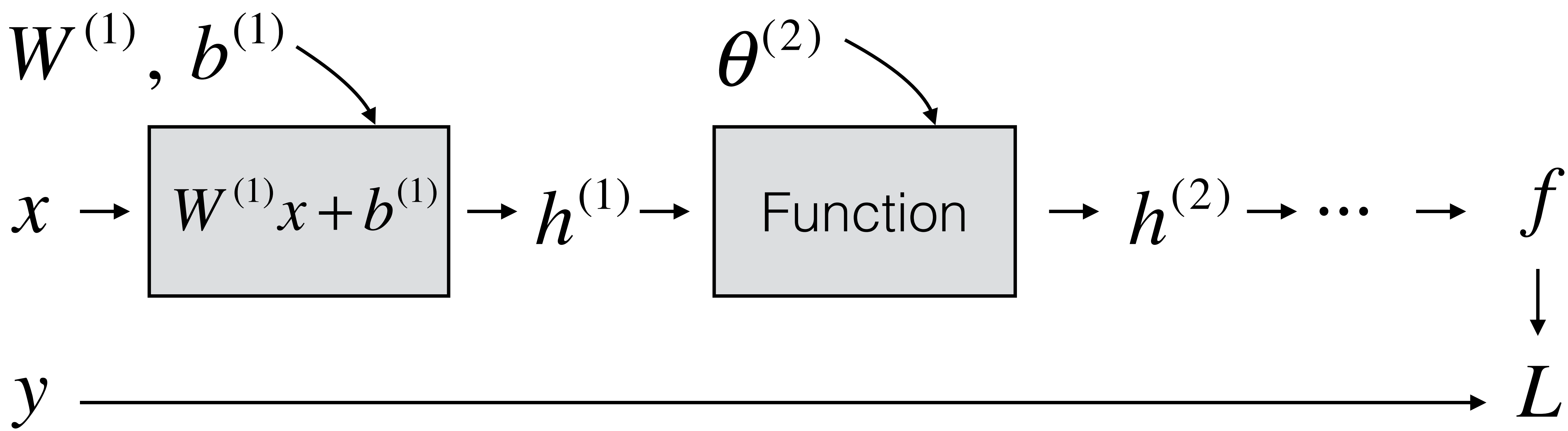


Review: Setup



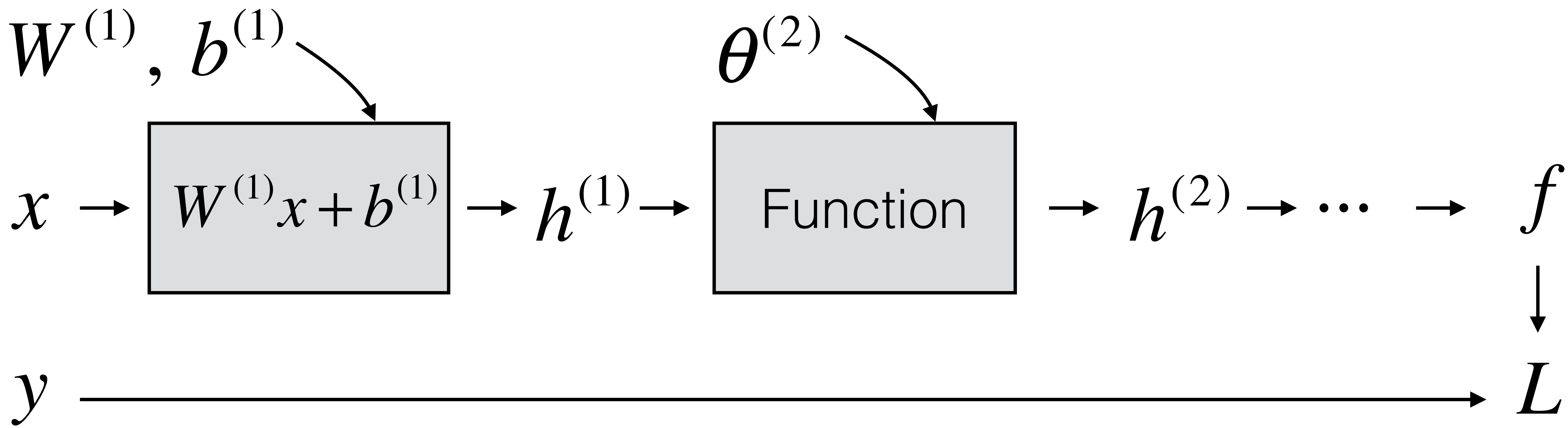
- **Goal:** Find a value for parameters ($\theta^{(1)}$, $\theta^{(2)}$, …), so that the loss (L) is small

Review: Setup

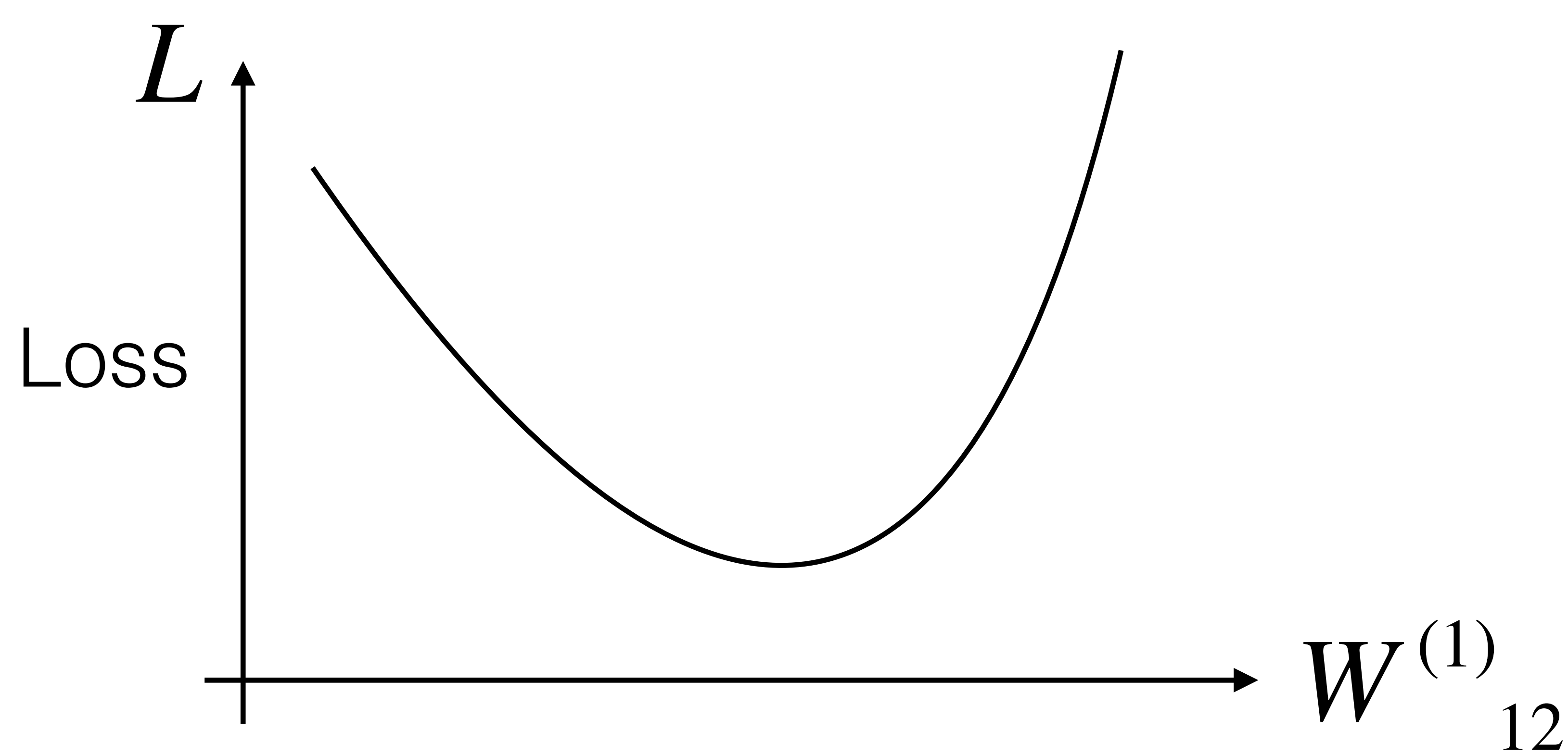


Toy Example:

Review: Setup

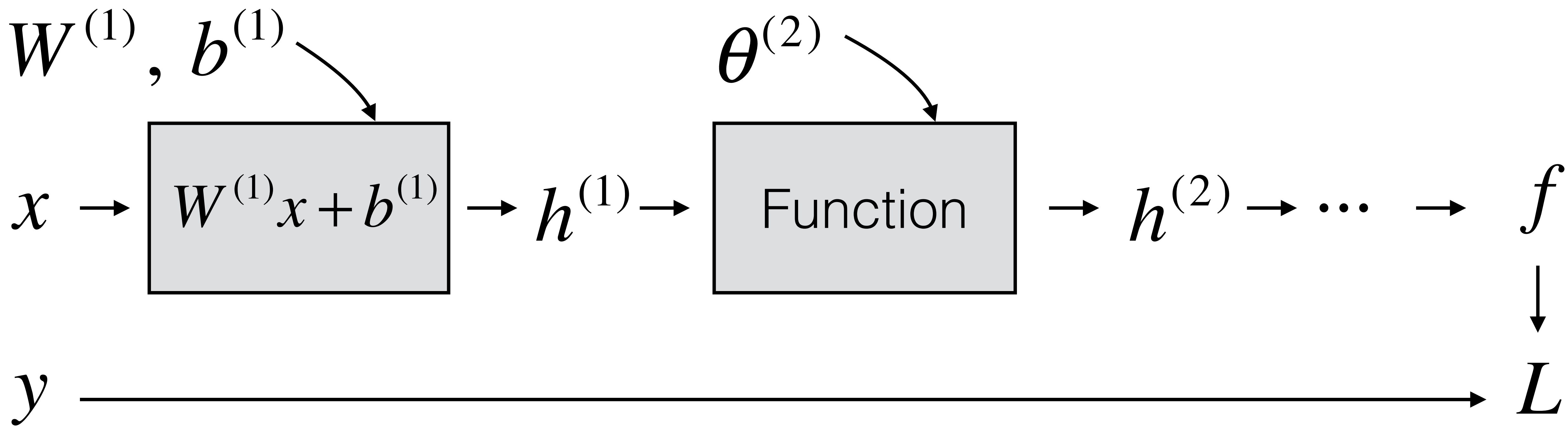


Toy Example:

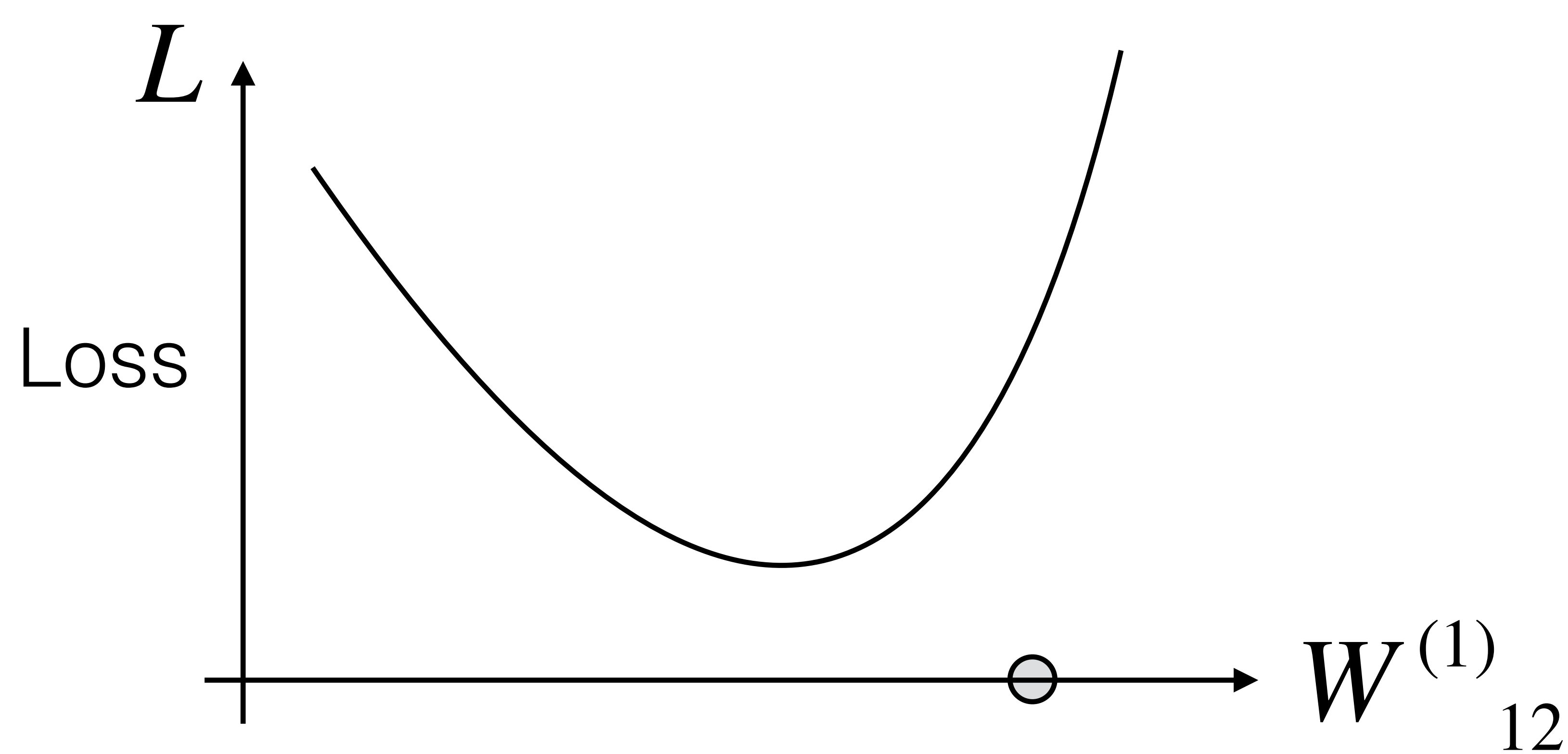


A weight somewhere in the network

Review: Setup

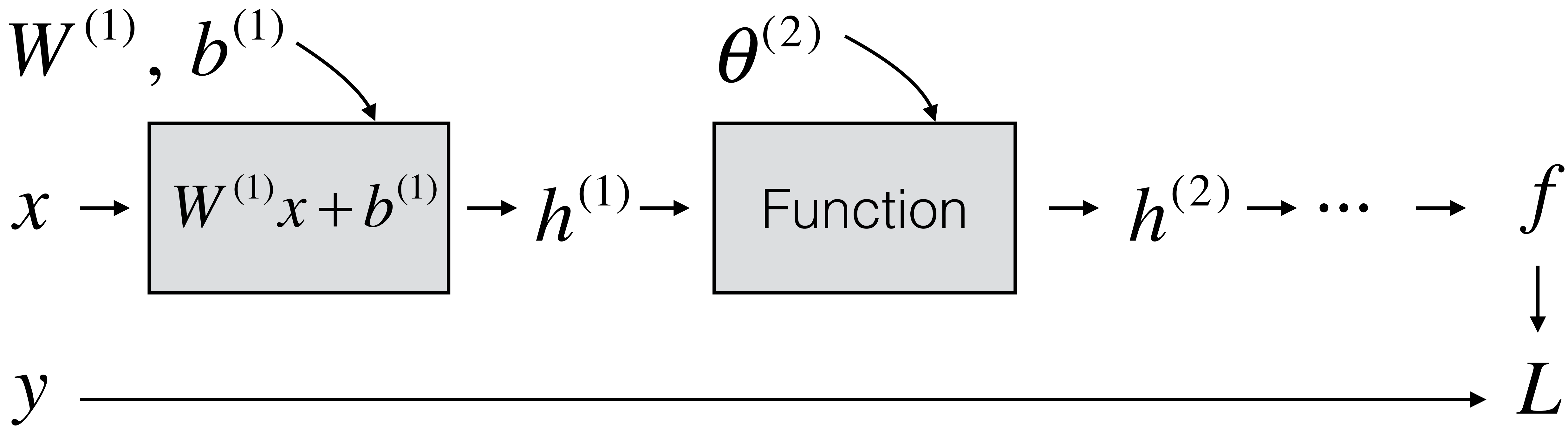


Toy Example:

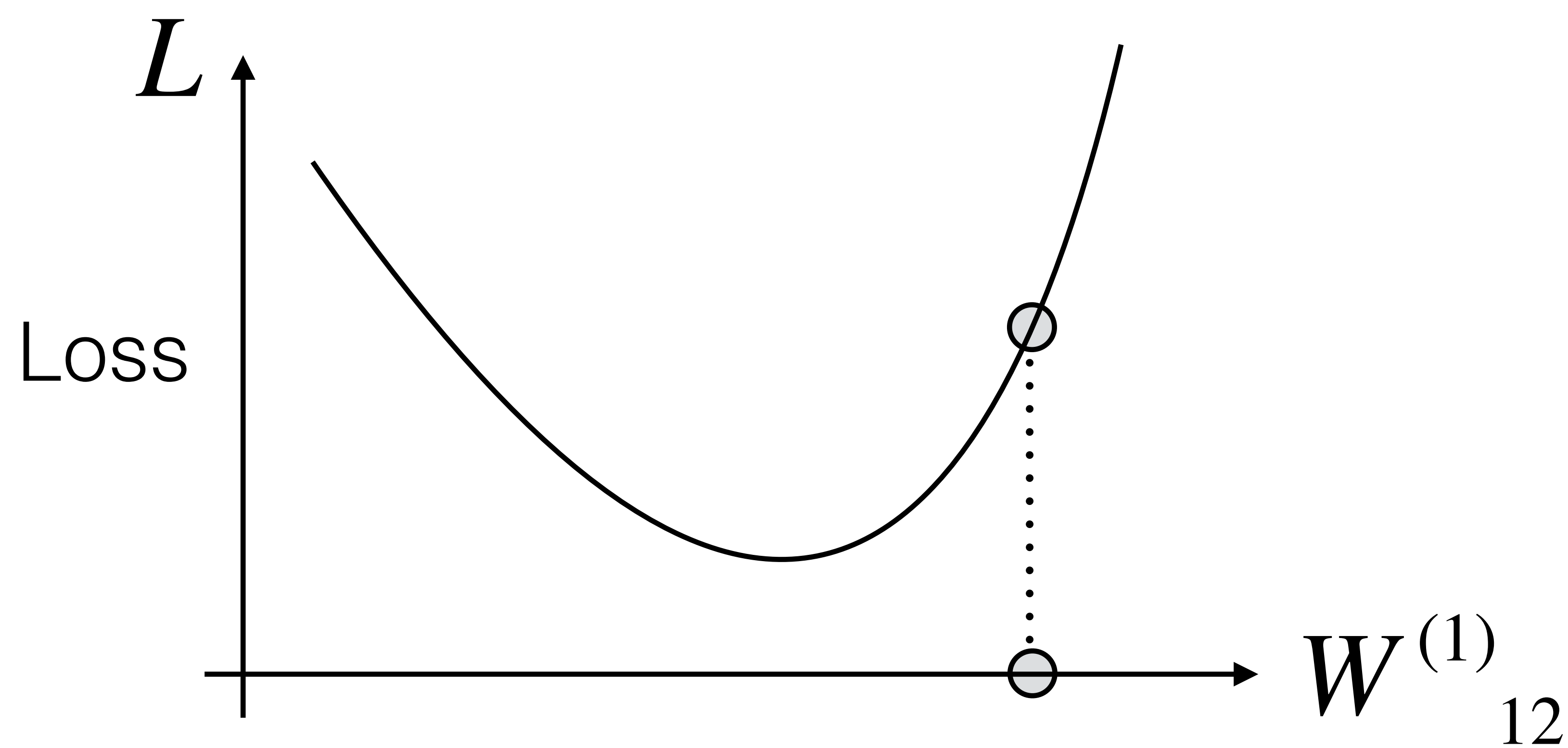


A weight somewhere in the network

Review: Setup

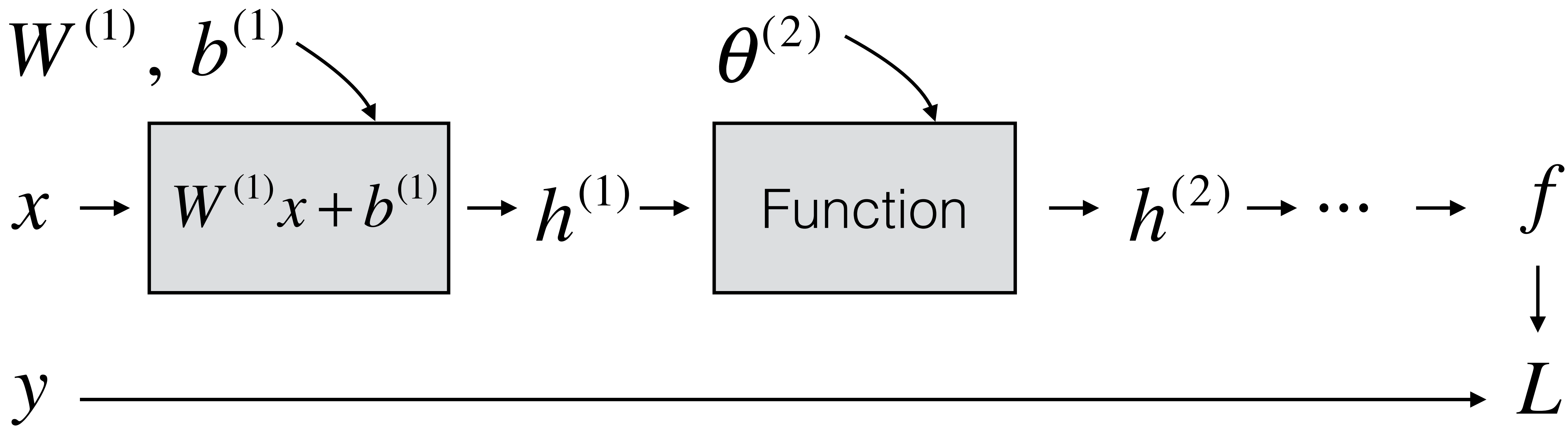


Toy Example:

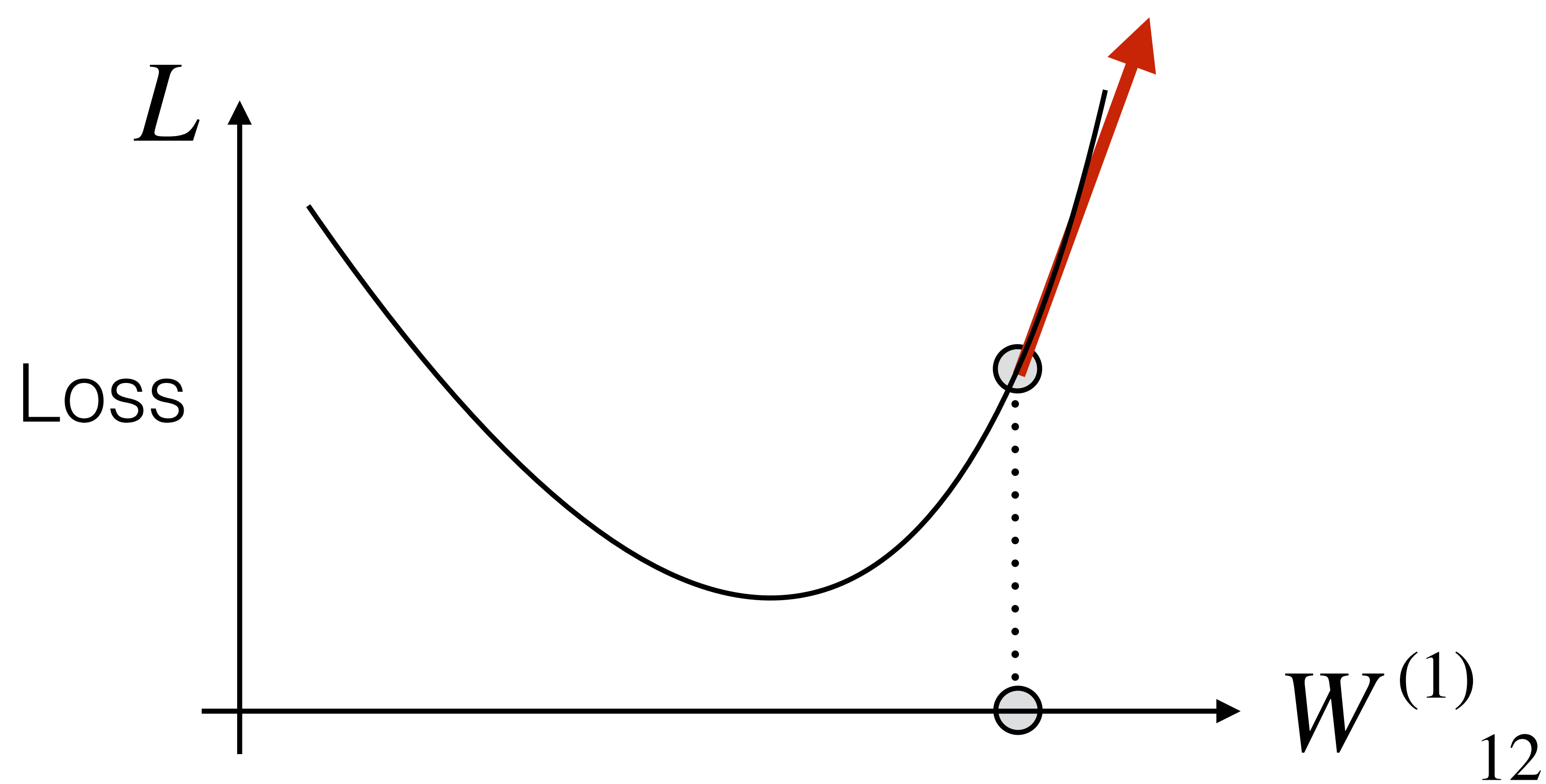


A weight somewhere in the network

Review: Setup

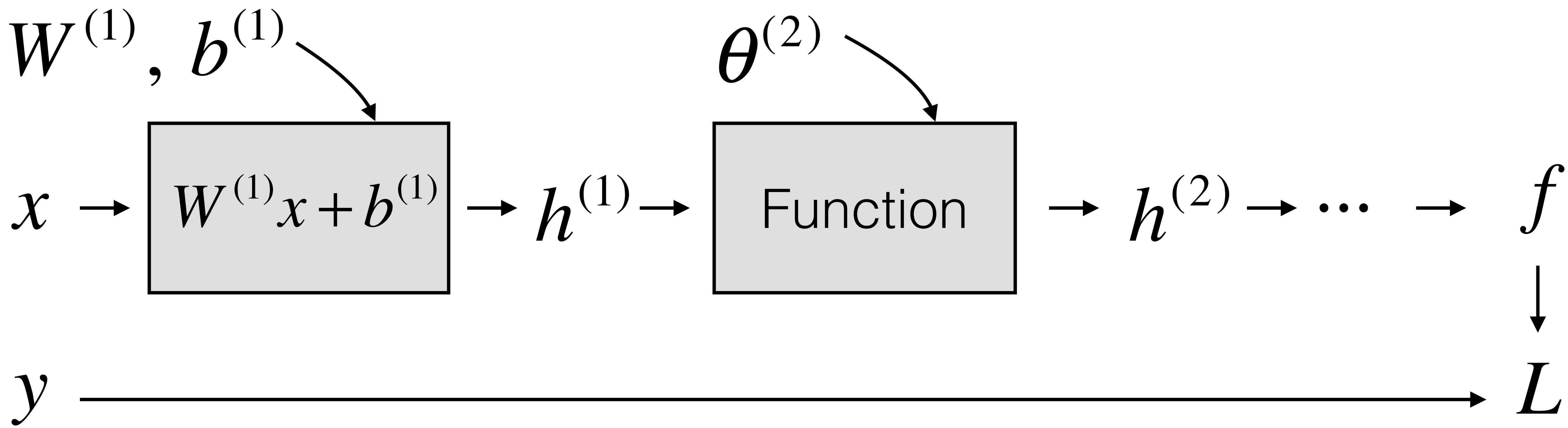


Toy Example:

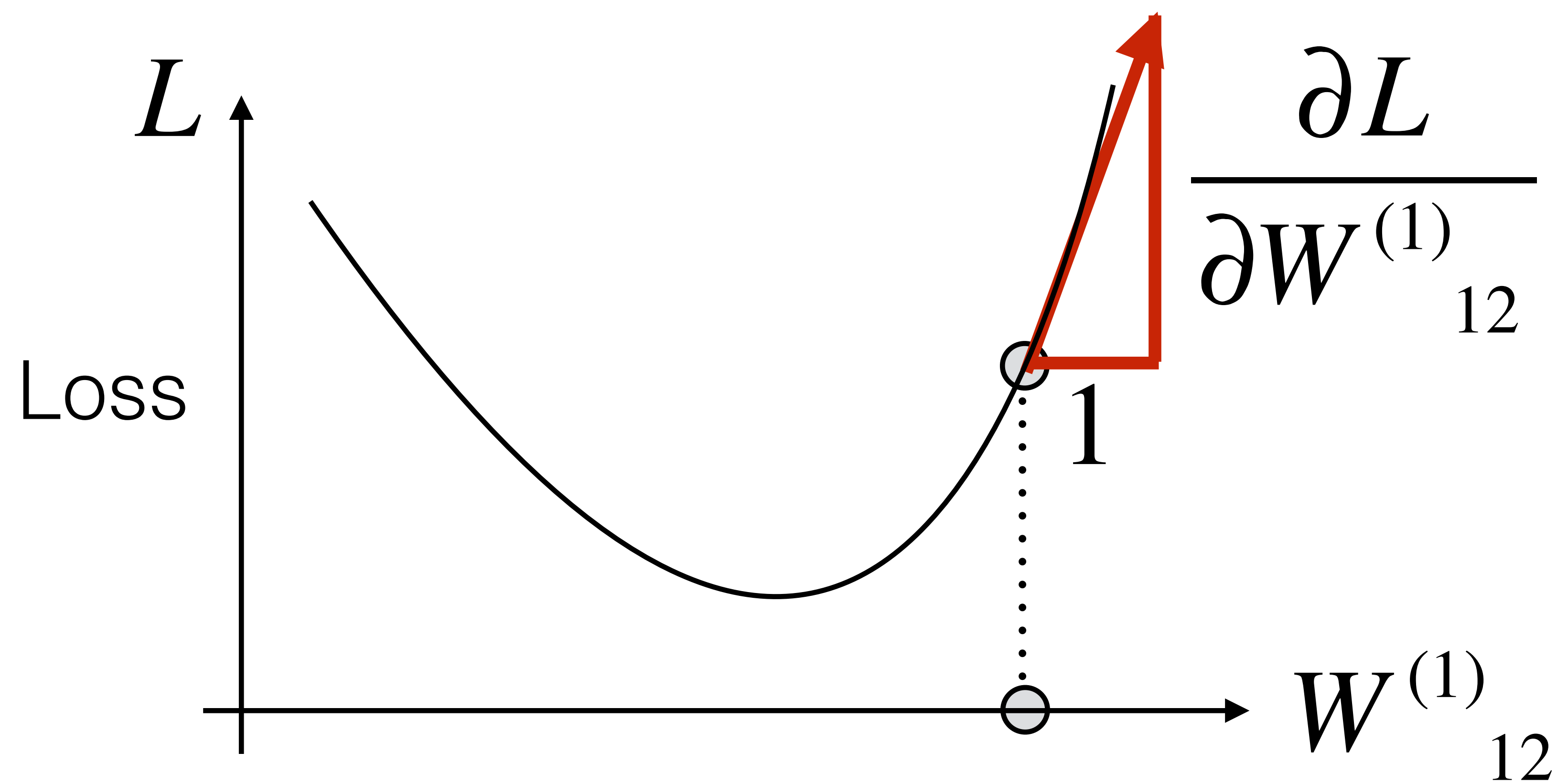


A weight somewhere in the network

Review: Setup

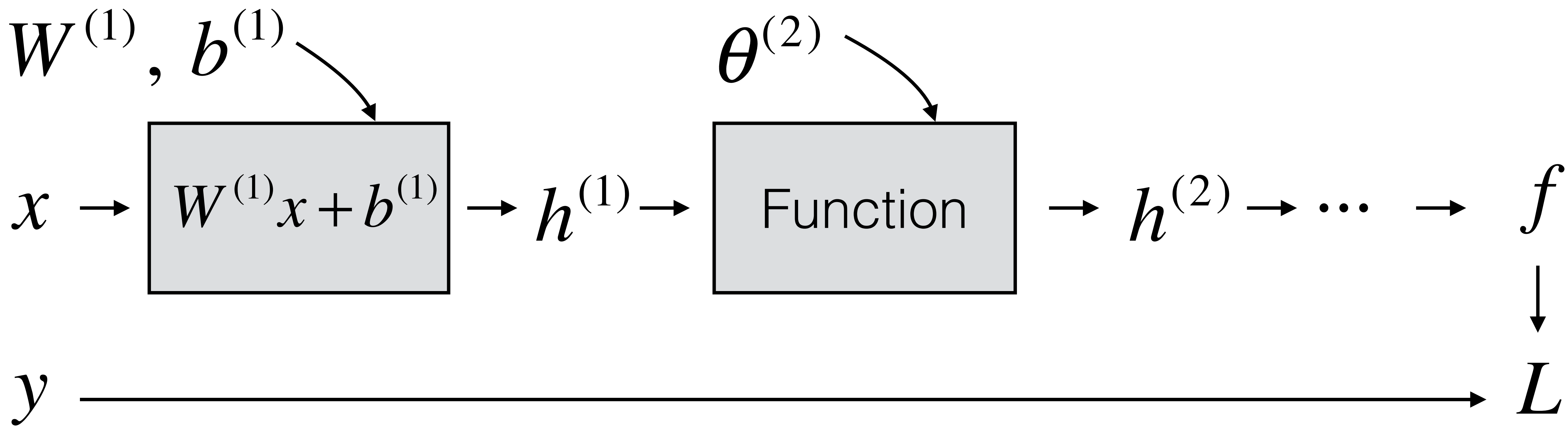


Toy Example:

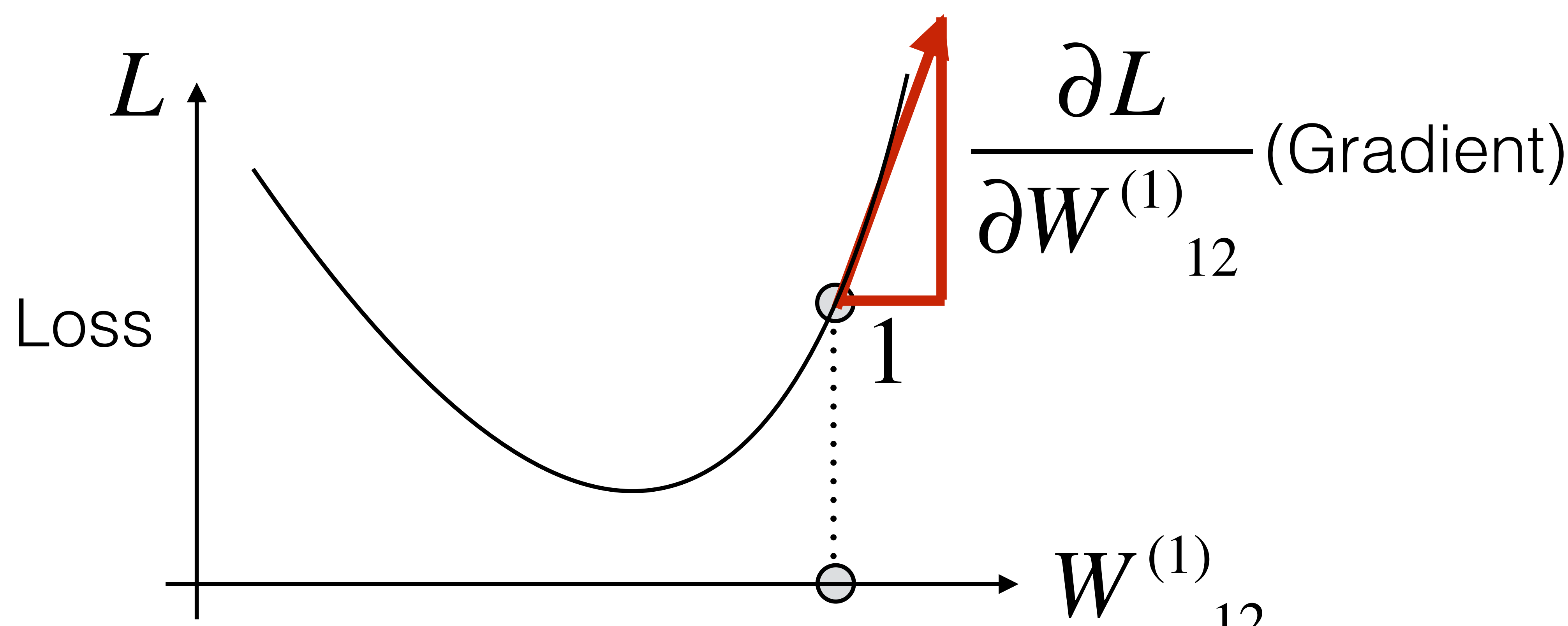


A weight somewhere in the network

Review: Setup

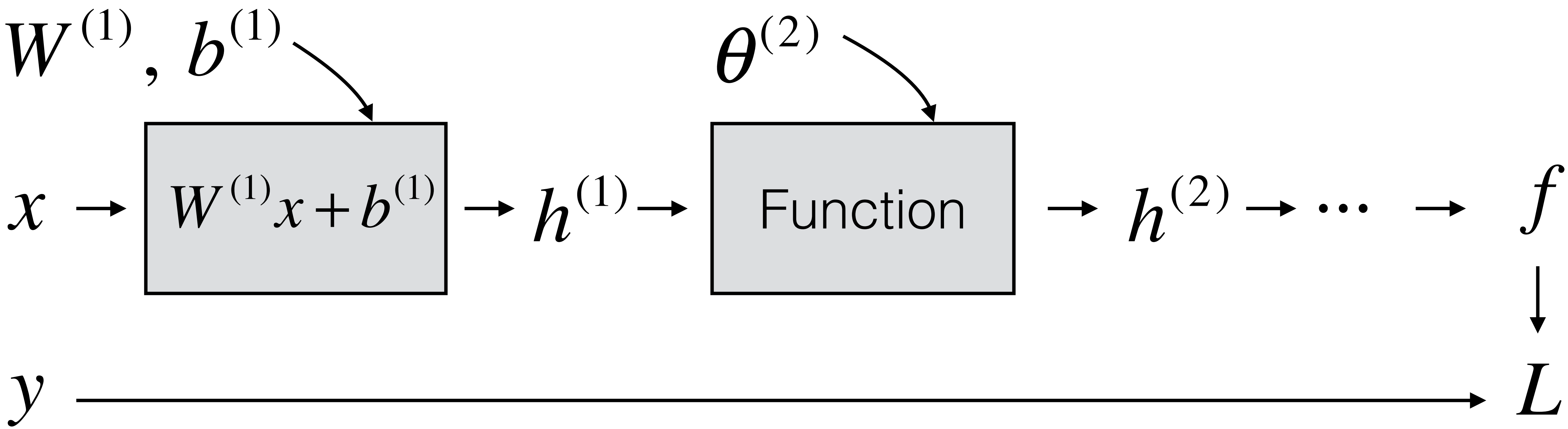


Toy Example:

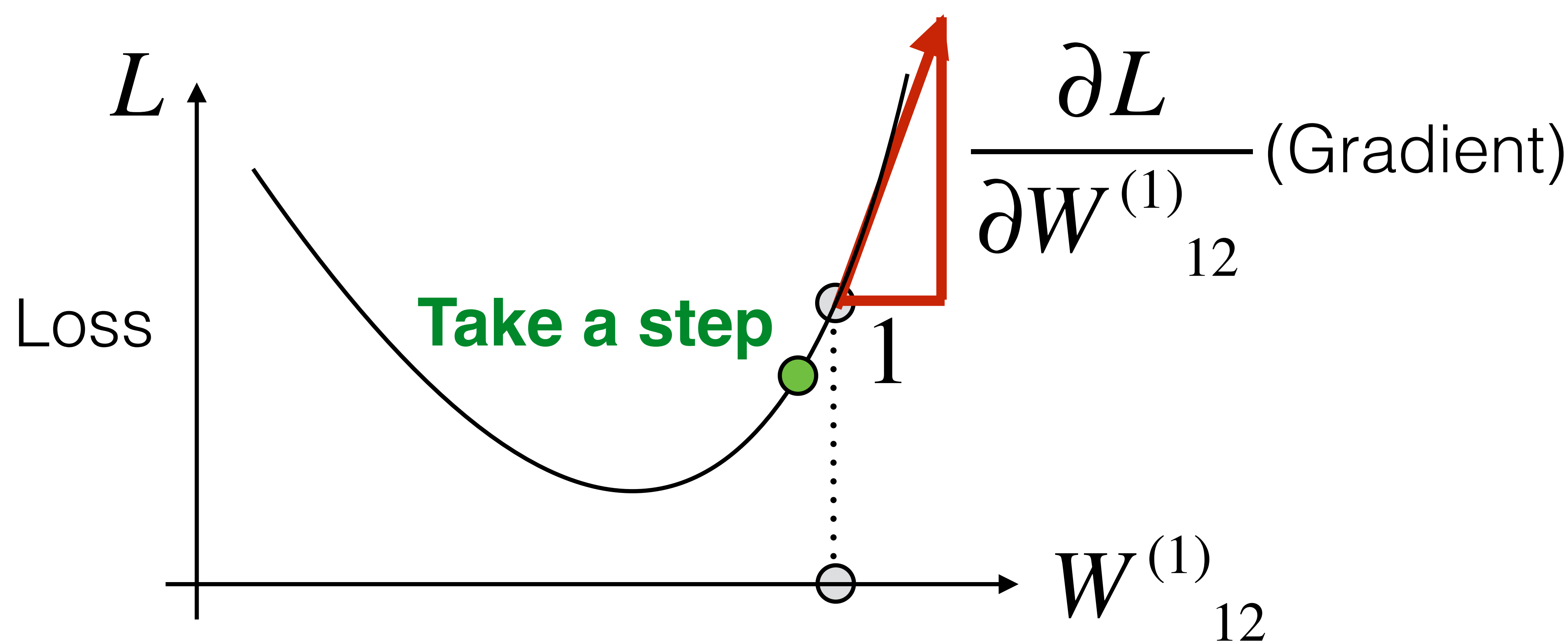


A weight somewhere in the network

Review: Setup

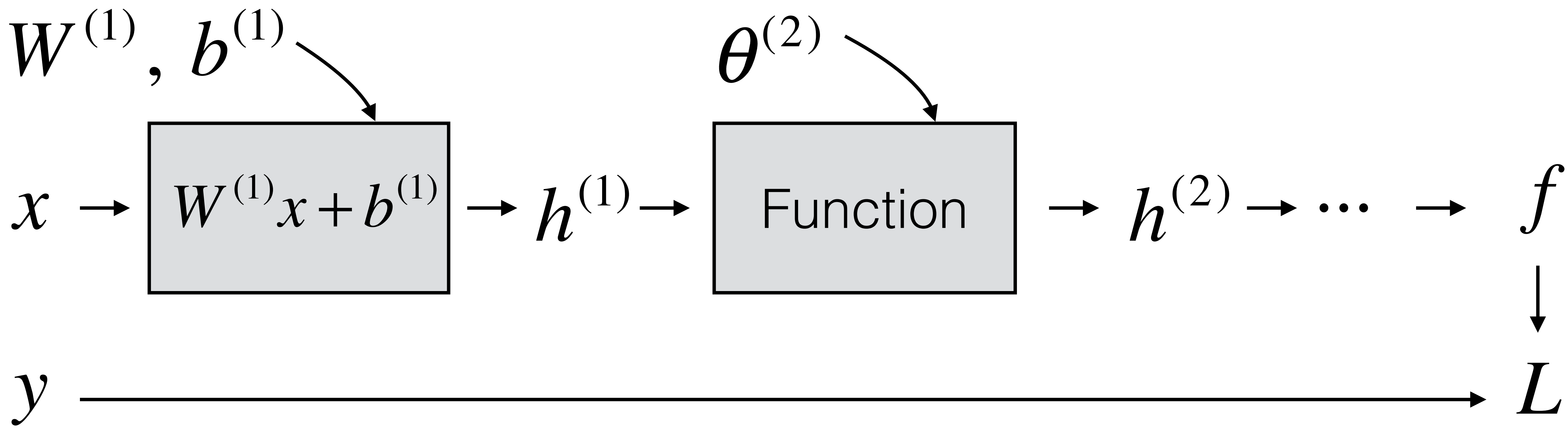


Toy Example:

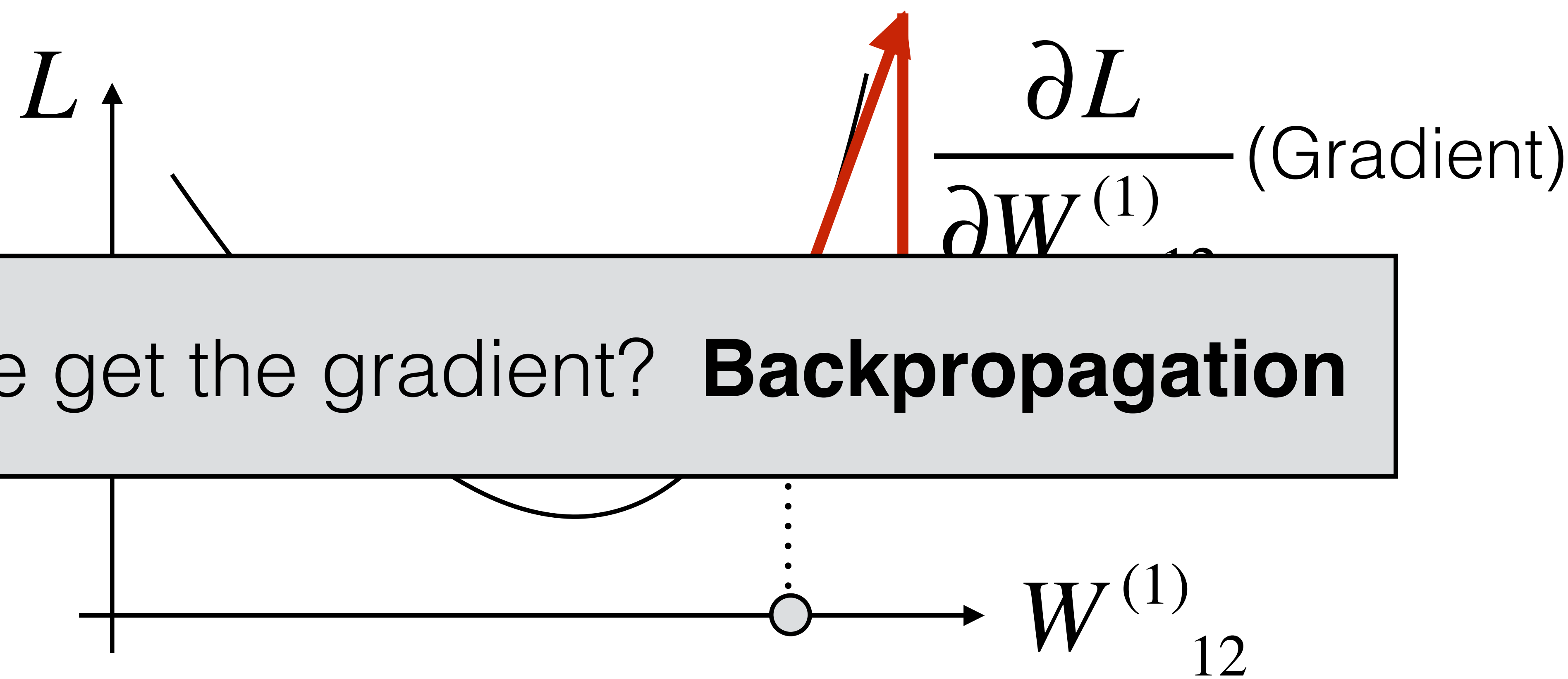


A weight somewhere in the network

Review: Setup



Toy Example:



