

**Deep Learning & Convolutional Neural Networks**  
**DS 4400 | Machine Learning and Data Mining I**  
**Zohair Shafi**  
**Spring 2026**

**Wednesday | March 18th, 2026**

# Today's Outline

- Deep Learning
- Convolutional Neural Networks

# Deep Learning

## Activation Functions

- Sigmoid Function

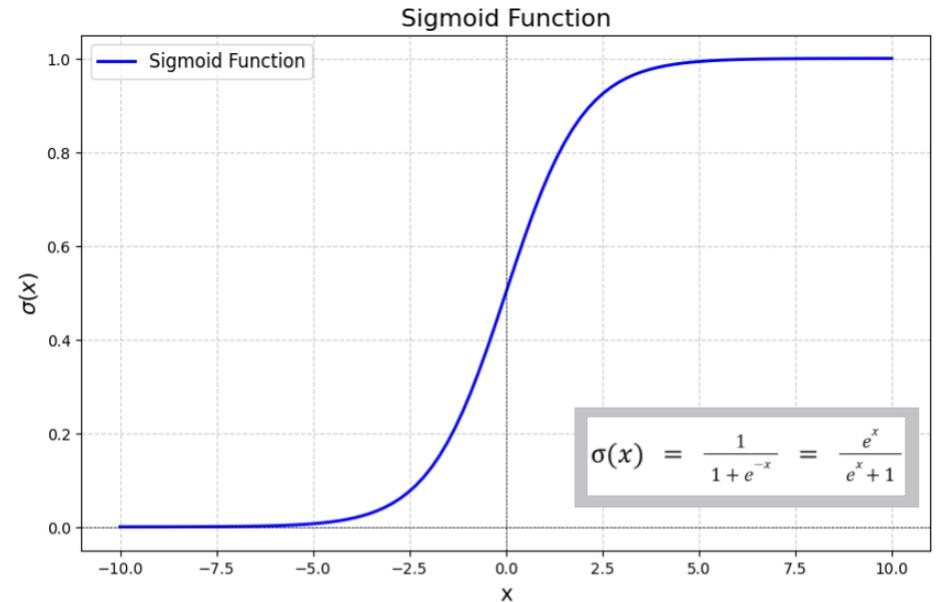
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

**Range:** [0, 1]

**Pros:** Smooth, bounded, interpretable as probability

**Cons:** Vanishing gradients (saturates at extremes), not zero-centered, expensive exponential operation

**Use:** Output layer for binary classification



# Deep Learning

## Activation Functions

- Tanh Function

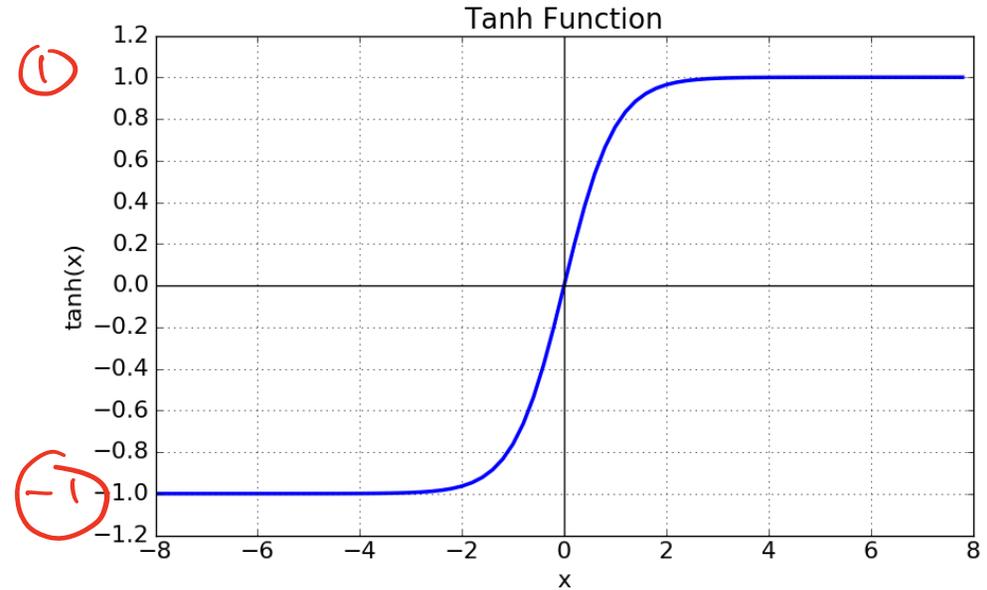
$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

**Range:** [-1, 1]

**Pros:** Zero-centered (better than sigmoid), stronger gradients

**Cons:** Still has vanishing gradient problem

**Use:** Hidden layers (historically), RNNs



# Deep Learning

## Activation Functions

- ReLU Function (Rectified Linear Unit)

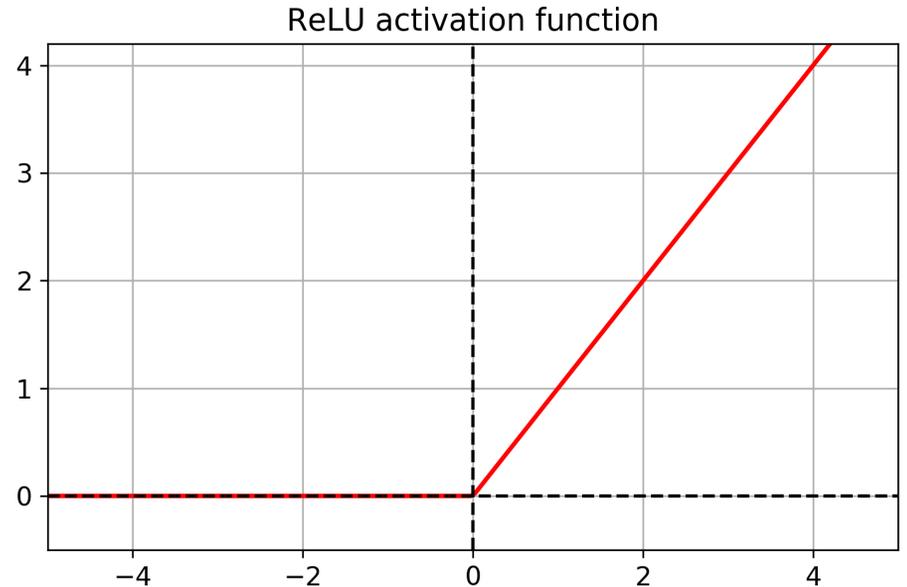
$$\text{ReLU}(x) = \max(0, x) = \begin{cases} x & x > 0 \\ 0 & x \leq 0 \end{cases}$$

**Range:**  $[0, \infty)$

**Pros:** No vanishing gradient (for  $x > 0$ ), computationally efficient, sparse activation

**Cons:** “Dying ReLU” problem (neurons stuck at 0), not zero-centered

**Use:** Default for hidden layers in most networks



# Deep Learning

## Activation Functions

Softmax

$$\frac{e^{z_k}}{\sum_{i=1}^K e^{z_i}}$$

If we have multi-class probability, what loss function can we use?

Categorical Cross Entropy Loss:

$$\ell_{\theta}(x) = -\frac{1}{N} \sum_{i=1}^N \sum_{k=1}^K y_{ik} \log(\hat{y}_{ik})$$



Pedestrian



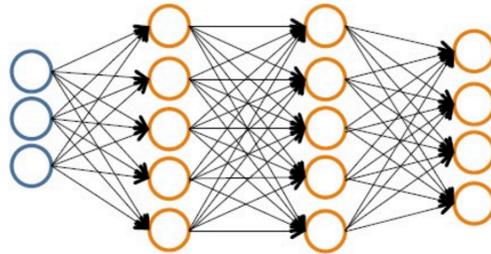
Car



Motorcycle



Truck



$$h_{\theta}(x) \in \mathbb{R}^K$$

$y = \text{car} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$   
 $y = \text{truck} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$

We want:

$$h_{\theta}(x) \approx \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

when pedestrian

$$h_{\theta}(x) \approx \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

when car

$$h_{\theta}(x) \approx \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

when motorcycle

$$h_{\theta}(x) \approx \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

when truck

# Deep Learning

## Regularization

- Deep networks have **millions** of parameters - they can **memorize** training data.
- We need regularization to improve generalization.

# Deep Learning

## Regularization

- $L_2$  Regularization

*reduce magnitude*

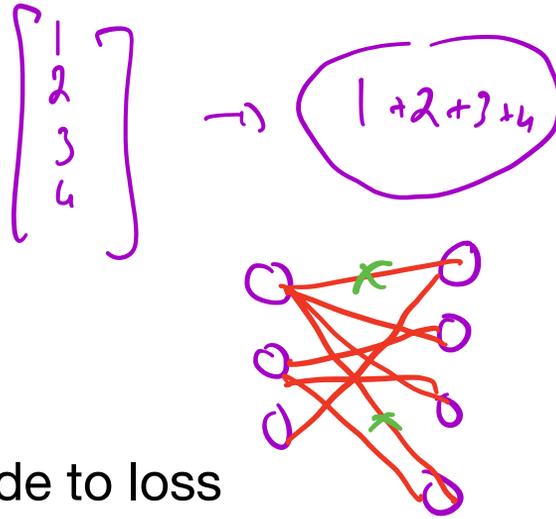
- Add penalty on weight magnitude to loss

$$L_{\theta}(x) = L_{\theta}(x) + \lambda \sum_l \|W^{[l]}\|_F^2$$

- **Effect:** Pushes weights toward zero, preferring simpler models.

# Deep Learning

## Regularization



- $L_1$  Regularization
  - Add penalty on weight magnitude to loss

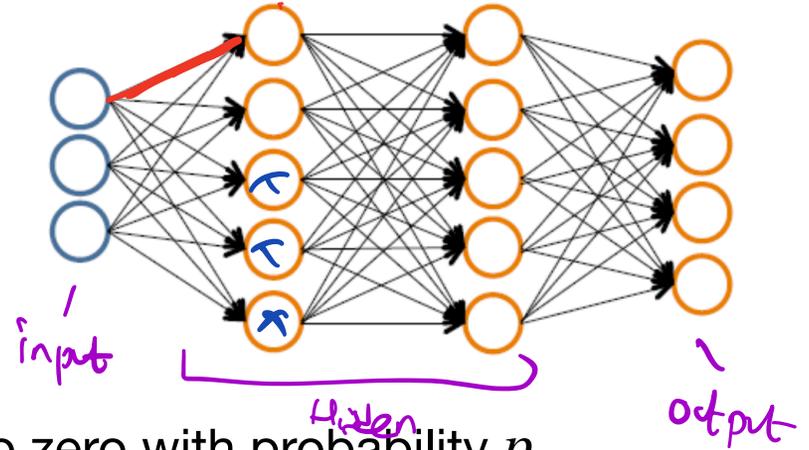
$$L_{\theta}(x) = L_{\theta}(x) + \lambda \sum_l \|W^{[l]}\|_1$$

- **Effect:** Encourages sparsity (many weights become exactly zero)

# Deep Learning

## Regularization

$p = 20\%$

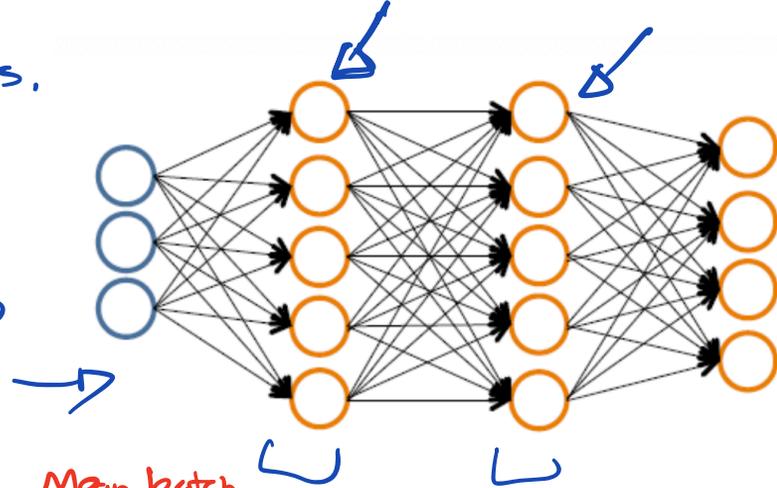


- Dropout Regularization
  - During training: randomly set activations to zero with probability  $p$
  - During inference: Use all neurons, no dropout
  - **Intuition:**
    - Forces network to not rely on any single neuron
    - Approximately trains an ensemble of  $2^n$  sub-networks
    - Prevents co-adaptation of neurons

# Deep Learning

## Regularization

$X \rightarrow 1000$  rows.  
 $B \rightarrow 128$   
 0 - 100  
 50 - 150



### Batch Normalization

- Normalize **activations** within each mini-batch

$$\hat{z} = \frac{z - \mu_B}{\sqrt{\sigma_B^2}}$$

Main batch ← va. factor.

$$\tilde{z} = \gamma \cdot \hat{z} + \beta$$

### • Benefits:

- Reduces internal covariate shift
- Allows higher learning rates
- Acts as regularization (noise from batch statistics)
- Reduces sensitivity to initialization