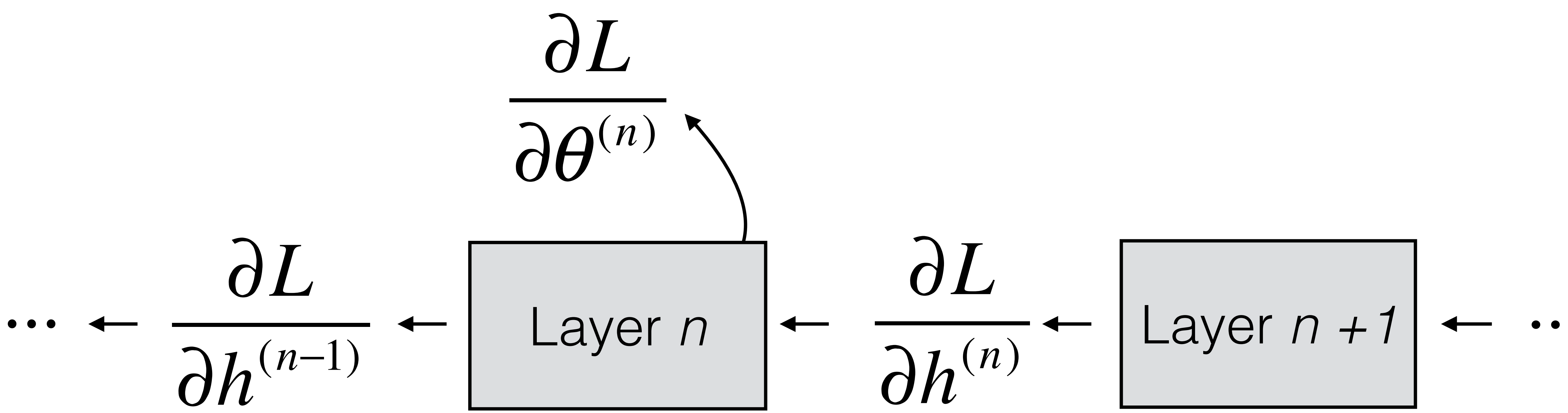
How do we get the gradient? **Backpropagation**

A weight somewhere in the network

Backprop

It’s just the chain rule

Backprop


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

Backprop

This is what we want for each layer

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

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

Backprop

This is what we want for each layer

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

To compute it, we need to propagate this gradient

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

Backprop

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:

Backprop

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

Backprop

This is what we want for each layer

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

To compute it, we need to propagate this gradient

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

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

Backprop

This is what we want for each layer

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

To compute it, we need to propagate this gradient

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

given to us

For each layer:

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

What we want

This is just the local gradient of layer *n*

Backprop

This is what we want for each layer

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

To compute it, we need to propagate this gradient

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

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*

Backprop

This is what we want for each layer

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

To compute it, we need to propagate this gradient

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

given to us

For each layer:

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

What we want

This is just the local gradient of layer *n*

Backprop

This is what we want for each layer

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

To compute it, we need to propagate this gradient

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

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*

Backprop

For each layer, we compute:

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

Backprop

For each layer, we compute:

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

(Can compute immediately)

Backprop

For each layer, we compute:

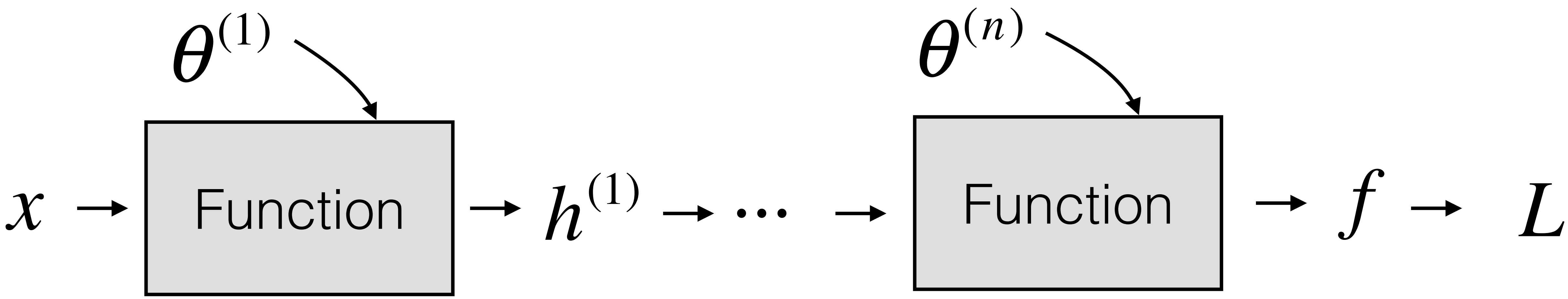
$$[\text{Propagated gradient to the left}] = [\text{Propagated gradient from right}] \cdot [\text{Local gradient}]$$

(Received during backprop)

(Can compute immediately)

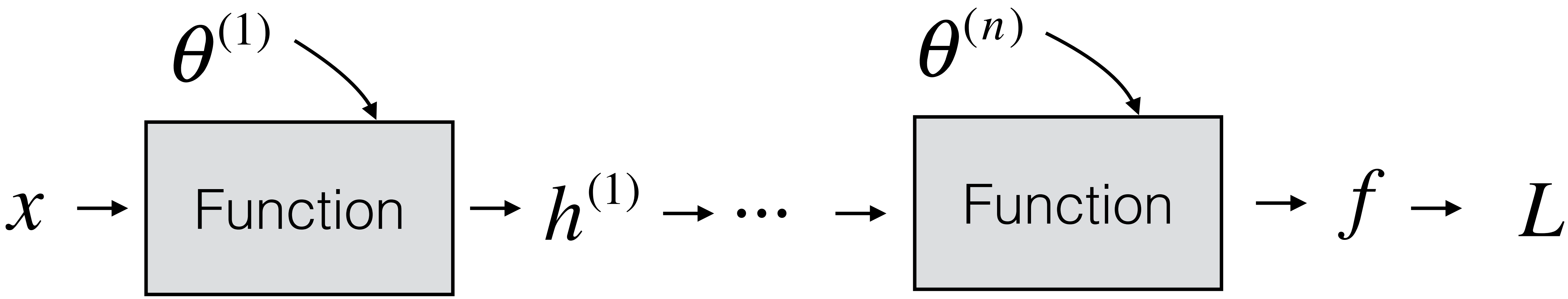
Backprop

Forward Propagation:



Backprop

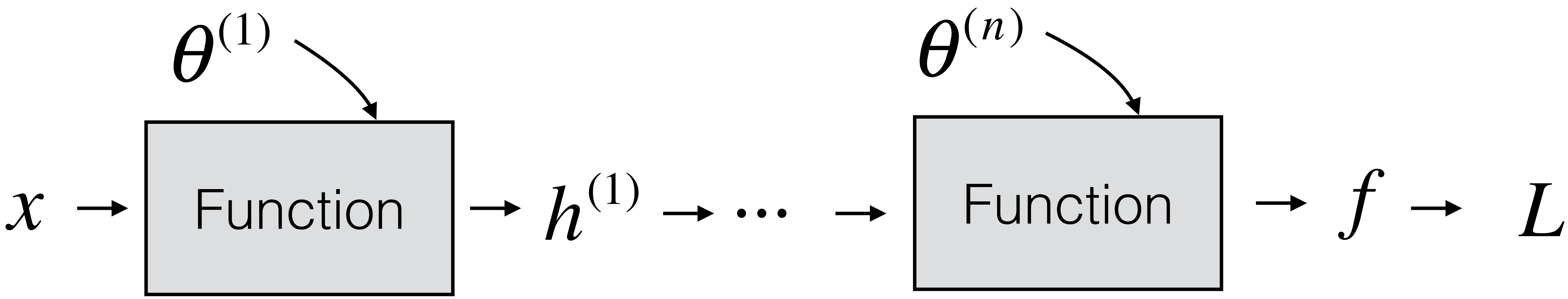
Forward Propagation:



Backward Propagation:

Backprop

Forward Propagation:

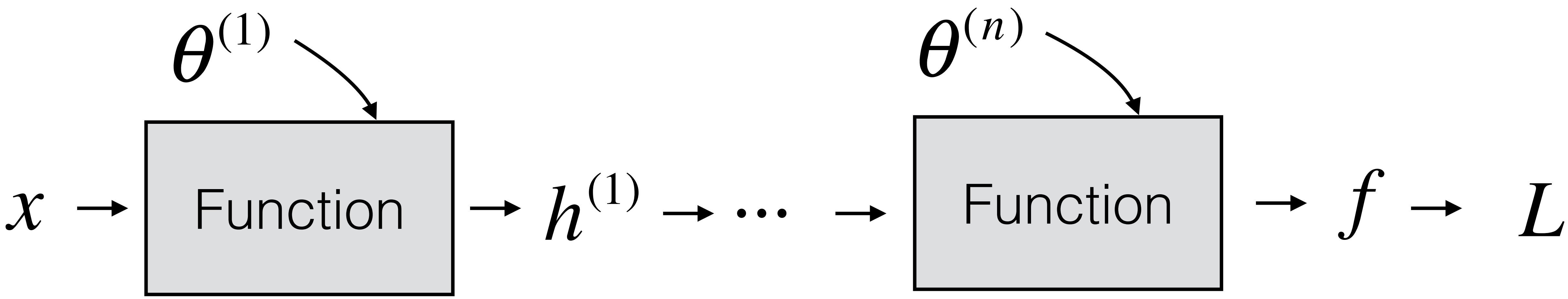


Backward Propagation:

L

Backprop

Forward Propagation:



Backward Propagation:

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

Backprop

Forward Propagation:

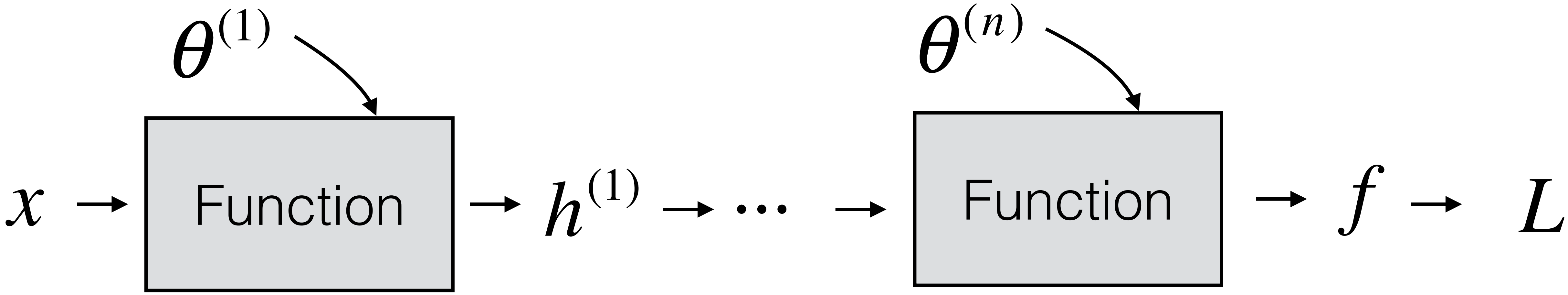
$$\begin{tikzcd} & \theta^{(1)} \arrow[d, bend left] & & & \theta^{(n)} \arrow[d, bend left] & & \\ x \arrow[r] & \boxed{\text{Function}} \arrow[r] & h^{(1)} \arrow[r] & \cdots \arrow[r] & \boxed{\text{Function}} \arrow[r] & f \arrow[r] & L \end{tikzcd}$$

Backward Propagation:

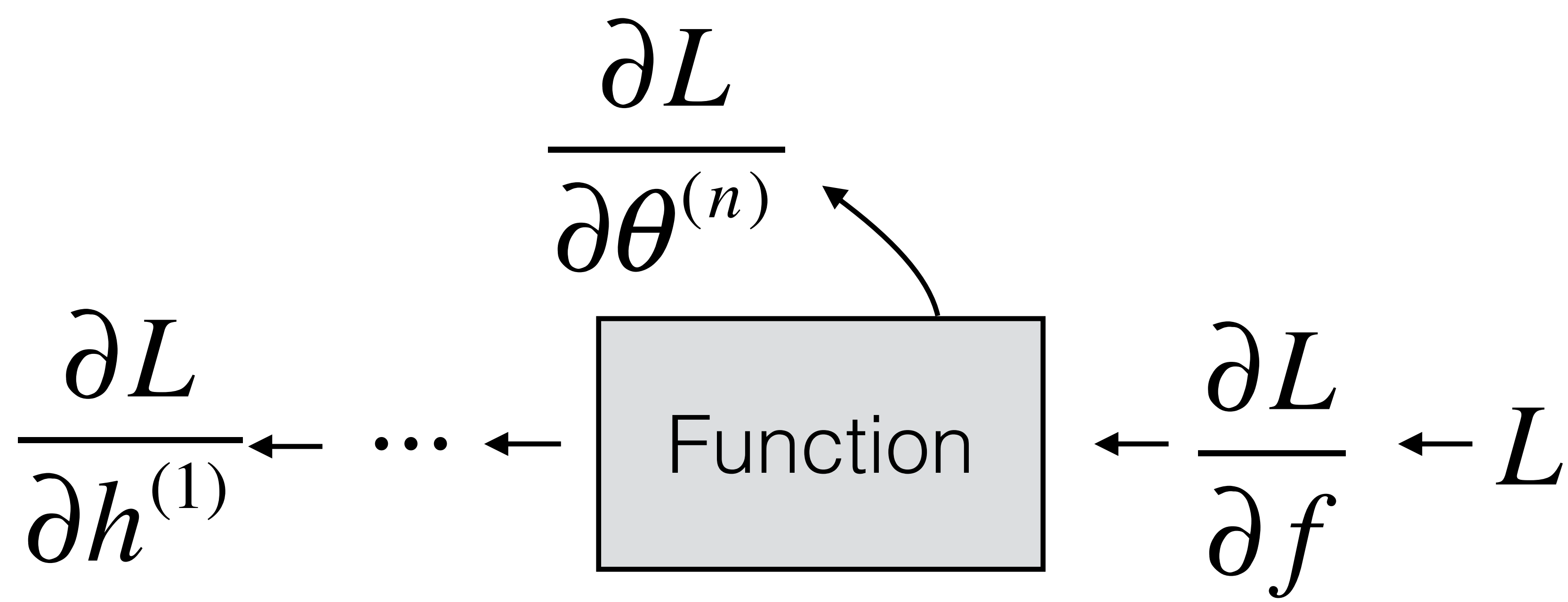
$$\begin{tikzcd} \frac{\partial L}{\partial \theta^{(n)}} & & \\ \boxed{\text{Function}} \arrow[u, bend right] & \frac{\partial L}{\partial f} \arrow[l] & L \arrow[l] \end{tikzcd}$$

Backprop

Forward Propagation:



Backward Propagation:



Backprop

Forward Propagation:

$$\begin{tikzcd} & \theta^{(1)} \arrow[d, bend left] & & & \theta^{(n)} \arrow[d, bend left] & & \\ x \arrow[r] & \boxed{\text{Function}} \arrow[r] & h^{(1)} \arrow[r] & \cdots \arrow[r] & \boxed{\text{Function}} \arrow[r] & f \arrow[r] & L \end{tikzcd}$$

Backward Propagation:

$$\begin{tikzcd} & \dfrac{\partial L}{\partial \theta^{(1)}} & & & \dfrac{\partial L}{\partial \theta^{(n)}} & & \\ \dfrac{\partial L}{\partial x} & \boxed{\text{Function}} \arrow[l] \arrow[u, bend right] & \dfrac{\partial L}{\partial h^{(1)}} \arrow[l] & \cdots \arrow[l] & \boxed{\text{Function}} \arrow[l] \arrow[u, bend right] & \dfrac{\partial L}{\partial f} \arrow[l] & L \arrow[l] \end{tikzcd}$$

Backprop

It’s easy to write down the chain rule for higher dimensions — *just add more subscripts and more summations*

Backprop

It's easy to write down the chain rule for higher 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

It's easy to write down the chain rule for higher 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

It's easy to write down the chain rule for higher 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

It's easy to write down the chain rule for higher 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

Example: Mean Subtraction

(for a single input)

Example: Mean Subtraction
(for a single input)

- Ok, so how do we actually derive the backwards pass? Let's walk through an example together.

Example: Mean Subtraction
(for a single input)

- Ok, so how do we actually derive the backwards pass? Let's walk through an example together.

- Example layer: mean subtraction:

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

Example: Mean Subtraction
(for a single input)

- Ok, so how do we actually derive the backwards pass? Let's walk through an example together.

- 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)

- Ok, so how do we actually derive the backwards pass? Let's walk through an example together.

- Example layer: mean subtraction:

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

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

- For backprop, we just need the local derivative

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: Mean Subtraction
(for a single input)

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

Let's code this up in NumPy:

Example: Mean Subtraction
(for a single input)

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

Let's code this up in NumPy:

```
def forward(X):
    return X - np.mean(X, axis=1)
```

Example: Mean Subtraction

(for a single input)

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

Let's code this up in NumPy:

```
def forward(X):
    return X - np.mean(X, axis=1)
```

Dimension mismatch

Example: Mean Subtraction
(for a single input)

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

Let's code this up in NumPy:

```
def forward(X):
    return X - np.mean(X, axis=1)
```

Dimension mismatch

You need to broadcast properly:

```
def forward(X):
    return X - np.mean(X, axis=1)[:, np.newaxis]
```

Example: Mean Subtraction

(for a single input)

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

Let's code this up in NumPy:

```
def forward(X):
    return X - np.mean(X, axis=1)
```

Dimension mismatch

You need to broadcast properly:

```
def forward(X):
    return X - np.mean(X, axis=1)[:, np.newaxis]
```

This also works:

```
def forward(X):
    return X - np.mean(X, axis=1, keepdims=True)
```

Example: Mean Subtraction
(for a single input)

The backward pass is easy:

```
def backward(dh):
    return forward(dh)
```

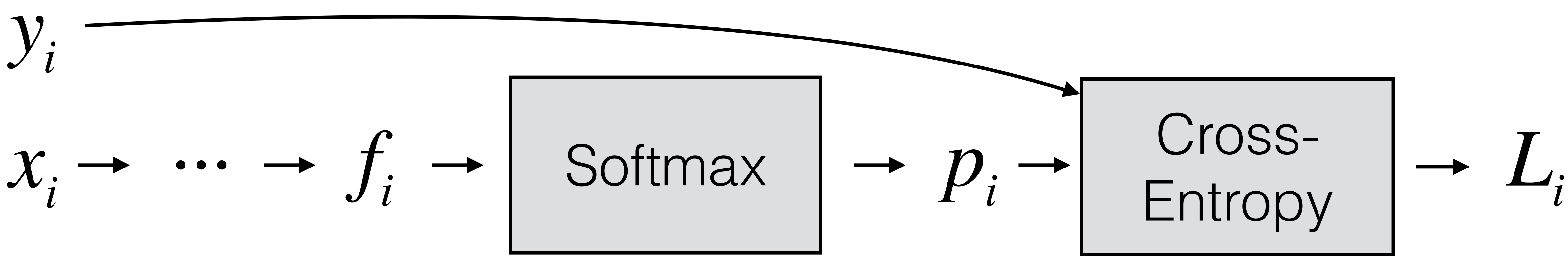
(Remember they're usually not the same)

Example: Softmax (for N inputs)

Let's assume we are using Softmax and Cross-entropy loss
(together this is often called "Softmax loss")

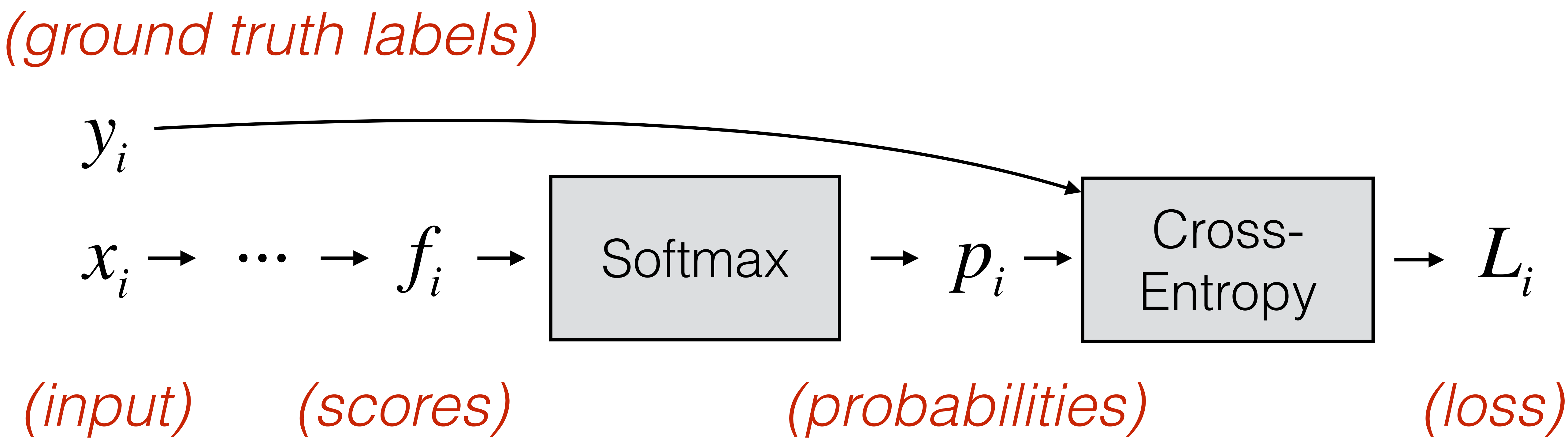
Example: Softmax (for N inputs)

Let's assume we are using Softmax and Cross-entropy loss
(together this is often called "Softmax loss")



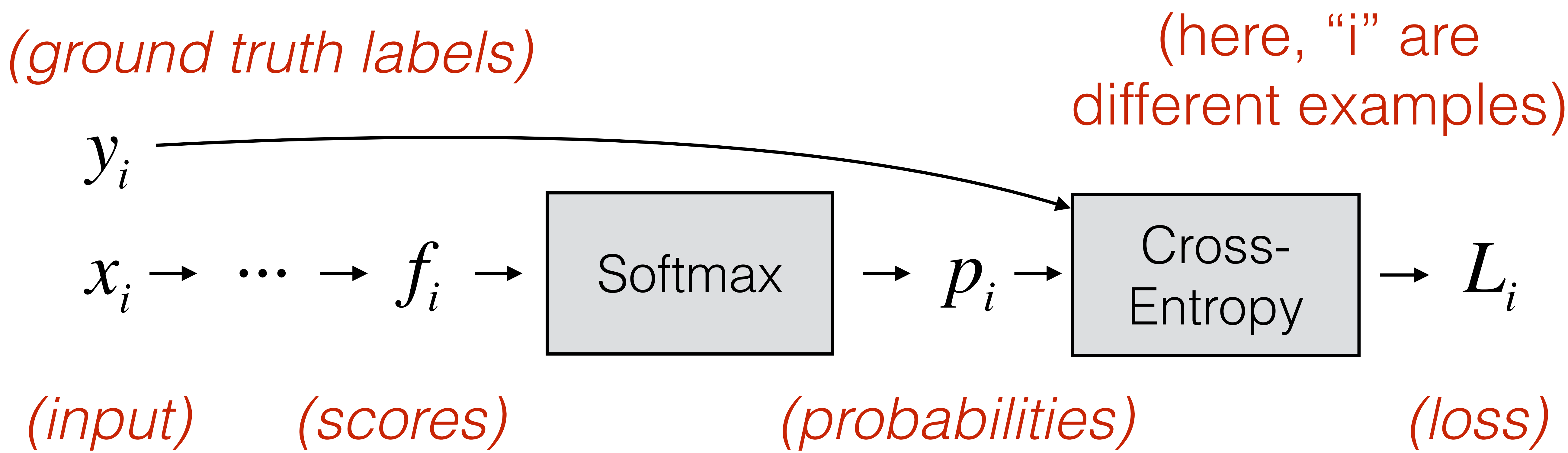
Example: Softmax (for N inputs)

Let's assume we are using Softmax and Cross-entropy loss
(together this is often called "Softmax loss")



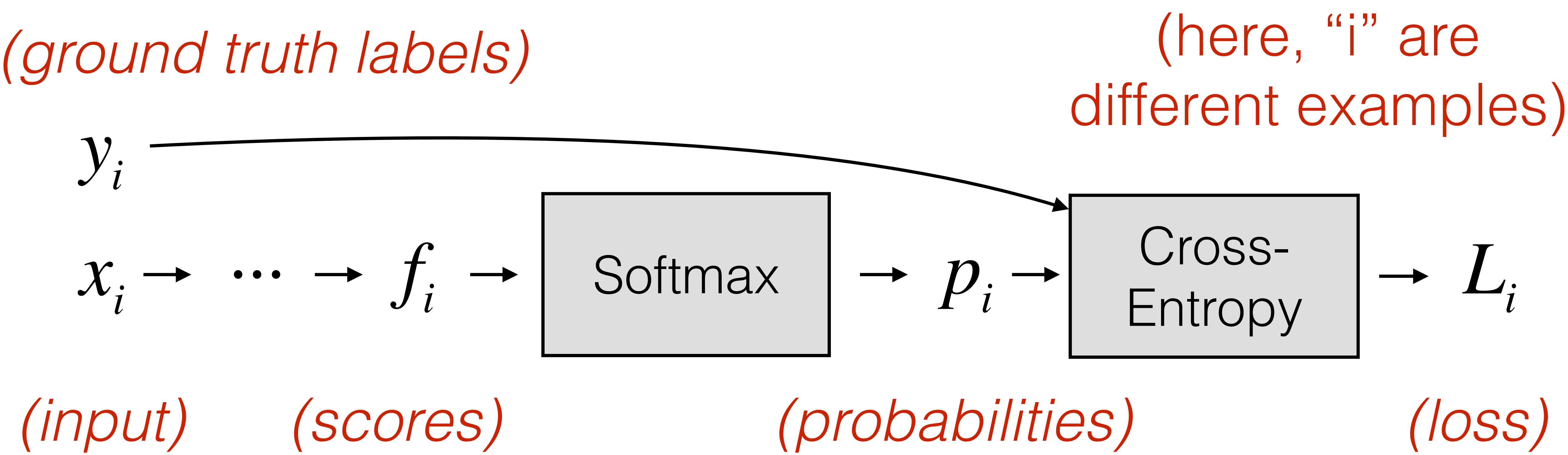
Example: Softmax (for N inputs)

Let's assume we are using Softmax and Cross-entropy loss
(together this is often called "Softmax loss")



Example: Softmax (for N inputs)

Let's assume we are using Softmax and Cross-entropy loss
(together this is often called "Softmax loss")

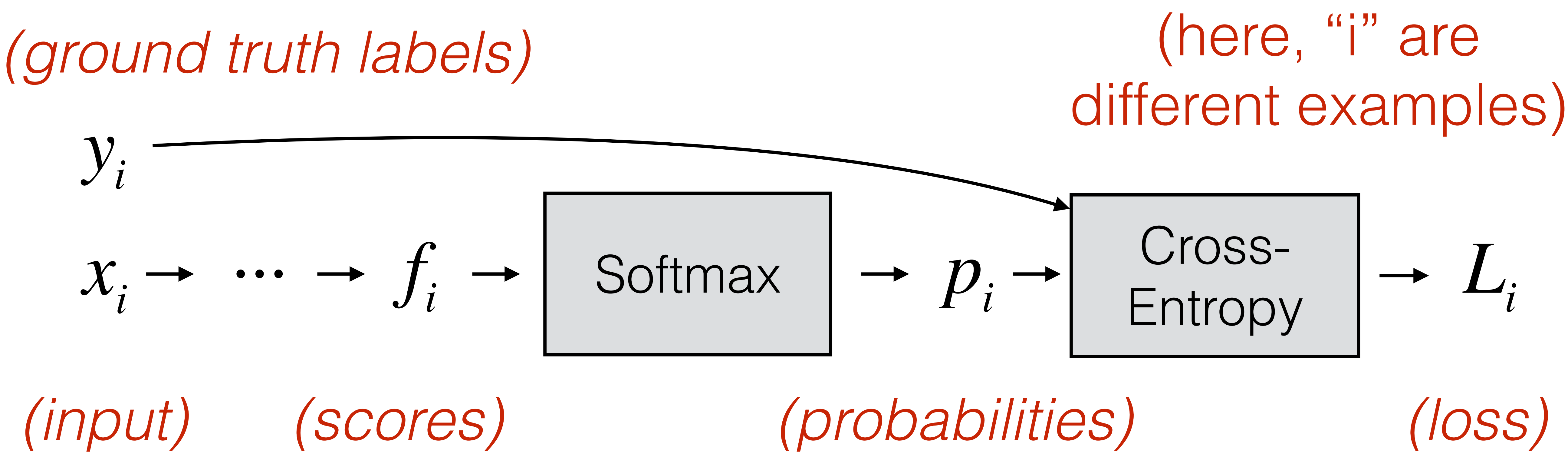


$$p_{i,j} = \frac{e^{f_{i,j}}}{\sum_k e^{f_{i,k}}}$$

(Softmax)

Example: Softmax (for N inputs)

Let's assume we are using Softmax and Cross-entropy loss (together this is often called "Softmax loss")



$$p_{i,j} = \frac{e^{f_{i,j}}}{\sum_k e^{f_{i,k}}}$$

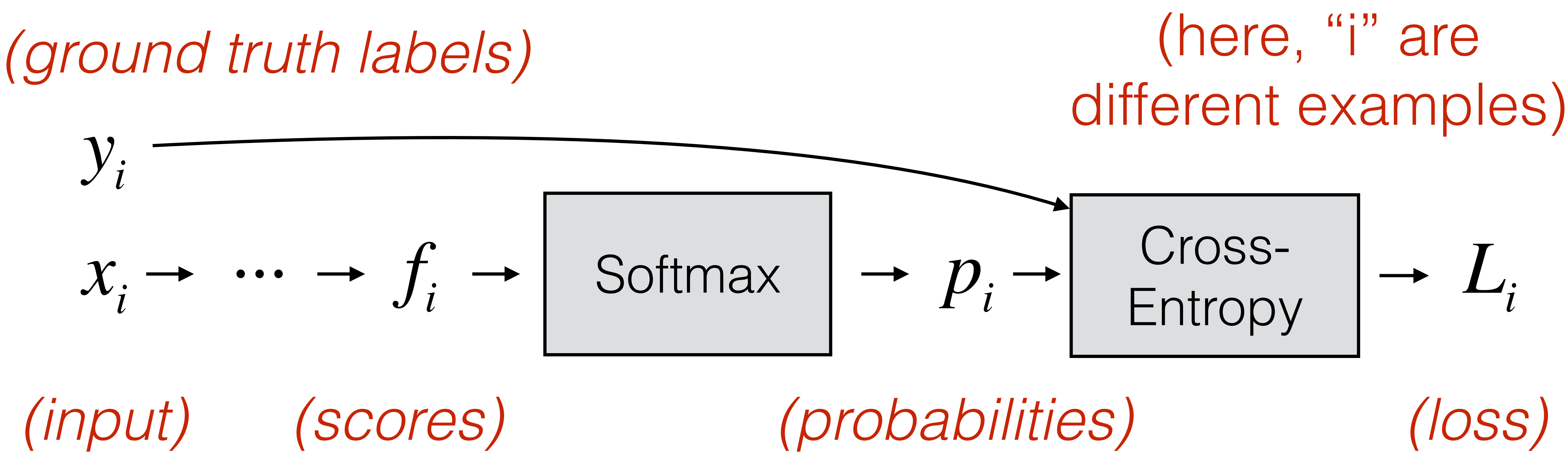
(Softmax)

$$L_i = -\log p_{i,y_i}$$

(Cross-entropy)

Example: Softmax (for N inputs)

Let's assume we are using Softmax and Cross-entropy loss
(together this is often called "Softmax loss")



$$p_{i,j} = \frac{e^{f_{i,j}}}{\sum_k e^{f_{i,k}}}$$

(Softmax)

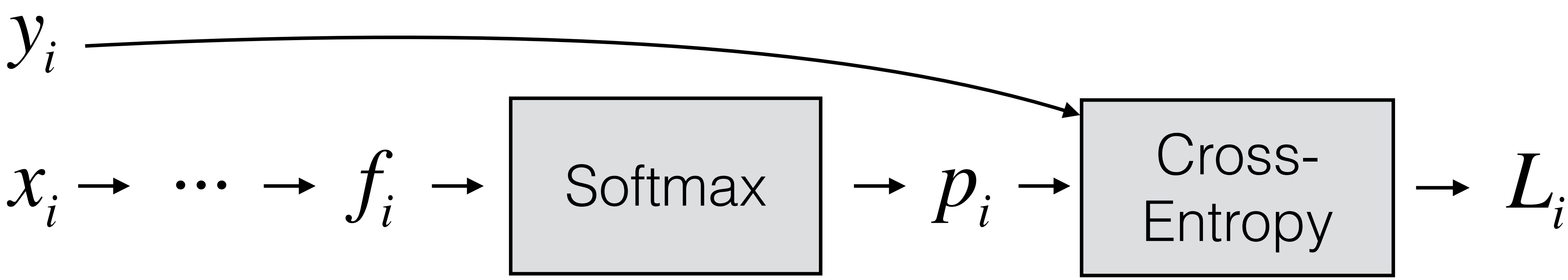
$$L_i = -\log p_{i,y_i}$$

(Cross-entropy)

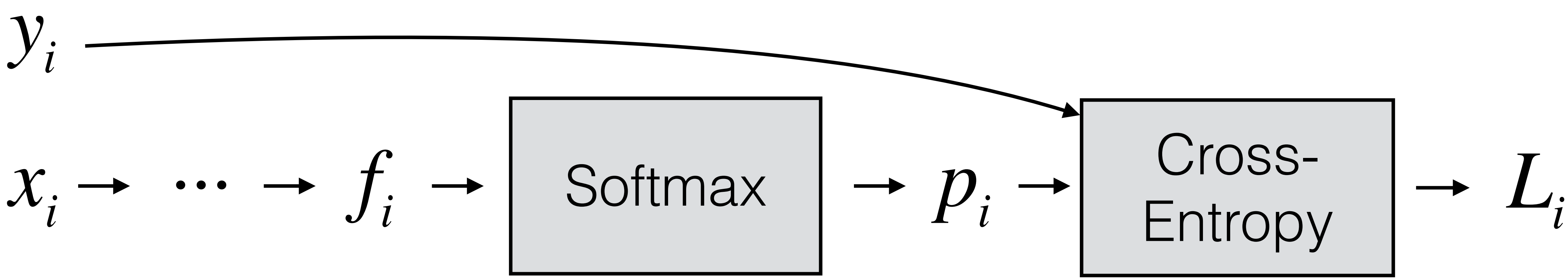
$$L = \frac{1}{N} \sum_i L_i$$

(Avg. over examples)

Example: Softmax (for N inputs)

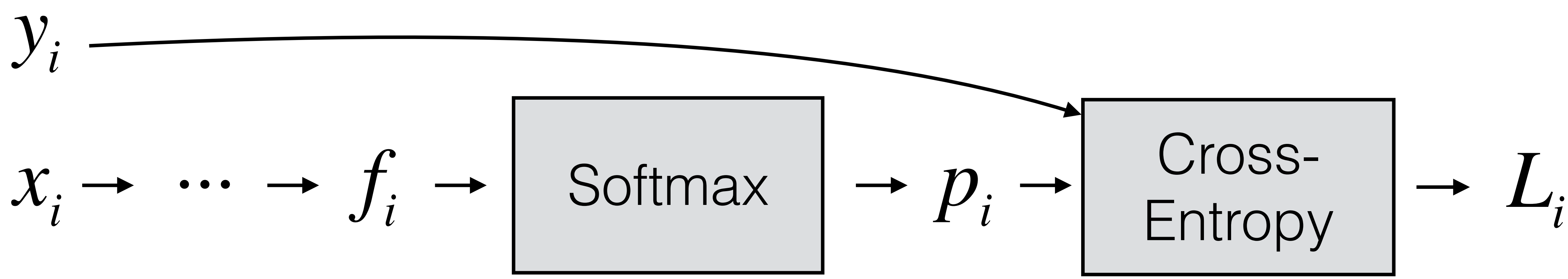


Example: Softmax (for N inputs)



Derivative: $$\frac{\partial L}{\partial f_{i,j}} = \frac{p_{i,j} - t_{i,j}}{N}$$

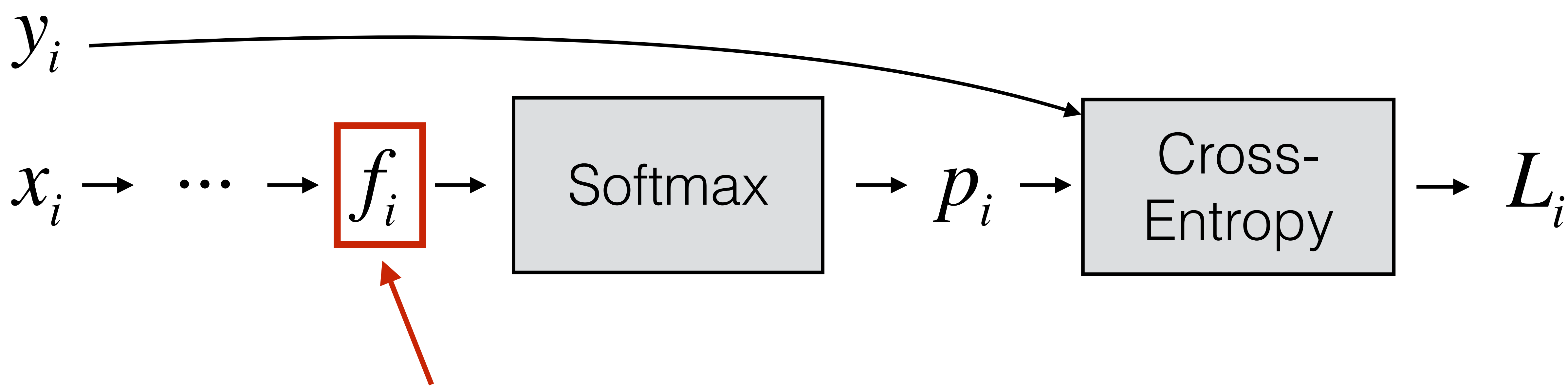
Example: Softmax (for N inputs)



Derivative: $$\frac{\partial L}{\partial f_{i,j}} = \frac{p_{i,j} - t_{i,j}}{N}$$ where $t_i = [0 \; \ldots \; 1 \; \ldots \; 0]$

(Entry y_i set to 1)

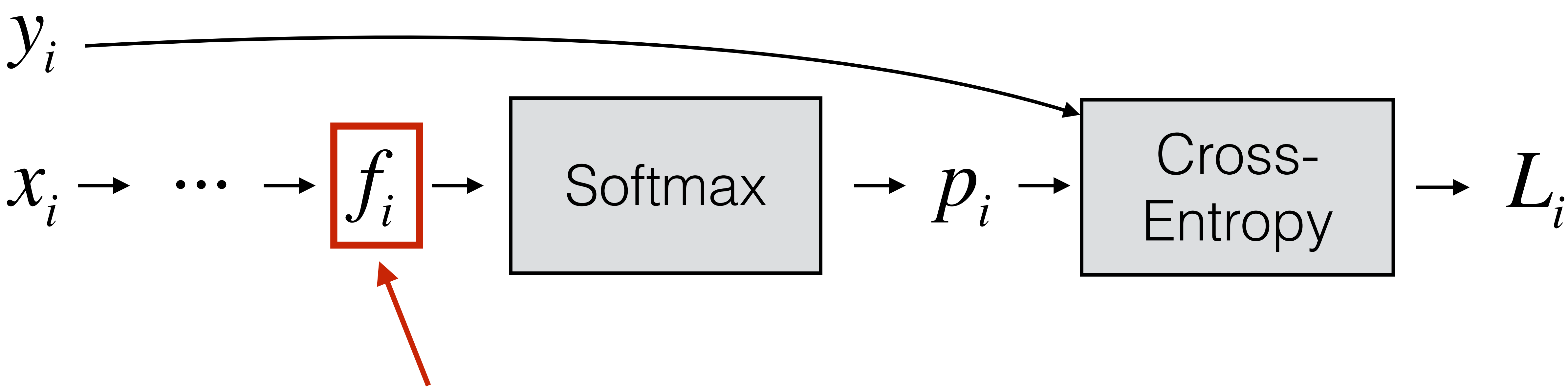
Example: Softmax (for N inputs)



Derivative: $$\boxed{\frac{\partial L}{\partial f_{i,j}}} = \frac{p_{i,j} - t_{i,j}}{N}$$ where $t_i = [0 \ \ldots \ 1 \ \ldots \ 0]$

(Entry y_i set to 1)

Example: Softmax (for N inputs)

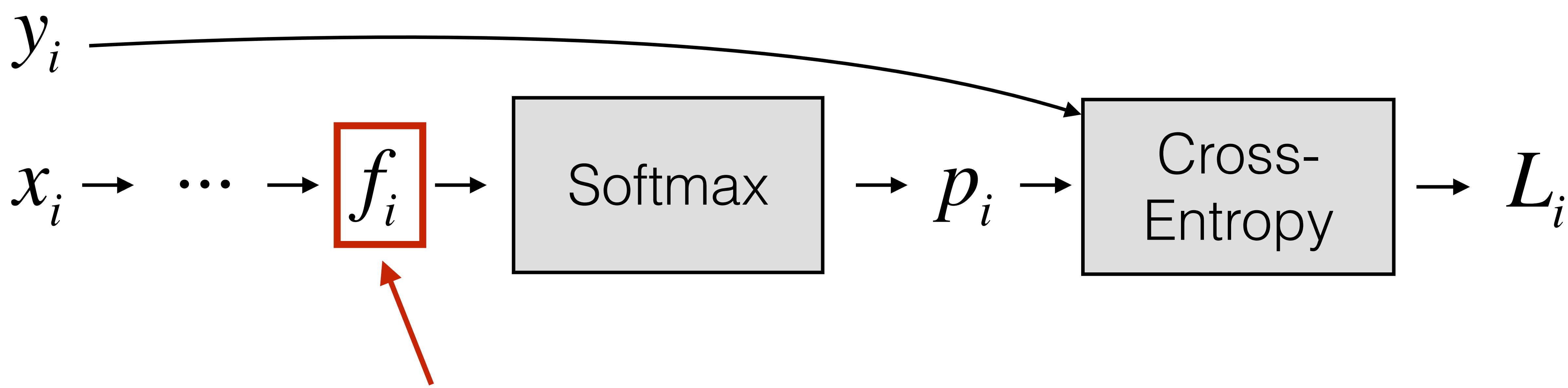


Derivative: $\boxed{\dfrac{\partial L}{\partial f_{i,j}}} = \dfrac{p_{i,j} - t_{i,j}}{N}$ where $t_i = [0 \;\ldots\; 1 \;\ldots\; 0]$

(Entry y_i set to 1)

(Try deriving this — it's tricky but not too hard)

Example: Softmax (for N inputs)



Derivative: $\boxed{\dfrac{\partial L}{\partial f_{i,j}}} = \dfrac{p_{i,j} - t_{i,j}}{N}$ where $t_i = [0 \;\ldots\; 1 \;\ldots\; 0]$

(Entry y_i set to 1)

(Try deriving this — it's tricky but not too hard)

Now we can continue backpropagating to the layer before "f"

Example: Softmax (for N inputs)

$$p_{i,j} = \frac{e^{f_{i,j}}}{\sum_k e^{f_{i,k}}}$$

Let's code this up in NumPy:

```
def softmax(f):
    exp_f = np.exp(f)
    return exp_f / np.sum(exp_f, axis=1, keepdims=True)
```

Example: Softmax (for N inputs)

$$p_{i,j} = \frac{e^{f_{i,j}}}{\sum_k e^{f_{i,k}}}$$

Let's code this up in NumPy:

```
def softmax(f):
    exp_f = np.exp(f)
    return exp_f / np.sum(exp_f, axis=1, keepdims=True)
```

Doesn't work — what's the problem this time?

Example: Softmax (for N inputs)

$$p_{i,j} = \frac{e^{f_{i,j}}}{\sum_k e^{f_{i,k}}}$$

Let's code this up in NumPy:

```
def softmax(f):
    exp_f = np.exp(f)
    return exp_f / np.sum(exp_f, axis=1, keepdims=True)
```

Doesn't work — what's the problem this time?

- What if there is the value 1000 appears in "f"?

Example: Softmax (for N inputs)

$$p_{i,j} = \frac{e^{f_{i,j}}}{\sum_k e^{f_{i,k}}}$$

Let's code this up in NumPy:

```
def softmax(f):
    exp_f = np.exp(f)
    return exp_f / np.sum(exp_f, axis=1, keepdims=True)
```

Doesn't work — what's the problem this time?

- What if there is the value 1000 appears in "f"?

Overflow —> we get inf/inf = NaN

Example: Softmax (for N inputs)

$$p_{i,j} = \frac{e^{f_{i,j}}}{\sum_k e^{f_{i,k}}}$$

Let's code this up in NumPy:

```
def softmax(f):
    exp_f = np.exp(f)
    return exp_f / np.sum(exp_f, axis=1, keepdims=True)
```

Doesn't work — what's the problem this time?

- What if there is the value 1000 appears in "f"?

  Overflow —> we get inf/inf = NaN

- What if the largest value is -1000?

Example: Softmax (for N inputs)

$$p_{i,j} = \frac{e^{f_{i,j}}}{\sum_k e^{f_{i,k}}}$$

Let's code this up in NumPy:

```
def softmax(f):
    exp_f = np.exp(f)
    return exp_f / np.sum(exp_f, axis=1, keepdims=True)
```

Doesn't work — what's the problem this time?

- What if there is the value 1000 appears in "f"?

  Overflow —> we get inf/inf = NaN

- What if the largest value is -1000?

  Underflow —> we get 0/0 = NaN

Example: Softmax (for N inputs)

$$p_{i,j} = \frac{e^{f_{i,j}}}{\sum_k e^{f_{i,k}}}$$

Let's code this up in NumPy:

```
def softmax(f):
    exp_f = np.exp(f)
    return exp_f / np.sum(exp_f, axis=1, keepdims=True)
```

Doesn't work — what's the problem this time?

- What if there is the value 1000 appears in "f"?

  Overflow —> we get inf/inf = NaN

- What if the largest value is -1000?

  Underflow —> we get 0/0 = NaN

This expression is numerically unstable

Example: Softmax (for N inputs)

Example: Softmax (for N inputs)

Observation: subtracting a constant does not change “p”:

Example: Softmax (for N inputs)

Observation: subtracting a constant does not change "p":

$$p_{i,j} = \frac{e^{f_{i,j}-C}}{\sum_k e^{f_{j,k}-C}} = \frac{e^{-C}e^{f_{i,j}}}{\sum_k e^{-C}e^{f_{i,k}}} = \frac{e^{f_{i,j}}}{\sum_k e^{f_{i,k}}}$$

Example: Softmax (for N inputs)

Observation: subtracting a constant does not change "p":

$$p_{i,j} = \frac{e^{f_{i,j}-C}}{\sum_k e^{f_{j,k}-C}} = \frac{e^{-C} e^{f_{i,j}}}{\sum_k e^{-C} e^{f_{i,k}}} = \frac{e^{f_{i,j}}}{\sum_k e^{f_{i,k}}}$$

If we choose "C" to be the max, then it works:

Example: Softmax (for N inputs)

Observation: subtracting a constant does not change "p":

$$p_{i,j} = \frac{e^{f_{i,j}-C}}{\sum_k e^{f_{j,k}-C}} = \frac{e^{-C}e^{f_{i,j}}}{\sum_k e^{-C}e^{f_{i,k}}} = \frac{e^{f_{i,j}}}{\sum_k e^{f_{i,k}}}$$

If we choose "C" to be the max, then it works:

- If a large value appears in "f", then that value will become 1 and all others will be 0 (avoiding overflow)

Example: Softmax (for N inputs)

Observation: subtracting a constant does not change "p":

$$p_{i,j} = \frac{e^{f_{i,j}-C}}{\sum_k e^{f_{j,k}-C}} = \frac{e^{-C}e^{f_{i,j}}}{\sum_k e^{-C}e^{f_{i,k}}} = \frac{e^{f_{i,j}}}{\sum_k e^{f_{i,k}}}$$

If we choose "C" to be the max, then it works:

- If a large value appears in "f", then that value will become 1 and all others will be 0 (avoiding overflow)
- If all values in "f" are large negative, then they will be shifted up towards 0 (avoiding underflow)

Example: Softmax (for N inputs)

Observation: subtracting a constant does not change "p":

$$p_{i,j} = \frac{e^{f_{i,j}-C}}{\sum_k e^{f_{j,k}-C}} = \frac{e^{-C}e^{f_{i,j}}}{\sum_k e^{-C}e^{f_{i,k}}} = \frac{e^{f_{i,j}}}{\sum_k e^{f_{i,k}}}$$

If we choose "C" to be the max, then it works:

- If a large value appears in "f", then that value will become 1 and all others will be 0 (avoiding overflow)
- If all values in "f" are large negative, then they will be shifted up towards 0 (avoiding underflow)

```
def softmax(f):
    exp_f = np.exp(f - np.max(f, axis=1, keepdims=True))
    return exp_f / np.sum(exp_f, axis=1, keepdims=True)
```

What about the weights?