$$z^{[1]} = w^{[1]} \cdot a^{[0]} + b^{[1]}$$

$$z^{[1]} = \frac{z^{[1]} - \mu_B}{\sqrt{\sigma_B^2}}$$

$$z^{[1]} = \gamma z^{[1]} + \beta$$

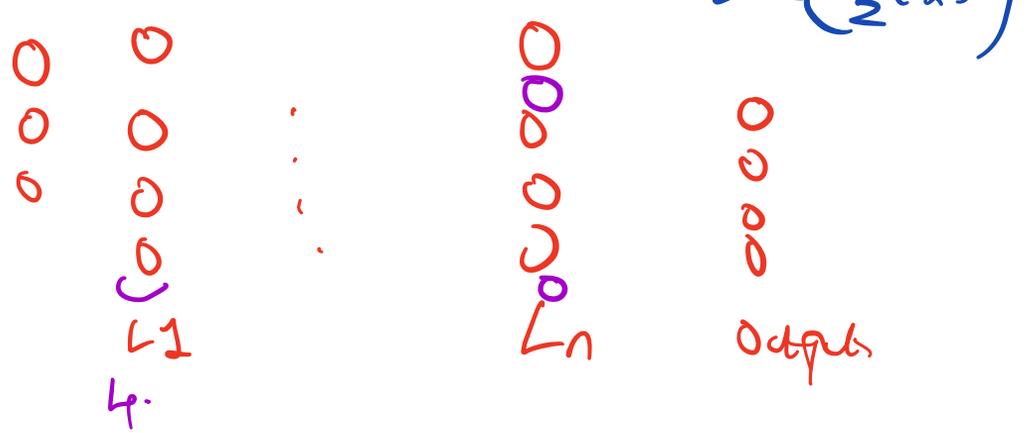
$$a^{[1]} = \sigma(z^{[1]})$$

$$z^{[2]} = w^{[2]} \cdot a^{[1]} + b^{[2]}$$

$$a^{[2]} = \sigma(z^{[2]})$$

# Deep Learning

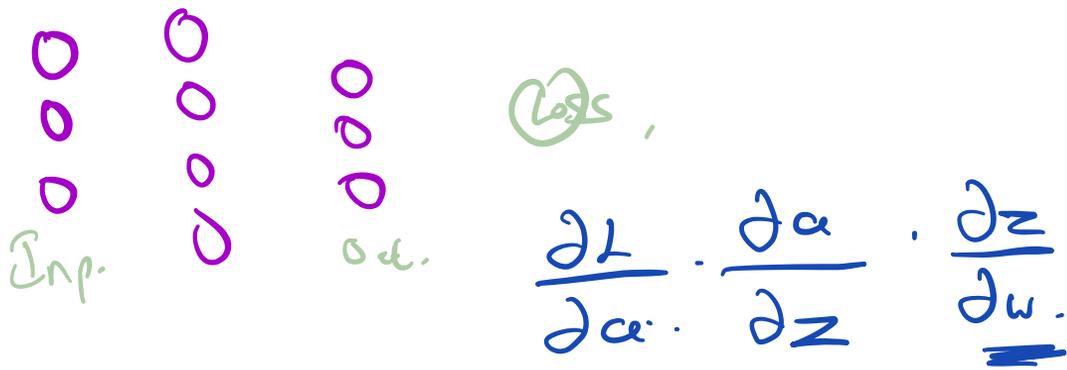
## Hyperparameters



- Number of layers —
- Number of hidden units (neurons) per layer —
- Activation function — Sigmoid / ReLU.
- Initialization — He, Xavier.
- Regularization —

# Deep Learning

## Backpropagation



- Compute gradients for each weight using the chain rule
- Consider a 2-layer MLP with the following forward pass:

**Layer 1 (Hidden Layer):**

$$z^{[1]} = W^{[1]}x + b^{[1]}$$
$$a^{[1]} = \text{ReLU}(z^{[1]})$$

**Layer 2 (Output Layer):**

$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$$
$$\hat{y} = \sigma(z^{[2]})$$

# Deep Learning

## Backpropagation

①  $z^{[1]} = a^{[0]} \cdot w^{[1]} + b^{[1]}$   
 $a^{[1]} = \sigma(z^{[1]})$   
 $z^{[2]} = a^{[1]} \cdot w^{[2]} + b^{[2]}$   
 $\hat{y} = \sigma(z^{[2]})$

Forward Pass.

$$\frac{\partial \mathcal{L}}{\partial w^{[1]}} = \frac{\partial \mathcal{L}}{\partial a^{[1]}} \cdot \frac{\partial a^{[1]}}{\partial z^{[1]}} \cdot \frac{\partial z^{[1]}}{\partial w^{[1]}}$$

$$\frac{\partial \mathcal{L}}{\partial w^{[2]}}$$

Backward Pass.

# Deep Learning

## Backpropagation

$$z = \frac{z - \mu_B}{\sqrt{\sigma_B^2}} \rightarrow \text{Batch.}$$

**Layer 1 (Hidden Layer):**

$$z^{[1]} = W^{[1]}x + b^{[1]}$$

$$a^{[1]} = \text{ReLU}(z^{[1]})$$

**Layer 2 (Output Layer):**

$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$$

$$\hat{y} = \sigma(z^{[2]})$$

**Loss Function:**  $L = - [y \log \hat{y} + (1 - y) \log(1 - \hat{y})]$

**Forward Pass:** Compute and store all intermediate values:

$$x \xrightarrow{W^{[1]}, b^{[1]}} z^{[1]} \xrightarrow{\text{ReLU}} a^{[1]} \xrightarrow{W^{[2]}, b^{[2]}} z^{[2]} \xrightarrow{\sigma} \hat{y} \rightarrow L$$

# Deep Learning

## Backpropagation

**Forward Pass:** Compute and store all intermediate values:

$$x \xrightarrow{W^{[1]}, b^{[1]}} z^{[1]} \xrightarrow{\text{ReLU}} a^{[1]} \xrightarrow{W^{[2]}, b^{[2]}} z^{[2]} \xrightarrow{\sigma} \hat{y} \rightarrow L$$

**Backward Pass:** We compute gradients layer by layer, from output to input, using the chain rule.

We need  $\frac{\partial L}{\partial z^{[2]}}$ . Using the chain rule:

$$\frac{\partial L}{\partial z^{[2]}} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z^{[2]}}$$

or binary cross-entropy loss:

$$\frac{\partial L}{\partial \hat{y}} = -\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}}$$

# Deep Learning

## Backpropagation

**Backward Pass:** We compute gradients layer by layer, from output to input, using the chain rule.

We need  $\frac{\partial L}{\partial z^{[2]}}$ . Using the chain rule:

$$\frac{\partial L}{\partial z^{[2]}} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z^{[2]}}$$

or binary cross-entropy loss:

$$\frac{\partial L}{\partial \hat{y}} = -\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}}$$

Combining these (with simplification):

$$\delta^{[2]} = \frac{\partial L}{\partial z^{[2]}} = \hat{y} - y$$

# Deep Learning

## Backpropagation

**Backward Pass:** We compute gradients layer by layer, from output to input, using the chain rule.

Step 2: Gradients for  $W^{[2]}$  and  $b^{[2]}$

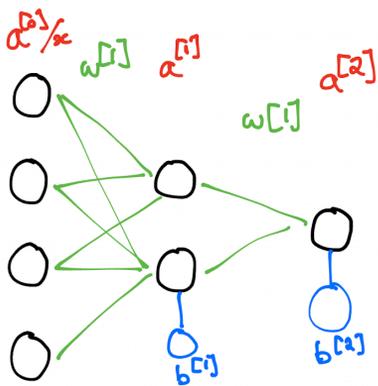
Since  $z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$ , we have:

$$\text{Weight gradient: } \frac{\partial L}{\partial W^{[2]}} = \frac{\partial L}{\partial z^{[2]}} \cdot \frac{\partial z^{[2]}}{\partial W^{[2]}} = \delta^{[2]} \cdot \frac{\partial}{\partial W^{[2]}} (W^{[2]}a^{[1]} + b^{[2]})$$

$$\frac{\partial L}{\partial W^{[2]}} = \delta^{[2]}(a^{[1]})^T$$

# Deep Learning

## Backpropagation



Equations -

$$z^{[1]} = w^{[1]} a^{[0]} + b^{[1]}$$

$$a^{[1]} = \sigma(z^{[1]})$$

$$z^{[2]} = w^{[2]} a^{[1]} + b^{[2]}$$

$$a^{[2]} = \sigma(z^{[2]})$$

$$a = \frac{1}{2} (y - \hat{y})^2$$

Forward Pass

$$\begin{aligned} \frac{\partial L}{\partial a^{[2]}} &= \frac{\partial}{\partial a^{[2]}} \left( \frac{1}{2} (y - \hat{y})^2 \right) \\ &= \frac{\partial}{\partial a^{[2]}} \left( \frac{1}{2} (y - a^{[2]})^2 \right) \\ &= -(y - a^{[2]}) \end{aligned}$$

$$\begin{aligned} \frac{\partial a^{[2]}}{\partial z^{[2]}} &= \frac{\partial}{\partial z^{[2]}} (\sigma(z^{[2]})) \\ &= \sigma(z^{[2]}) (1 - \sigma(z^{[2]})) \end{aligned}$$

$$\begin{aligned} \frac{\partial z^{[2]}}{\partial a^{[1]}} &= \frac{\partial}{\partial a^{[1]}} (w^{[2]} a^{[1]} + b^{[2]}) \\ &= w^{[2]} \end{aligned}$$

$$\begin{aligned} \frac{\partial z^{[2]}}{\partial w^{[2]}} &= \frac{\partial}{\partial w^{[2]}} (w^{[2]} a^{[1]} + b^{[2]}) \\ &= a^{[1]} \end{aligned}$$

$$\begin{aligned} \frac{\partial a^{[1]}}{\partial z^{[1]}} &= \frac{\partial}{\partial z^{[1]}} (\sigma(z^{[1]})) \\ &= \sigma(z^{[1]}) (1 - \sigma(z^{[1]})) \end{aligned}$$

$$\begin{aligned} \frac{\partial z^{[1]}}{\partial a^{[0]}} &= \frac{\partial}{\partial a^{[0]}} (w^{[1]} a^{[0]} + b^{[1]}) \\ &= w^{[1]} \end{aligned}$$

$$\begin{aligned} \frac{\partial z^{[1]}}{\partial w^{[1]}} &= \frac{\partial}{\partial w^{[1]}} (w^{[1]} a^{[0]} + b^{[1]}) \\ &= a^{[0]} \end{aligned}$$

Chain Rule -

$$da^{[2]} = \frac{\partial L}{\partial a^{[2]}} = -(y - a^{[2]})$$

$$\begin{aligned} dz^{[2]} &= \frac{\partial L}{\partial z^{[2]}} = \left( \frac{\partial L}{\partial a^{[2]}} \right) \left( \frac{\partial a^{[2]}}{\partial z^{[2]}} \right) \\ &= da^{[2]} g'(z^{[2]}) \end{aligned}$$

$$\begin{aligned} dw^{[2]} &= \frac{\partial L}{\partial w^{[2]}} = \left( \frac{\partial L}{\partial a^{[2]}} \right) \left( \frac{\partial a^{[2]}}{\partial z^{[2]}} \right) \left( \frac{\partial z^{[2]}}{\partial w^{[2]}} \right) \\ &= -(y - a^{[2]}) (\sigma(z^{[2]}) (1 - \sigma(z^{[2]}))) a^{[1]} \\ &= dz^{[2]} a^{[1]} \end{aligned}$$

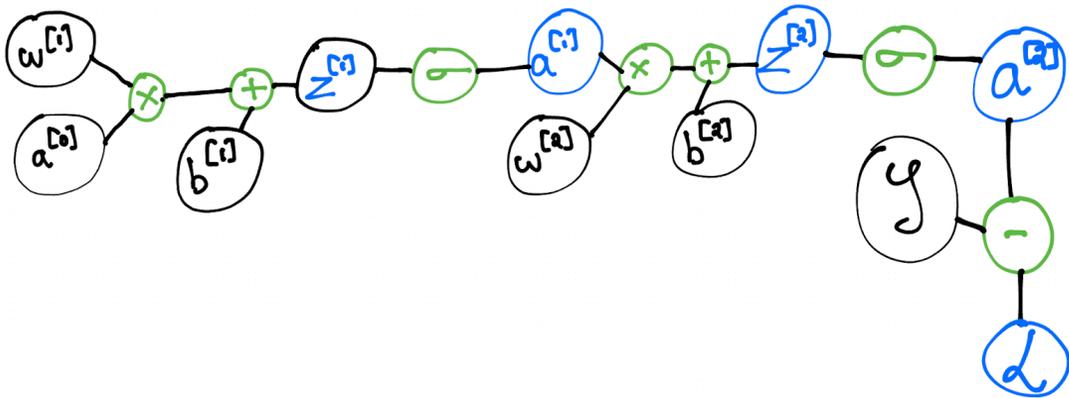
$$da^{[1]} = \frac{\partial L}{\partial a^{[1]}} = \left( \frac{\partial L}{\partial a^{[2]}} \right) \left( \frac{\partial a^{[2]}}{\partial z^{[2]}} \right) \left( \frac{\partial z^{[2]}}{\partial a^{[1]}} \right)$$

$$dz^{[1]} = \frac{\partial L}{\partial z^{[1]}} = \left( \frac{\partial L}{\partial a^{[2]}} \right) \left( \frac{\partial a^{[2]}}{\partial z^{[2]}} \right) \left( \frac{\partial z^{[2]}}{\partial a^{[1]}} \right) \left( \frac{\partial a^{[1]}}{\partial z^{[1]}} \right)$$

$$dw^{[1]} = \frac{\partial L}{\partial w^{[1]}} = \left( \frac{\partial L}{\partial a^{[2]}} \right) \left( \frac{\partial a^{[2]}}{\partial z^{[2]}} \right) \left( \frac{\partial z^{[2]}}{\partial a^{[1]}} \right) \left( \frac{\partial a^{[1]}}{\partial z^{[1]}} \right) \left( \frac{\partial z^{[1]}}{\partial w^{[1]}} \right)$$

# Deep Learning

## Backpropagation



Final Equations -

$$\delta a^{[2]} = a^{[2]} - y$$

$$\delta z^{[2]} = \delta a^{[2]} g'(z^{[2]})$$

$$\delta w^{[2]} = \delta z^{[2]} a^{[1]}$$

$$\delta a^{[1]} = \delta z^{[2]} w^{[2]}$$

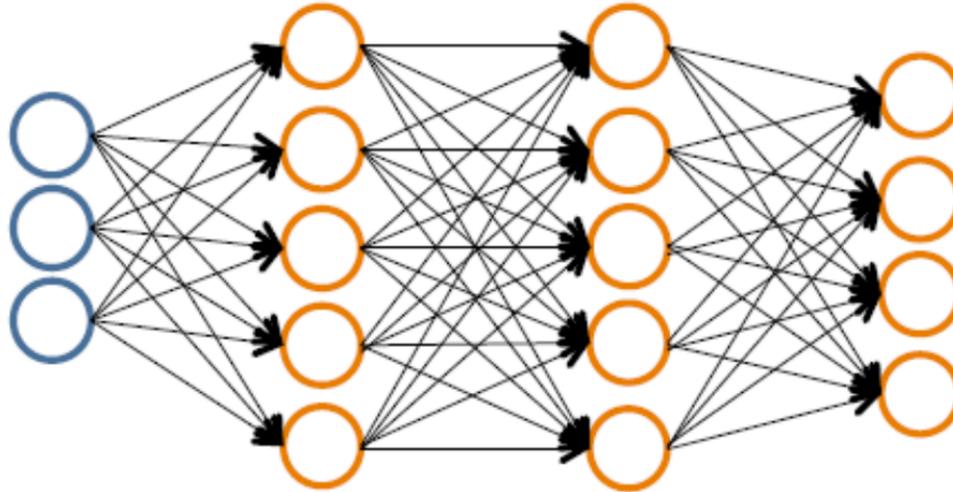
$$\delta z^{[1]} = \delta a^{[1]} g'(z^{[1]})$$