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

What about the weights?

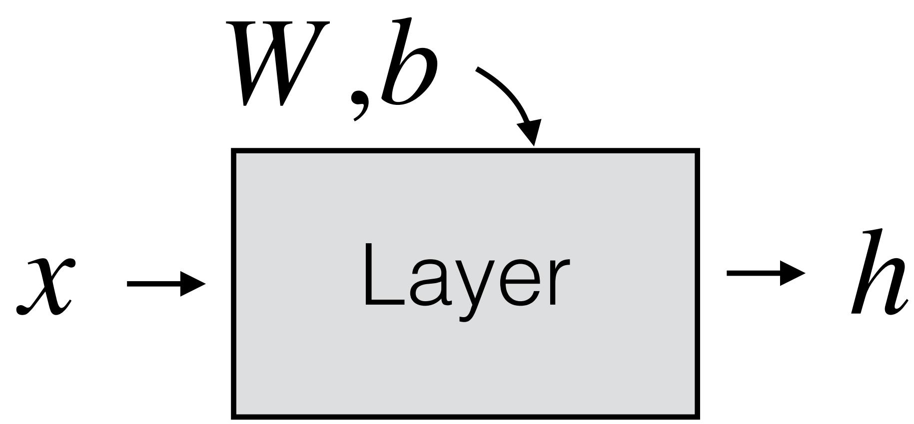
To get the derivative of the weights, use the chain rule again!

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:

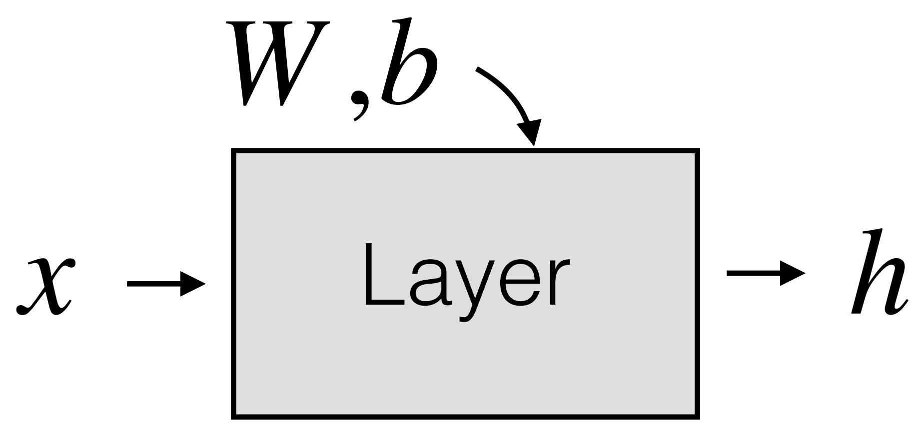


$$h = h(x; W)$$

What about the weights?

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

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



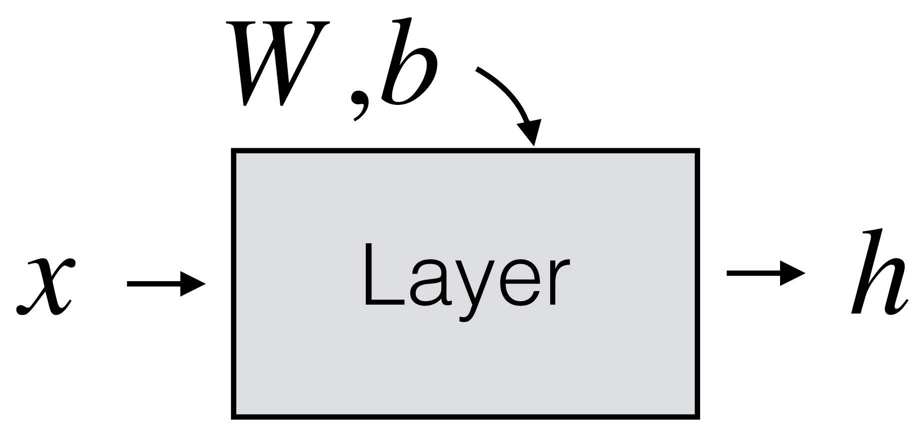
$$h = h(x;W)$$

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

What about the weights?

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

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



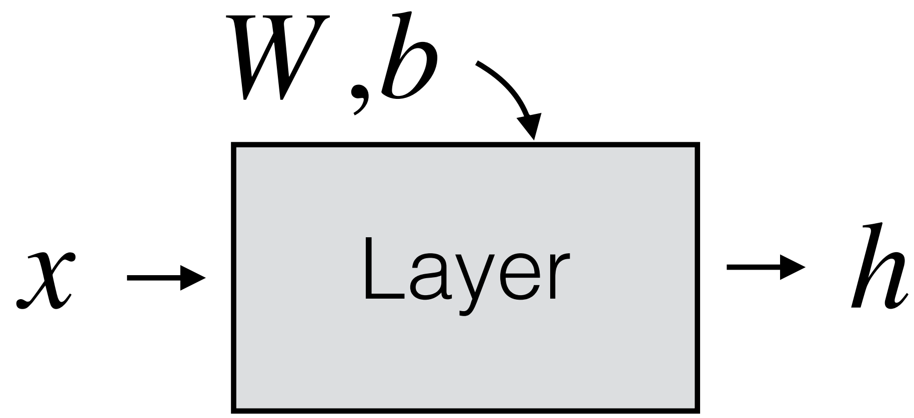
$$h = h(x;W)$$

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

What about the weights?

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

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



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

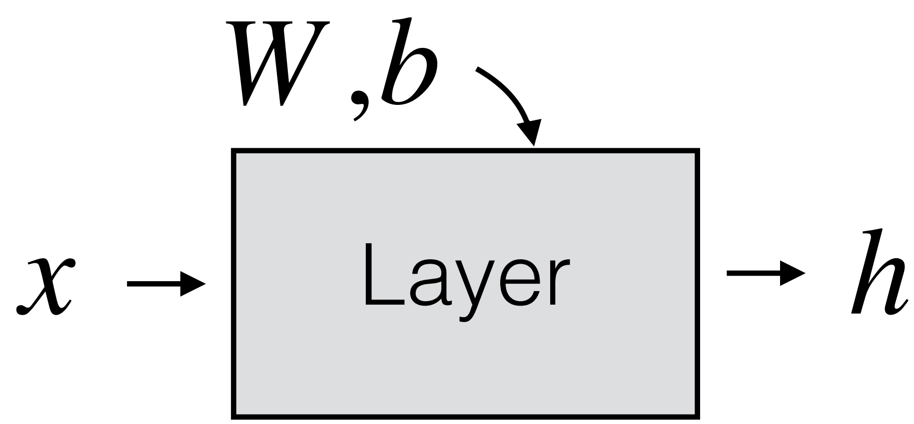
$$\frac{\partial L}{\partial W_{ij}} = \sum_k \frac{\partial L}{\partial h_k} \frac{\partial h_k}{\partial W_{ij}} \qquad \frac{\partial L}{\partial b_i} = \sum_k \frac{\partial L}{\partial h_k} \frac{\partial h_k}{\partial b_i}$$

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

What about the weights?

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

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



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

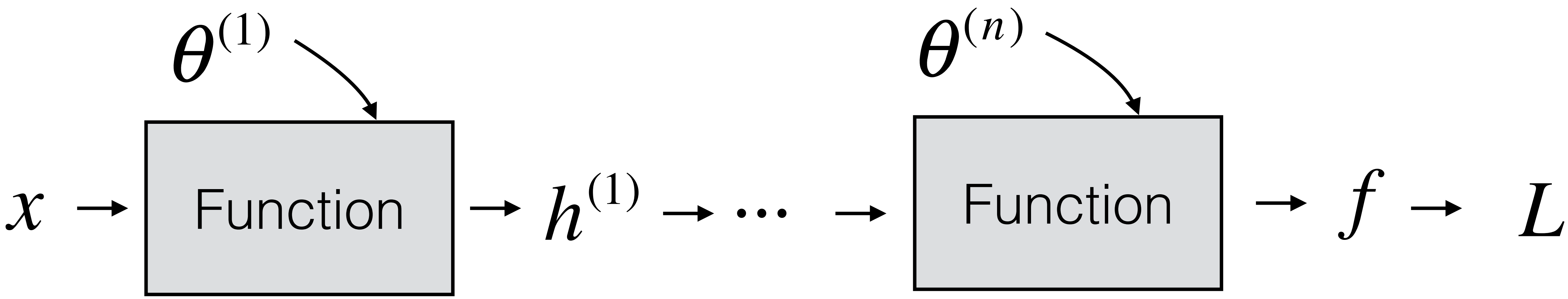
$$\frac{\partial L}{\partial W_{ij}} = \sum_k \frac{\partial L}{\partial h_k} \frac{\partial h_k}{\partial W_{ij}} \qquad \frac{\partial L}{\partial b_i} = \sum_k \frac{\partial L}{\partial h_k} \frac{\partial h_k}{\partial b_i}$$

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

HW2: you will derive this for various layers.

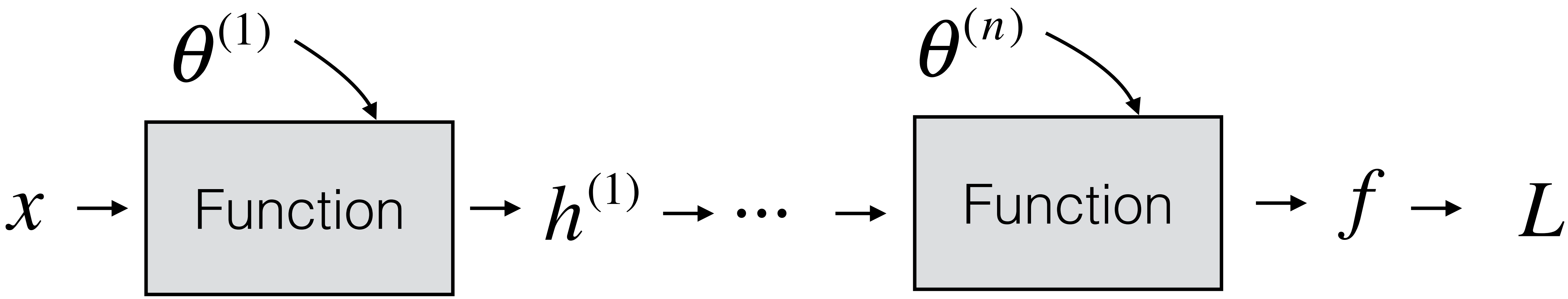
Recap

Forward Propagation:



Recap

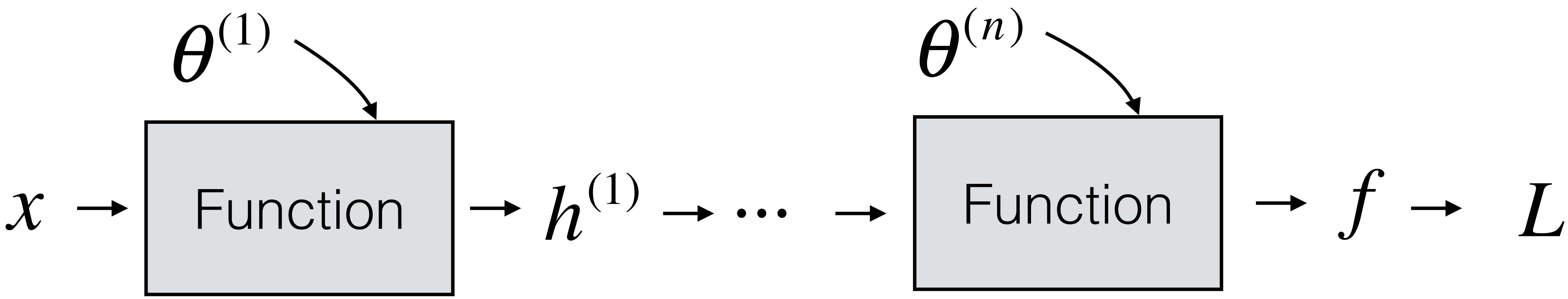
Forward Propagation:



Backward Propagation:

Recap

Forward Propagation:

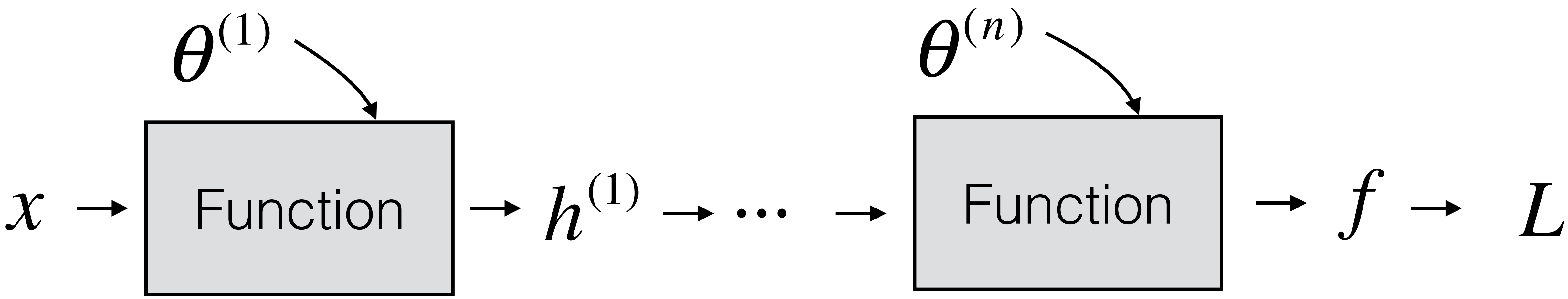


Backward Propagation:

L

Recap

Forward Propagation:

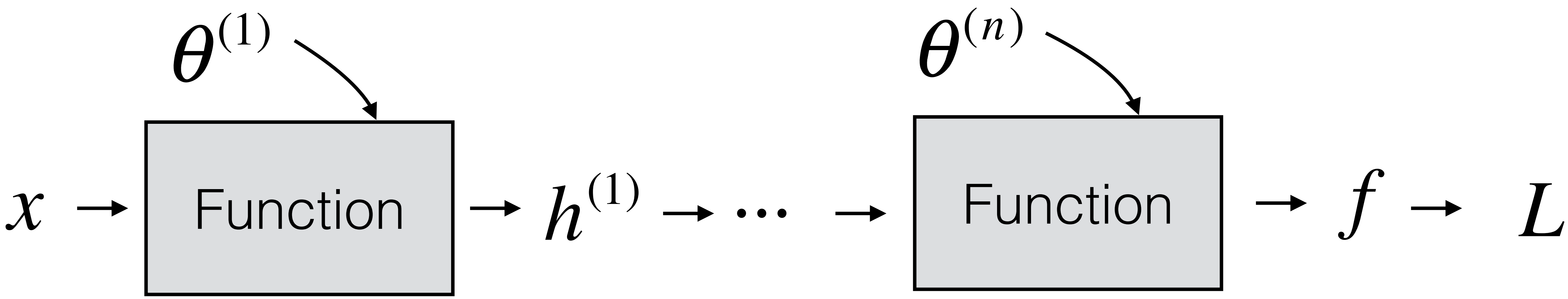


Backward Propagation:

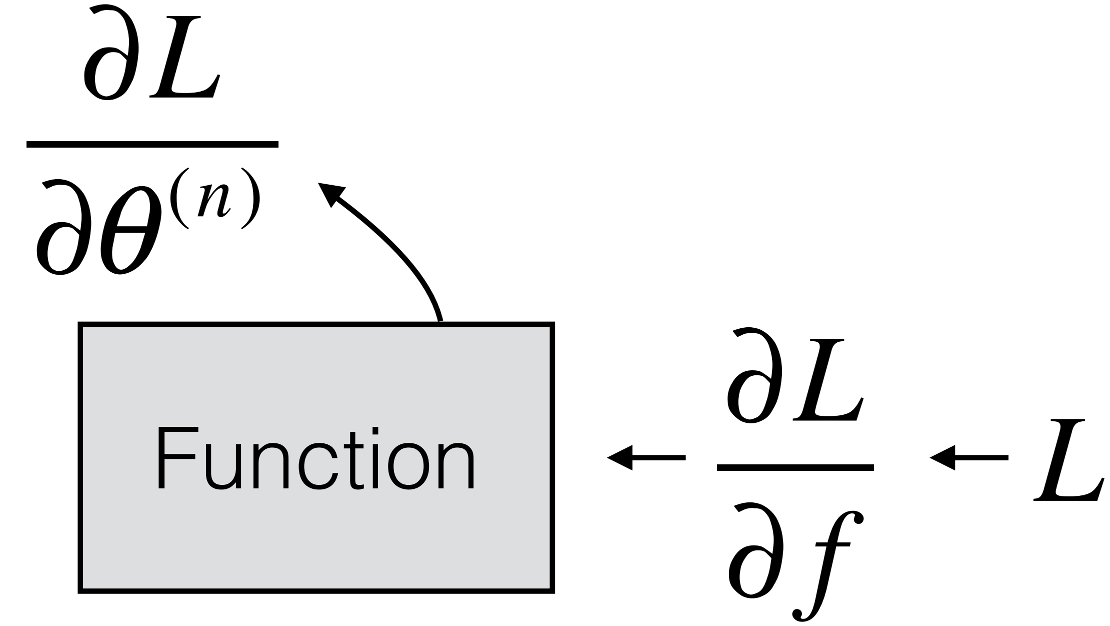
$$\frac{\partial L}{\partial f} \leftarrow L$$

Recap

Forward Propagation:



Backward Propagation:



Recap

Forward Propagation:

$$\begin{tikzcd} & \theta^{(1)} \arrow[d, bend left] & & & \theta^{(n)} \arrow[d, bend left] & & \\ x \arrow[r] & \boxed{\text{Function}} \arrow[r] & h^{(1)} \arrow[r] & \cdots \arrow[r] & \boxed{\text{Function}} \arrow[r] & f \arrow[r] & L \end{tikzcd}$$

Backward Propagation:

$$\begin{tikzcd} & & \dfrac{\partial L}{\partial \theta^{(n)}} & & \\ \dfrac{\partial L}{\partial h^{(1)}} & \cdots \arrow[l] & \boxed{\text{Function}} \arrow[l] \arrow[u, bend right] & \dfrac{\partial L}{\partial f} \arrow[l] & L \arrow[l] \end{tikzcd}$$

Recap

Forward Propagation:

$$\begin{tikzcd} & \theta^{(1)} \arrow[d] & & & \theta^{(n)} \arrow[d] & & \\ x \arrow[r] & \boxed{\text{Function}} \arrow[r] & h^{(1)} \arrow[r] & \cdots \arrow[r] & \boxed{\text{Function}} \arrow[r] & f \arrow[r] & L \end{tikzcd}$$

Backward Propagation:

$$\begin{tikzcd} & \dfrac{\partial L}{\partial \theta^{(1)}} & & & \dfrac{\partial L}{\partial \theta^{(n)}} & & \\ \dfrac{\partial L}{\partial x} & \boxed{\text{Function}} \arrow[l] \arrow[u] & \dfrac{\partial L}{\partial h^{(1)}} \arrow[l] & \cdots \arrow[l] & \boxed{\text{Function}} \arrow[l] \arrow[u] & \dfrac{\partial L}{\partial f} \arrow[l] & L \arrow[l] \end{tikzcd}$$

Questions?

30s cat picture break

http://stylonica.com/cat-pictures/

CNNs

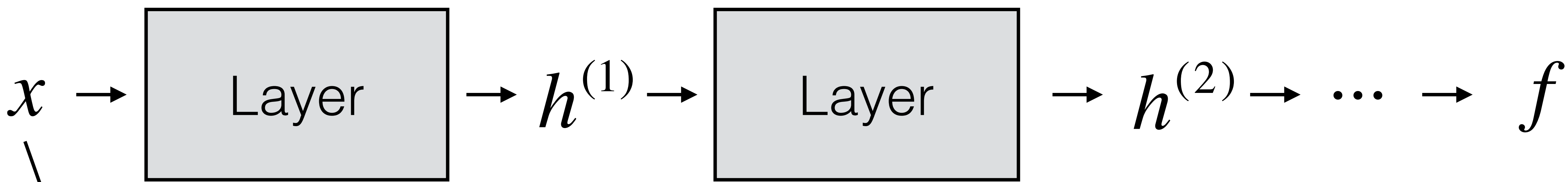
It’s just neural networks
with 3D activations

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)

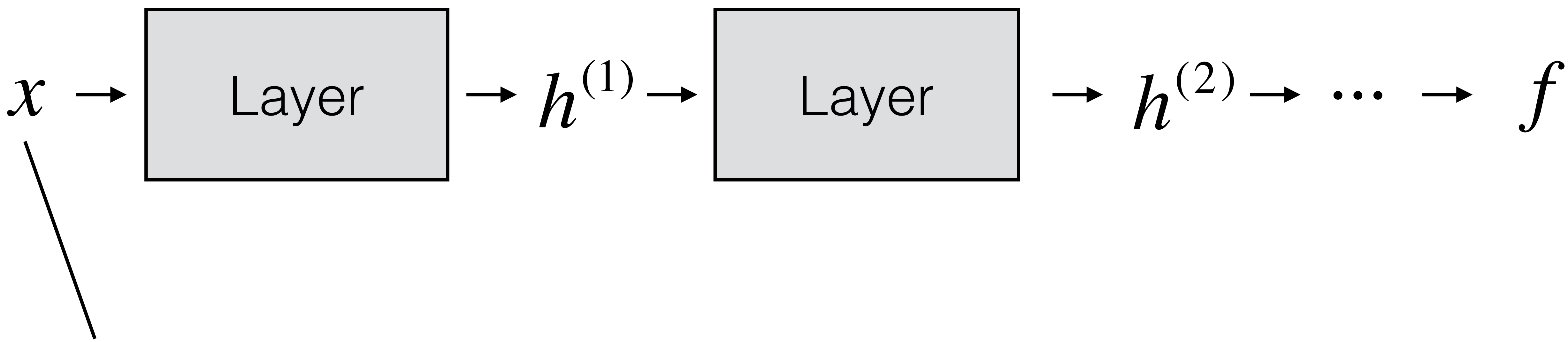
What shape should the activations have?



- 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?



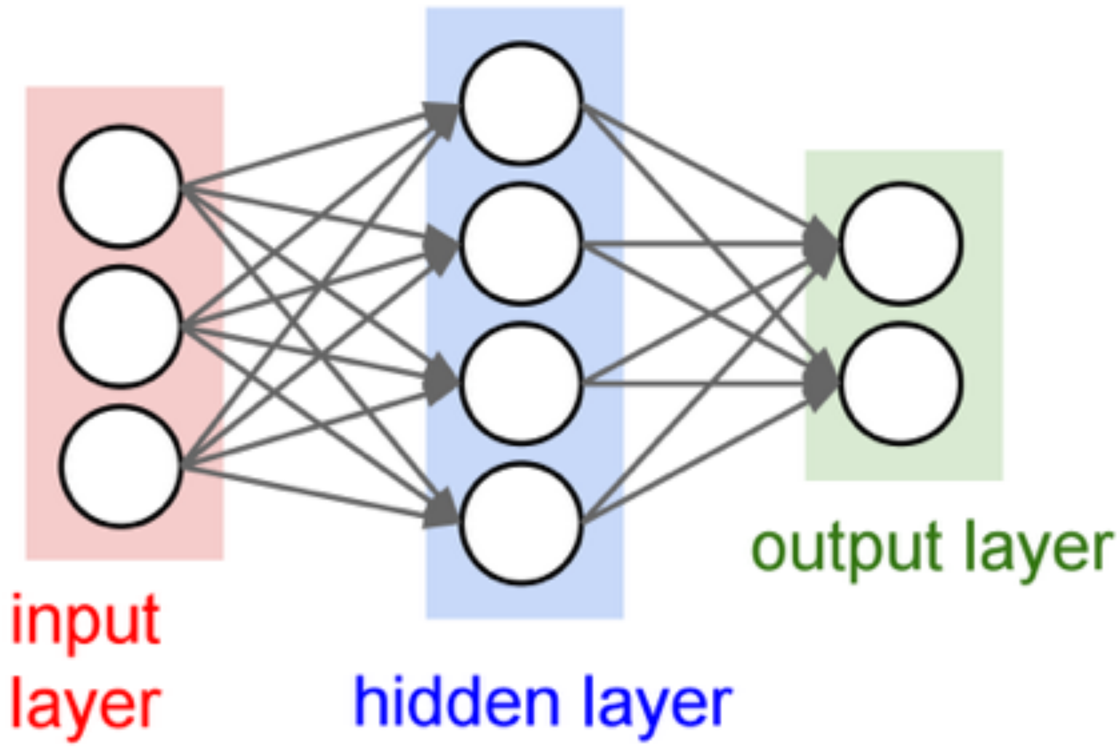
- The input is an image, which is 3D (RGB channel, height, width)

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

- What about keeping everything in 3D?

3D Activations

before:

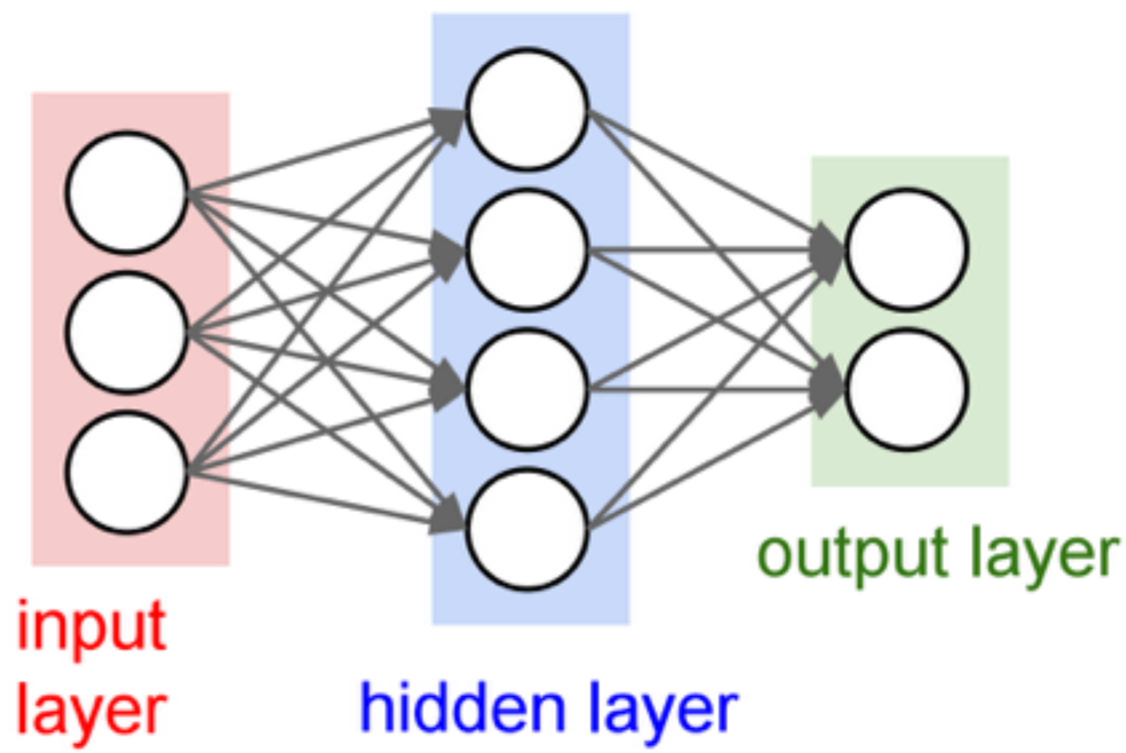


(1D vectors)

Figure: Andrej Karpathy

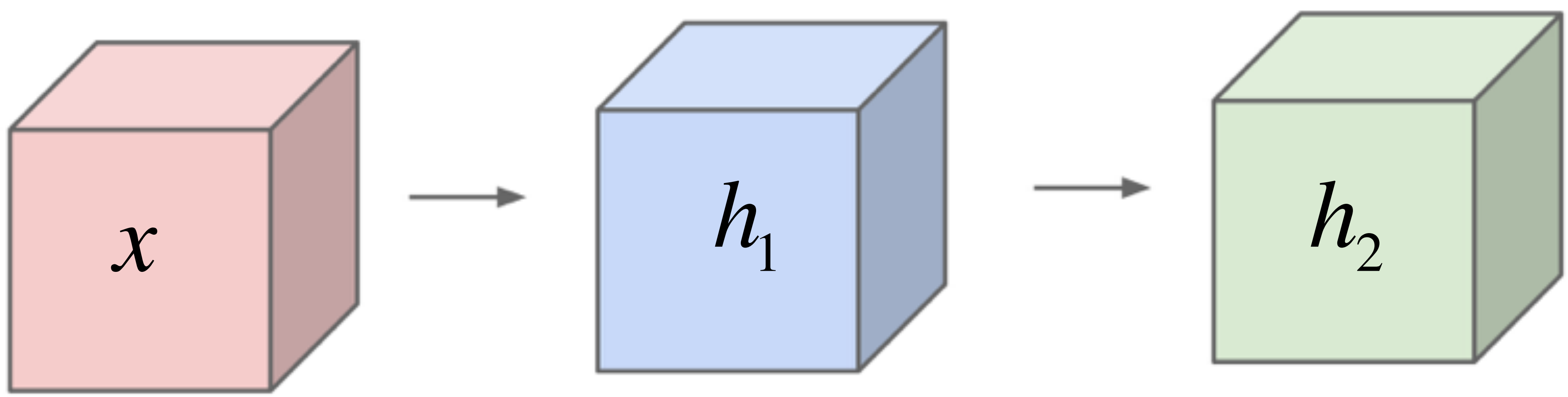
3D Activations

before:



(1D vectors)

now:



(3D arrays)

Figure: Andrej Karpathy

3D Activations

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

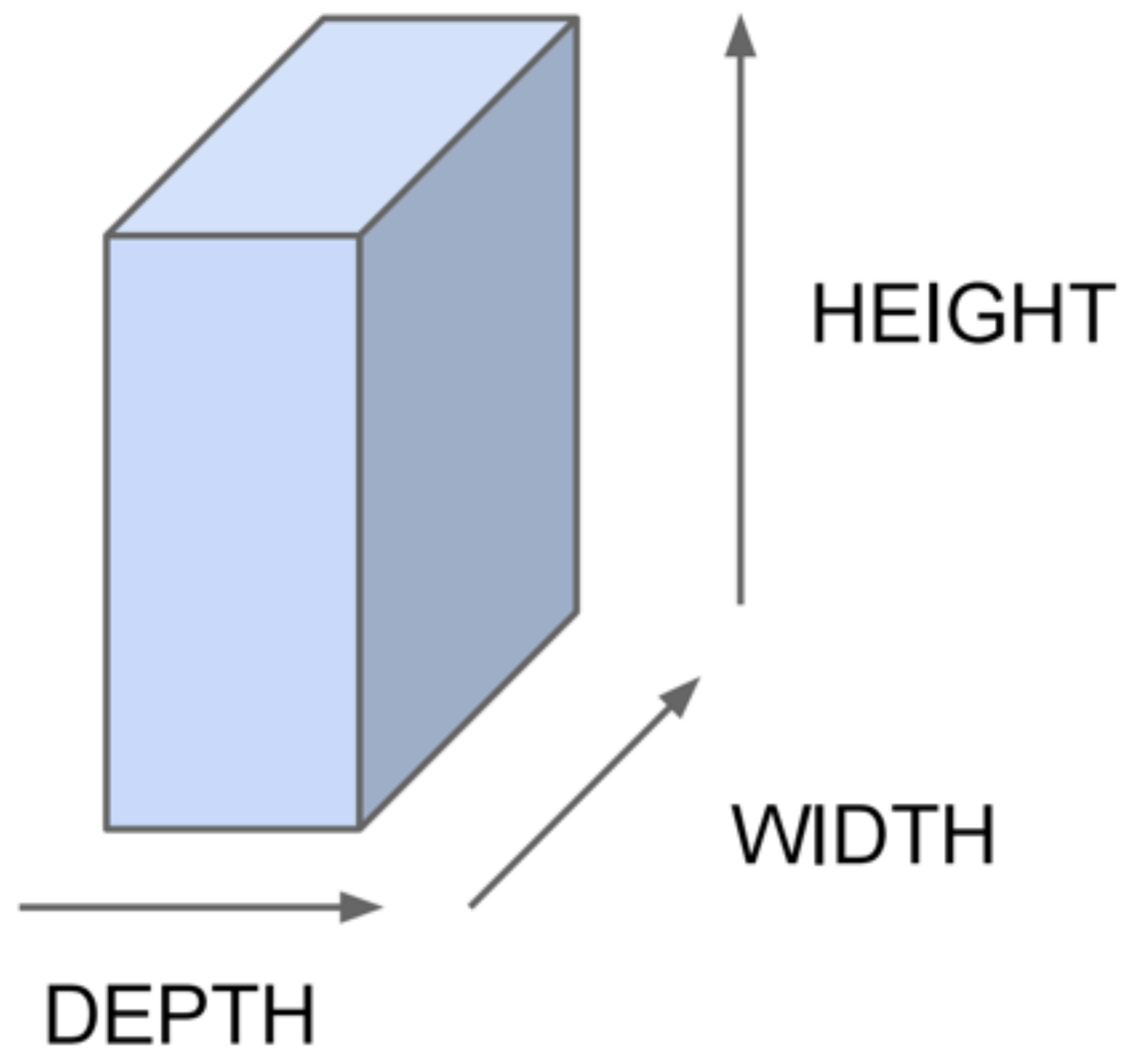
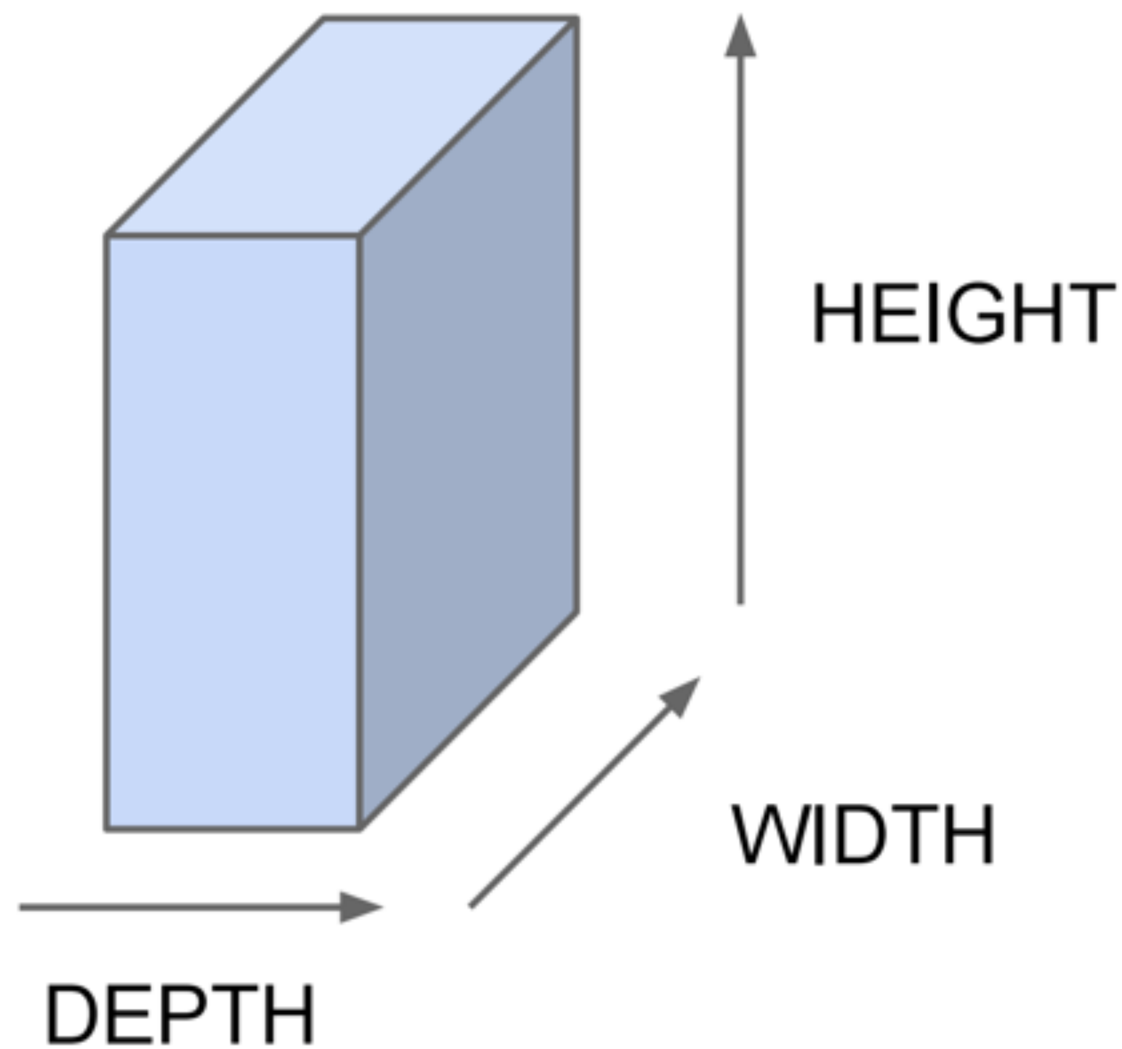


Figure: Andrej Karpathy

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:

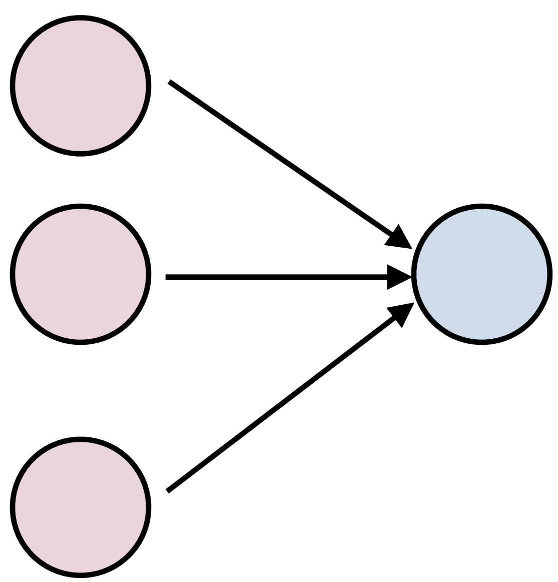
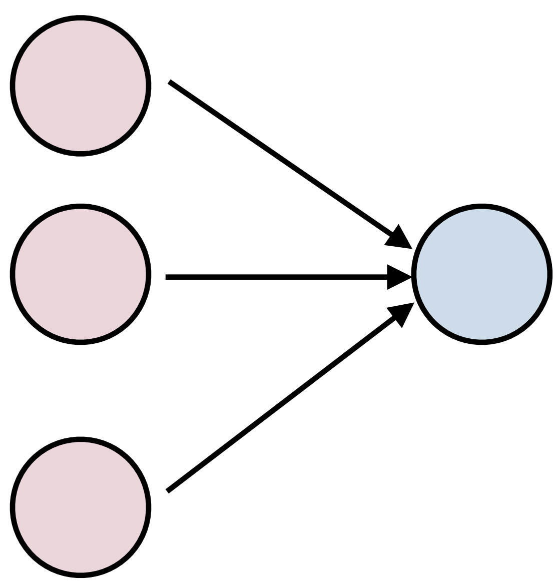


Figure: Andrej Karpathy

3D Activations

1D Activations:



3D Activations:

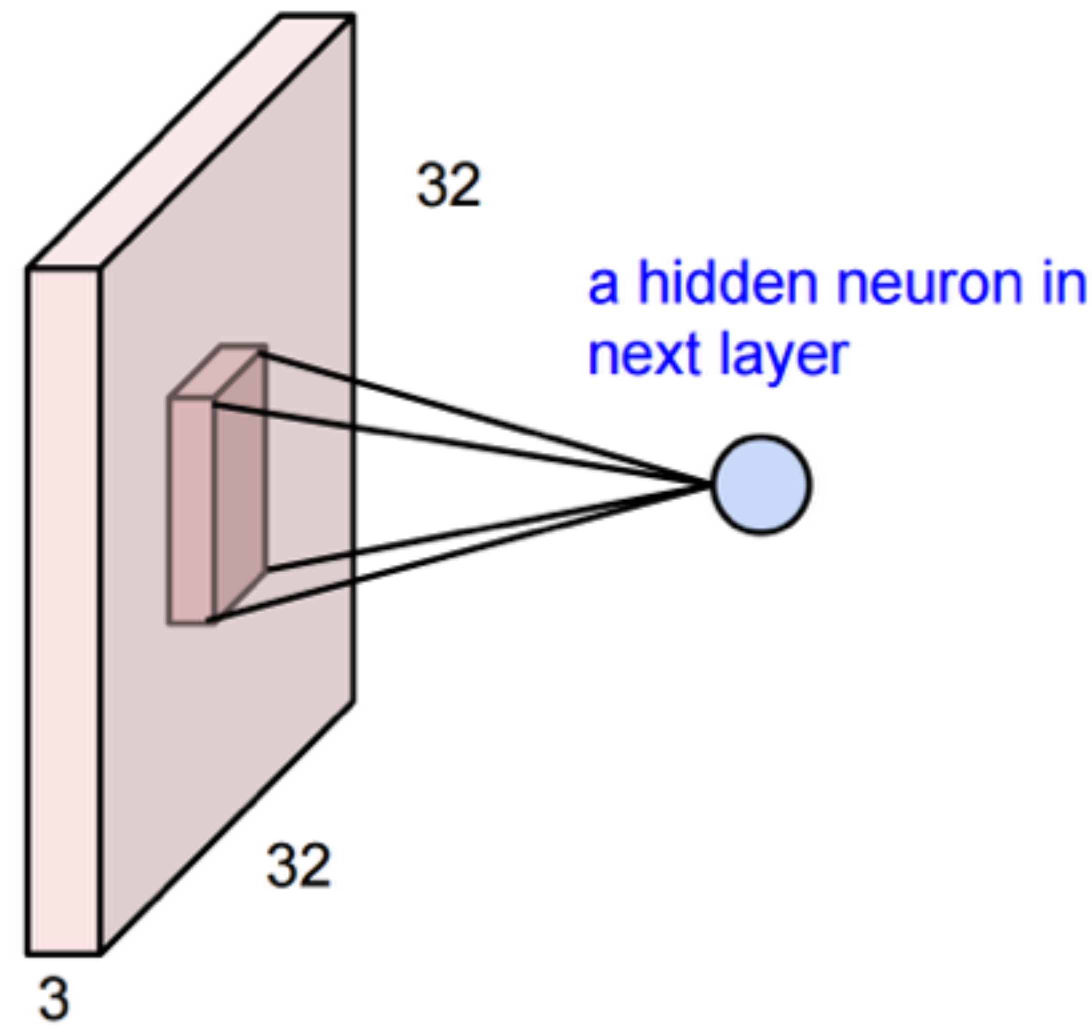
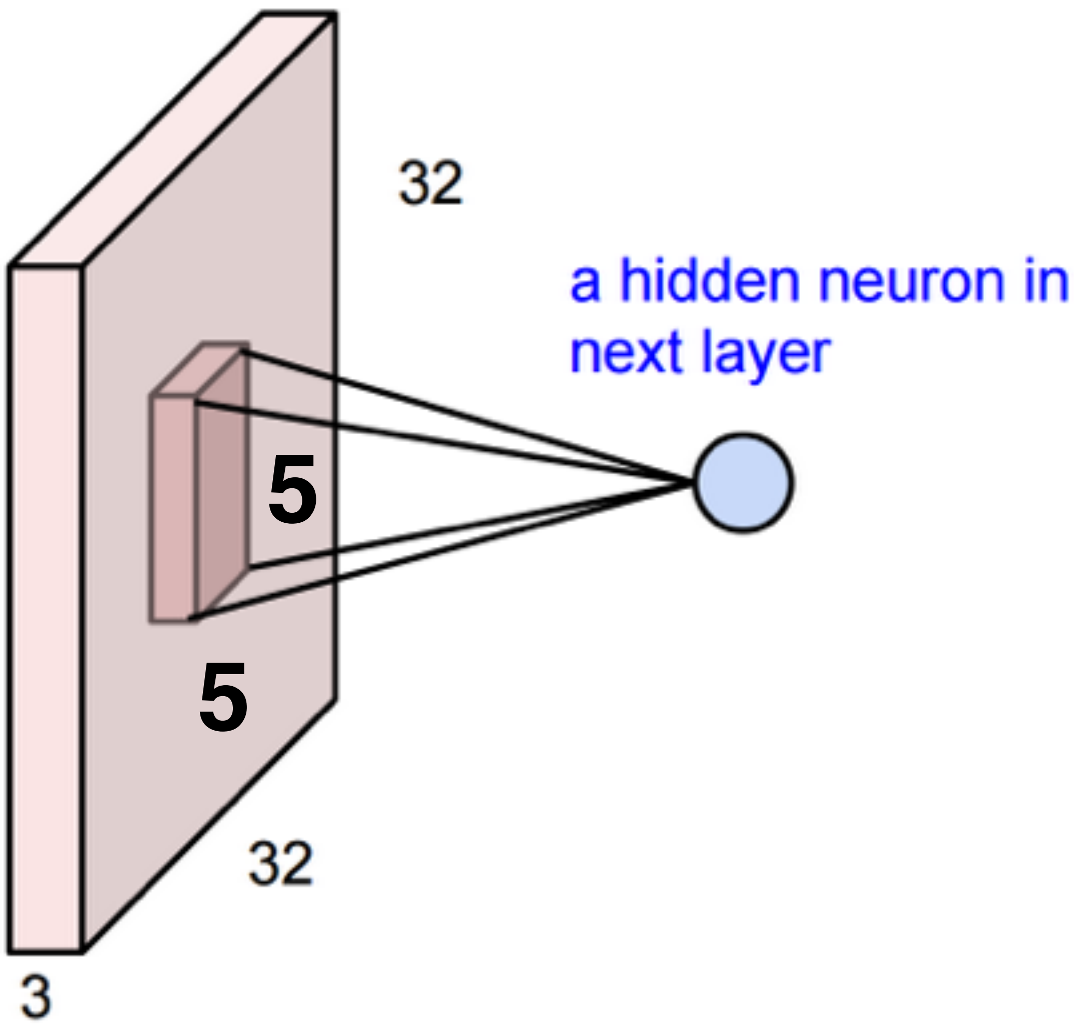


Figure: Andrej Karpathy

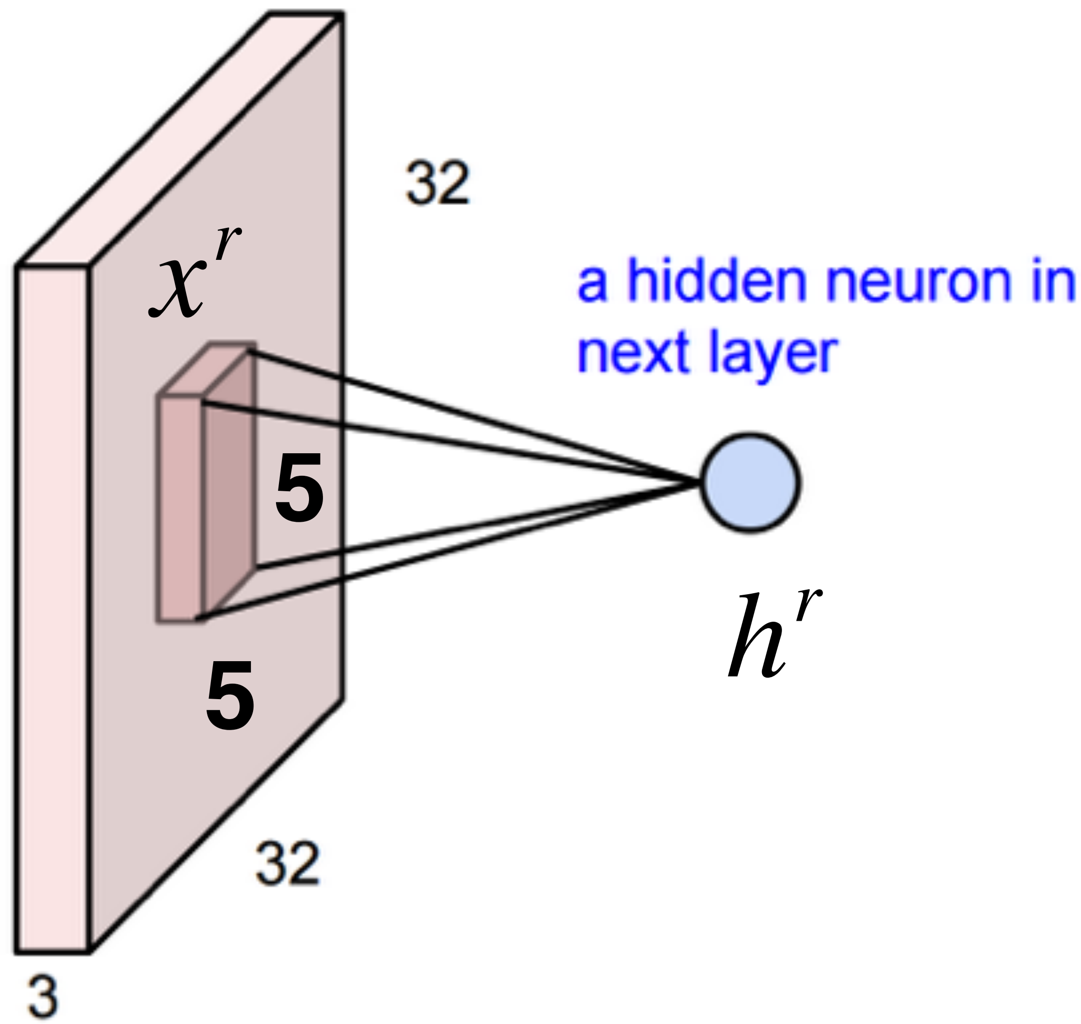
3D Activations



- 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

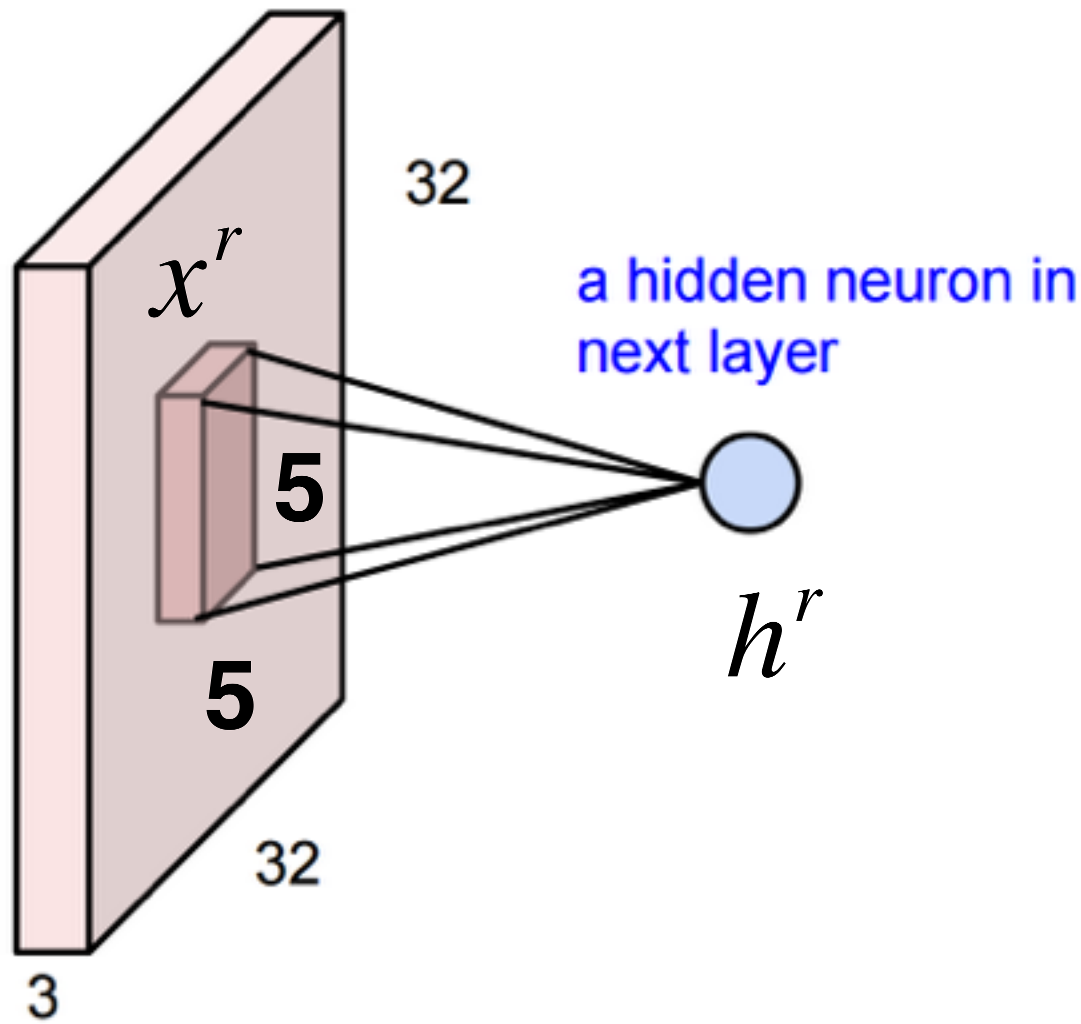


Example: consider the region of the input "x^r"

With output neuron h^r

Figure: Andrej Karpathy

3D Activations



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

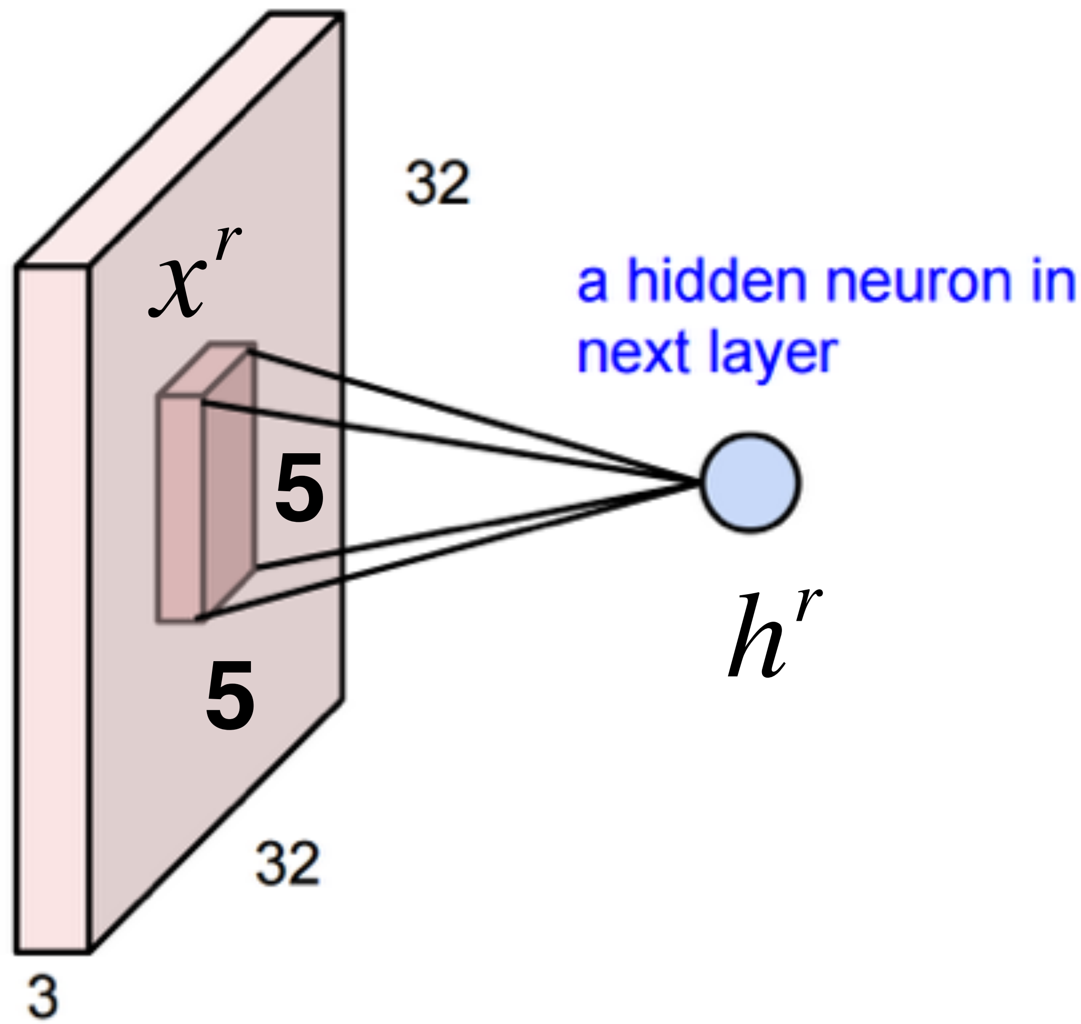
With output neuron h^r

Then the output is:

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

Figure: Andrej Karpathy

3D Activations



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

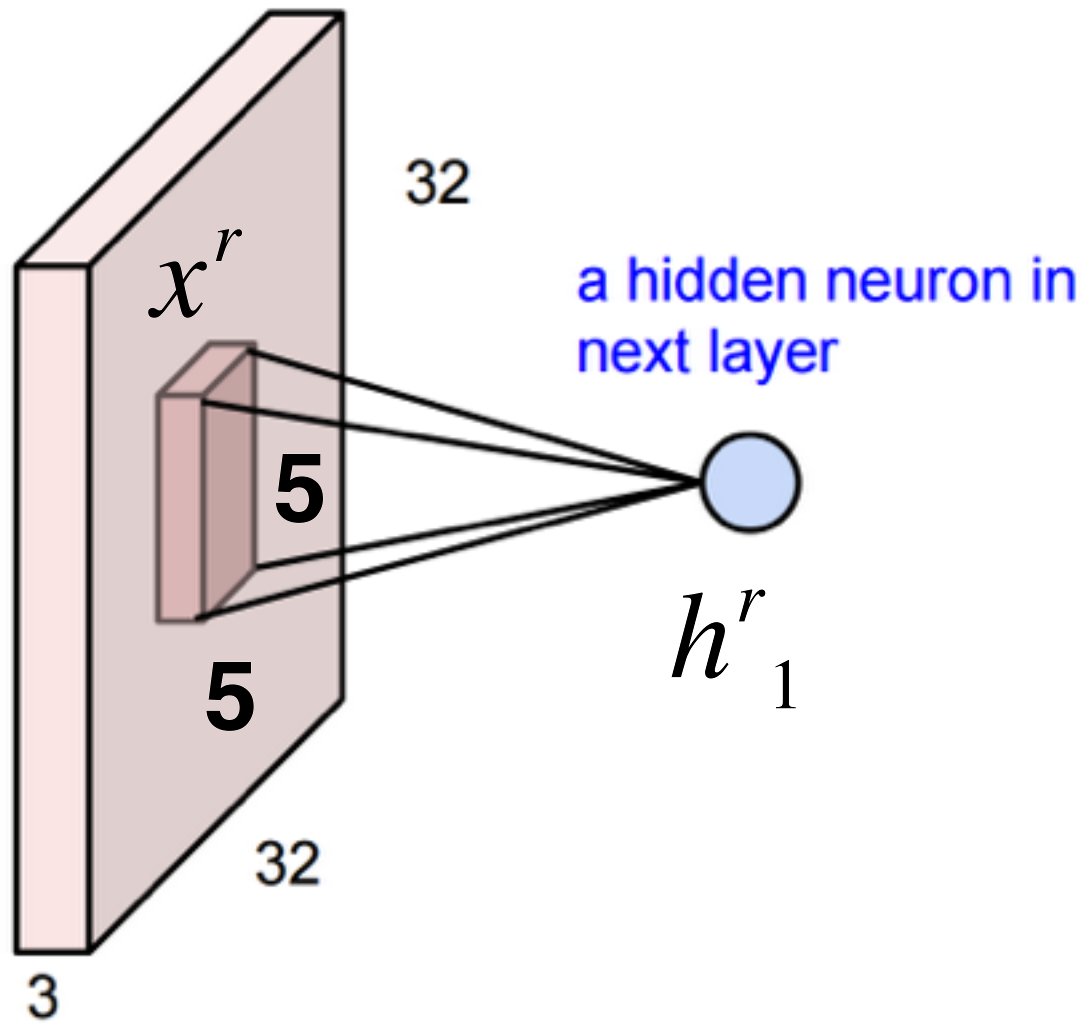


Figure: Andrej Karpathy

3D Activations

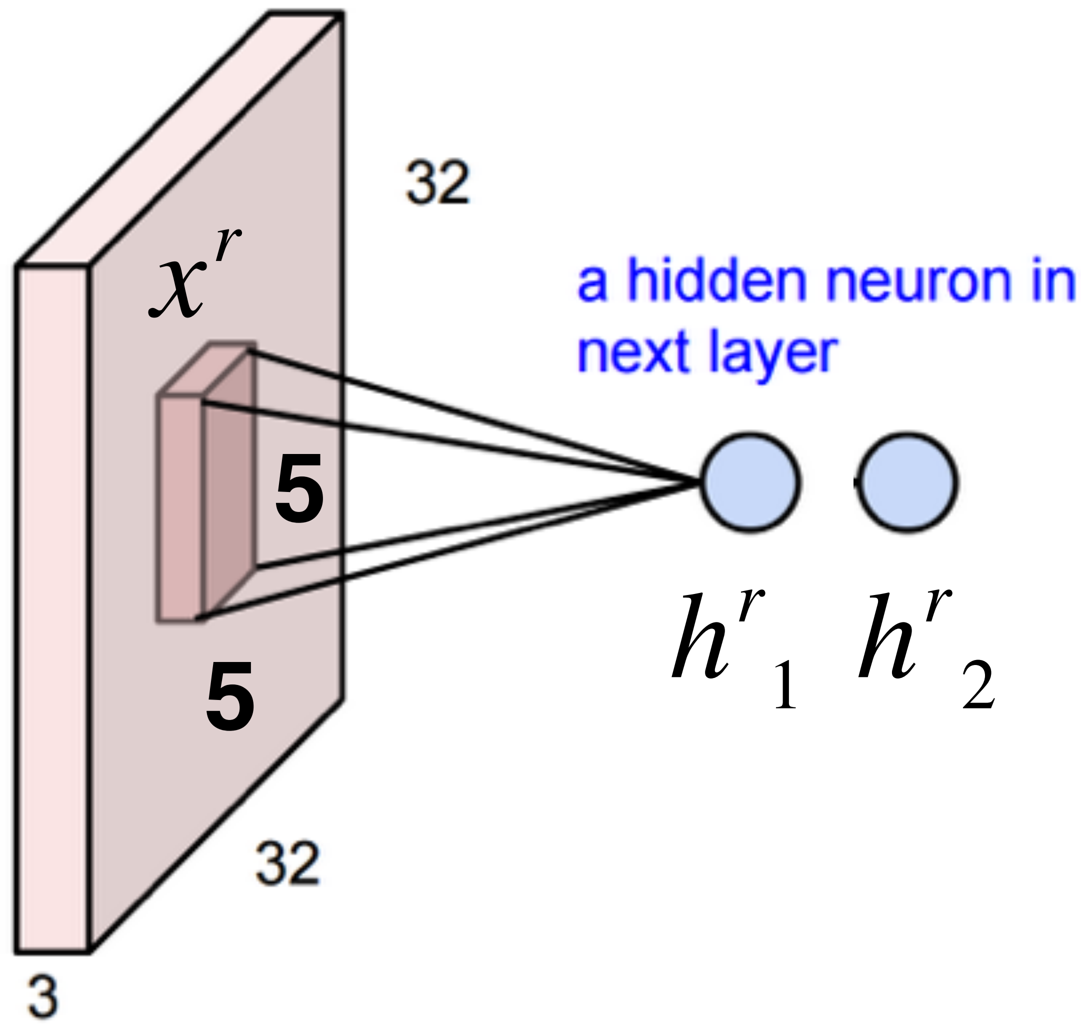
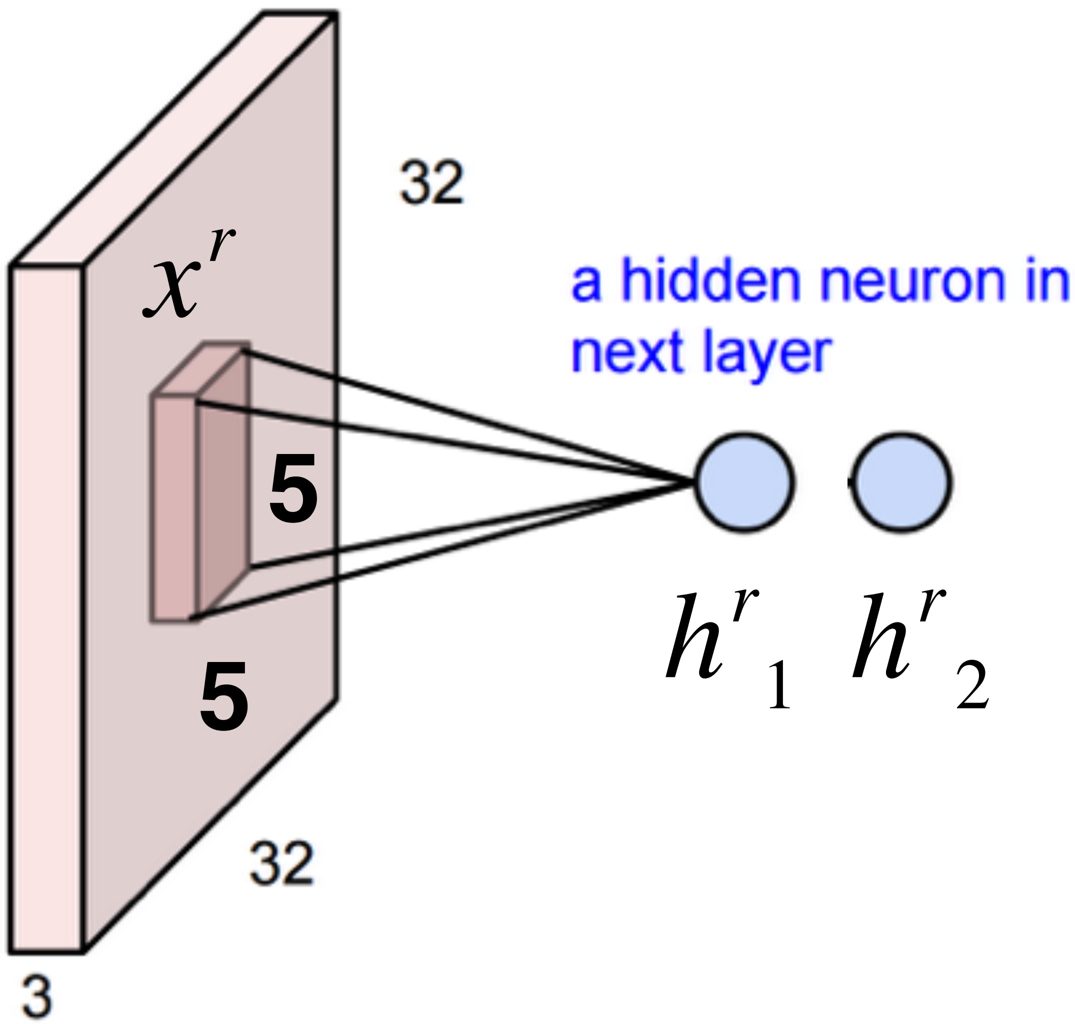


Figure: Andrej Karpathy

3D Activations



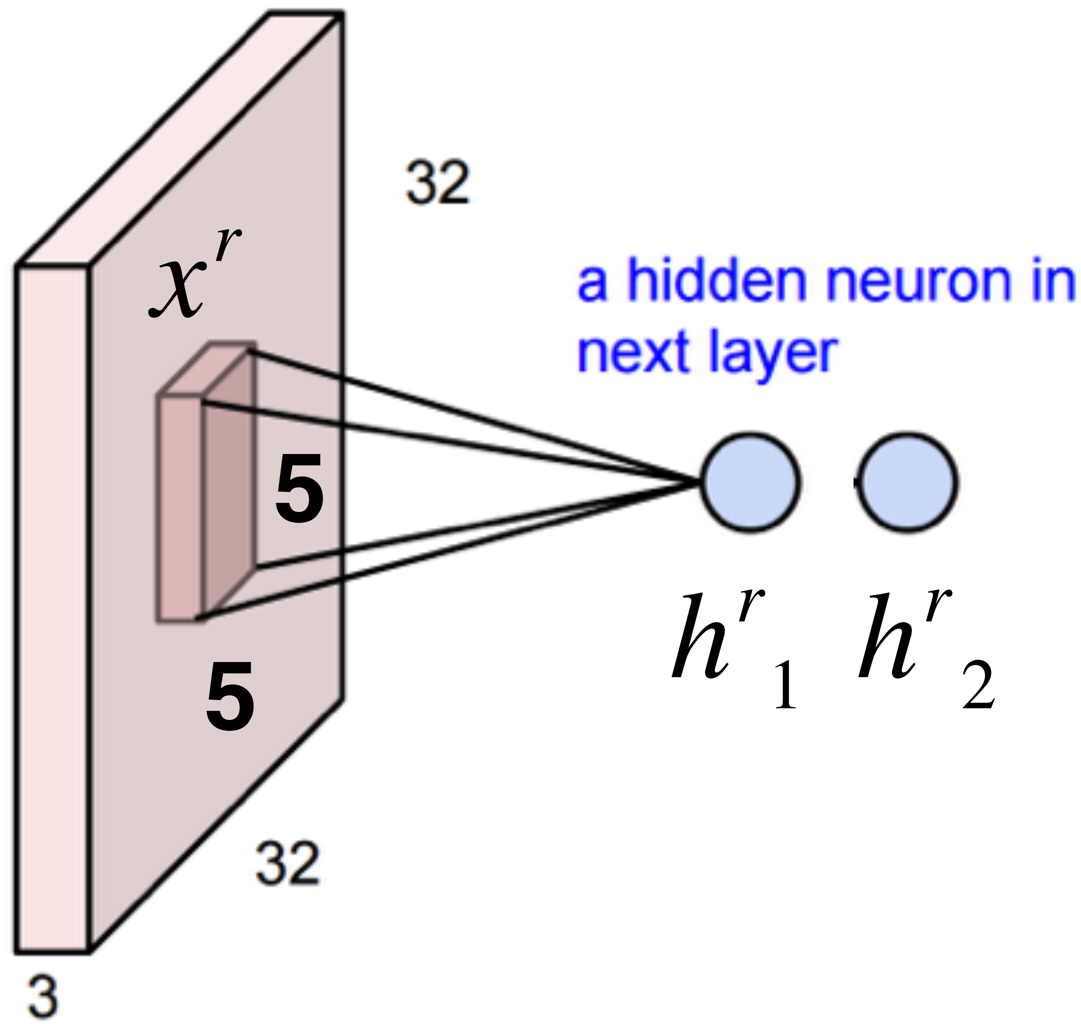
With **2** output neurons

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

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

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

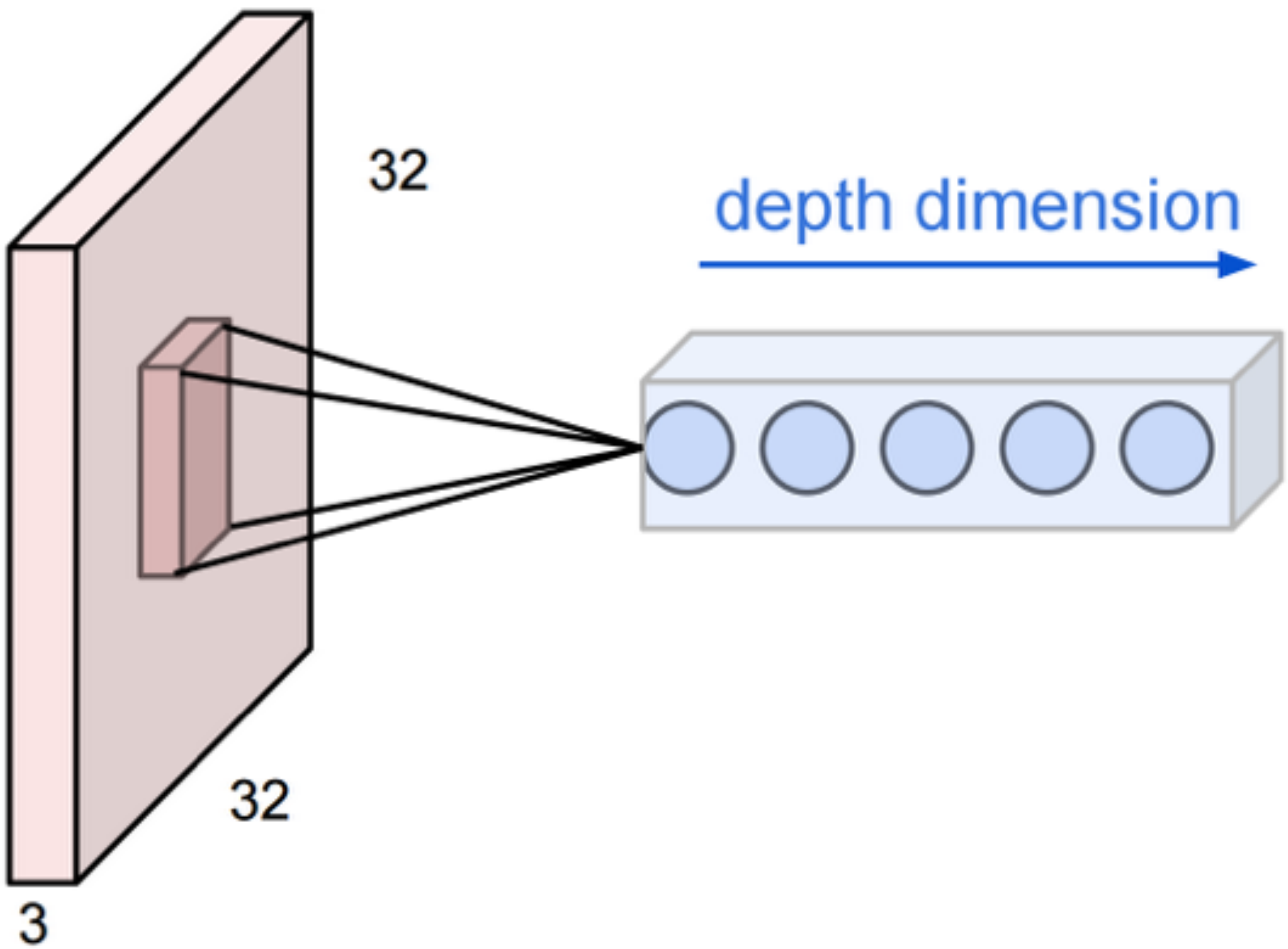
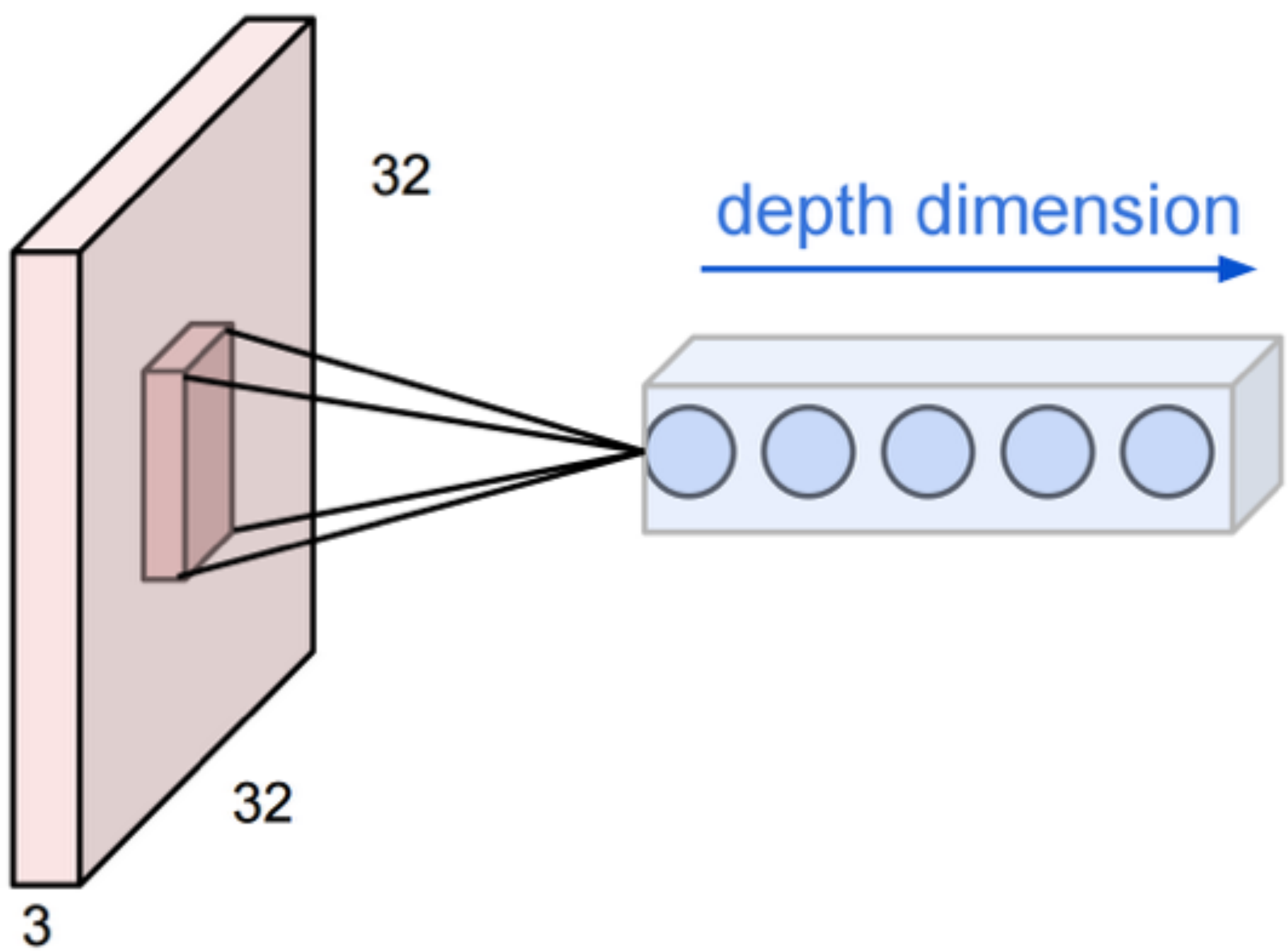


Figure: Andrej Karpathy

3D Activations

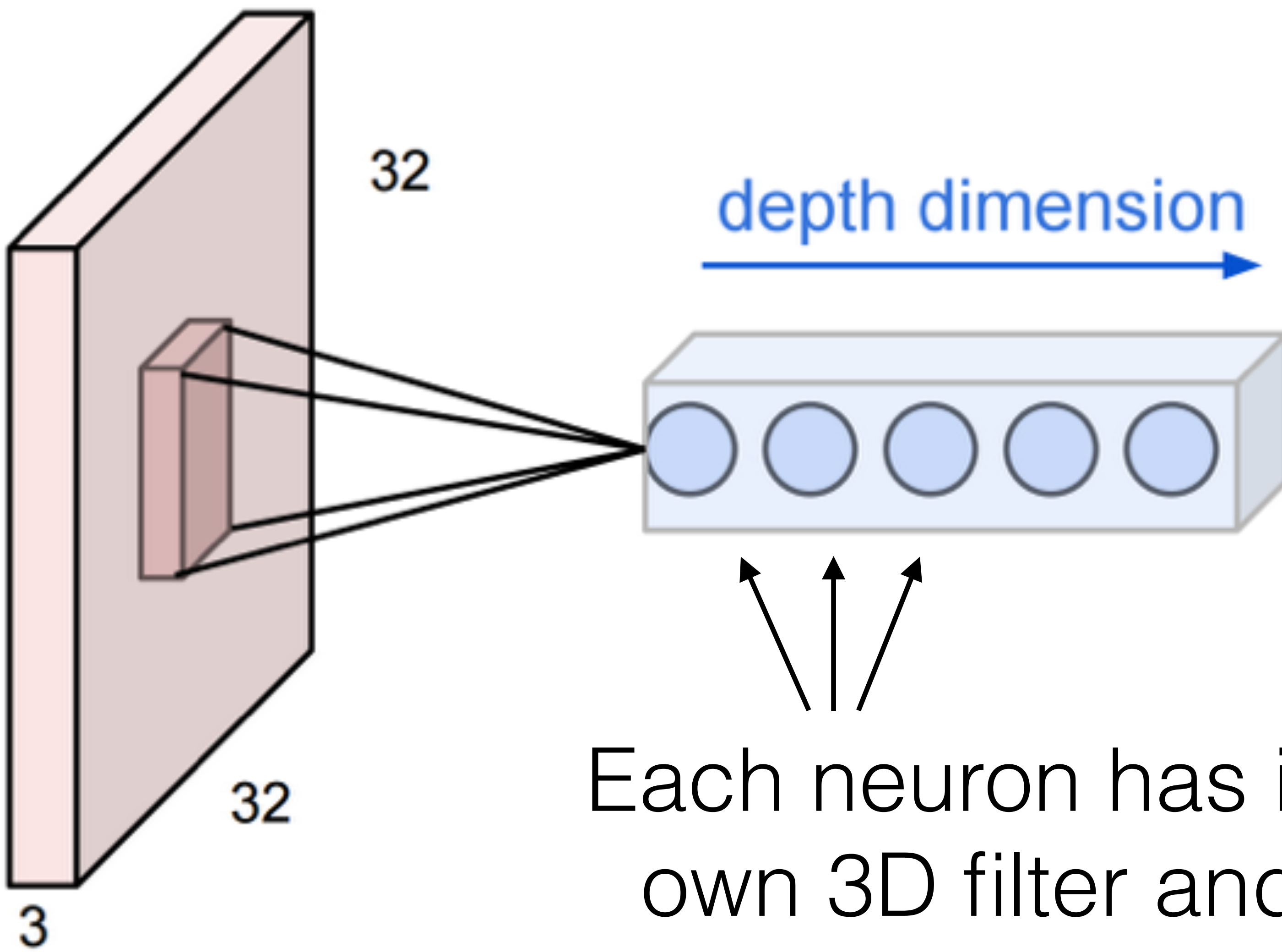


We can keep adding more outputs

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

Figure: Andrej Karpathy

3D Activations



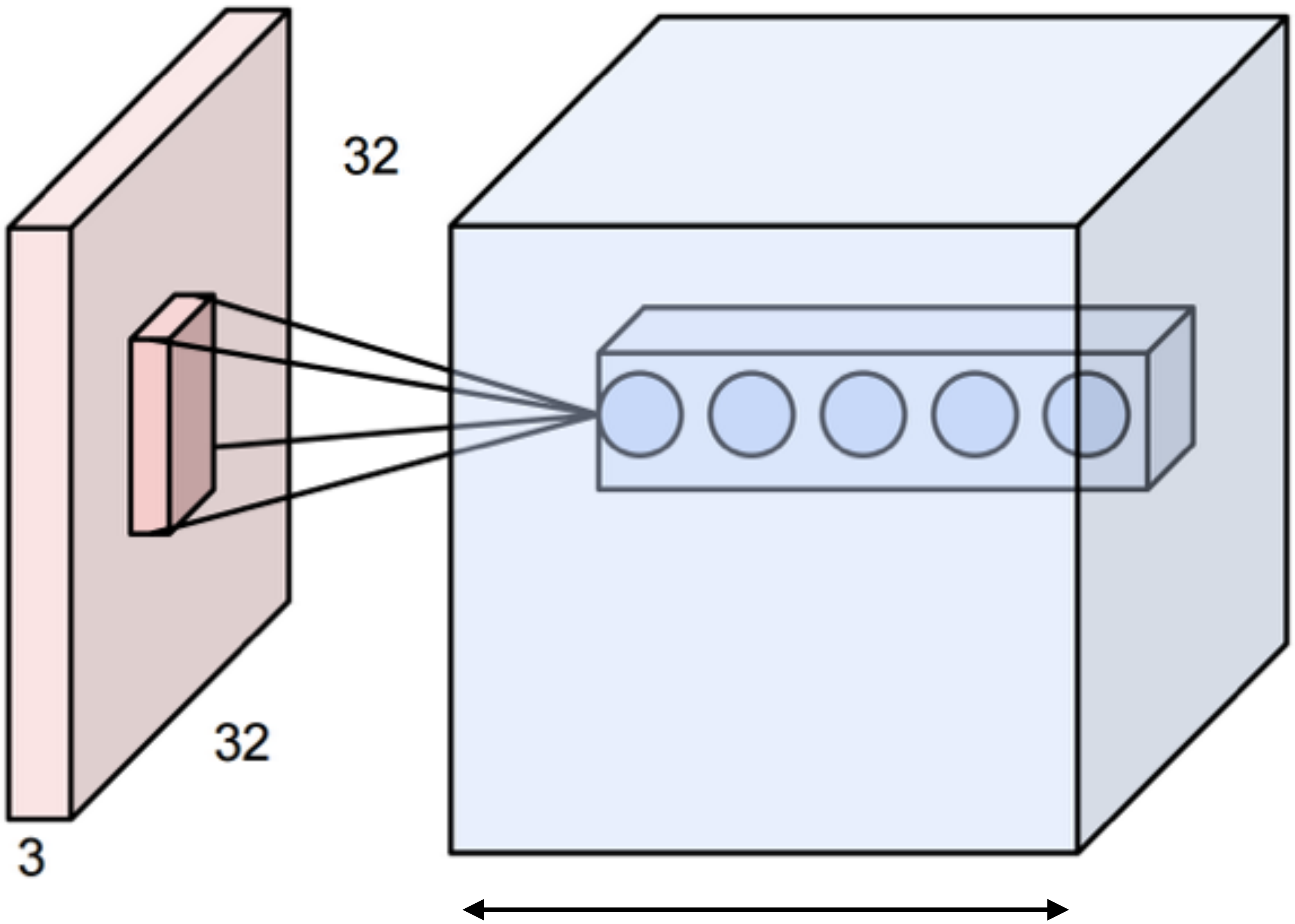
Each neuron has its own 3D filter and own (scalar) bias

We can keep adding more outputs

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

Figure: Andrej Karpathy

3D Activations

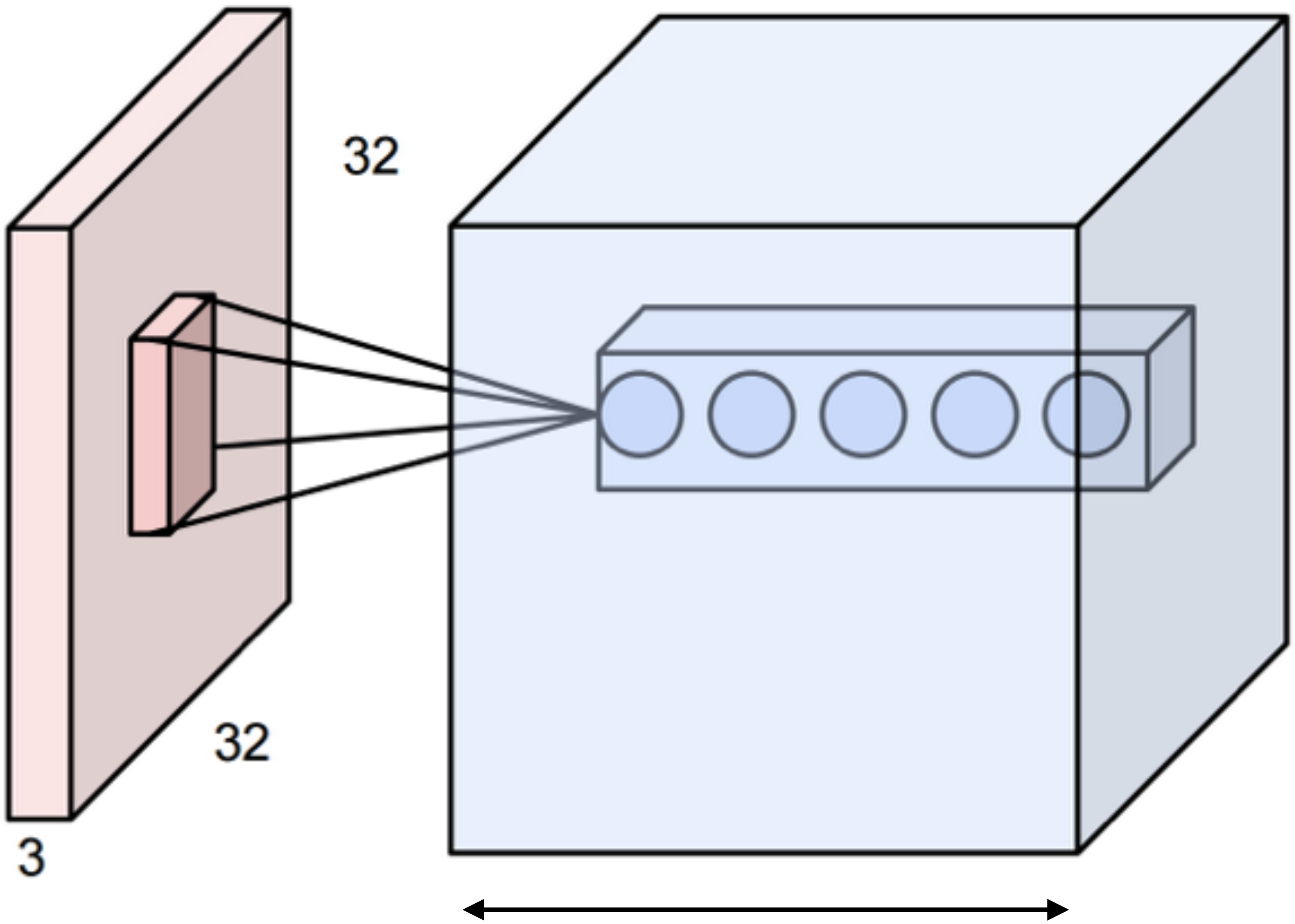


D sets of weights
(also called filters)

Now repeat this across the input

Figure: Andrej Karpathy

3D Activations



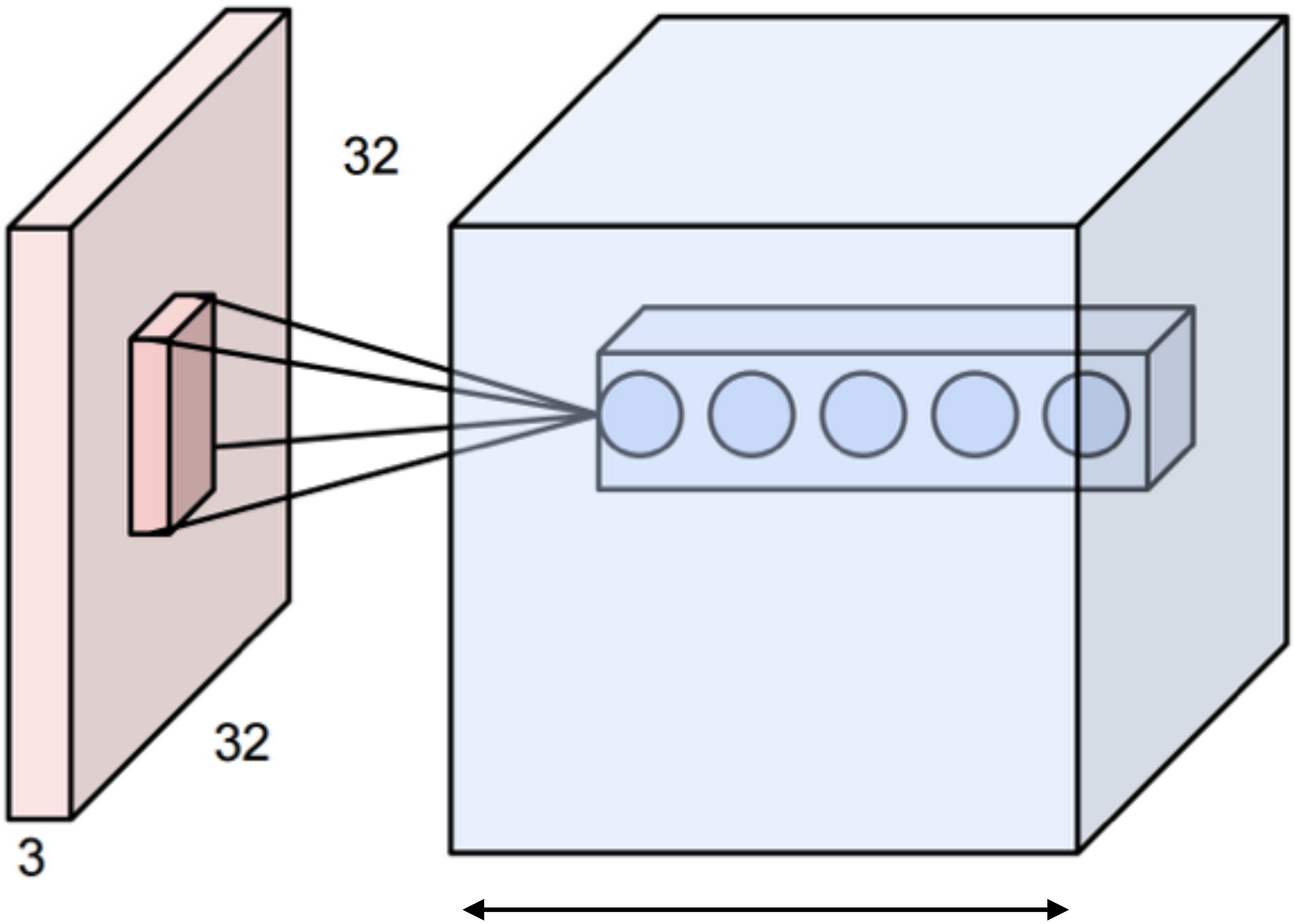
D sets of weights
(also called filters)