$$\delta w^{[1]} = \delta z^{[1]} a^{[0]}$$

# Deep Learning

## Backpropagation

### 1. Forward Propagation



### 2. Backward Propagation



Equations -

$$z^{[1]} = w^{[1]} a^{[0]} + b^{[1]}$$

$$a^{[1]} = \sigma(z^{[1]})$$

$$z^{[2]} = w^{[2]} a^{[1]} + b^{[2]}$$

$$a^{[2]} = \sigma(z^{[2]})$$

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

Final Equations -

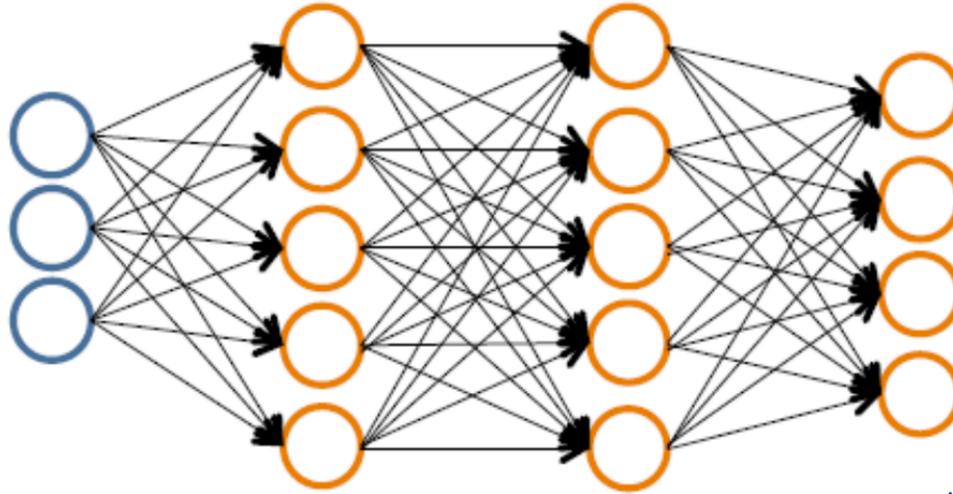
$$\begin{aligned} da^{[2]} &= a^{[2]} - y \\ dz^{[2]} &= da^{[2]} g'(z^{[2]}) \\ d\omega^{[2]} &= dz^{[2]} a^{[1]} \\ da^{[1]} &= dz^{[2]} \omega^{[2]} \\ dz^{[1]} &= da^{[1]} g'(z^{[1]}) \\ d\omega^{[1]} &= dz^{[1]} a^{[0]} \end{aligned}$$

3. ?

# Deep Learning

## Backpropagation

### 1. Forward Propagation



Final Equations-

$$d a^{[2]} = a^{[2]} - y$$
$$d z^{[2]} = d a^{[2]} g'(z^{[2]})$$
$$d \omega^{[2]} = d z^{[2]} a^{[1]}$$
$$d a^{[1]} = d z^{[2]} \omega^{[2]}$$
$$d z^{[1]} = d a^{[1]} g'(z^{[1]})$$
$$d \omega^{[1]} = d z^{[1]} a^{[0]}$$

$$\omega^{[1]} = \omega^{[1]} - \alpha \cdot d \omega^{[1]}$$

### 2. Backward Propagation



### 3. Update Weights - Gradient Descent

Equations -

$$z^{[1]} = \omega^{[1]} a^{[0]} + b^{[1]}$$

$$a^{[1]} = \sigma(z^{[1]})$$

$$z^{[2]} = \omega^{[2]} a^{[1]} + b^{[2]}$$

$$a^{[2]} = \sigma(z^{[2]})$$

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

# Today's Outline

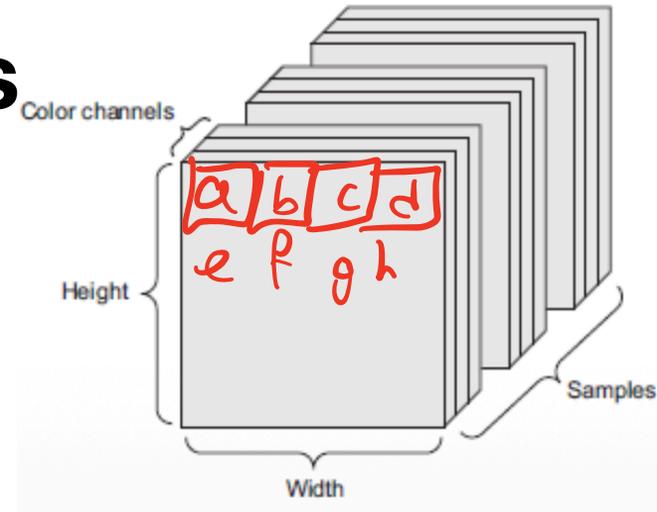
- Deep Learning
- Convolutional Neural Networks



# Convolutional Neural Networks

[a b c d e f g]

a 0  
b 0  
c 0  
d 0  
e 0  
f 0  
g 0  
j 0



- An image is represented as a grid of pixel values
- Lets say this image is 224x224 RGB image
  - Input size after flattening out image = 224 x 224 x 3 = 150,528
  - Input of neural network needs 150,528 neurons
  - Assume first hidden layer has 1000 neurons, first weight matrix will have  $150,528 \times 1000 = 150,528,000 \sim 150$  million parameters in the first layer alone.

15014  $\begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}$   $\begin{matrix} 0 \\ 0 \\ 0 \end{matrix}$   $\begin{matrix} 0 - \text{cat} \\ 0 - \text{dog} \end{matrix}$



# Convolutional Neural Networks

- Key insights from human vision
  - **Local connectivity:** Pixels that are close together are more related than distant pixels.
  - A neuron doesn't need to see the entire image.

# Convolutional Neural Networks

- Key insights from human vision
  - **Local connectivity:** Pixels that are close together are more related than distant pixels.
    - A neuron doesn't need to see the entire image.
  - **Translation invariance:** A feature (like an edge or eye) should be detected regardless of its position in the image.

# Convolutional Neural Networks

- Key insights from human vision
  - **Local connectivity:** Pixels that are close together are more related than distant pixels.
    - A neuron doesn't need to see the entire image.
  - **Translation invariance:** A feature (like an edge or eye) should be detected regardless of its position in the image.
  - **Hierarchical structure:** Simple features (edges) combine to form complex features (shapes → objects).

# Convolutional Neural Networks

- CNNs exploit these insights through:
  - • **Local receptive fields**: Each neuron sees only a small region
  - • **Weight sharing**: Same detector (filter) applied across entire image
  - • **Pooling**: Reduces spatial dimensions, adds invariance

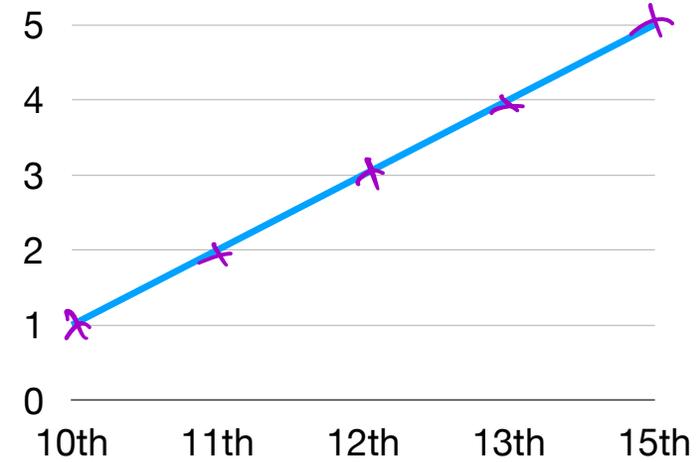
# Convolutional Neural Networks

- Understanding a 1D Filter:

- Signal: [1 2 3 4 5] ← data.