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

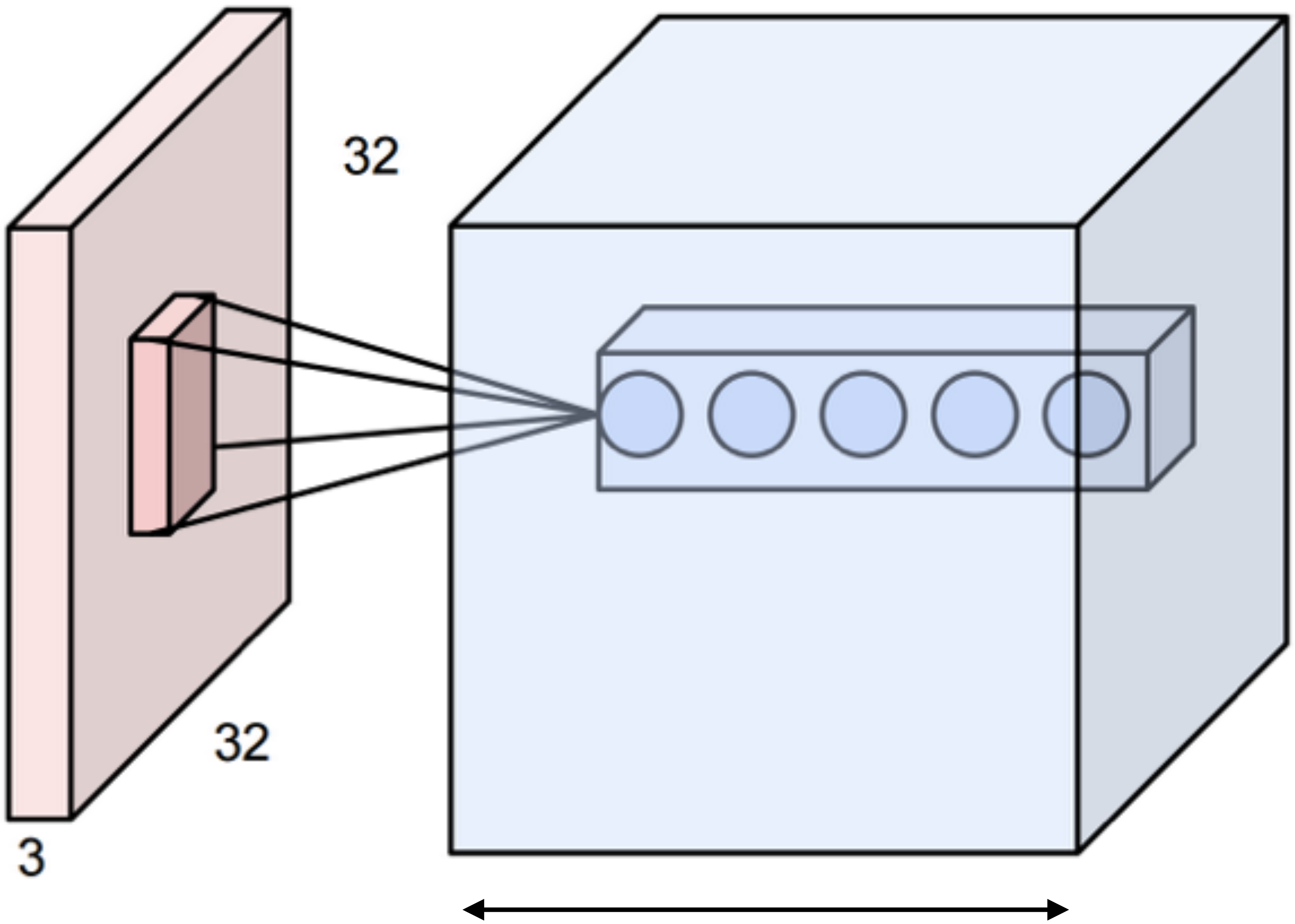
3D Activations



D sets of weights
(also called filters)

Figure: Andrej Karpathy

3D Activations

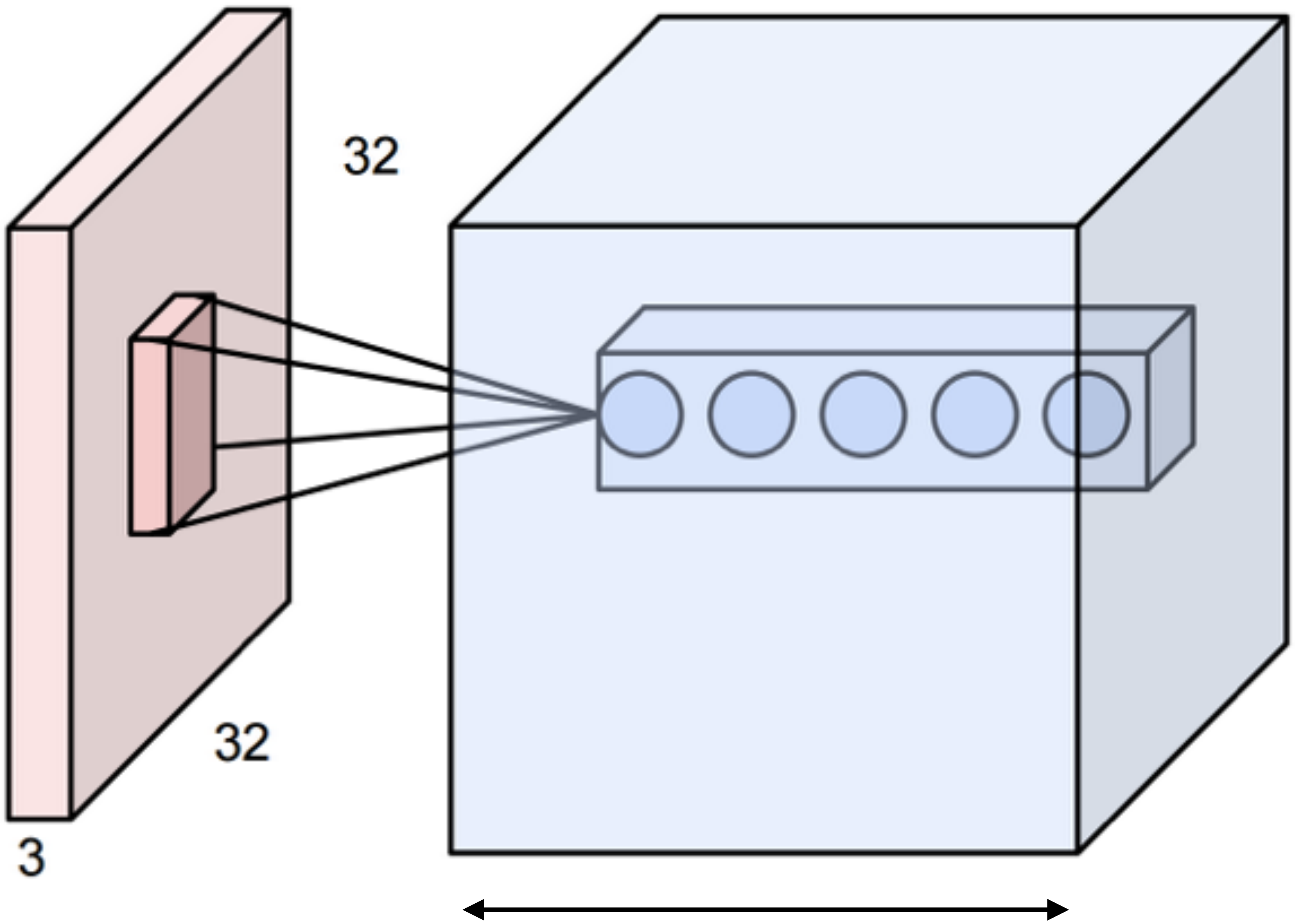


D sets of weights
(also called filters)

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

Figure: Andrej Karpathy

3D Activations



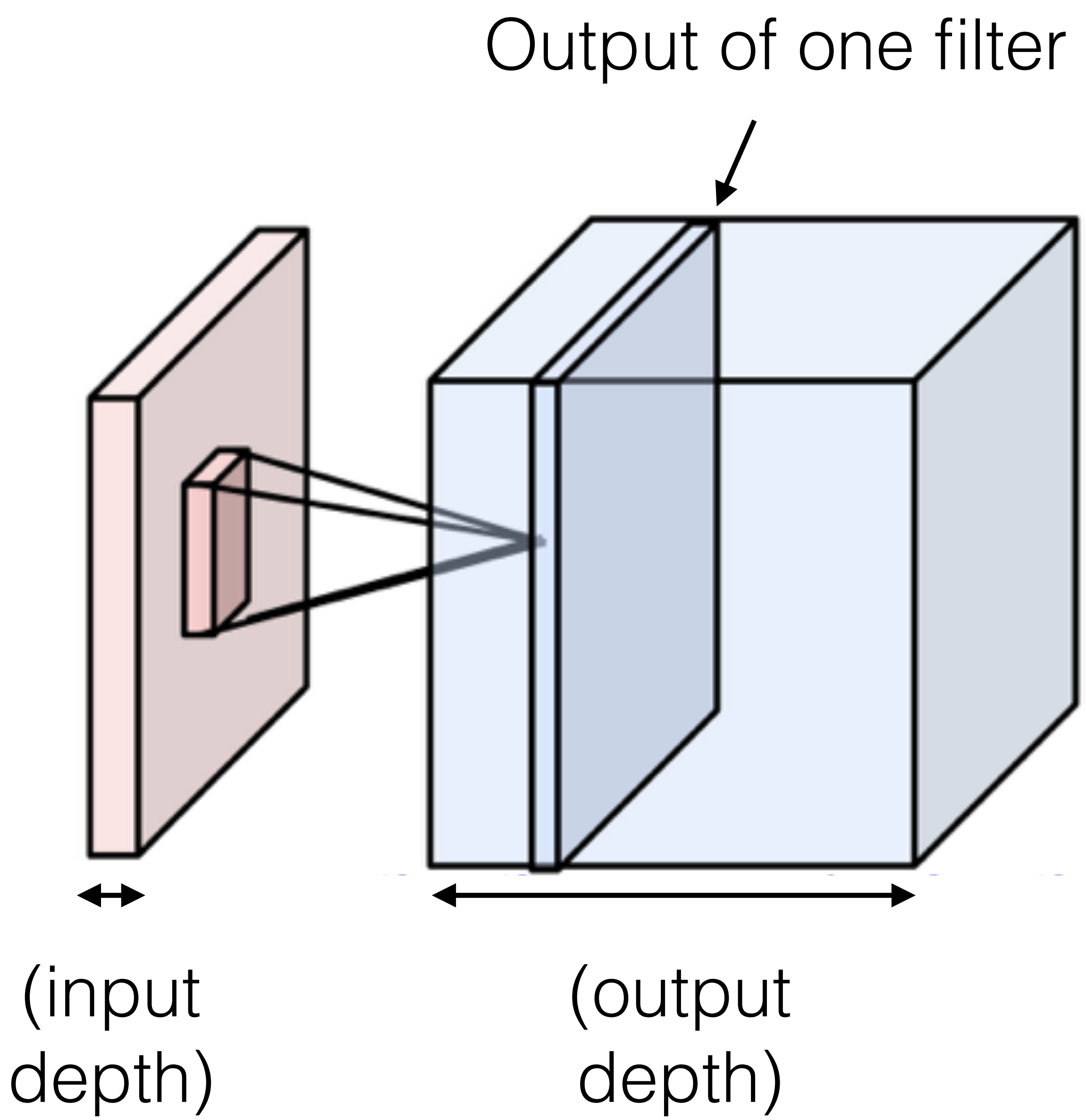
D sets of weights
(also called filters)

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

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

Figure: Andrej Karpathy

3D Activations

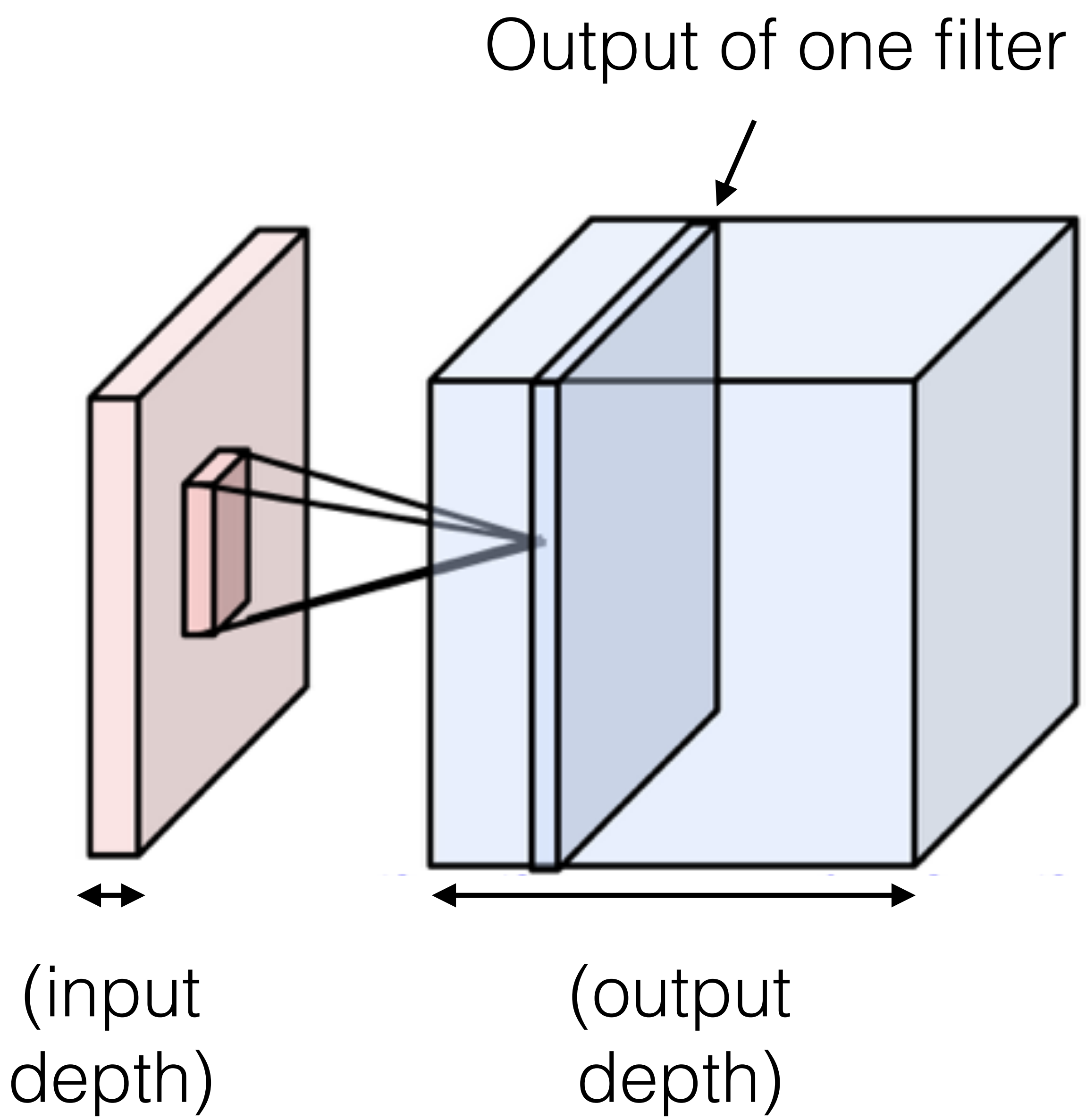


One set of weights gives one slice in the output

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

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

3D Activations



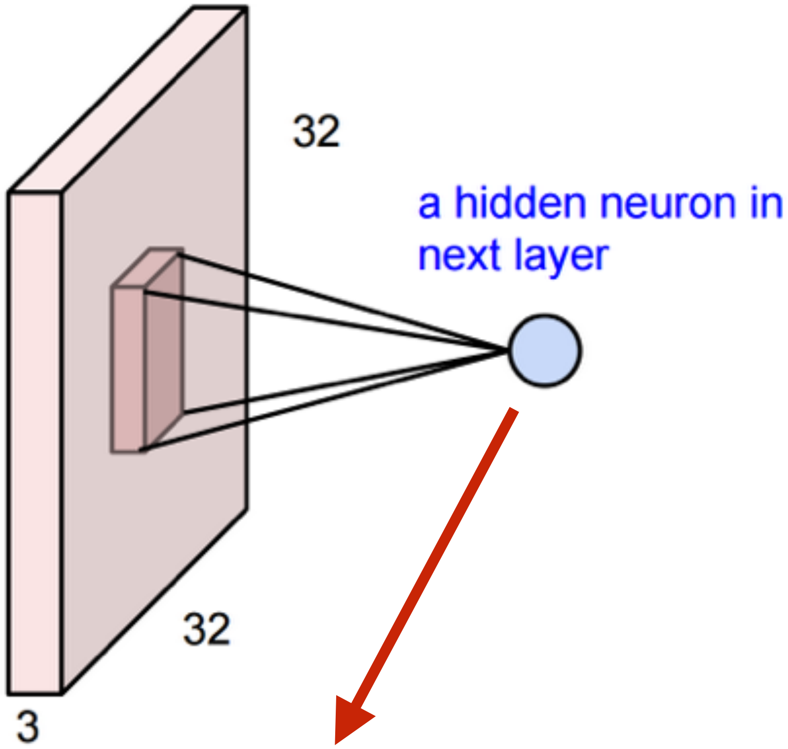
One set of weights gives one slice in the output

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

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

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

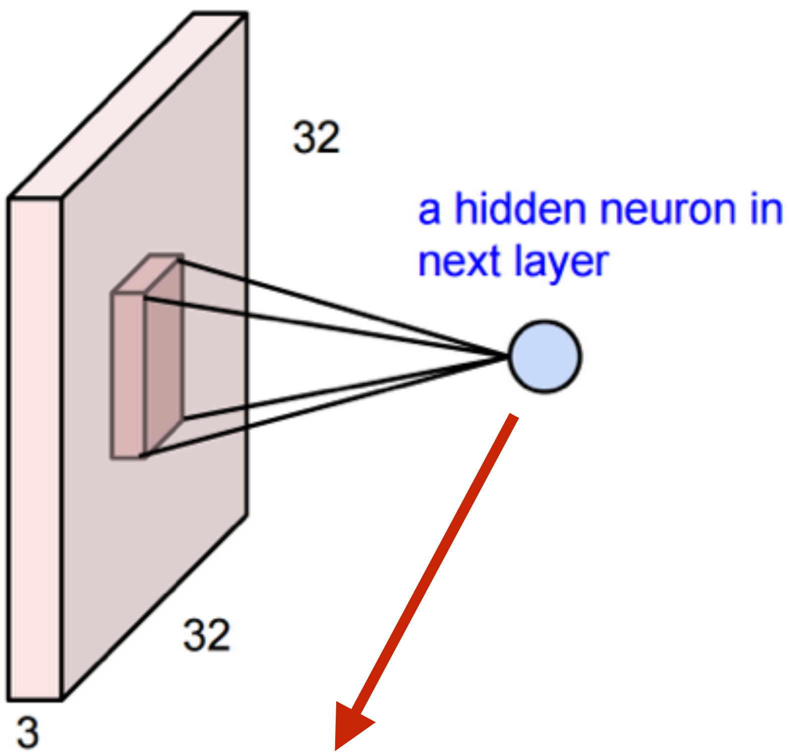
3D Activations




```
out[n, 0, r, c] =
```

Let's code this up in NumPy

3D Activations

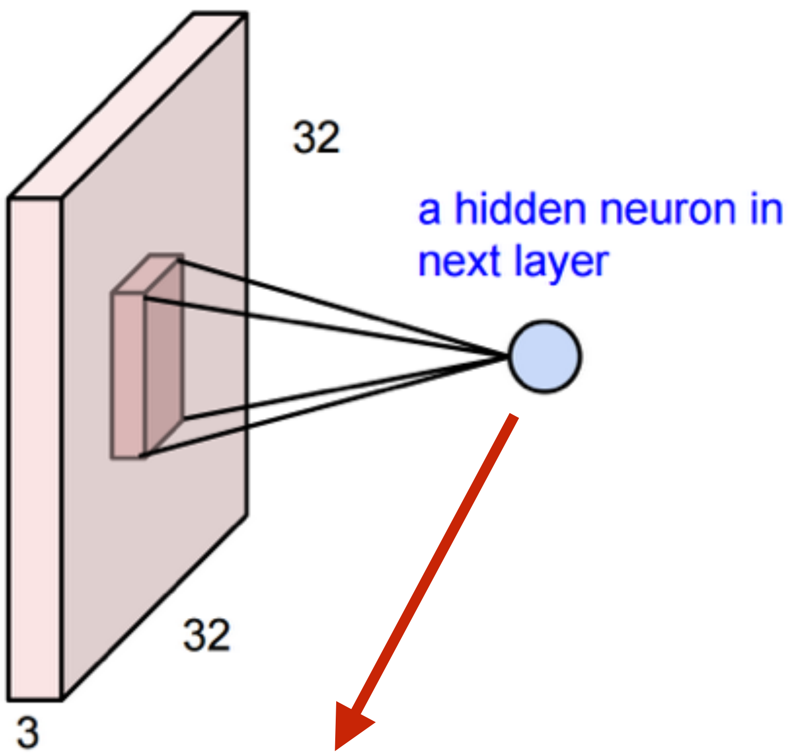


Let's code this up in NumPy

```
out[n, 0, r, c] =
```

n^{th} example

3D Activations



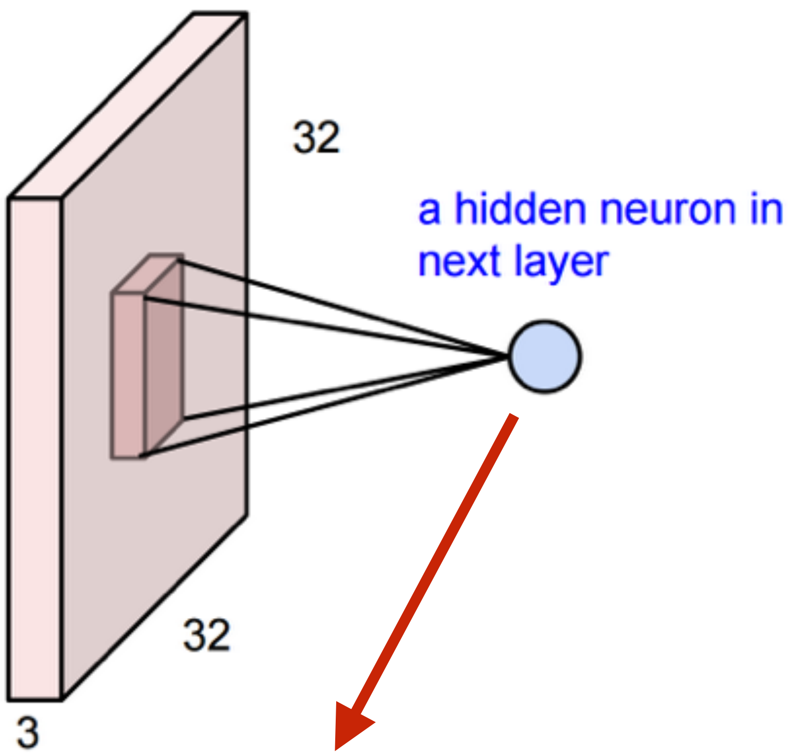
Let's code this up in NumPy

```
out[n, 0, r, c] =
```

first filter

n^{th} example

3D Activations



Let’s code this up in NumPy

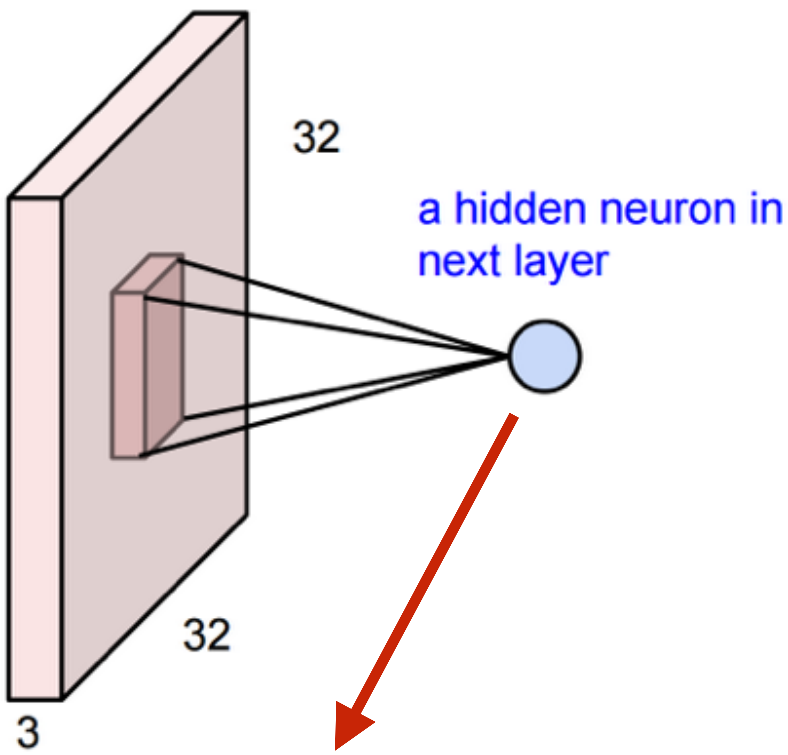
```
out[n, 0, r, c] =
```

output position

first filter

n^{th} example

3D Activations




```
out[n, 0, r, c] = np.sum(
```

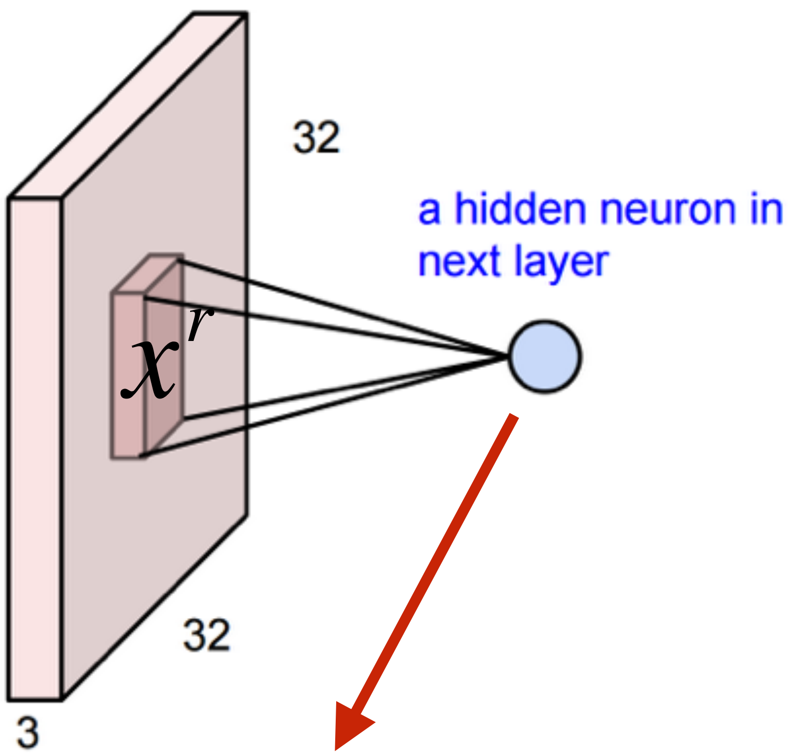
output position

first filter

n^{th} example

Let's code this up in NumPy

3D Activations



Let's code this up in NumPy

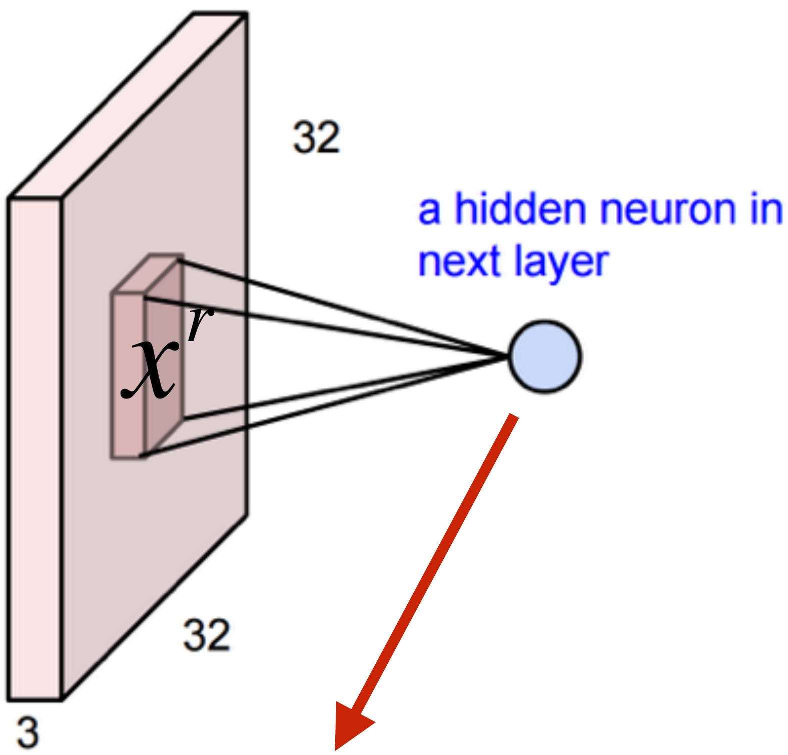
```
out[n, 0, r, c] = np.sum(X[n, :, r0:r1, c0:c1]
```

output position

first filter

nth example

3D Activations



Let's code this up in NumPy

```
out[n, 0, r, c] = np.sum(X[n, :, r0:r1, c0:c1]
```

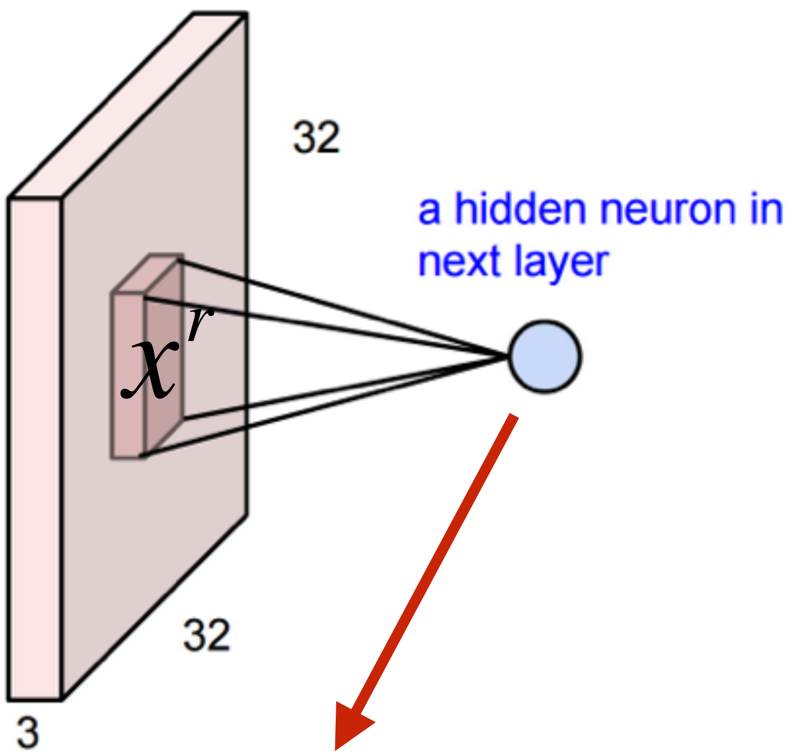
output position

first filter

n^{th} example

n^{th} example

3D Activations



Let's code this up in NumPy

```
out[n, 0, r, c] = np.sum(X[n, :, r0:r1, c0:c1]
```

output position

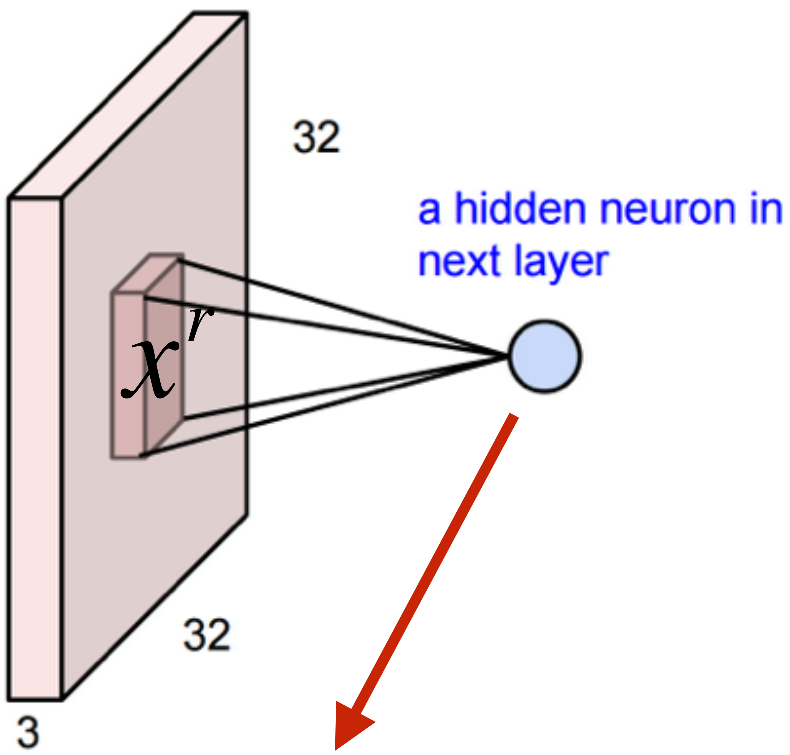
first filter

all input channels

n^{th} example

n^{th} example

3D Activations



Let's code this up in NumPy

```
out[n, 0, r, c] = np.sum(X[n, :, r0:r1, c0:c1]
```

output position

first filter

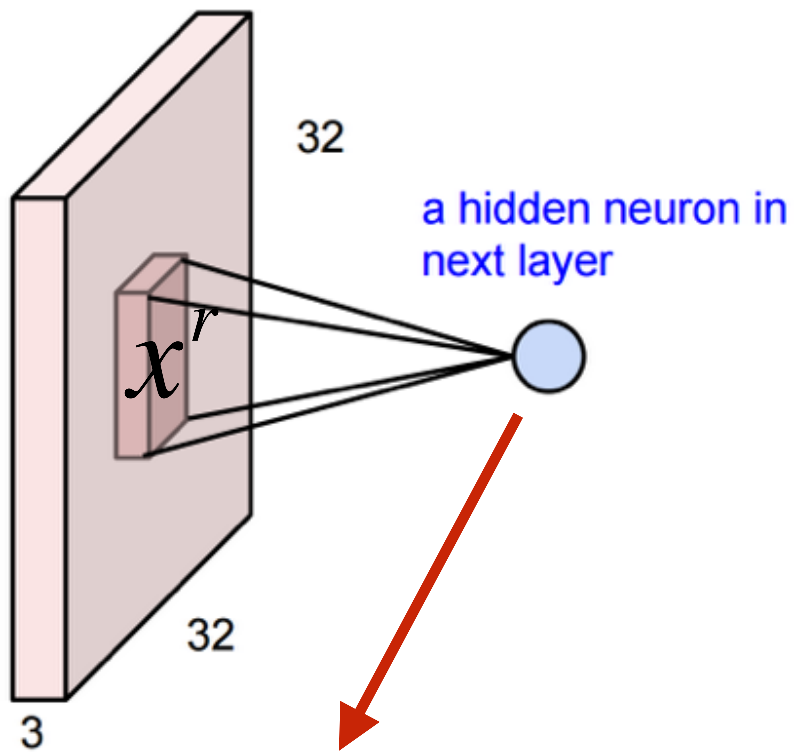
n^{th} example

input region

all input channels

n^{th} example

3D Activations



Let's code this up in NumPy

```
out[n, 0, r, c] = np.sum(X[n, :, r0:r1, c0:c1] * W[0, :, :, :]) + b[0]
```

output position

first filter

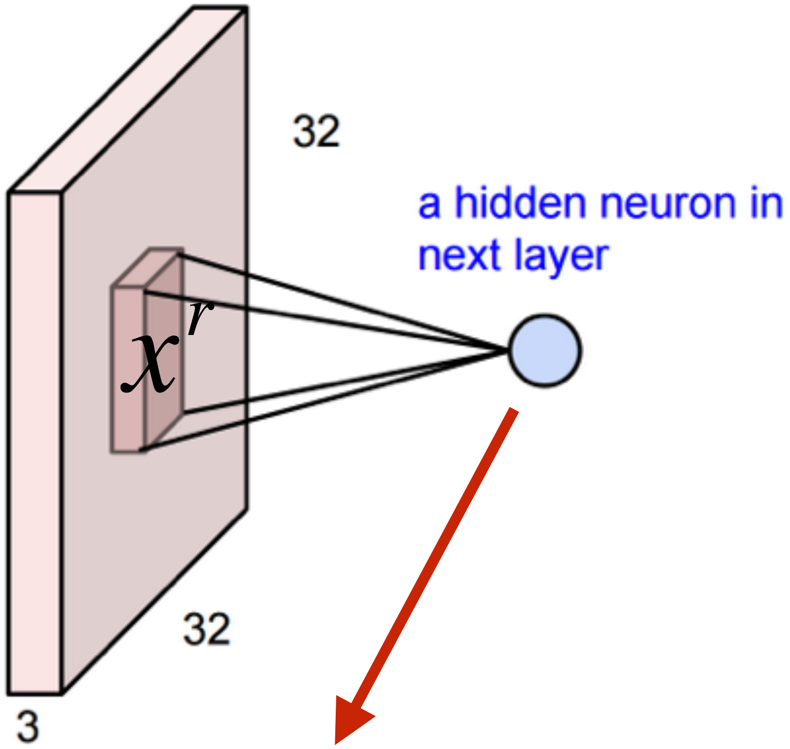
n^{th} example

input region

all input channels

n^{th} example

3D Activations



Let's code this up in NumPy

```
out[n, 0, r, c] = np.sum(X[n, :, r0:r1, c0:c1] * W[0, :, :, :]) + b[0]
```

output position

first filter

n^{th} example

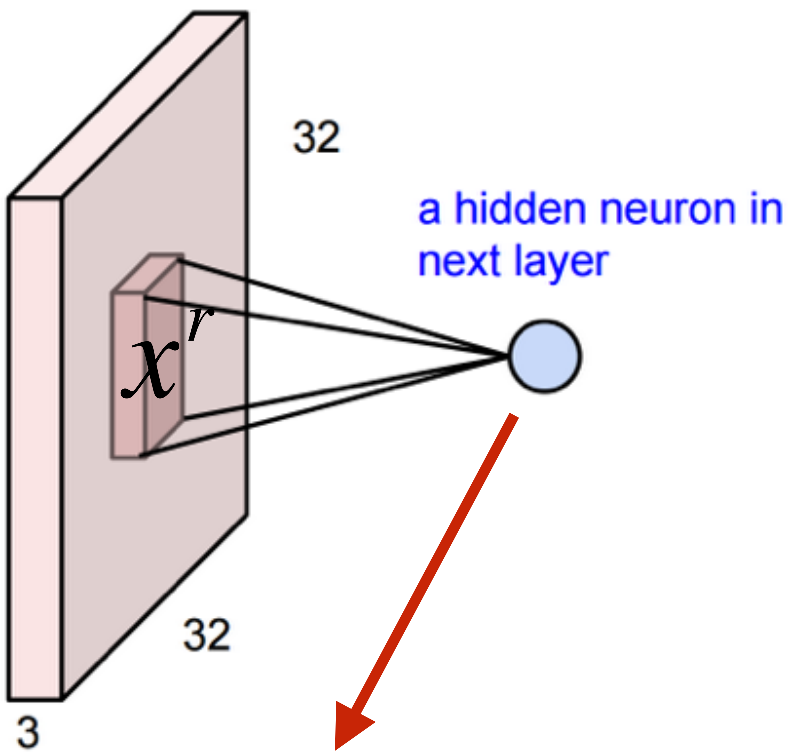
input region

all input channels

n^{th} example

first filter

3D Activations



Let's code this up in NumPy

```
out[n, 0, r, c] = np.sum(X[n, :, r0:r1, c0:c1] * W[0, :, :, :]) + b[0]
```

output position

first filter

nth example

input region

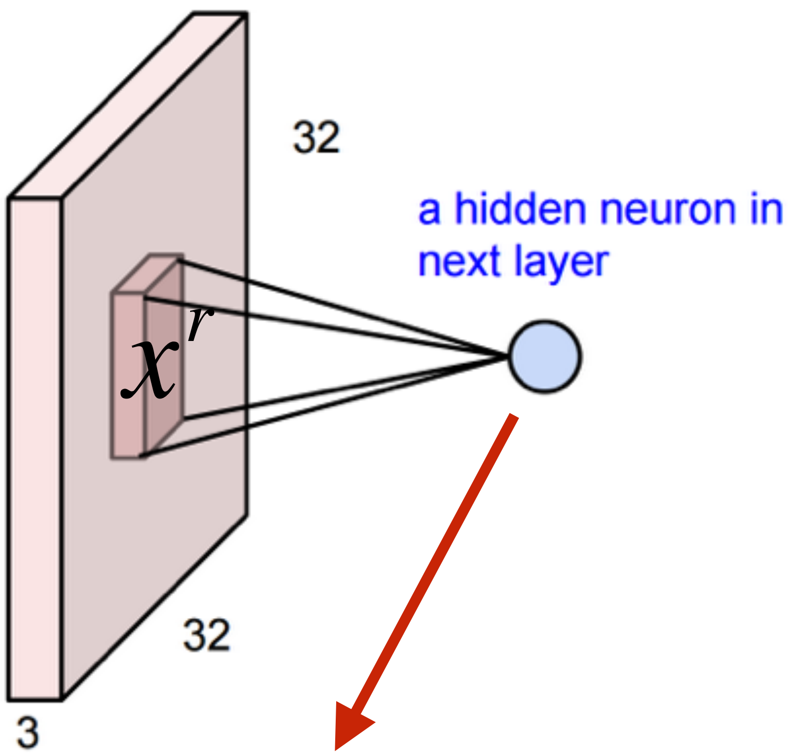
all input channels

nth example

all channels

first filter

3D Activations



Let's code this up in NumPy

```
out[n, 0, r, c] = np.sum(X[n, :, r0:r1, c0:c1] * W[0, :, :, :]) + b[0]
```

output position

first filter

n^th example

input region

all input channels

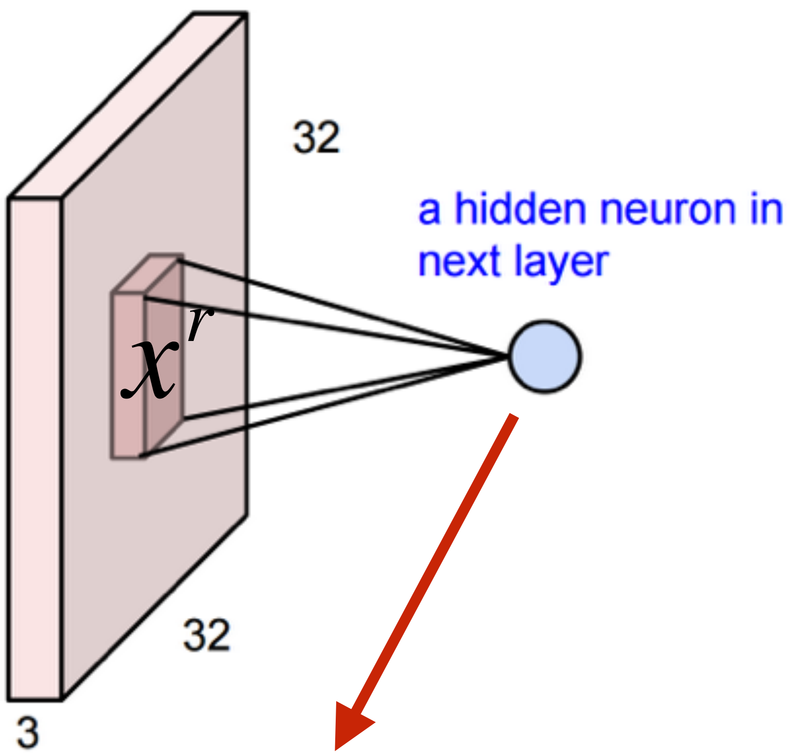
n^th example

all positions

all channels

first filter

3D Activations



Let's code this up in NumPy

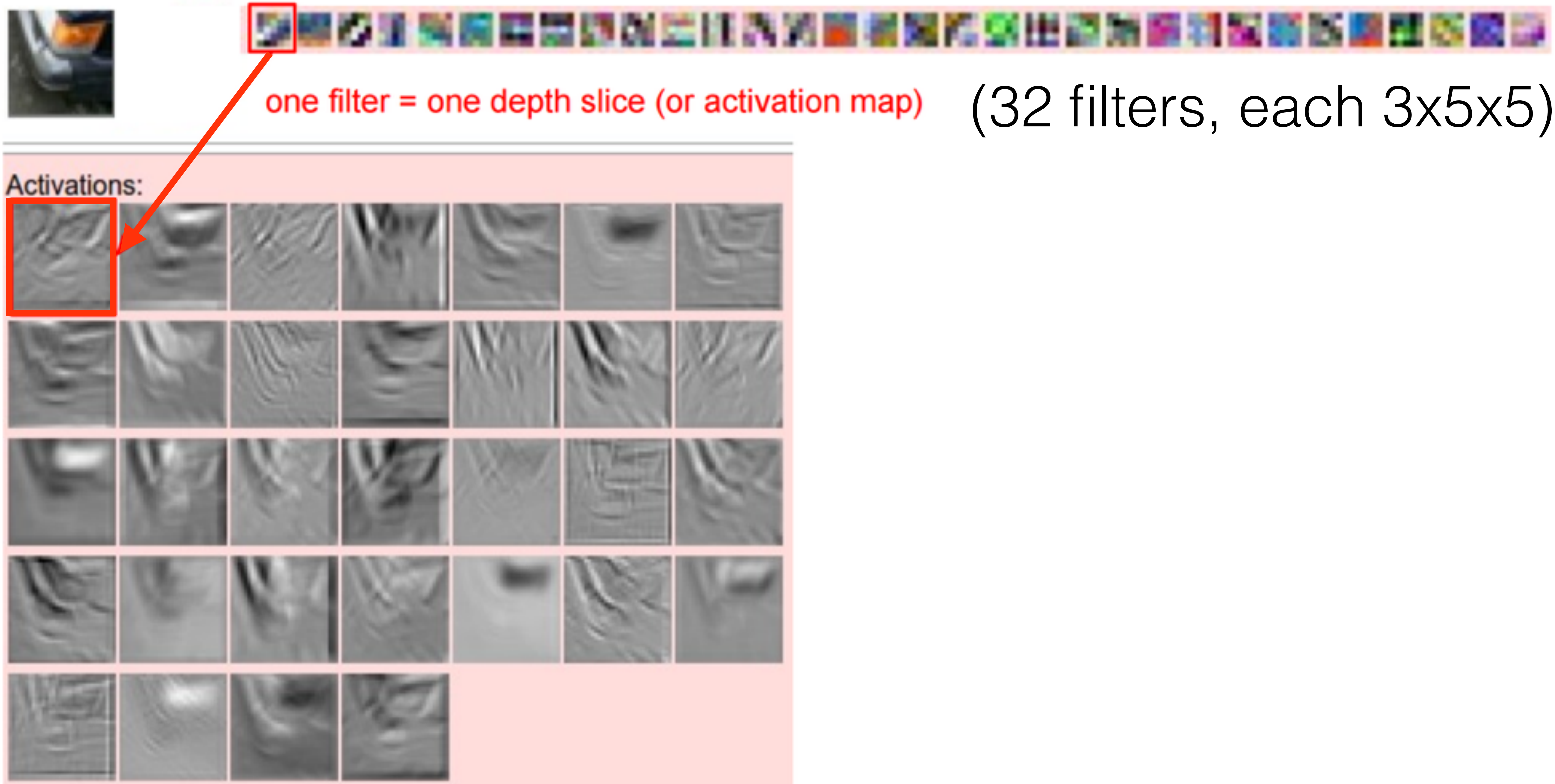
```
out[n, 0, r, c] = np.sum(X[n, :, r0:r1, c0:c1] * W[0, :, :, :]) + b[0]
```

- out[n, ...]: n^{th} example
- out[..., 0, ...]: first filter
- out[..., r, c]: output position
- X[n, ...]: n^{th} example
- X[..., :, ...]: all input channels
- X[..., r0:r1, c0:c1]: input region
- W[0, ...]: first filter
- W[..., :, ...]: all channels
- W[..., :, :]: all positions
- b[0]: bias

3D Activations

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

(Input)



(32 filters, each 3x5x5)

Figure: Andrej Karpathy

3D Activations

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

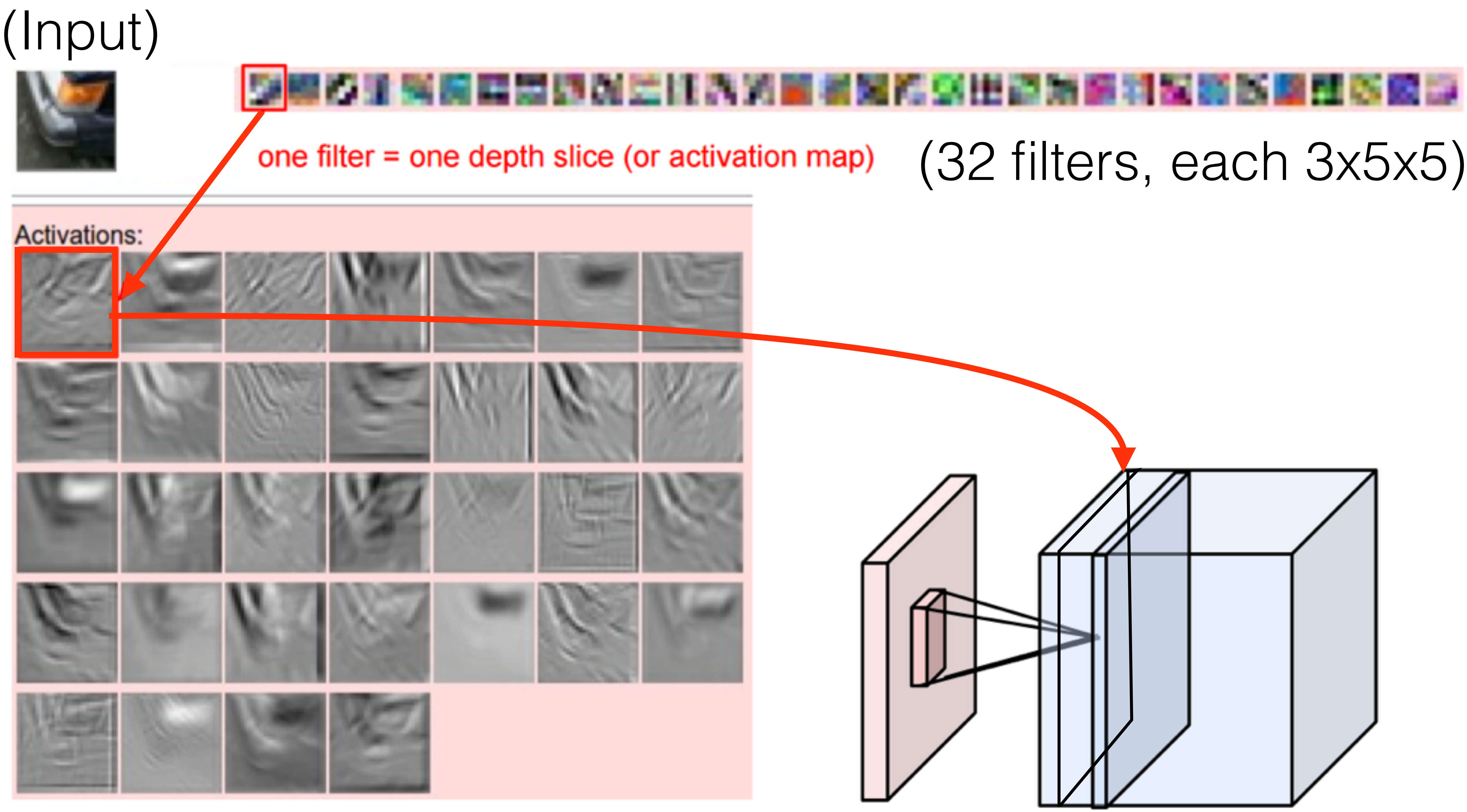


Figure: Andrej Karpathy

3D Activations

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

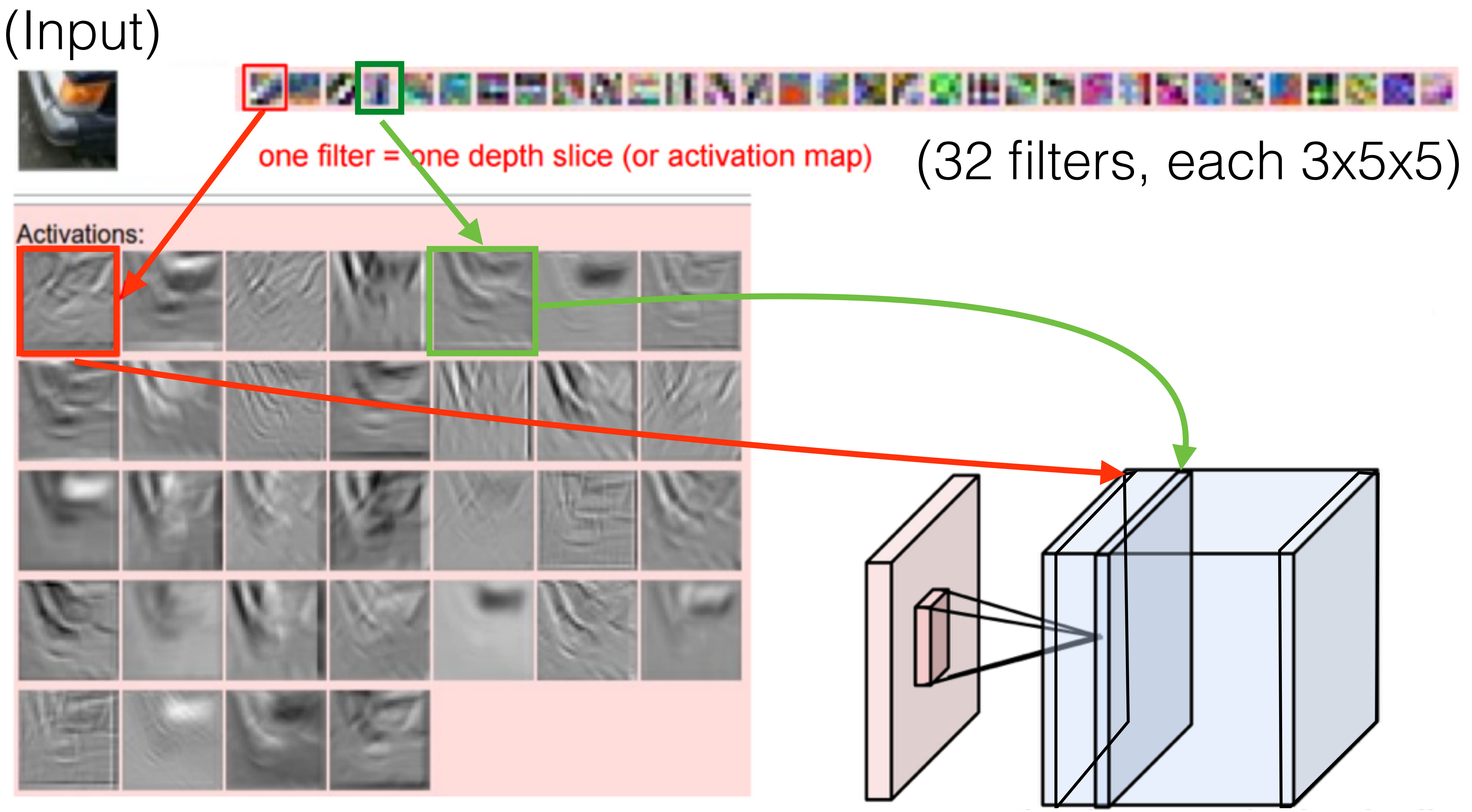


Figure: Andrej Karpathy

3D Activations

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

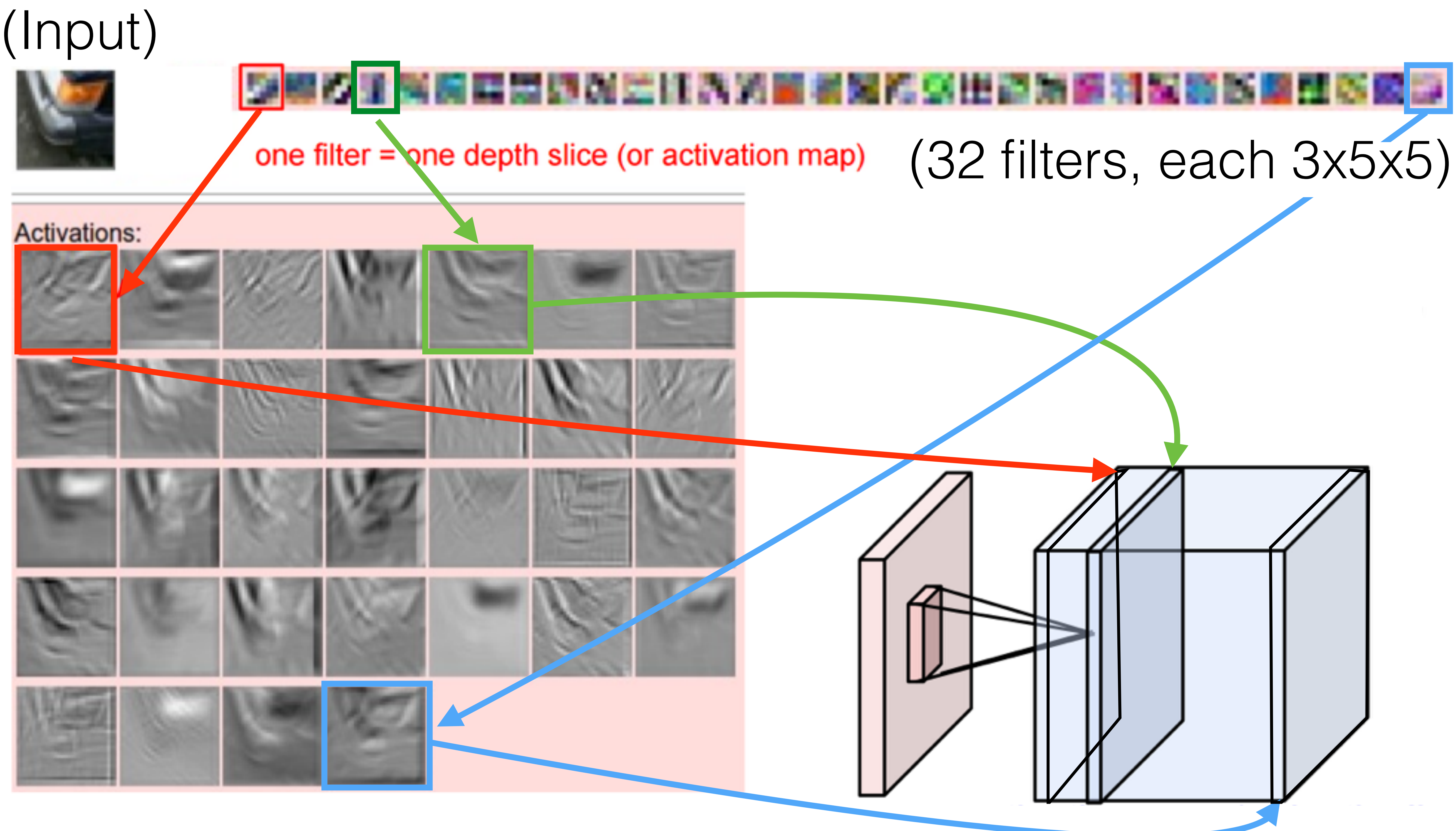
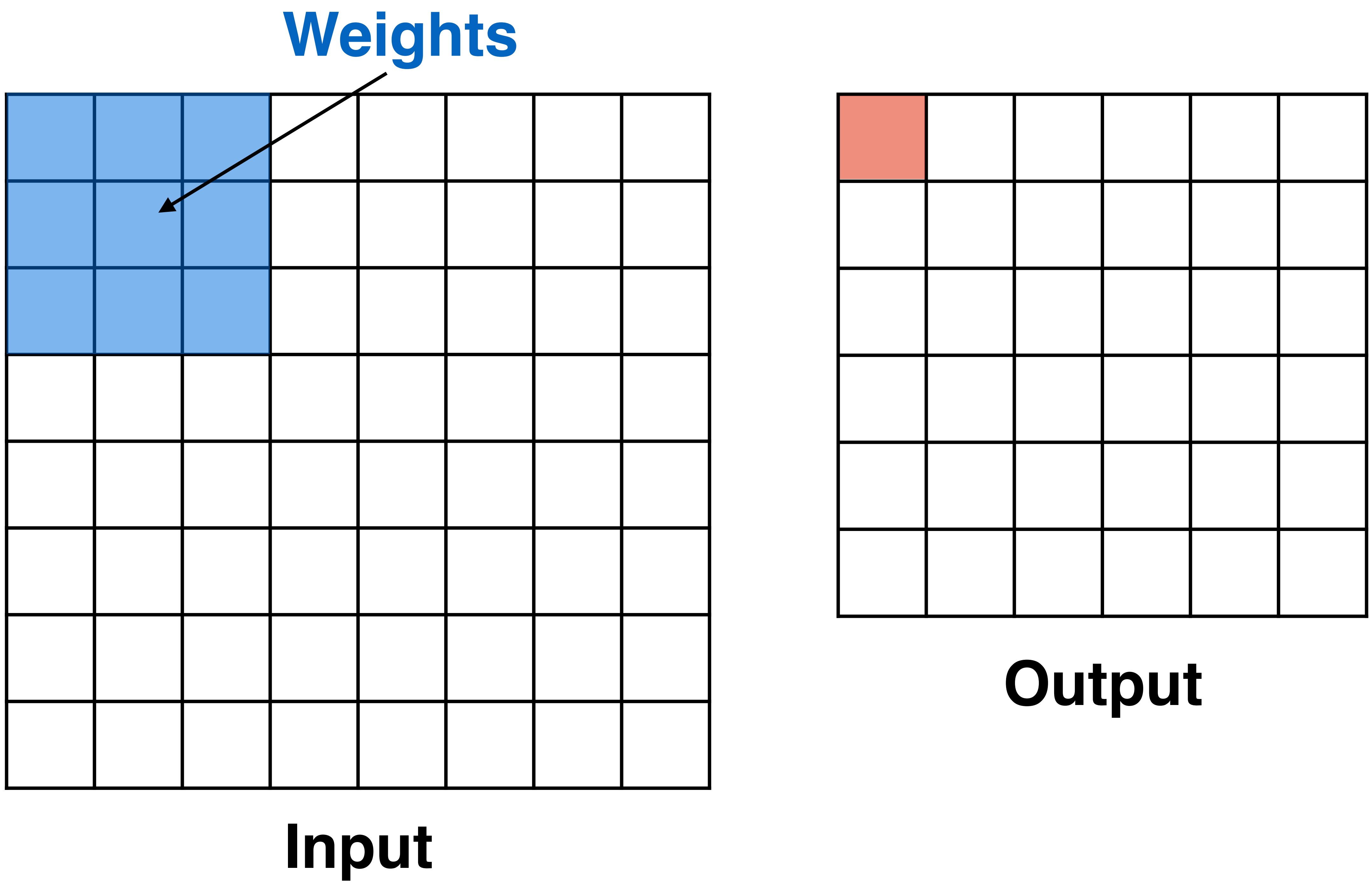


Figure: Andrej Karpathy

Questions?

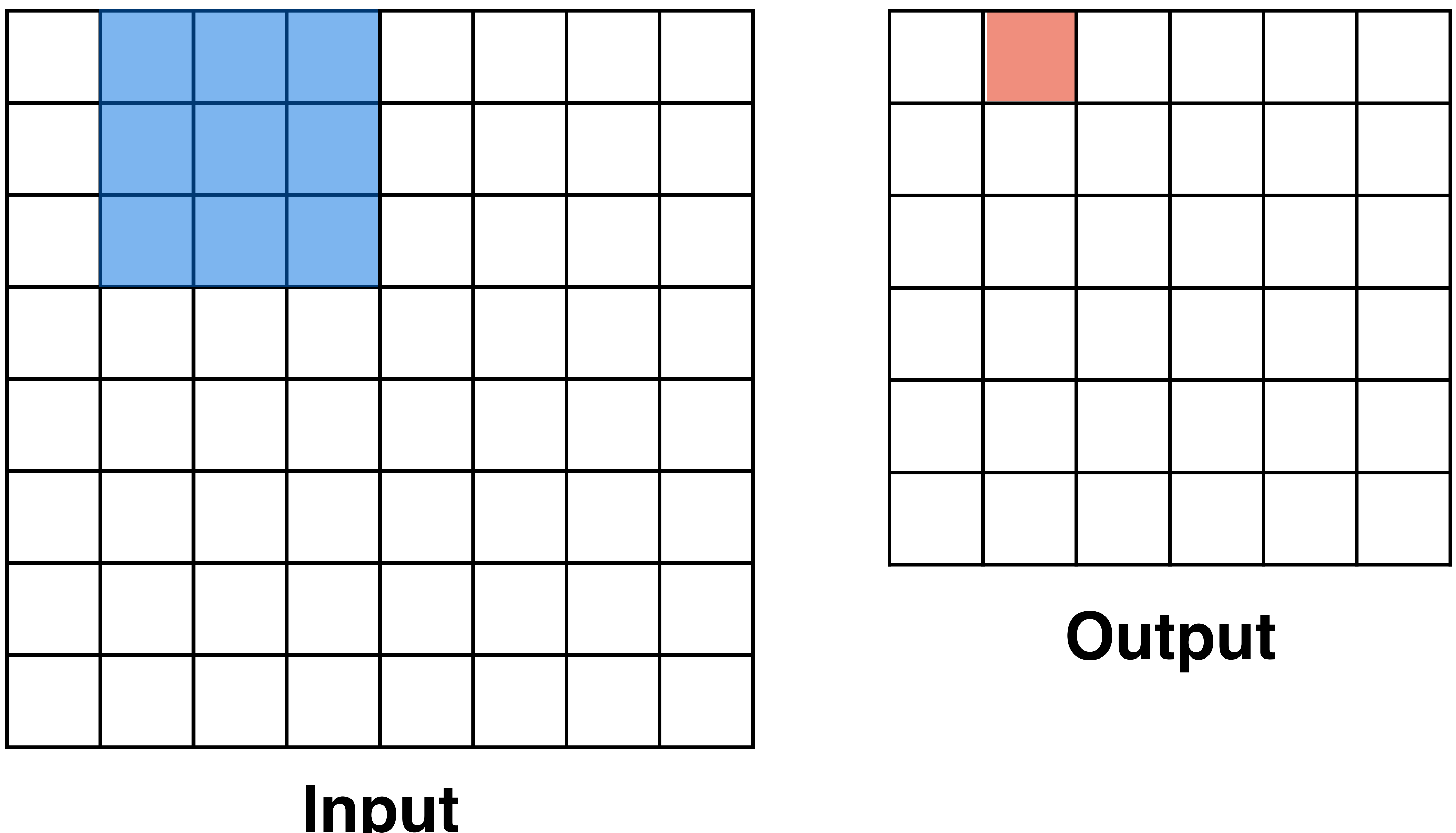
Convolution: Stride

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



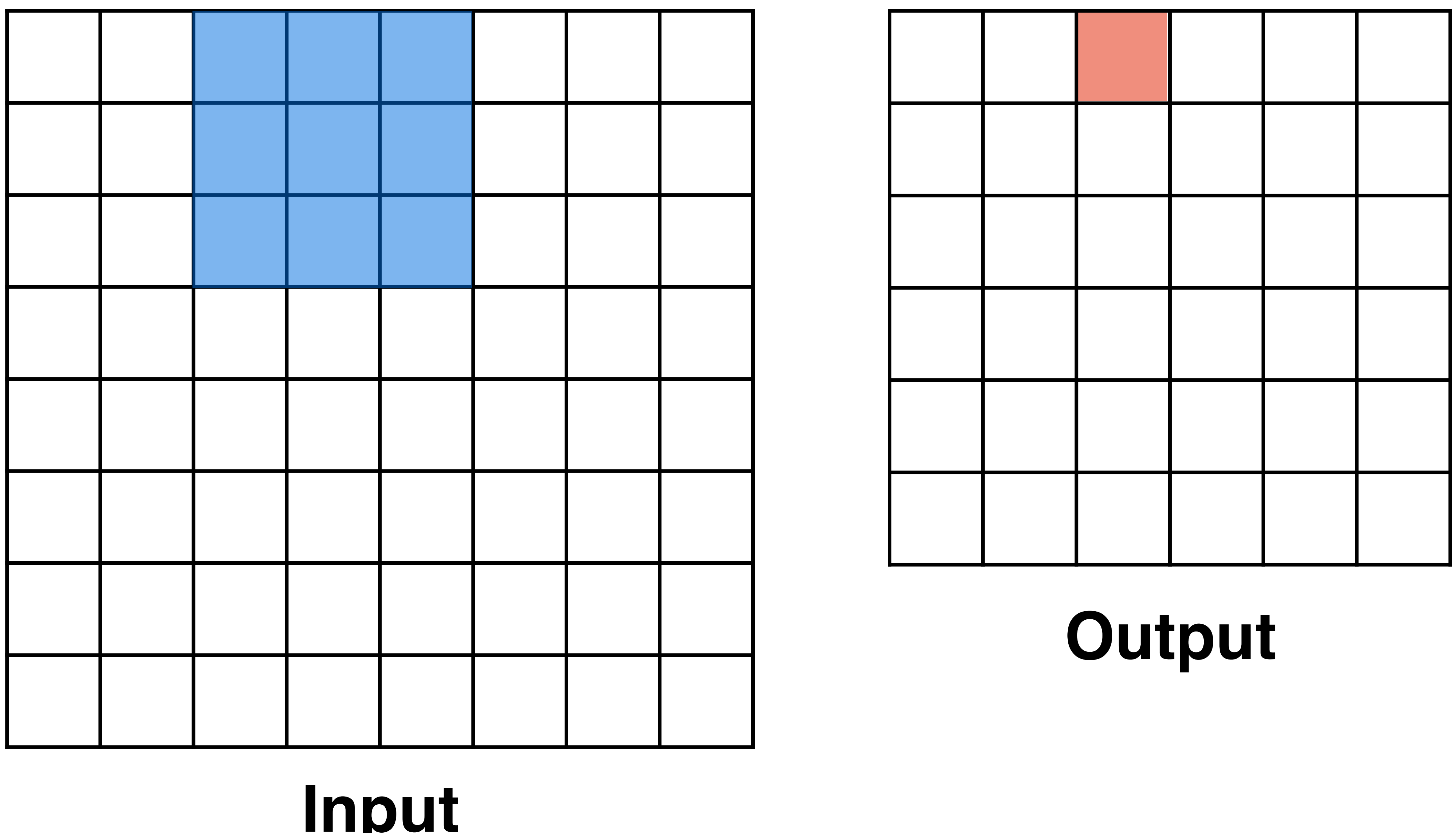
Convolution: Stride

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



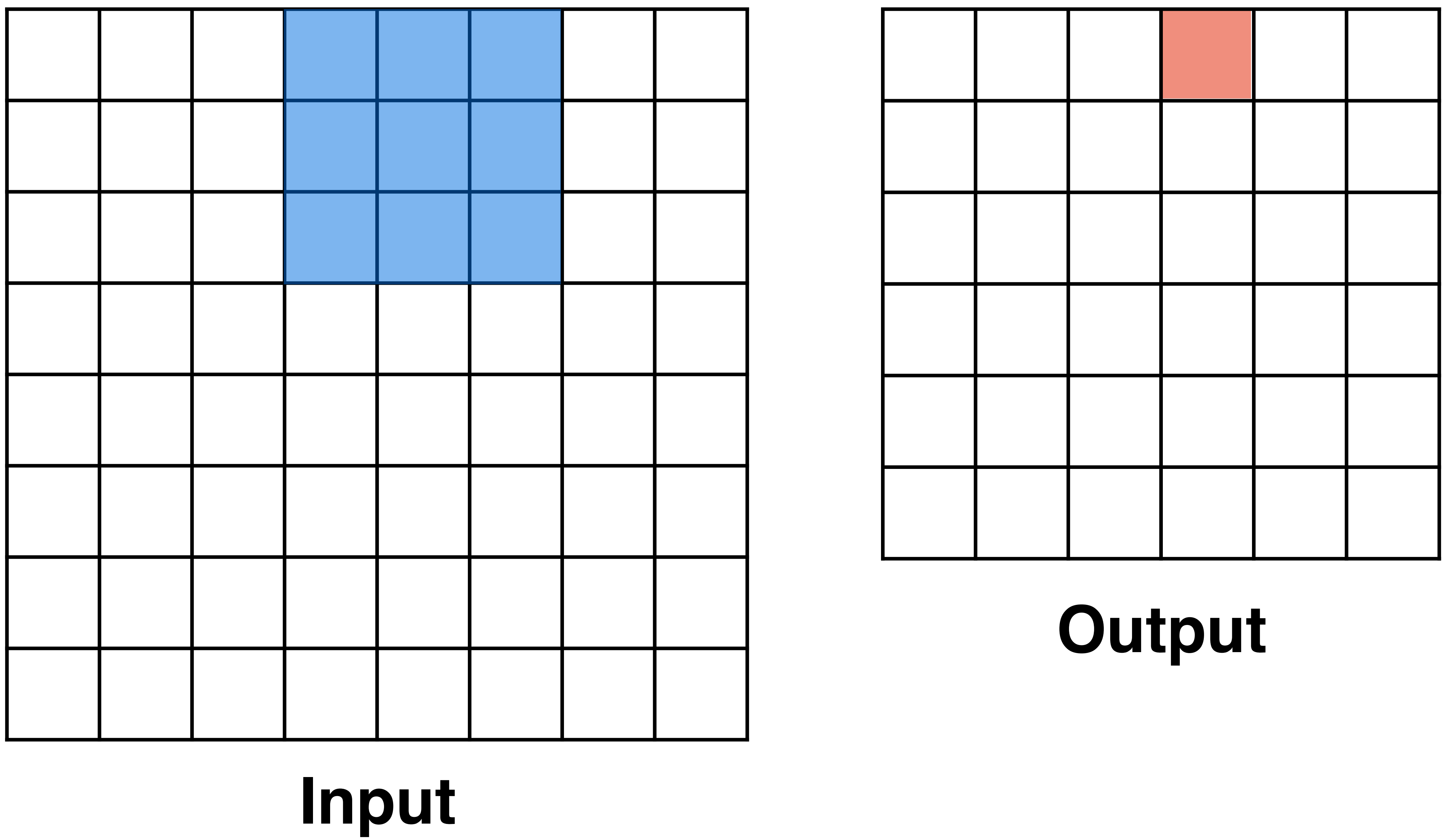
Convolution: Stride

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



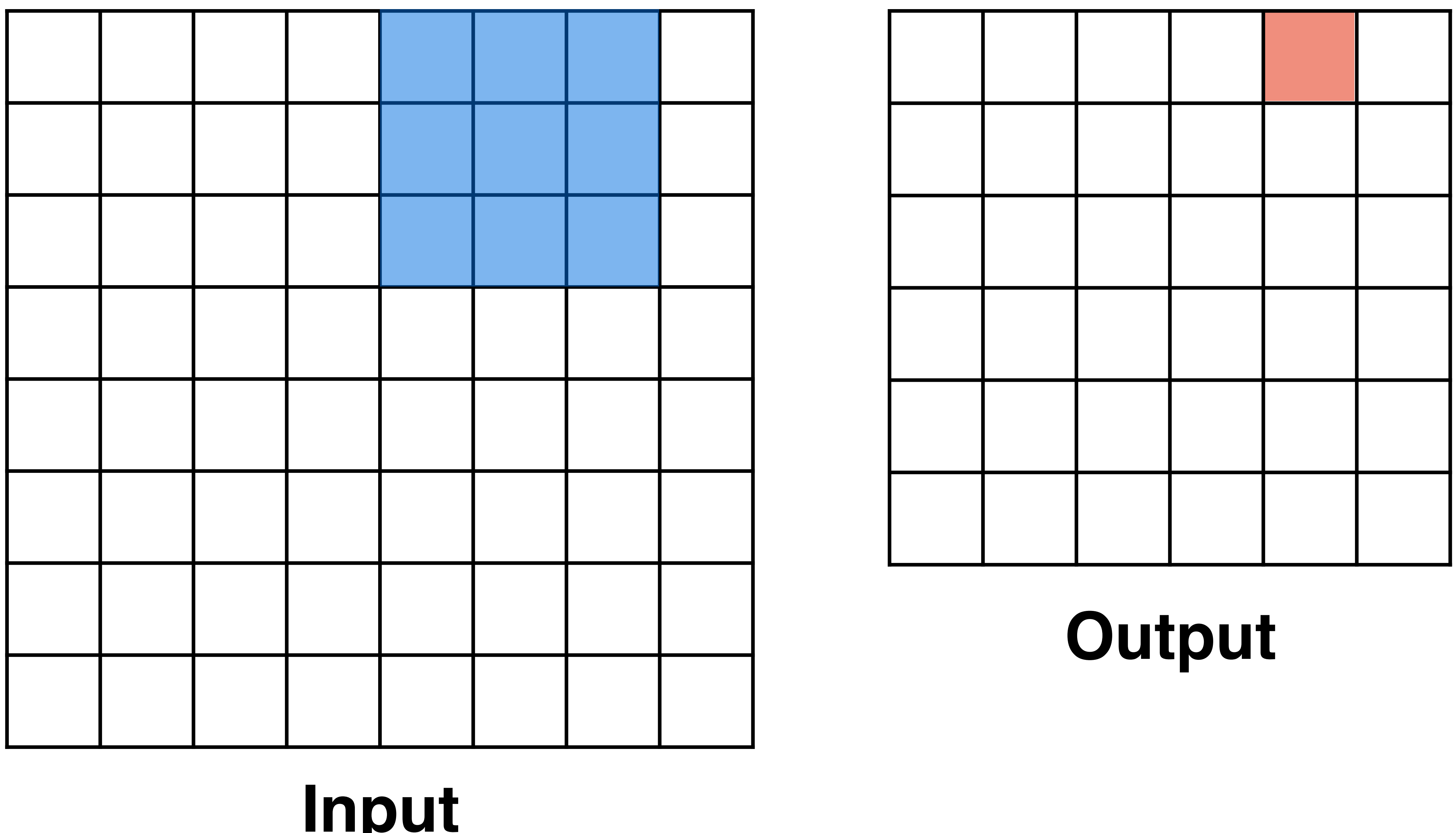
Convolution: Stride

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



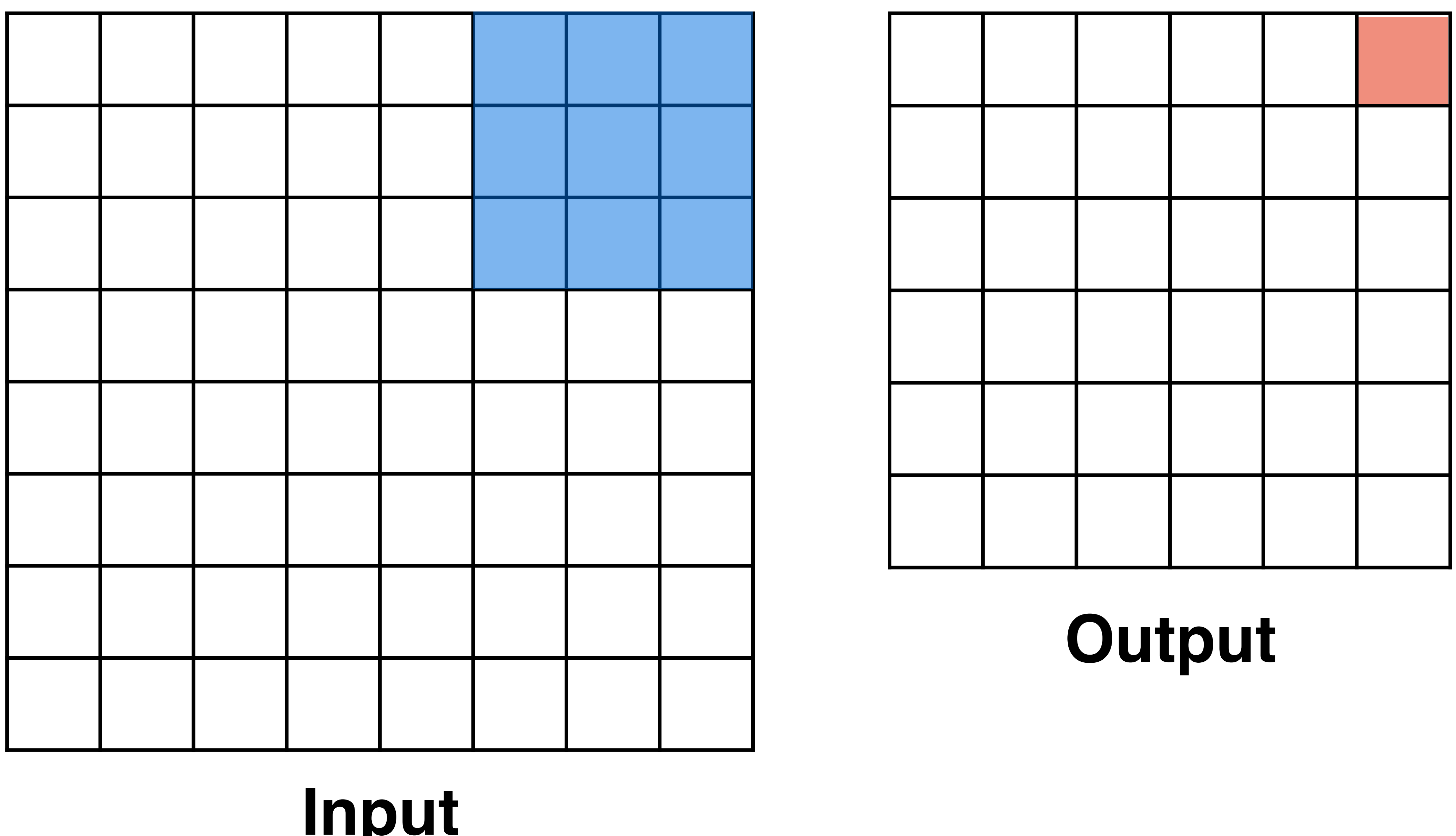
Convolution: Stride

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



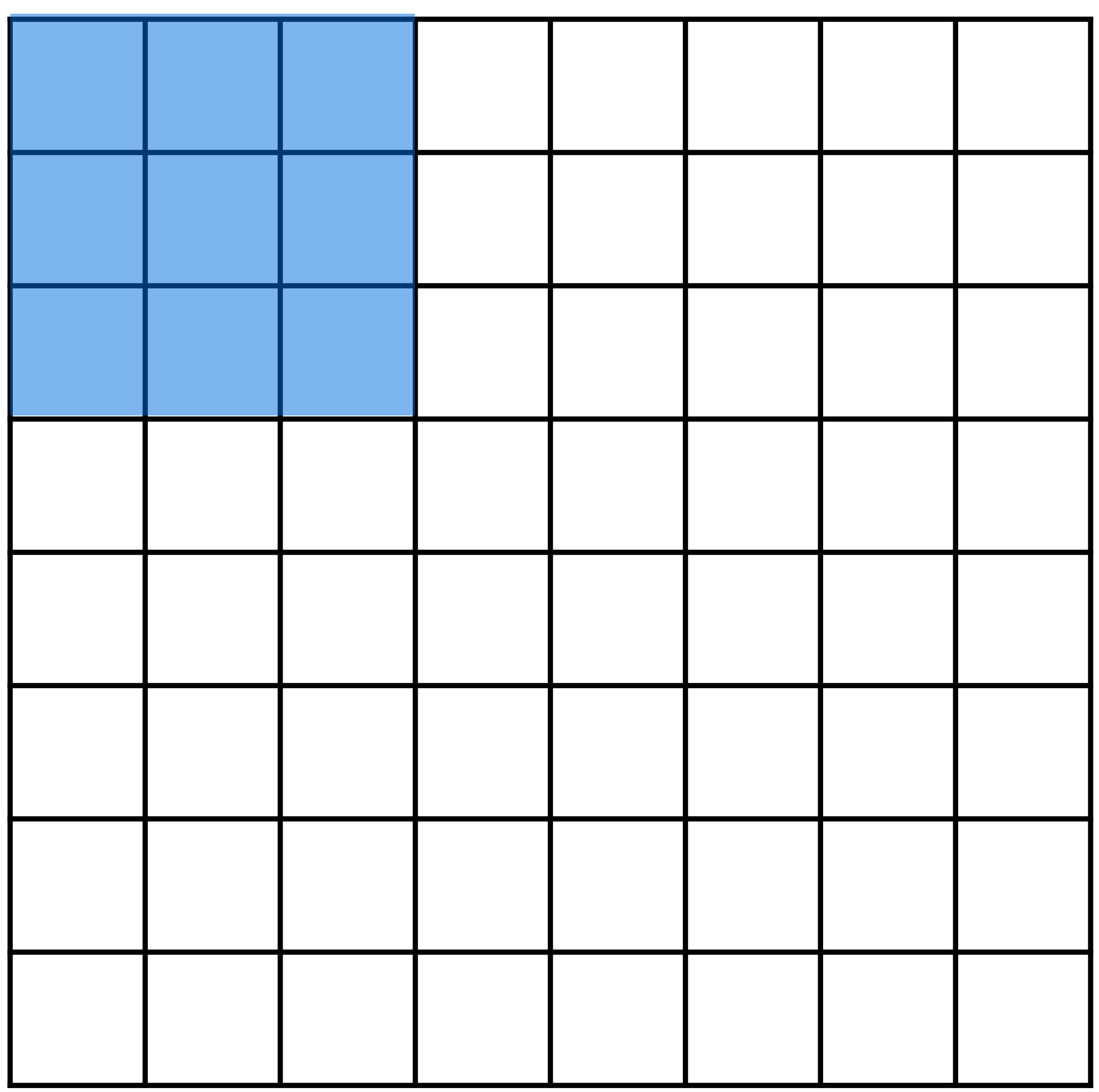
Convolution: Stride

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

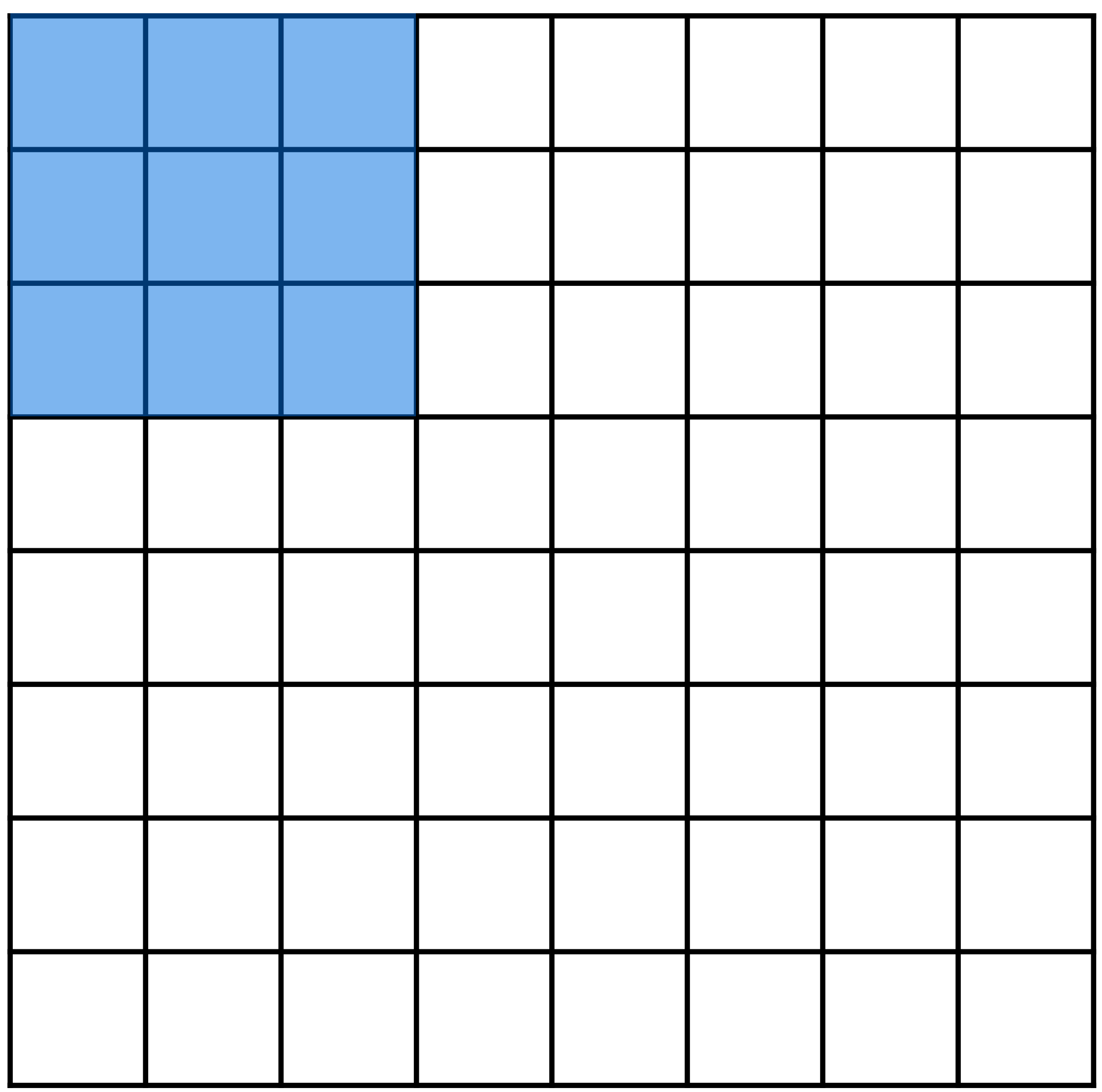
Recall that at each position, we are doing a **3D** sum:

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

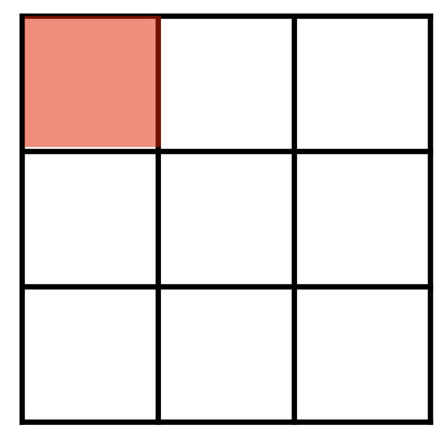
(channel, row, column)