- Filter: [1 0 -1]

$$(1 \times 1) + (0 \times 2) + (3 \times -1) = -2$$



# Convolutional Neural Networks

- Understanding a 1D Filter:

- Signal:  $[1 \ 2 \ 3 \ 4 \ 5]$

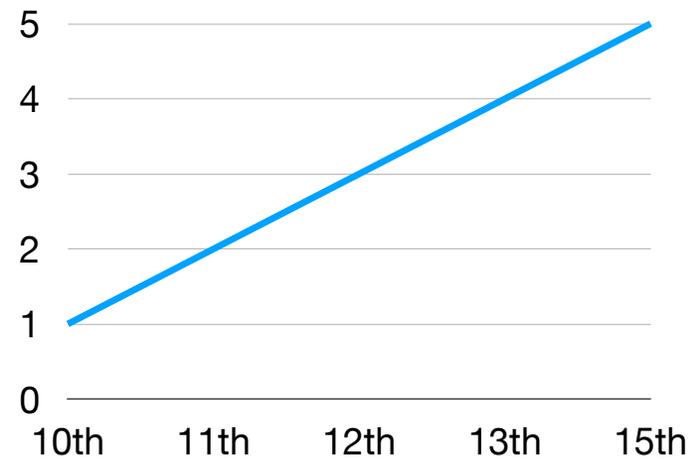
- Filter:  $[1 \ 0 \ -1]$

- Position 0:  $(1 \cdot 1) + (2 \cdot 0) + (2 \cdot -1) = -2$

- Position 1:  $(2 \cdot 1) + (3 \cdot 0) + (4 \cdot -1) = -2$

- Position 2:  $(3 \cdot 1) + (4 \cdot 0) + (5 \cdot -1) = -2$

- Output:  $[-2 \ -2 \ -2]$



# Convolutional Neural Networks

- Understanding a 1D Filter:

- Signal: [1 2 3 4 5]

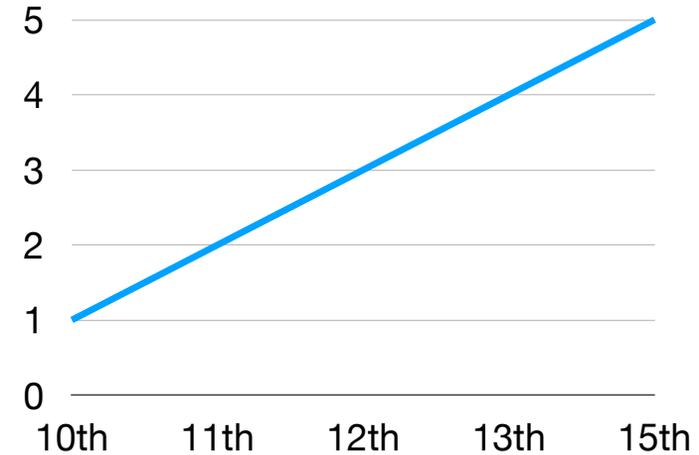
- Filter: [1 0 -1]

- Position 0:  $(1 \cdot 1) + (2 \cdot 0) + (2 \cdot -1) = -2$

- Position 1:  $(2 \cdot 1) + (3 \cdot 0) + (4 \cdot -1) = -2$

- Position 2:  $(3 \cdot 1) + (4 \cdot 0) + (5 \cdot -1) = -2$

- Output: [-2 -2 -2]



# Convolutional Neural Networks

- Understanding a 1D Filter:

- Signal: [1 2 3 4 5]

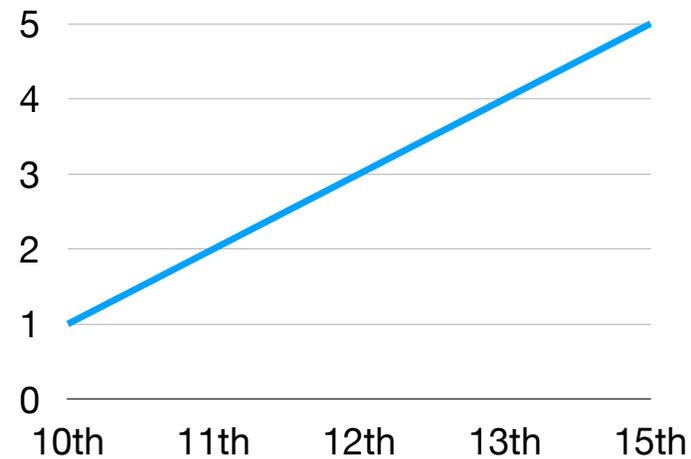
- Filter: [1 0 -1]

- Position 0:  $(1 \cdot 1) + (2 \cdot 0) + (3 \cdot -1) = -1$

- Position 1:  $(2 \cdot 1) + (3 \cdot 0) + (4 \cdot -1) = -2$

- Position 2:  $(3 \cdot 1) + (4 \cdot 0) + (5 \cdot -1) = -2$

- Output: [-2 -2 -2]

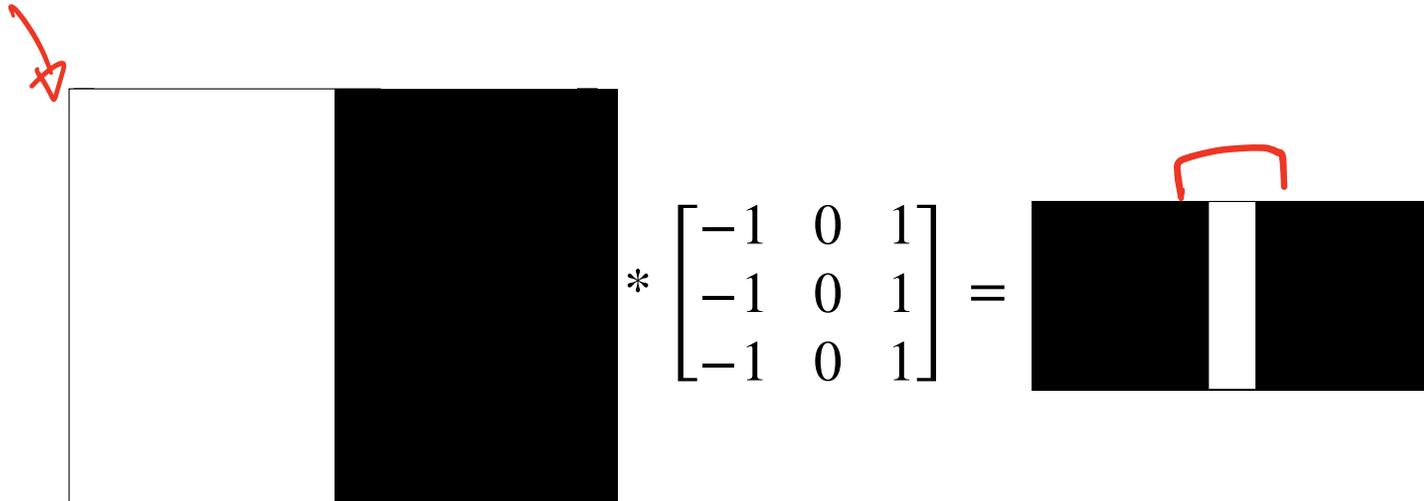


# Convolutional Neural Networks

- These filters “slide” over the entire dataset
- Each filter learns something about the data
- 2D filters similarly slide over an image

# Convolutional Neural Networks

$$\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \text{ — Vertical Edge Filter}$$



# Convolutional Neural Networks

$$\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \text{ — Vertical Edge Filter}$$

$$\begin{bmatrix} 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 30 & -30 & 0 \\ 0 & 30 & -30 & 0 \\ 0 & 30 & -30 & 0 \end{bmatrix}$$

# Convolutional Neural Networks

$$\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \text{ — Vertical Edge Filter}$$

$$\begin{bmatrix} 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 30 & -30 & 0 \\ 0 & 30 & -30 & 0 \\ 0 & 30 & -30 & 0 \end{bmatrix}$$