Convolution: Stride

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



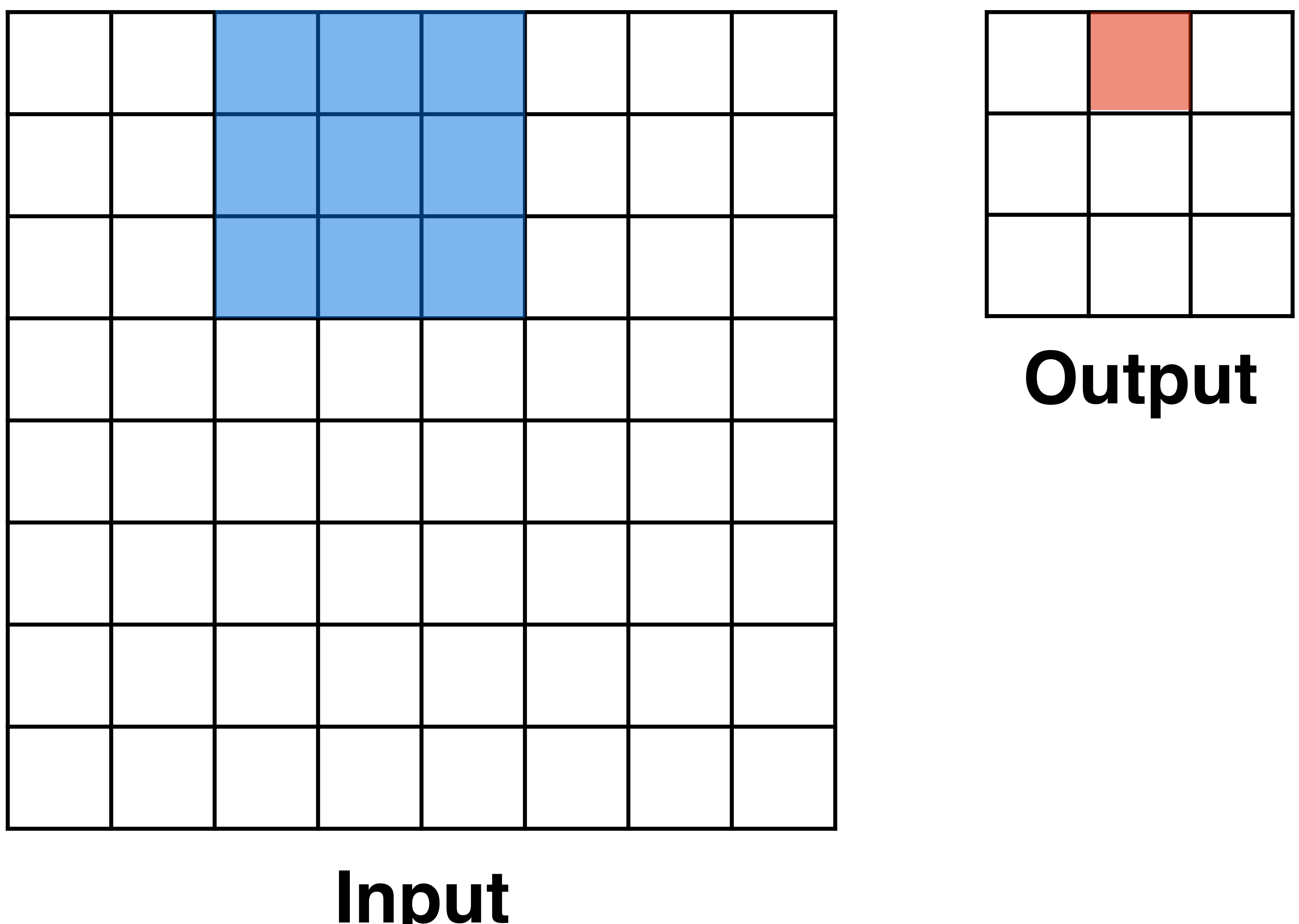
Input



Output

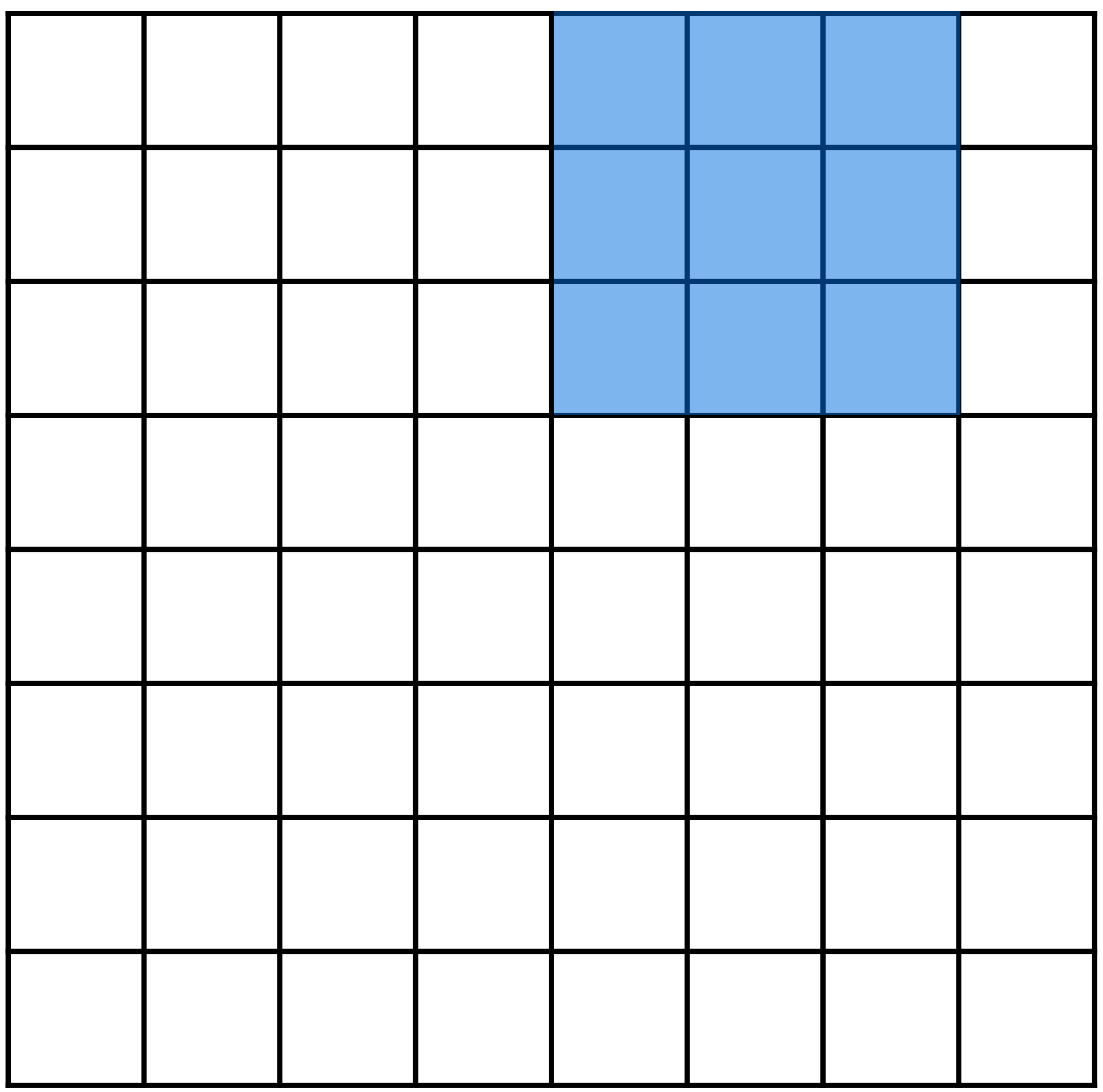
Convolution: Stride

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

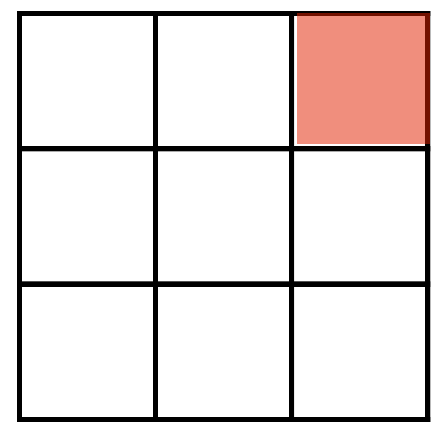


Convolution: Stride

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



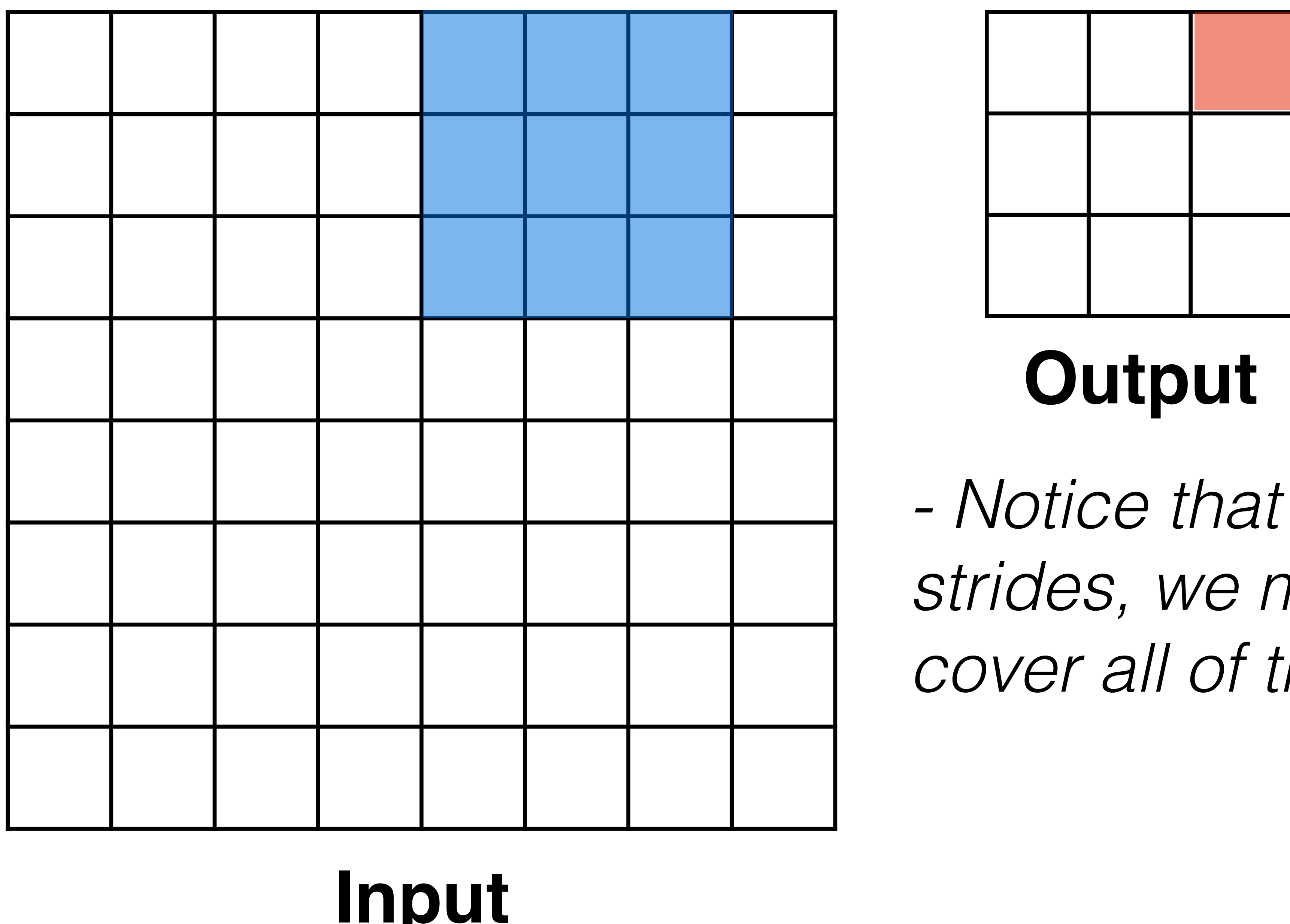
Input



Output

Convolution: Stride

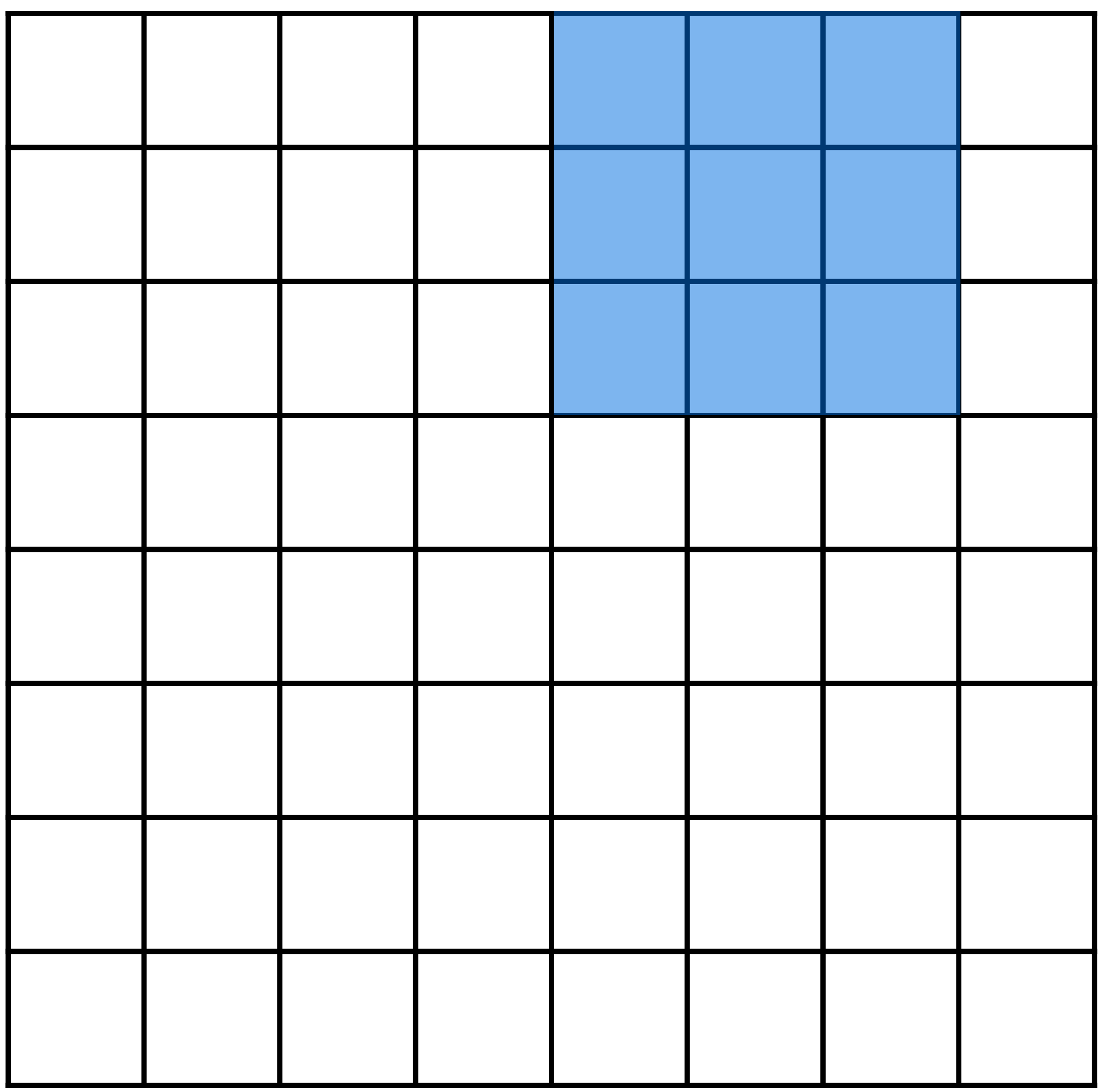
But we can also convolve with a **stride**, e.g. stride = 2



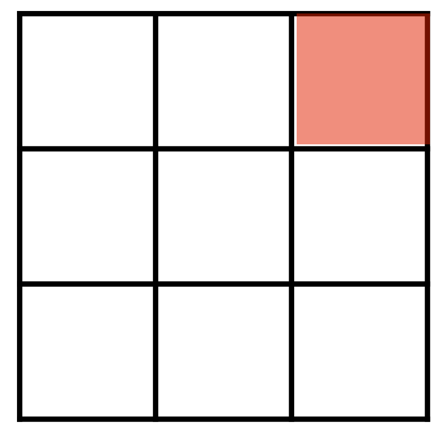
- Notice that with certain strides, we may not be able to cover all of the input

Convolution: Stride

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



Input



Output

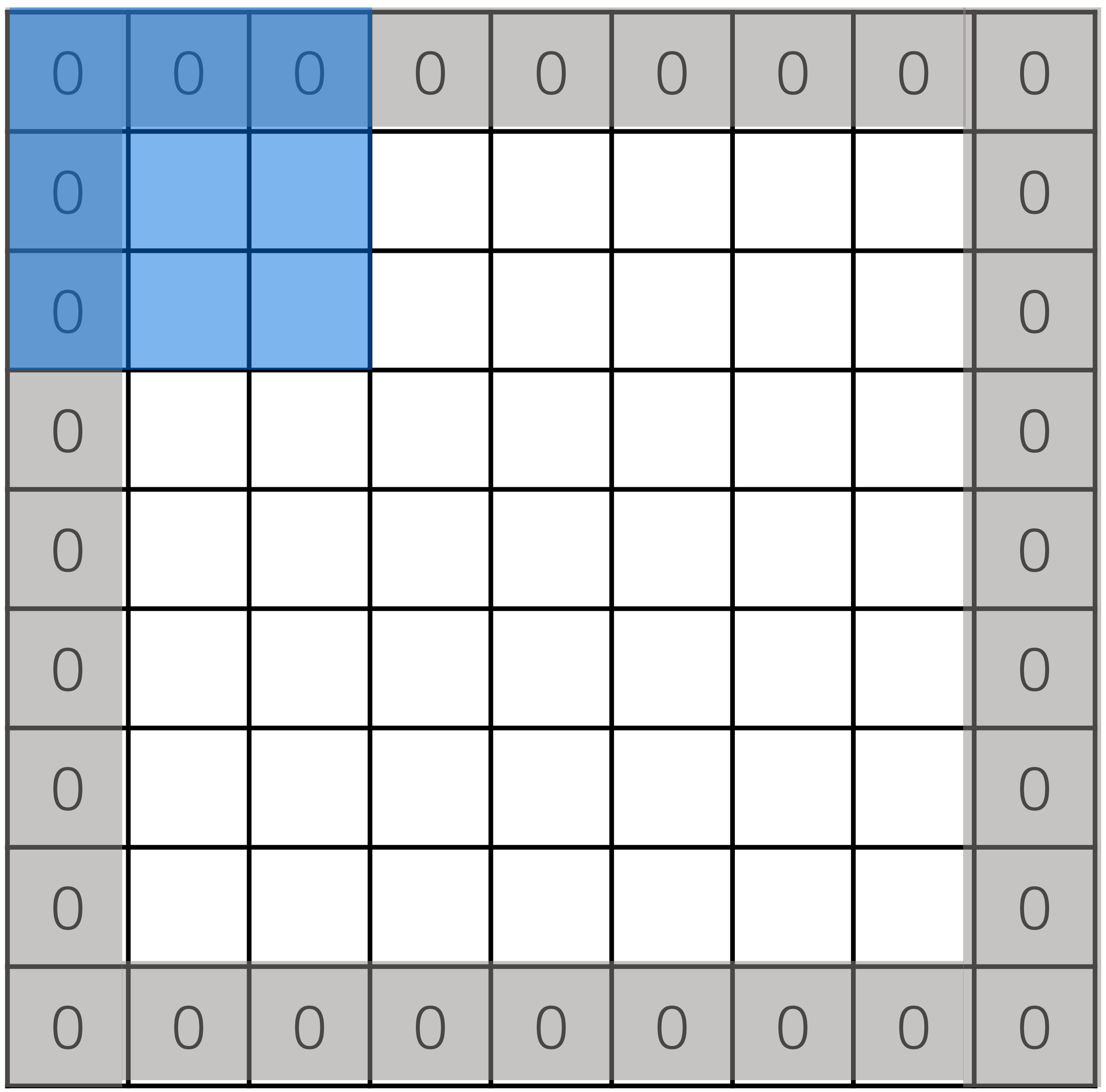
- *Notice that with certain strides, we may not be able to cover all of the input*

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

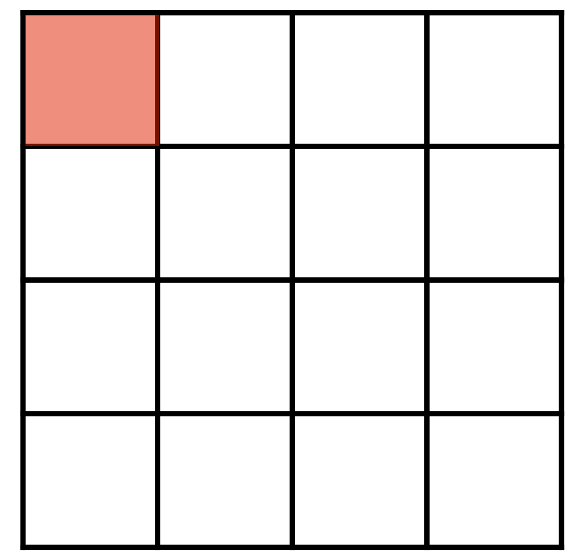
Convolution: Padding

We can also pad the input with zeros.

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



Input

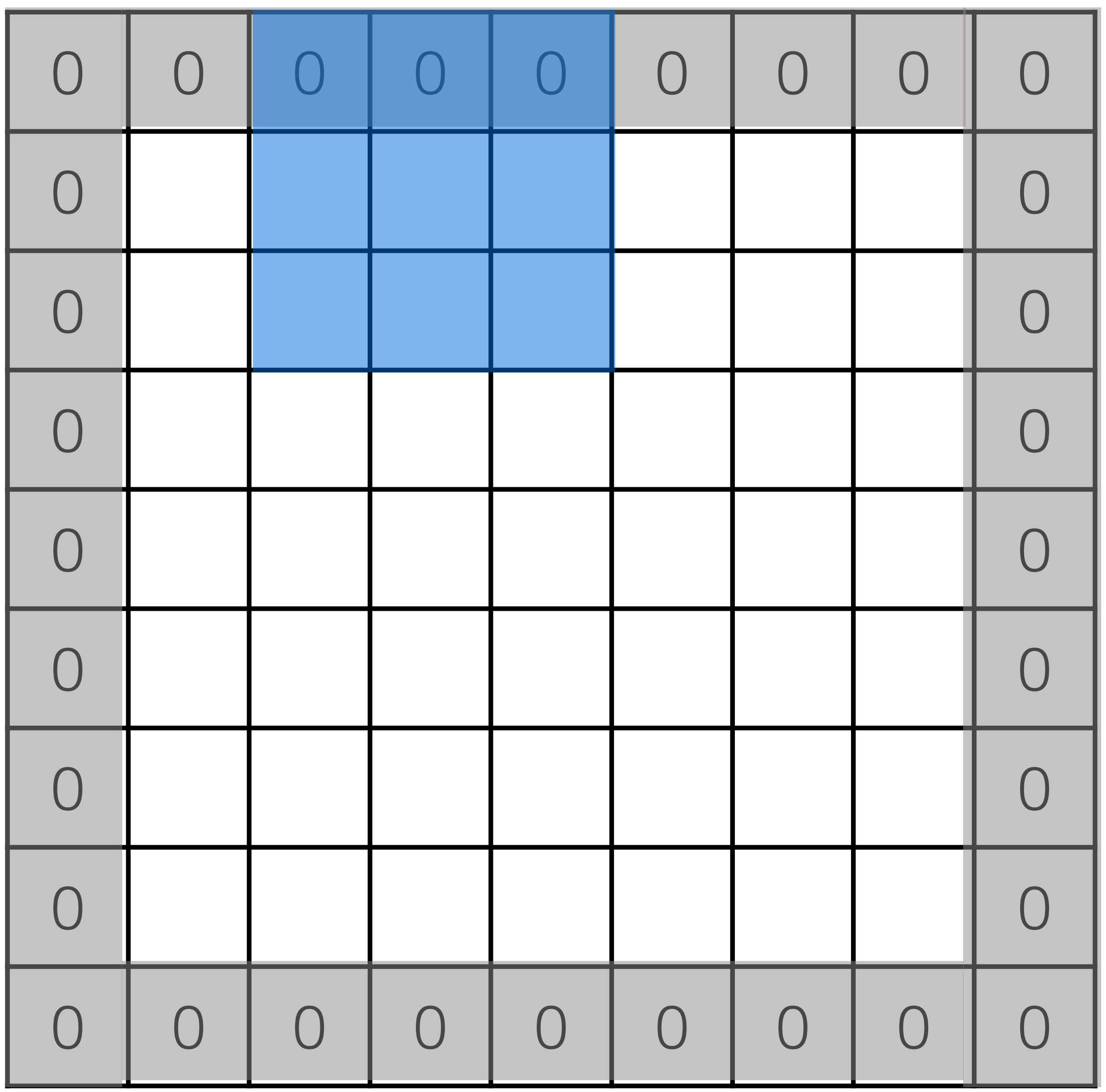


Output

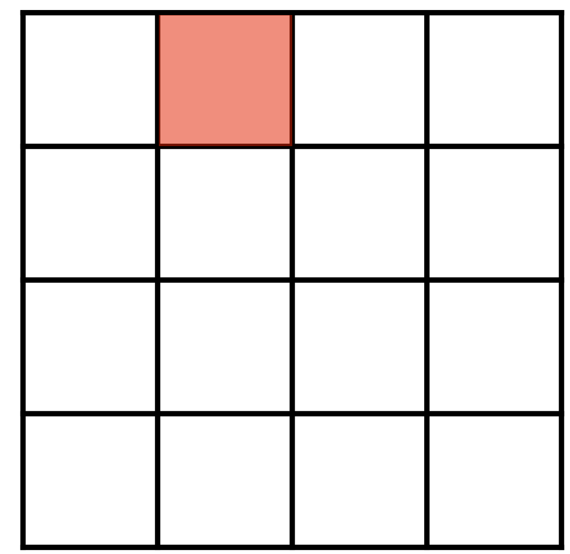
Convolution: Padding

We can also pad the input with zeros.

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



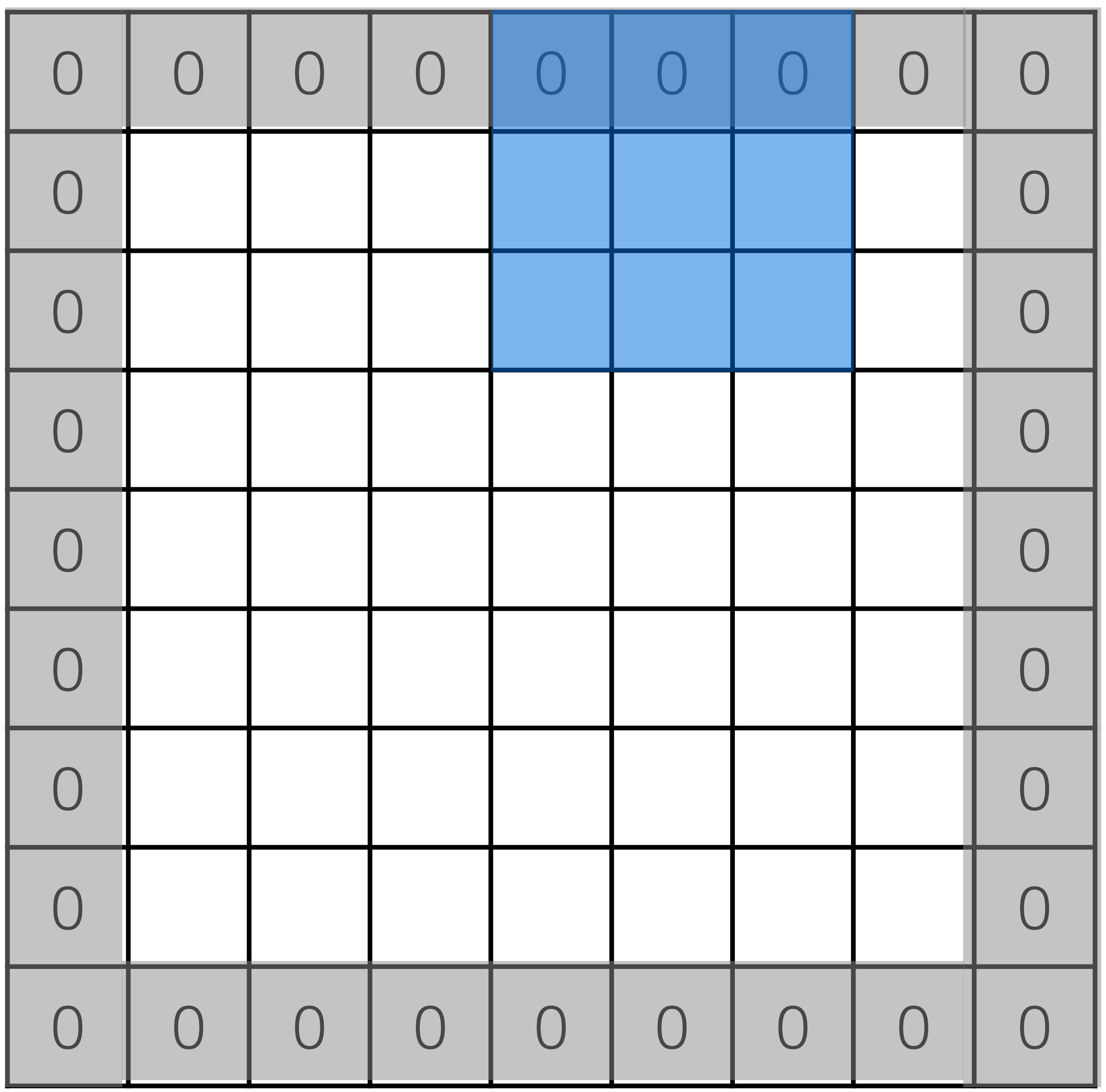
Input



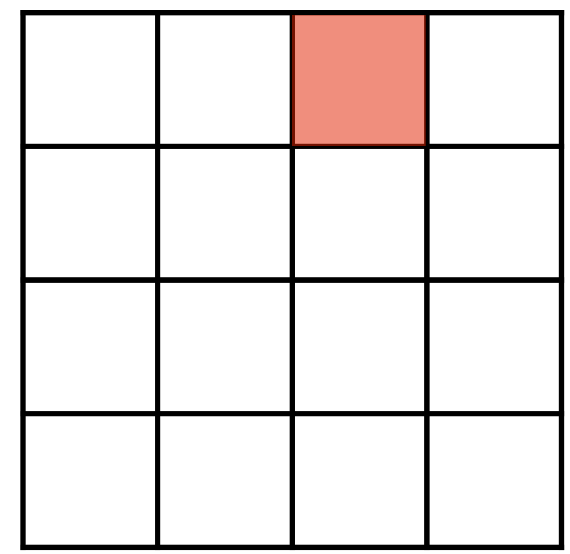
Output

Convolution: Padding

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



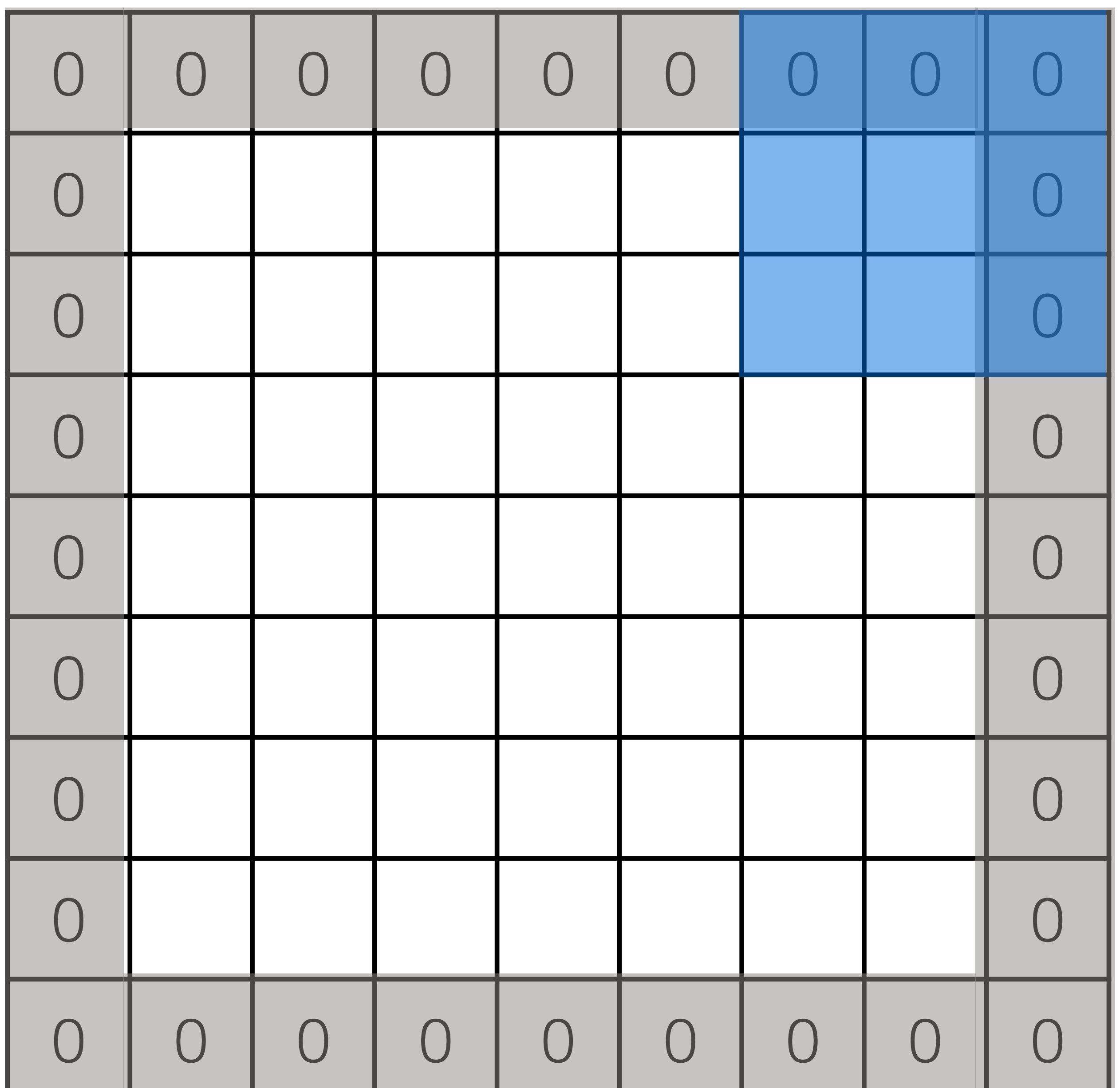
Input



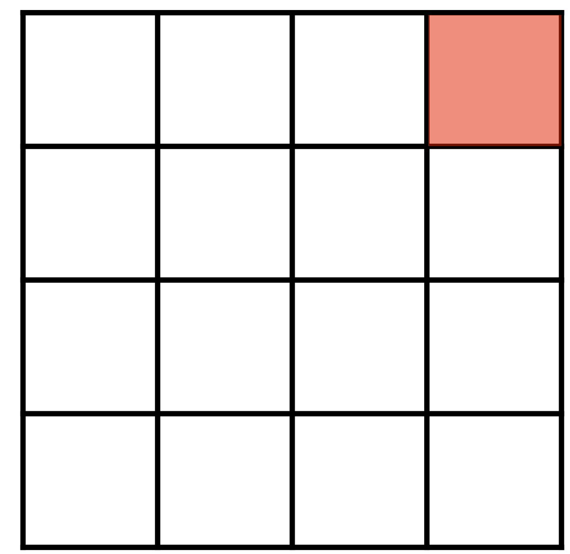
Output

Convolution: Padding

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



Input



Output

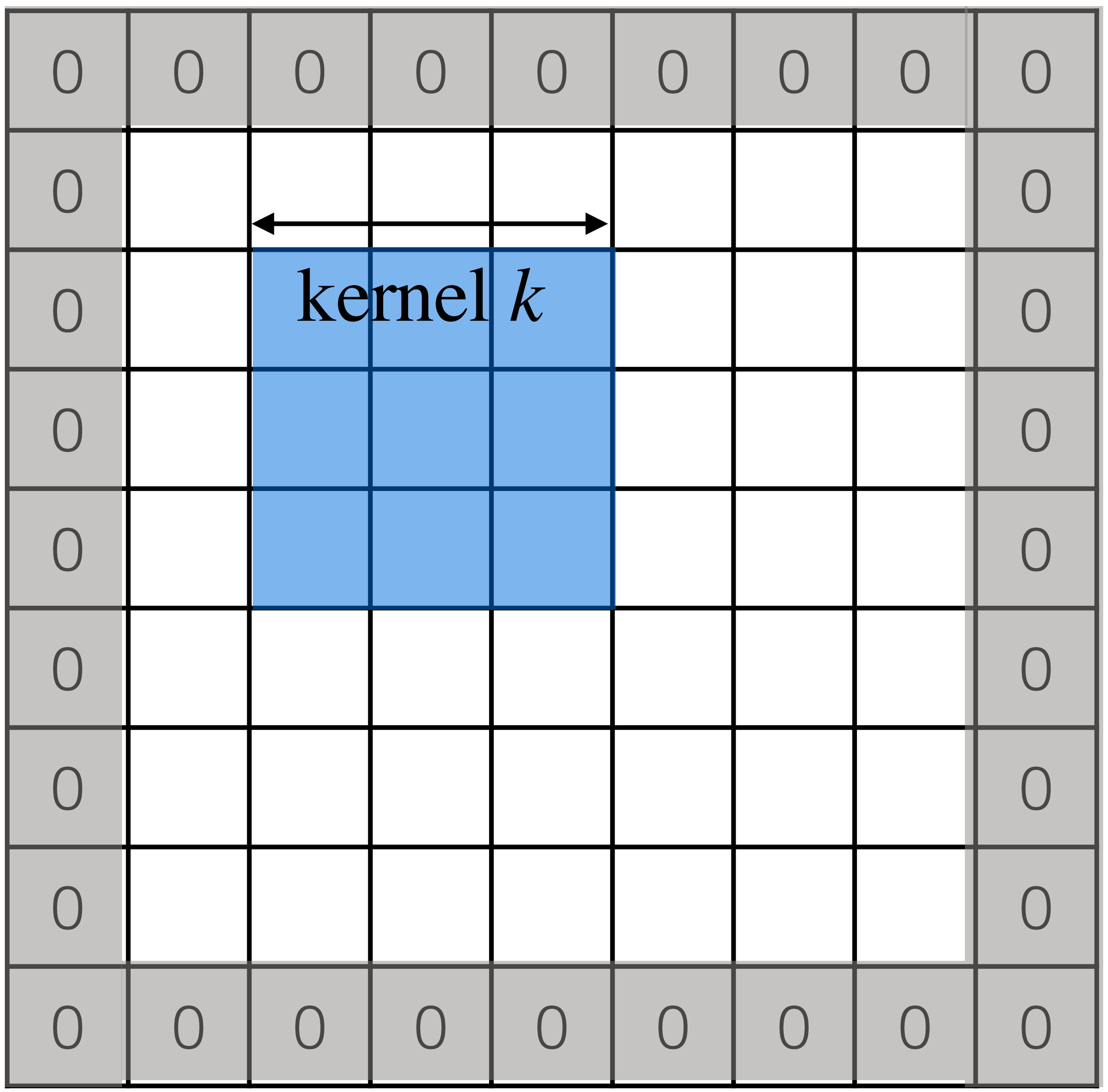
Convolution:

How big is the output?

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

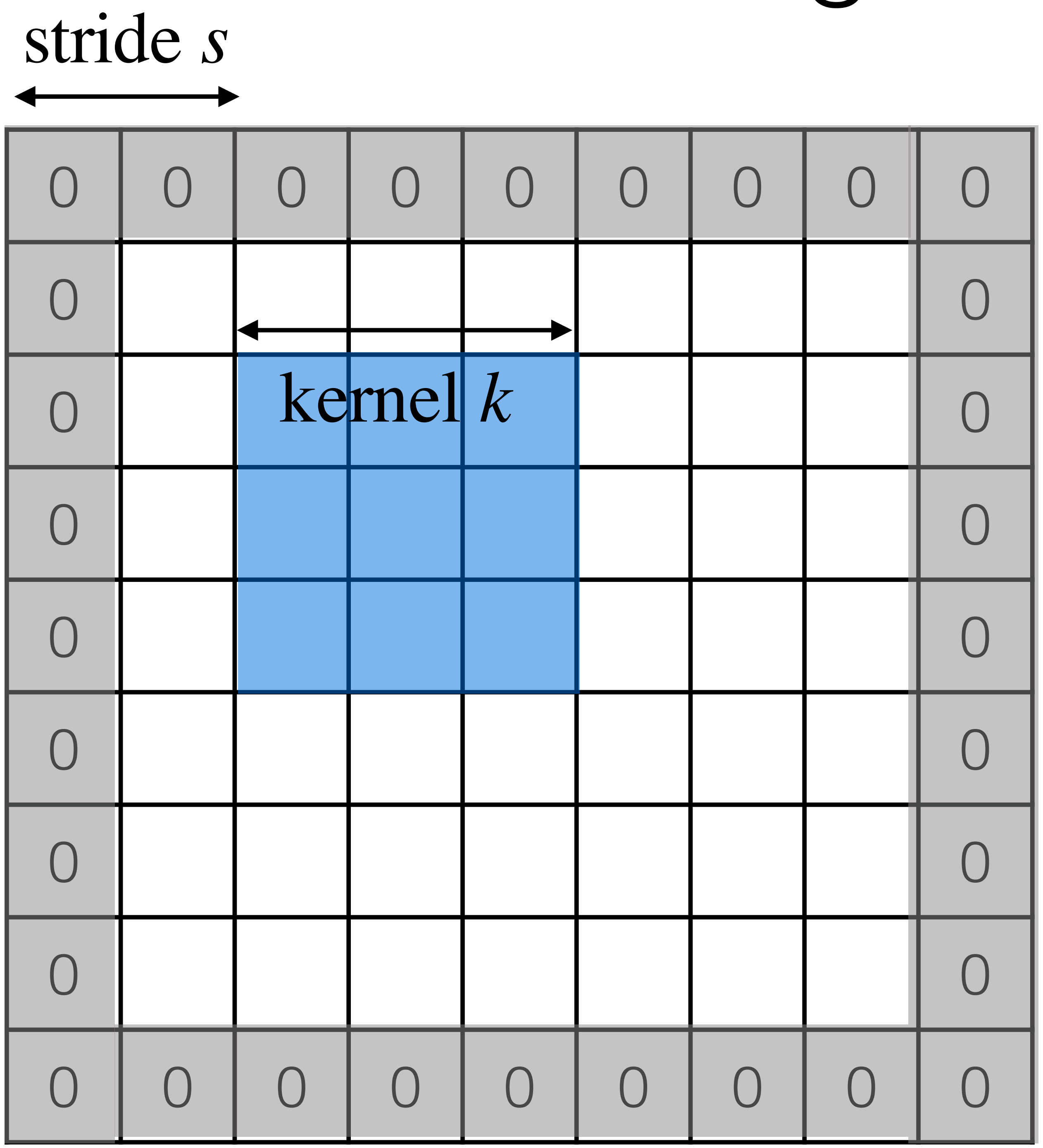
Convolution:

How big is the output?

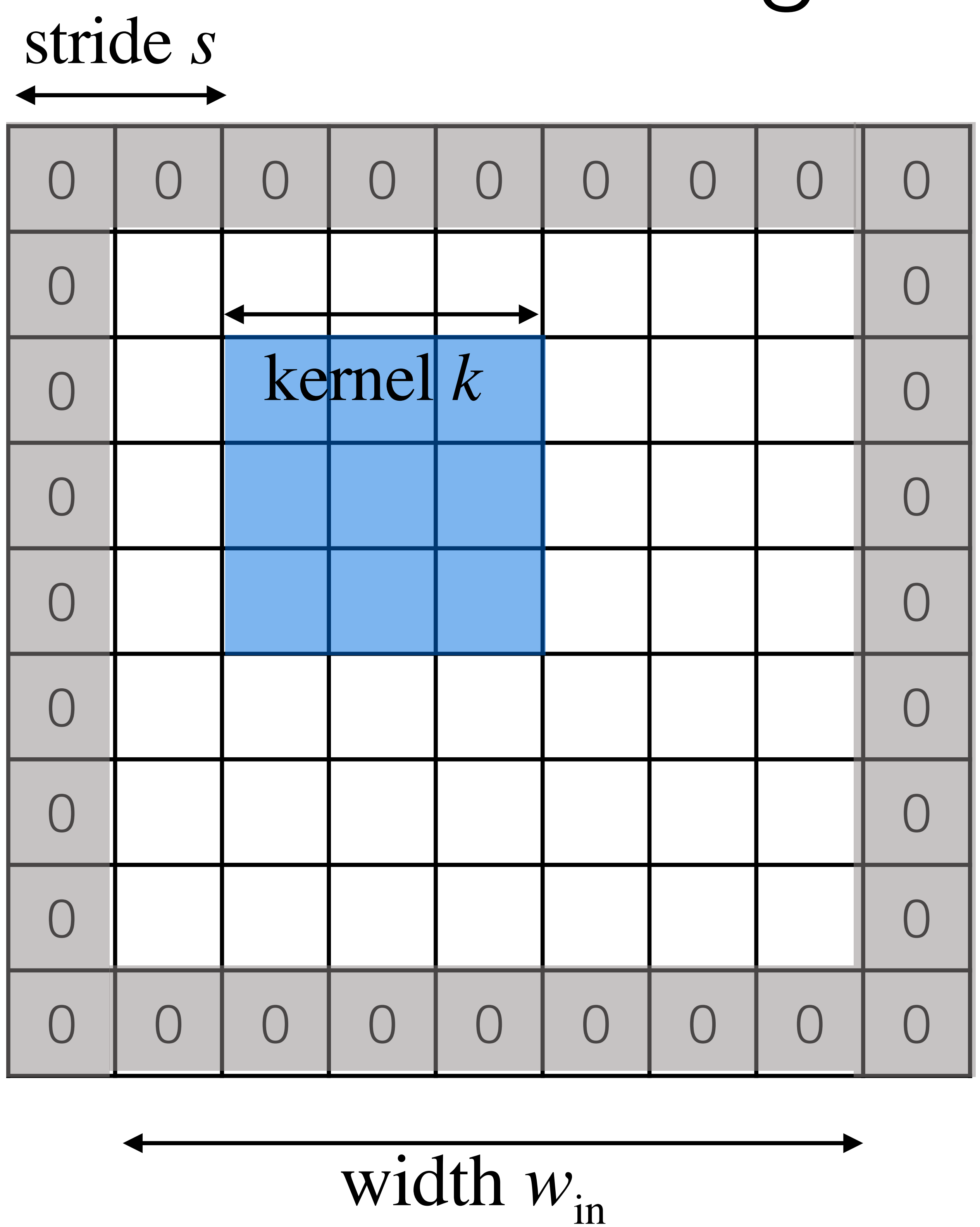


Convolution:

How big is the output?

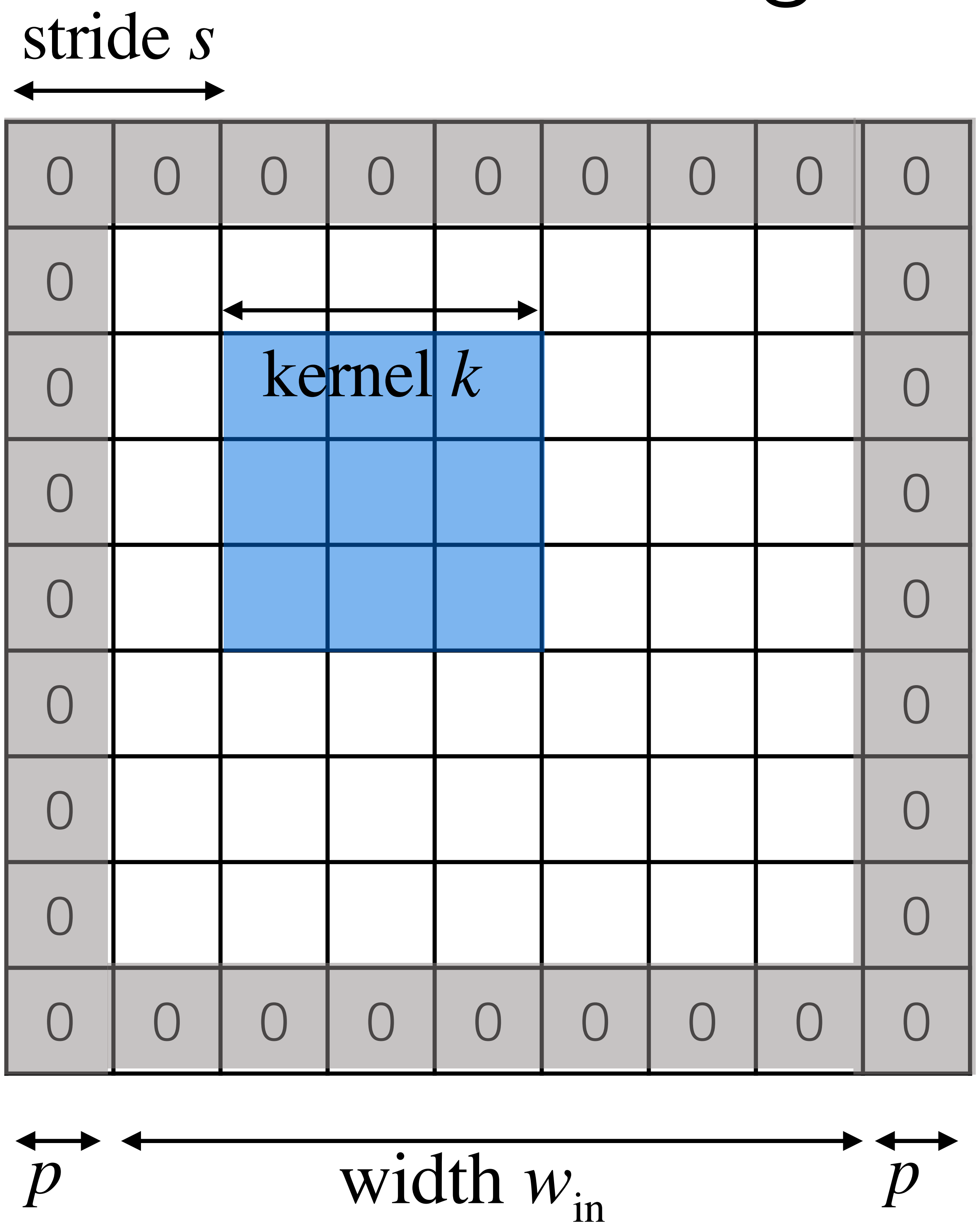


Convolution:
How big is the output?



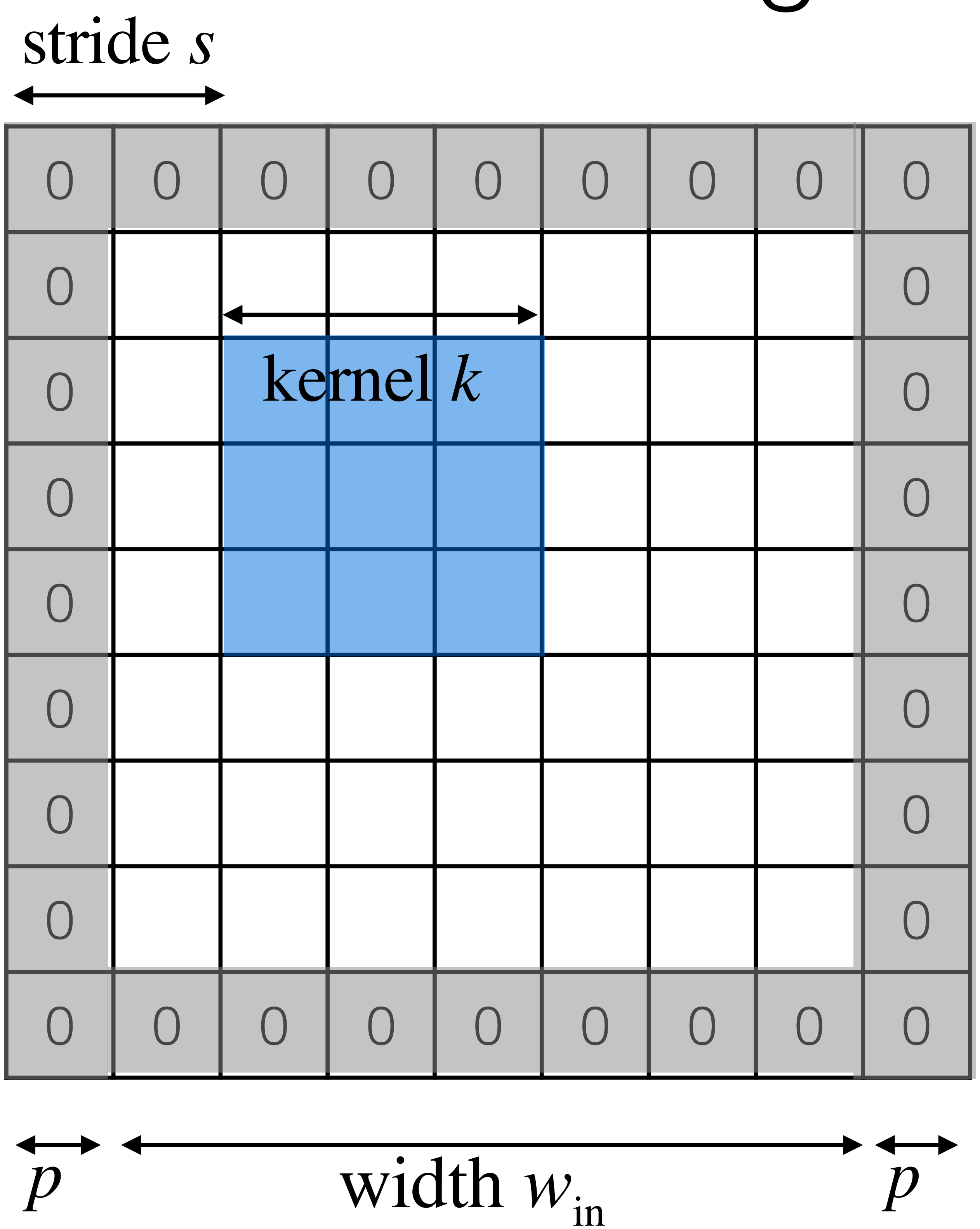
Convolution:

How big is the output?



Convolution:

How big is the output?

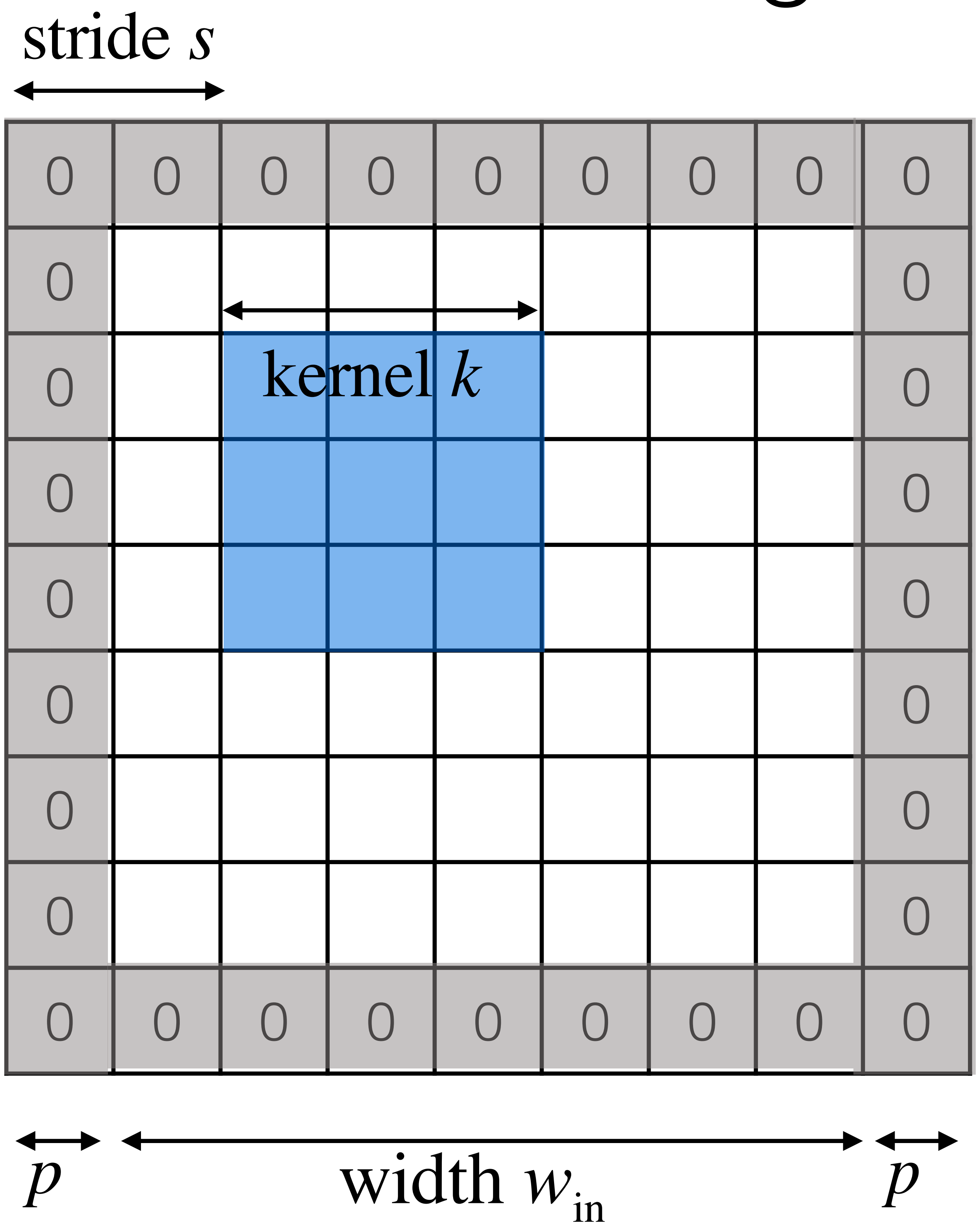


In general, the output has size:

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

Convolution:

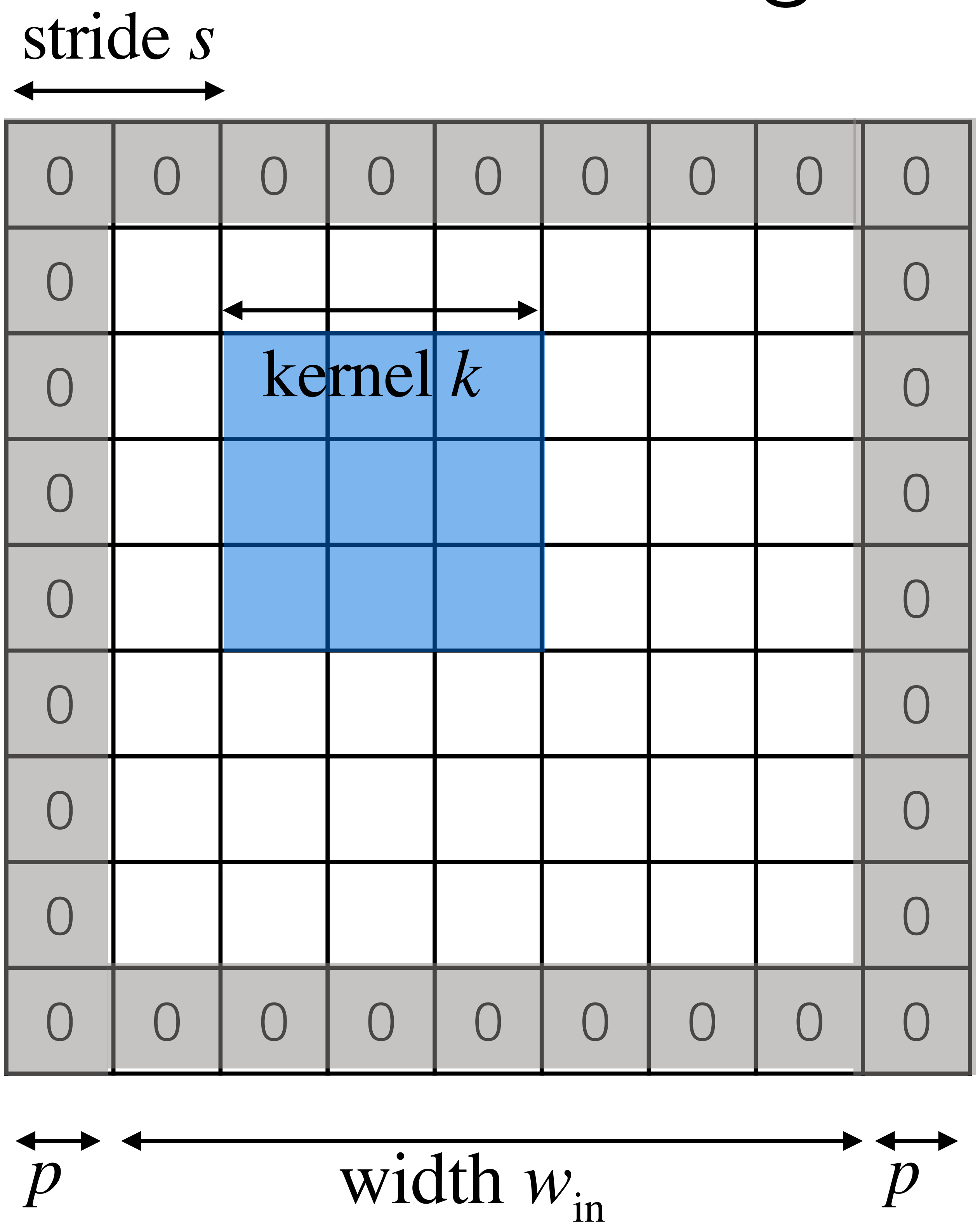
How big is the output?



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

Convolution:

How big is the output?



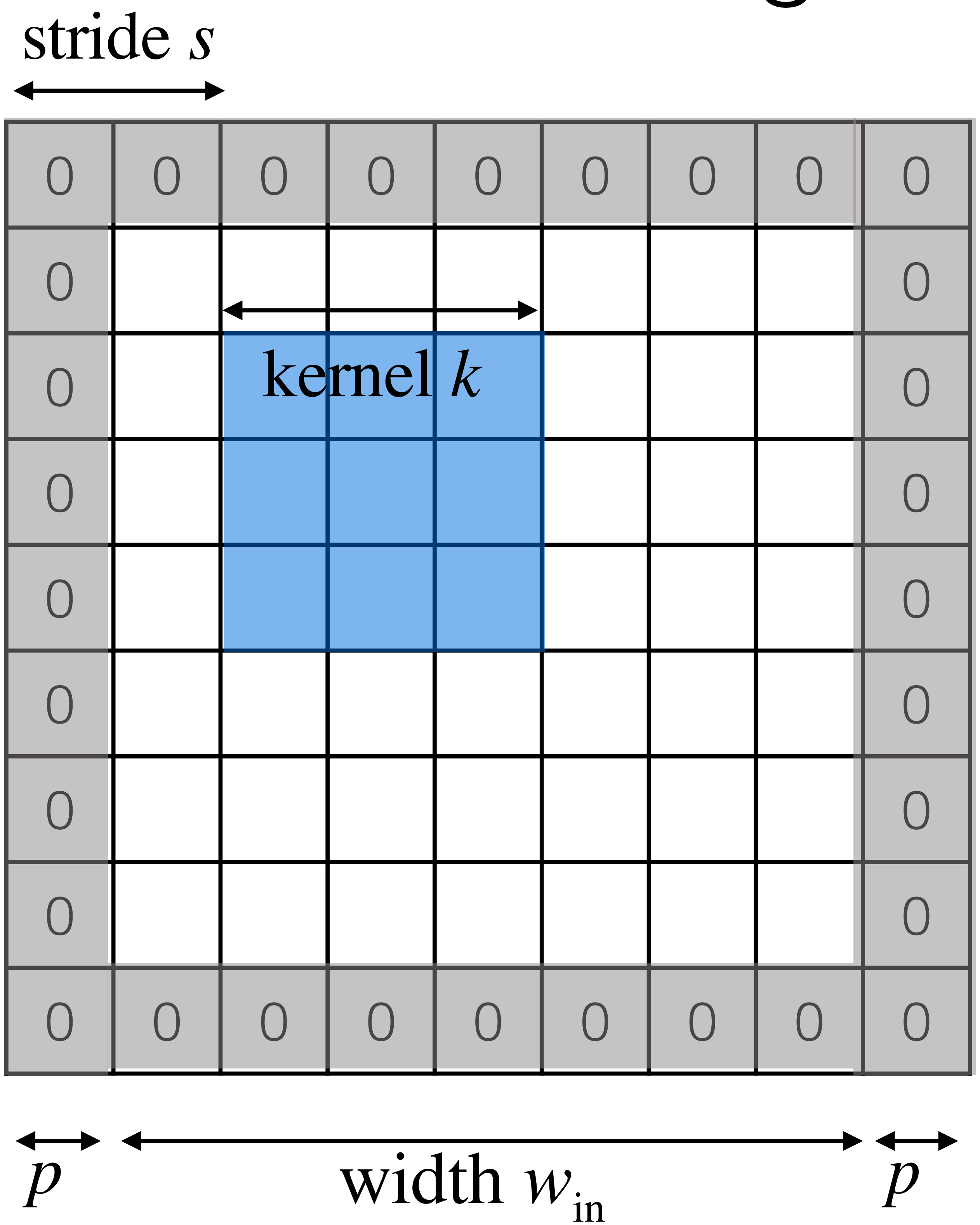
Example: k=3, s=1, p=1

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

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

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

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

Max Pooling

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

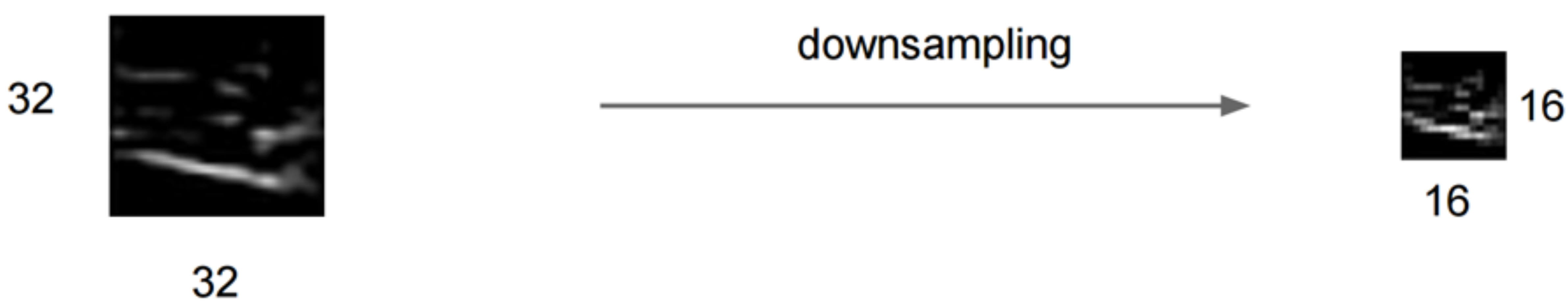


Figure: Andrej Karpathy

Max Pooling

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

- Creates a smaller representation while retaining the most important information

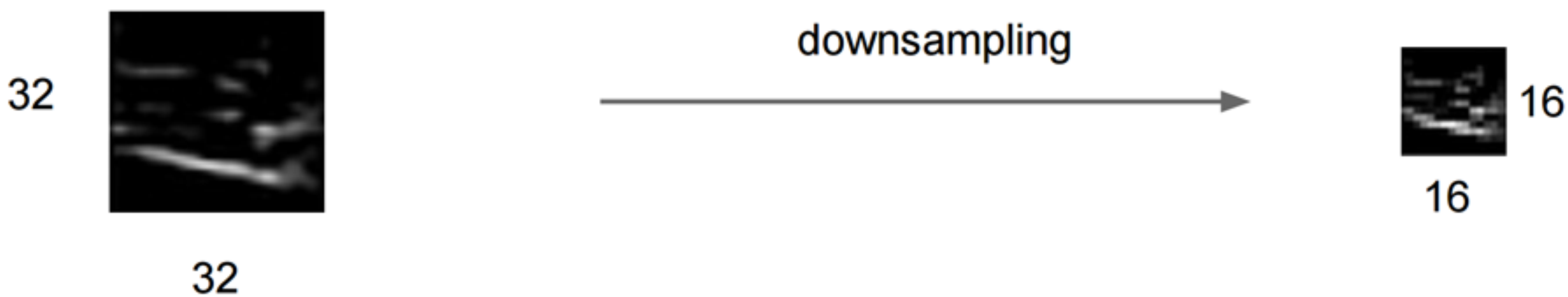


Figure: Andrej Karpathy

Max Pooling

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

- Creates a smaller representation while retaining the most important information
- The "max" operation is the most common

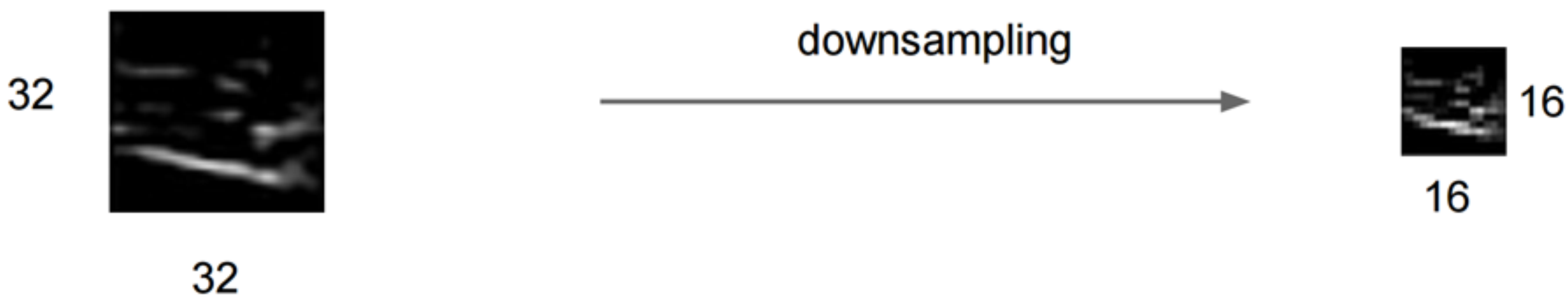


Figure: Andrej Karpathy

Max Pooling

For most CNNs, **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?

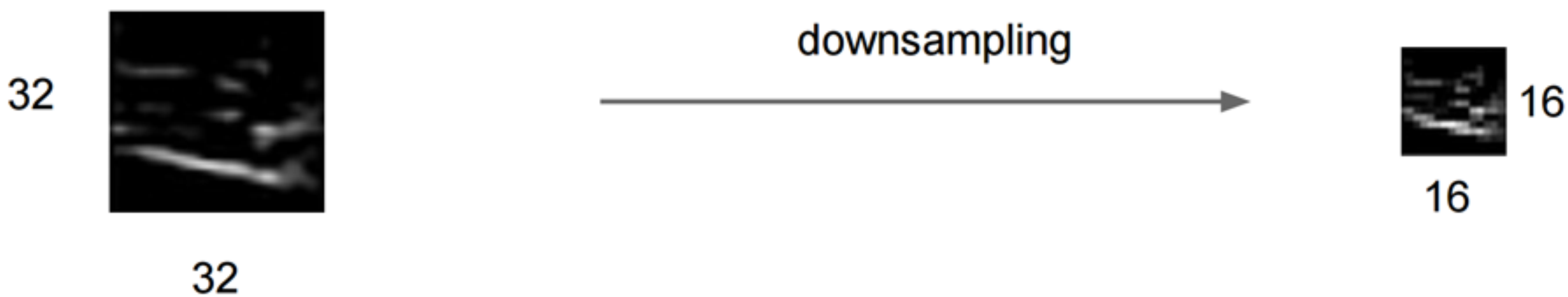


Figure: Andrej Karpathy

Max Pooling

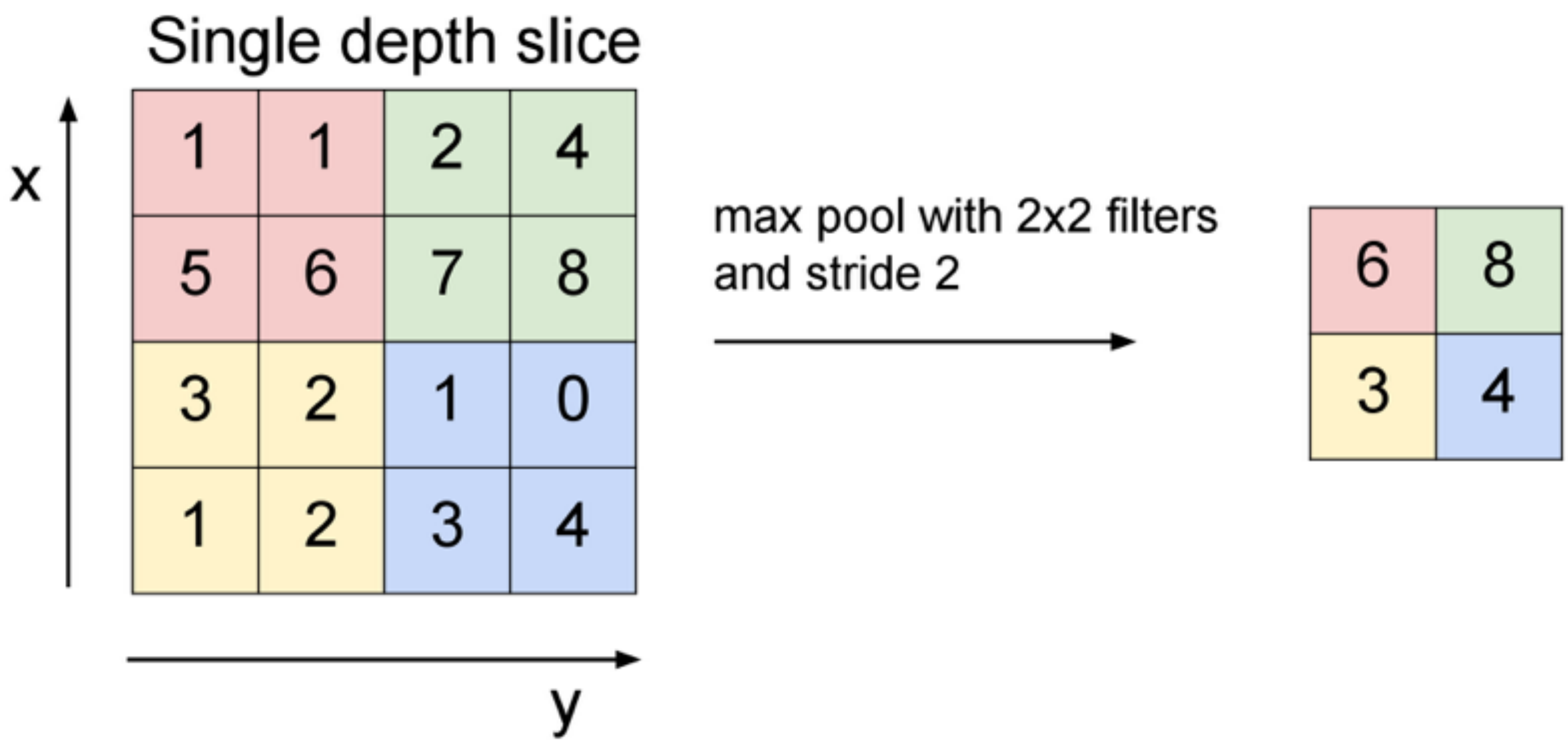
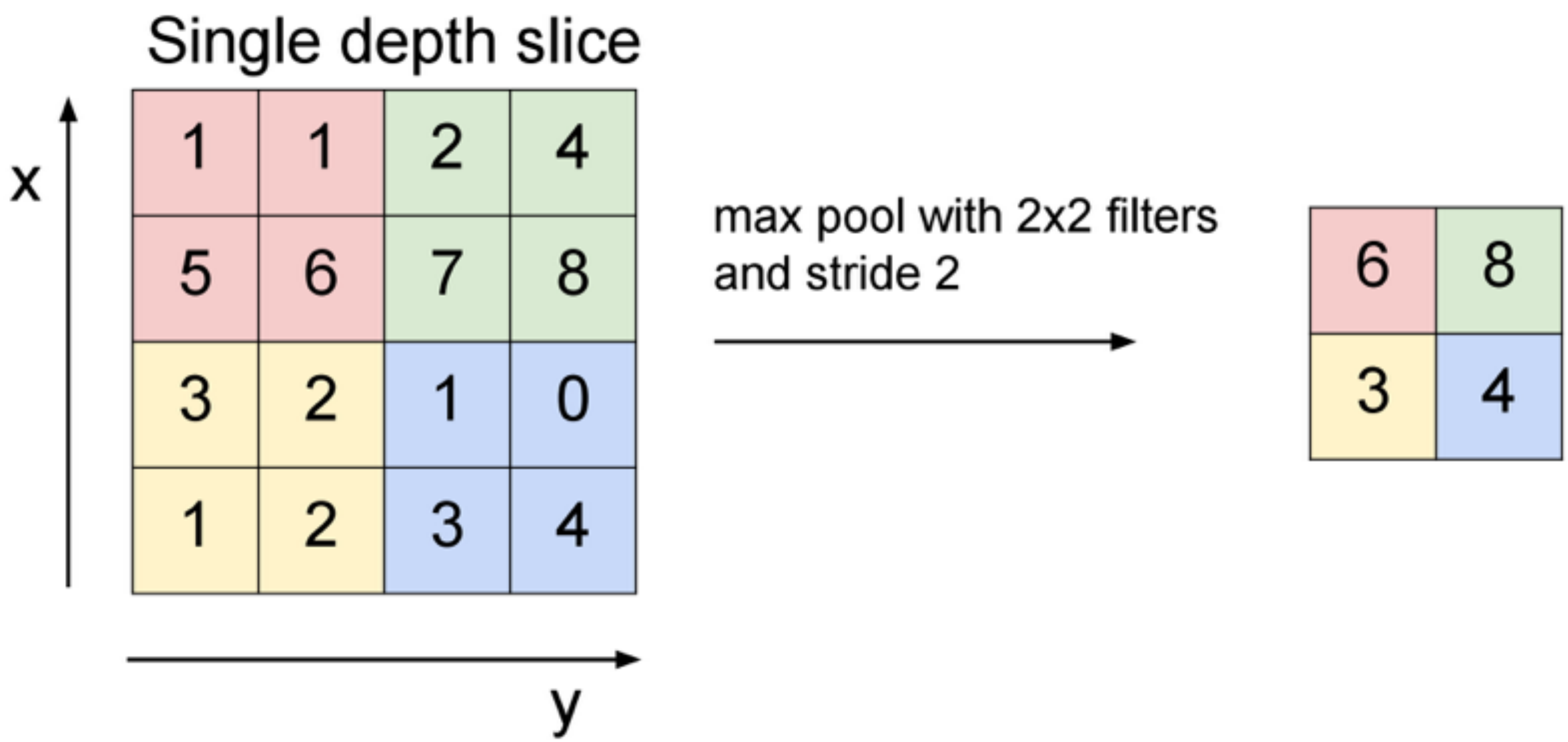


Figure: Andrej Karpathy

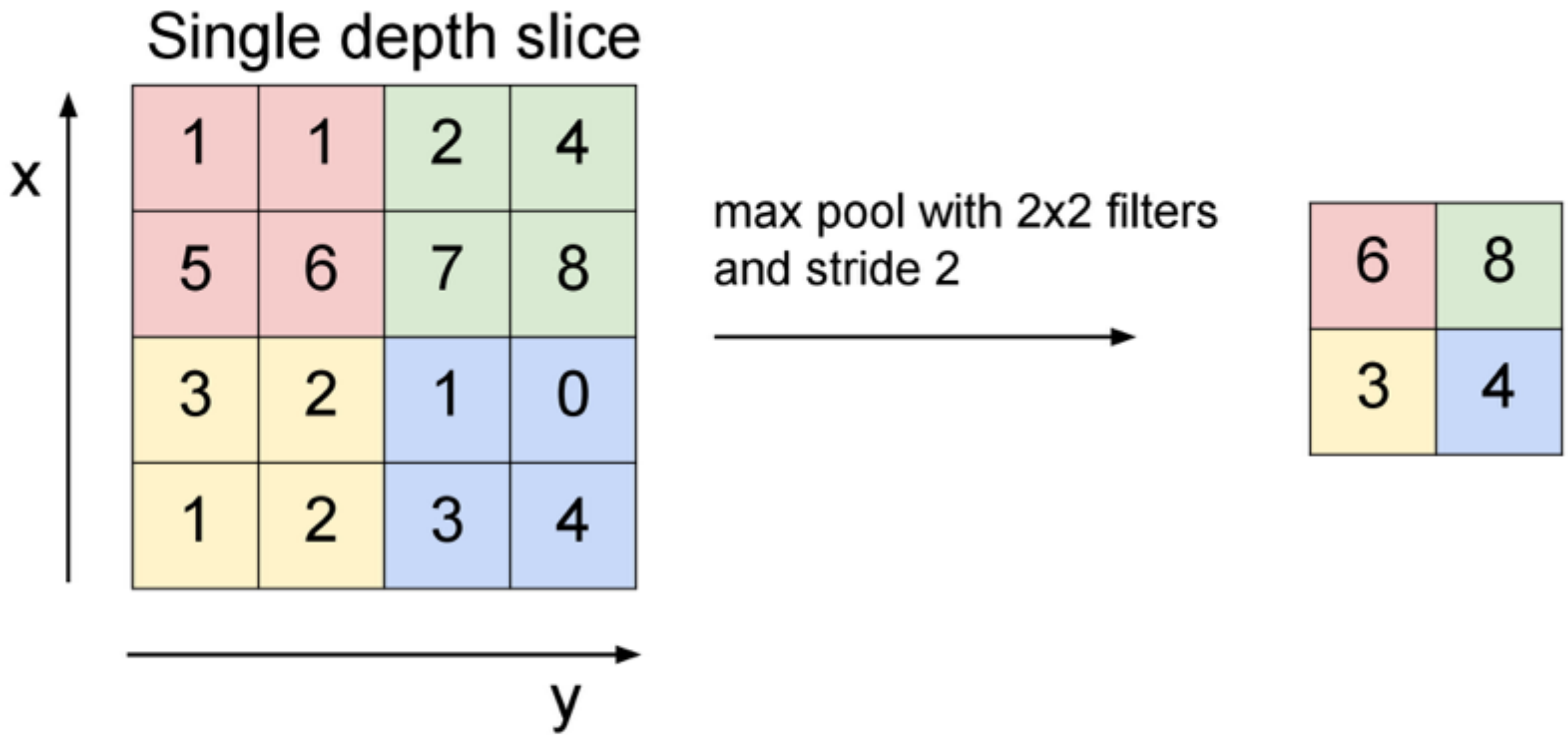
Max Pooling



What's the backprop rule for max pooling?

Figure: Andrej Karpathy

Max Pooling

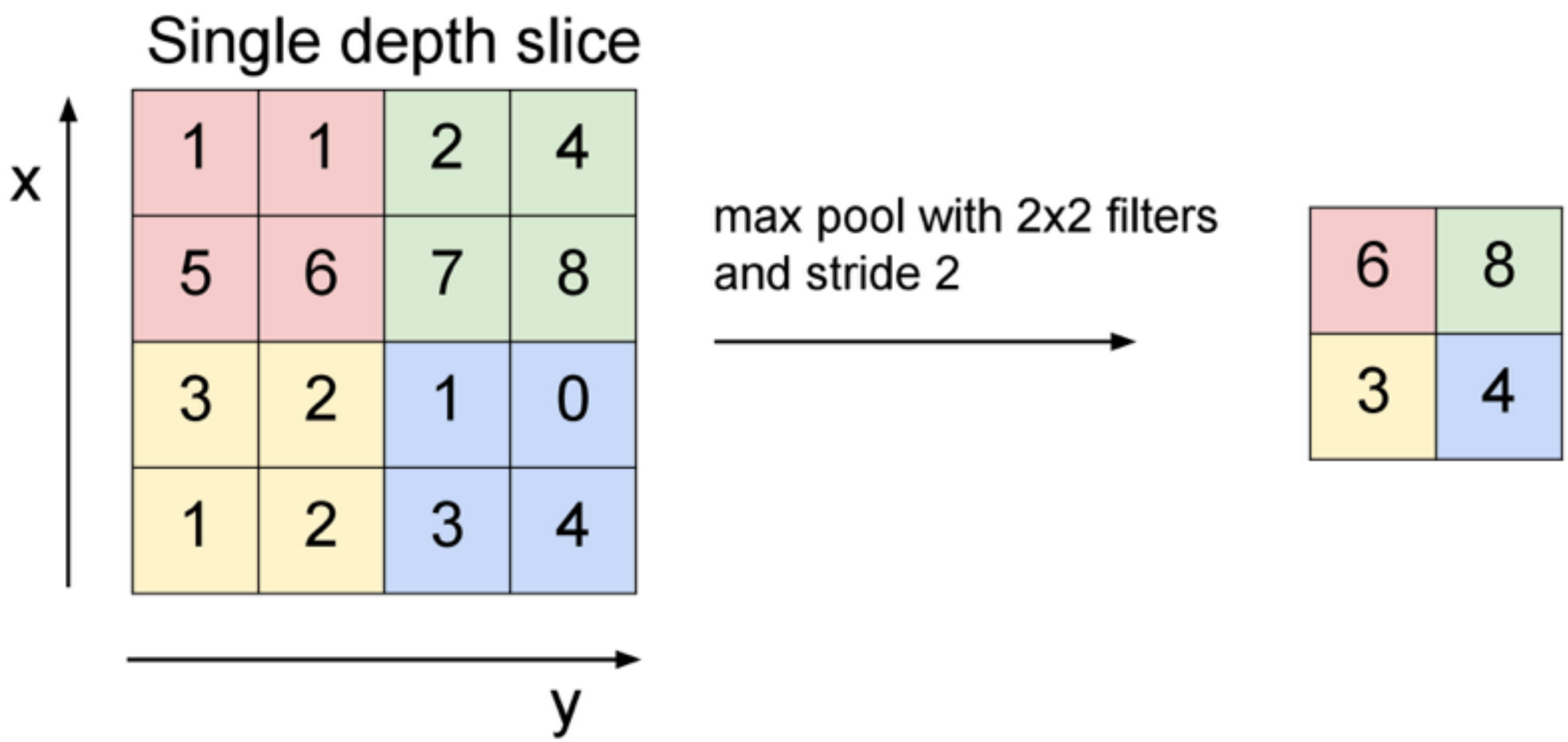


What's the backprop rule for max pooling?

- In the forward pass, store the index that took the max

Figure: Andrej Karpathy

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 CNN

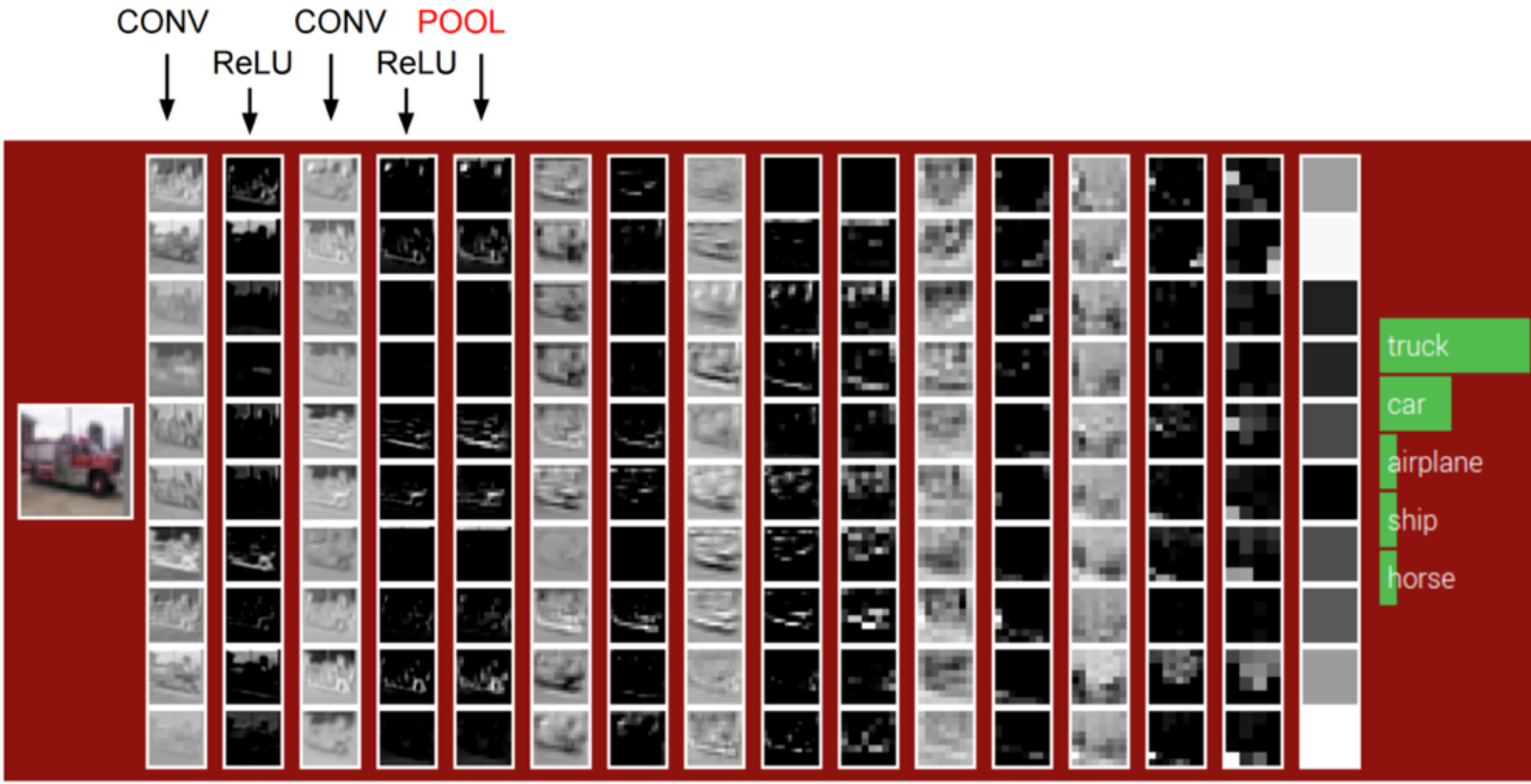


Figure: Andrej Karpathy

Example CNN

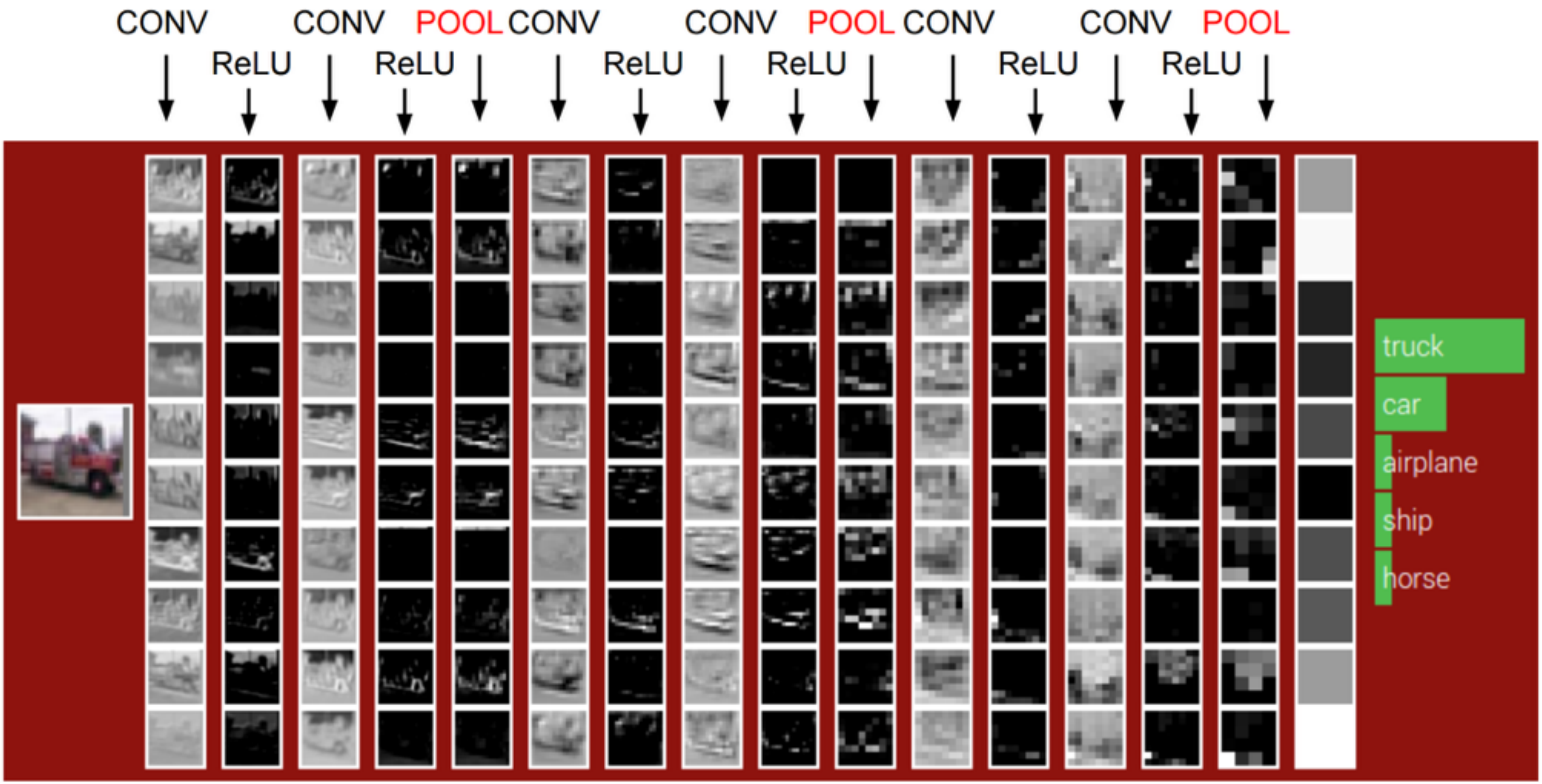


Figure: Andrej Karpathy

Example CNN

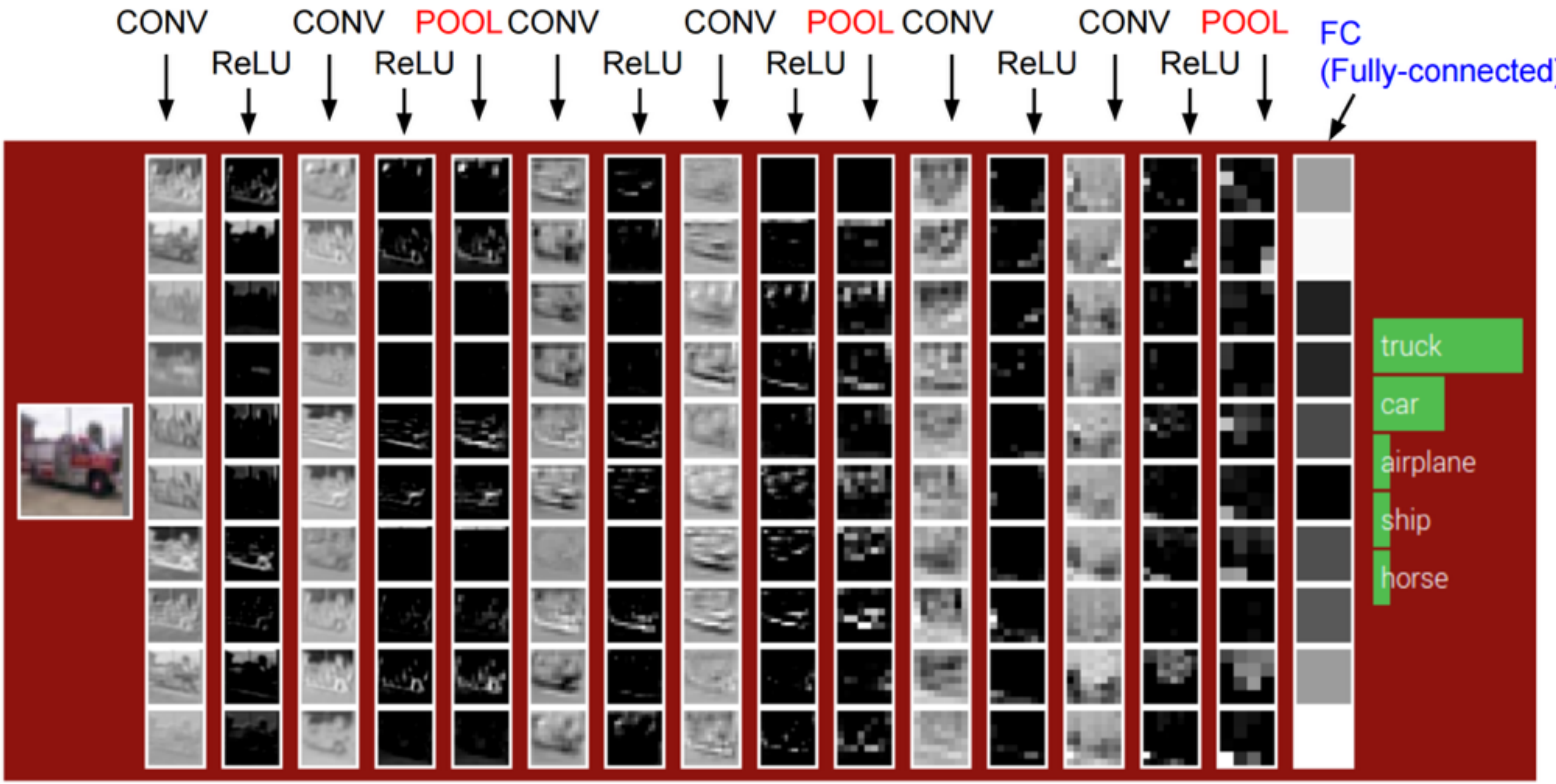
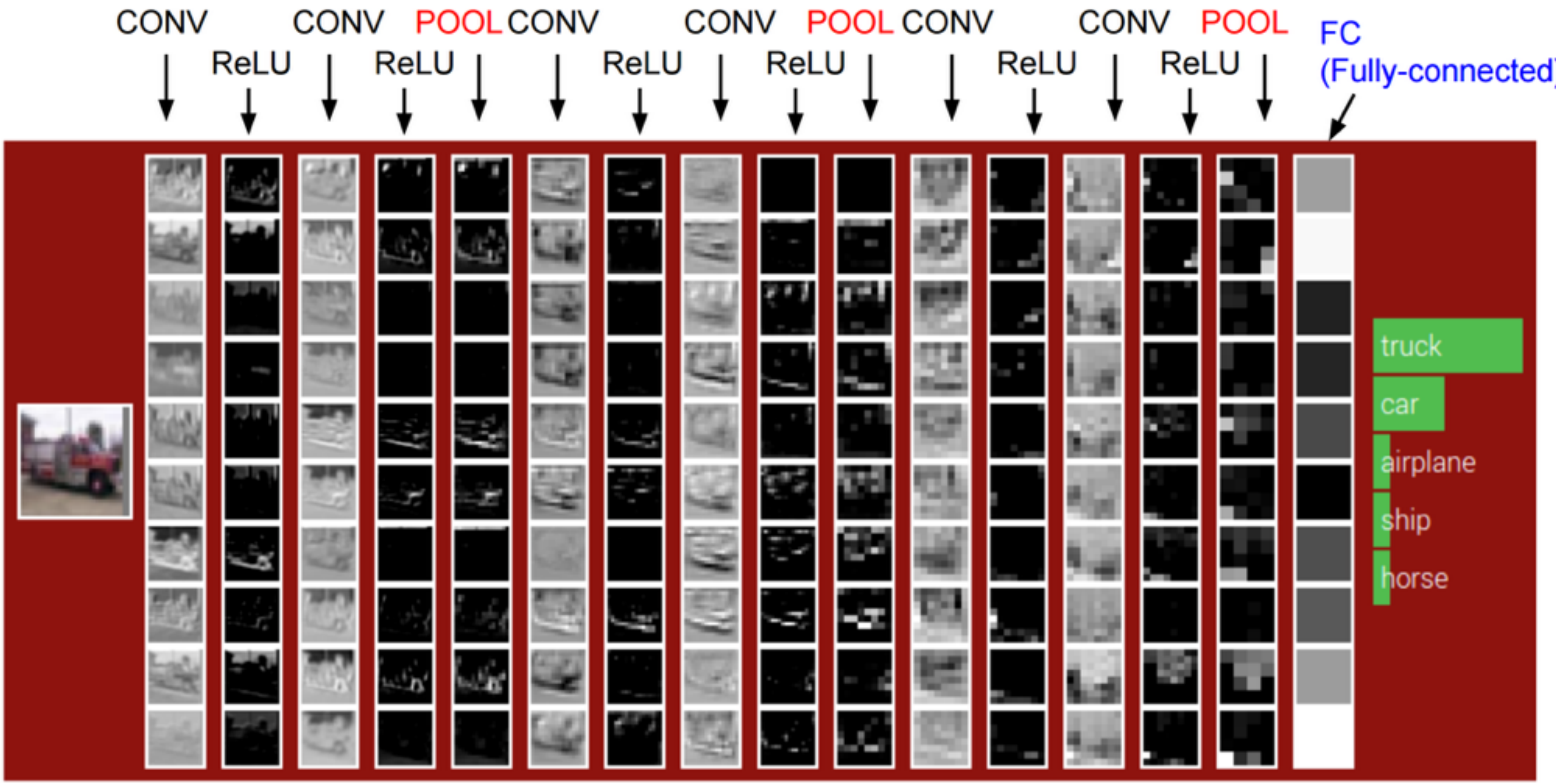


Figure: Andrej Karpathy

Example CNN



10x3x3 conv filters, stride 1, pad 1

2x2 pool filters, stride 2

Figure: Andrej Karpathy

Questions?