# Convolutional Neural Networks

$$\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \text{ -- Vertical Edge Filter}$$

$$\begin{aligned} & (-1)(10) + (-1)(10) + (-1)(0) \\ & + (0)(0) + (0)(0) + (0)(0) \\ & + (0)(0) + (0)(0) + (0)(0) \\ & = \underline{\underline{-30}} \end{aligned}$$

$$\begin{bmatrix} 10 & 10 & 10 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 \end{bmatrix} * \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 30 & -30 & 0 \\ 0 & 30 & -30 & 0 \\ 0 & 30 & -30 & 0 \end{bmatrix}$$

# Convolutional Neural Networks

$$\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \text{ — Vertical Edge Filter}$$

$$\begin{bmatrix} 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 30 & -30 & 0 \\ 0 & 30 & -30 & 0 \\ 0 & 30 & -30 & 0 \end{bmatrix}$$

# Convolutional Neural Networks

$$\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \text{ — Vertical Edge Filter}$$

$$\begin{bmatrix} 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 30 & -30 & 0 \\ 0 & 30 & -30 & 0 \\ 0 & 30 & -30 & 0 \end{bmatrix}$$

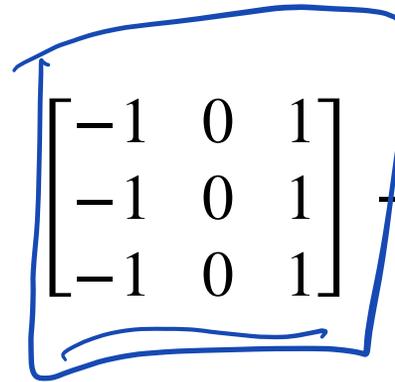
# Convolutional Neural Networks

$$\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \text{ -- Vertical Edge Filter}$$

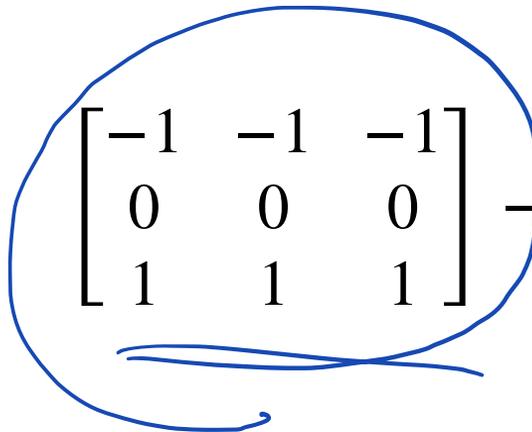


$$\begin{bmatrix} 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 30 & -30 & 0 \\ 0 & 30 & -30 & 0 \\ 0 & 30 & -30 & 0 \\ 0 & 30 & -30 & 0 \end{bmatrix}$$

# Convolutional Neural Networks

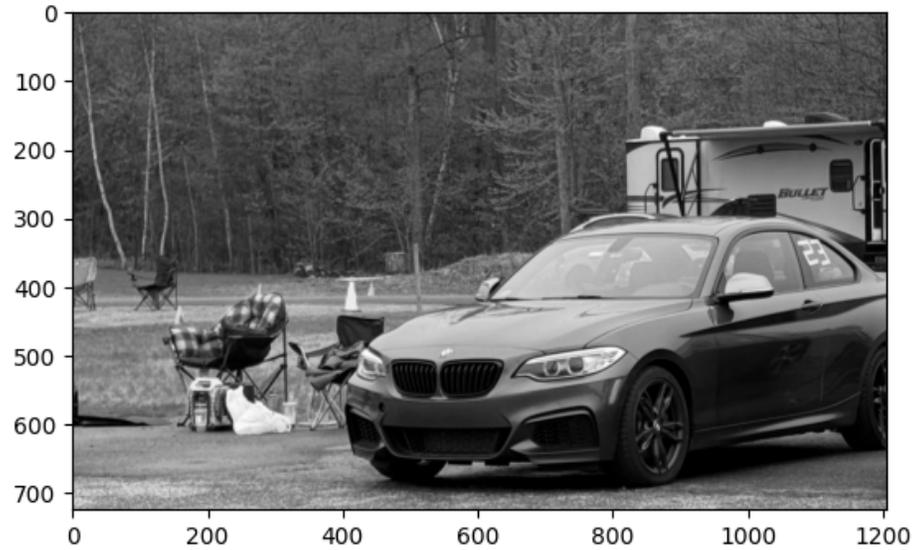

$$\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \text{ — Vertical Edge Filter}$$

$$\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \text{ — Blur (Averaging)}$$


$$\begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} \text{ — Horizontal Edge Filter}$$

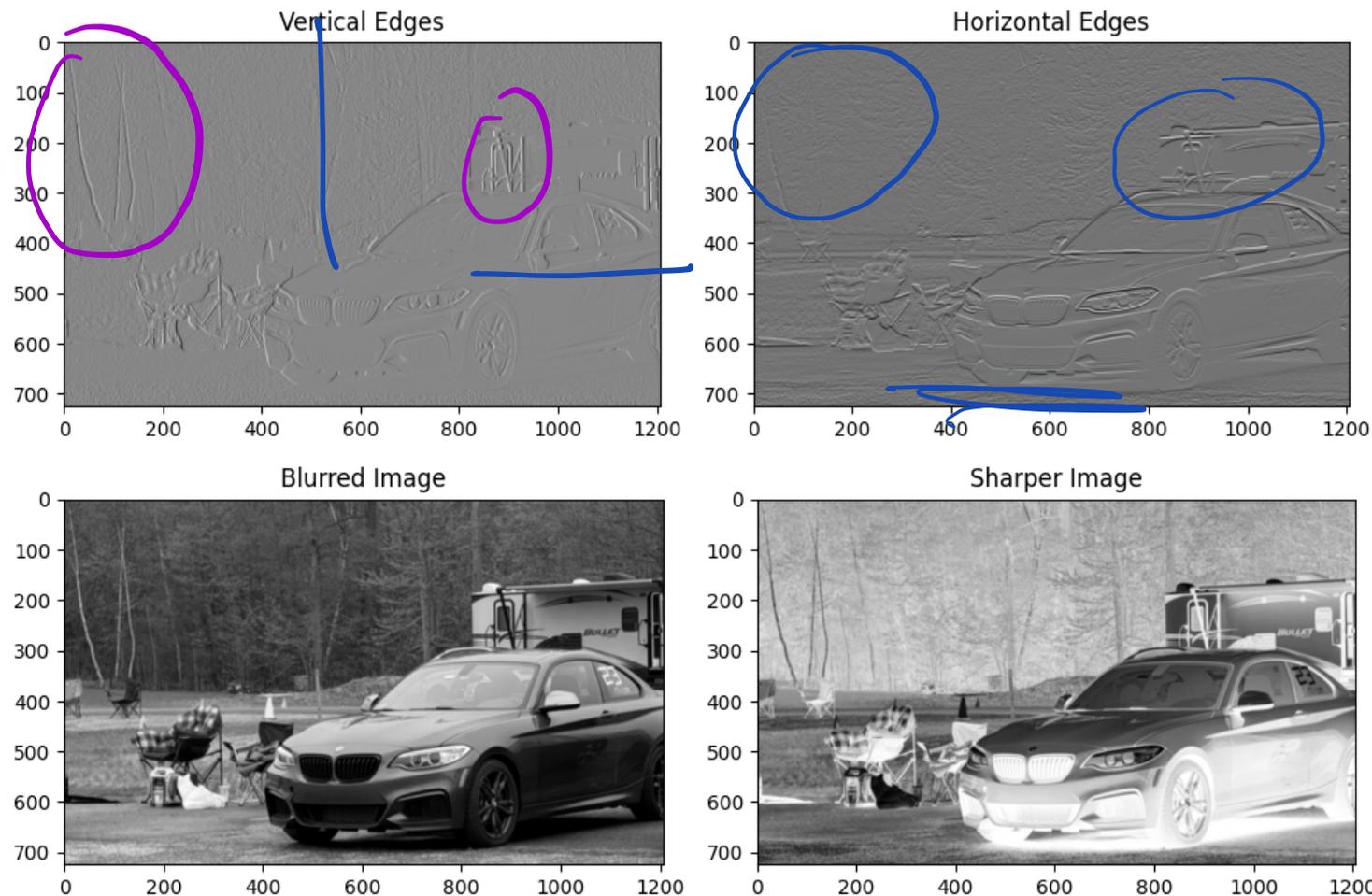
$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix} \text{ — Sharpen}$$

# Convolutional Neural Networks



Original

# Convolutional Neural Networks

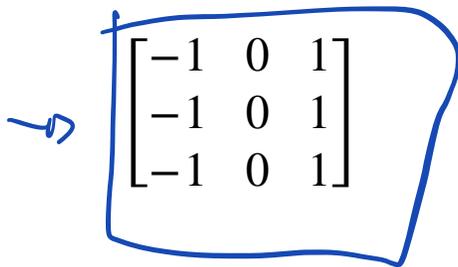


# Convolutional Neural Networks

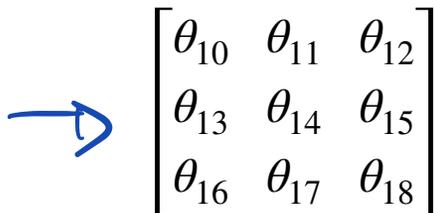
- Key idea of CNNs - **learn filter values** rather than hand designing them
- A convolutional layer has **multiple filters**, each producing one output channel (feature map).

# Convolutional Neural Networks

- Key idea of CNNs - **learn filter values** rather than hand designing them
- A convolutional layer has **multiple filters**, each producing one output channel (feature map).
- Instead of having a fixed vertical edge filter like


$$\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

- You learn filters like


$$\begin{bmatrix} \theta_{10} & \theta_{11} & \theta_{12} \\ \theta_{13} & \theta_{14} & \theta_{15} \\ \theta_{16} & \theta_{17} & \theta_{18} \end{bmatrix}$$

# Convolutional Neural Networks

- Key idea of CNNs - **learn filter values** rather than hand designing them
- A convolutional layer has **multiple filters**, each producing one output channel (feature map).
- Instead of having a fixed vertical edge filter like

$$\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

$5 \times 5 = 25$   
 $5 \times 5 \times 3 = 75$

- You learn filters like

$3 \times 3 = 9$

Filter 1:  $\begin{bmatrix} \theta_{10} & \theta_{11} & \theta_{12} \\ \theta_{13} & \theta_{14} & \theta_{15} \\ \theta_{16} & \theta_{17} & \theta_{18} \end{bmatrix}$  Filter 2:  $\begin{bmatrix} \theta_{20} & \theta_{21} & \theta_{22} \\ \theta_{23} & \theta_{24} & \theta_{25} \\ \theta_{26} & \theta_{27} & \theta_{28} \end{bmatrix}$

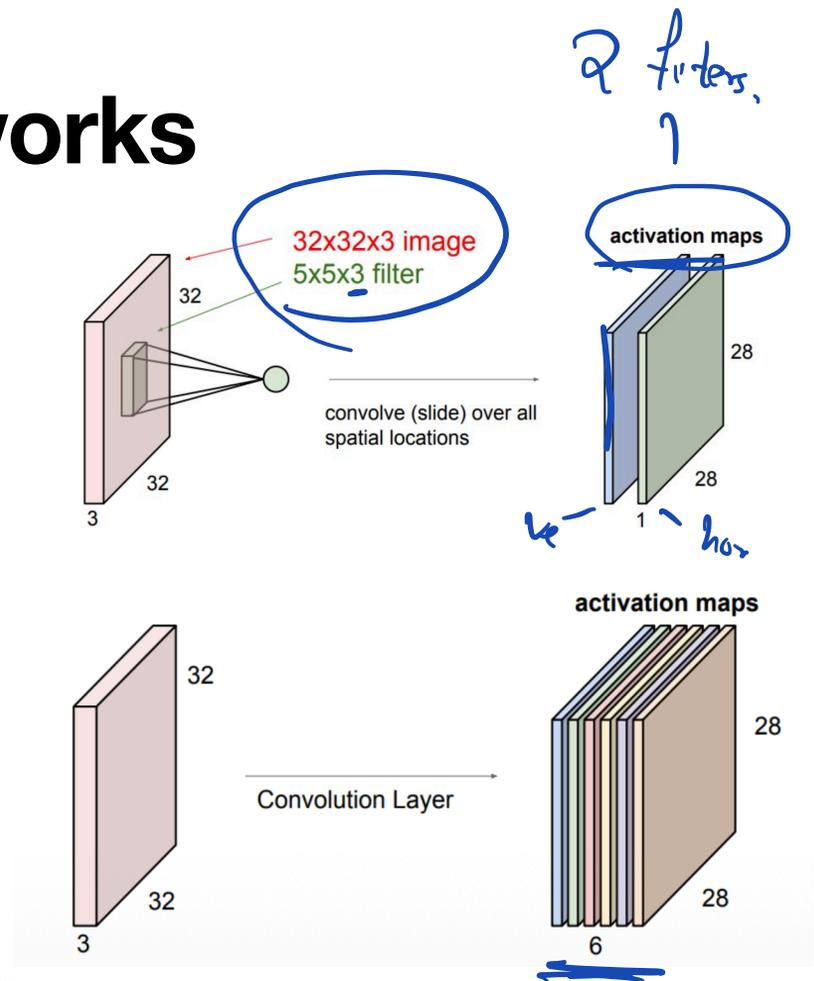


# Convolutional Neural Networks

- Input:  $H_{in} \times W_{in} \times C_{in}$  (height x width x channels)
- Output:  $H_{out} \times W_{out} \times C_{out}$  (one channel per filter)
- Each filter has shape  $k \times k \times C_{in}$

## • Example

- Input Image:  $32 \times 32 \times 3$  (RGB Image), 6 filters
- Each filter:  $5 \times 5 \times 3 = 75$  parameters
- Output Shape:  $28 \times 28 \times 6$  (why 28?)
- Total parameters =  $75 \times 6 = 450$ , compared to  $32 \times 32 \times 3 \times 1000 = 3M$  parameters for MLP



# Convolutional Neural Networks

## Stride

- Output Shape:  $28 \times 28 \times 16$  (why 28?)
- Stride captures how many pixels the filter moves each step

$$\begin{bmatrix} 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 30 & -30 & 0 \\ 0 & 30 & -30 & 0 \\ 0 & 30 & -30 & 0 \end{bmatrix}$$

# Convolutional Neural Networks

## Stride

- Output Shape:  $28 \times 28 \times 16$  (why 28?)
- Stride captures how many pixels the filter moves each step

$$\begin{bmatrix} 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 30 & -30 & 0 \\ 0 & 30 & -30 & 0 \\ 0 & 30 & -30 & 0 \end{bmatrix}$$

# Convolutional Neural Networks

## Stride

- Output Shape:  $28 \times 28 \times 16$  (why 28?)
- Stride captures how many pixels the filter moves each step

$$\begin{bmatrix} 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 30 & -30 & 0 \\ 0 & 30 & -30 & 0 \\ 0 & 30 & -30 & 0 \end{bmatrix}$$

# Convolutional Neural Networks

## Stride

- Output Shape:  $28 \times 28 \times 16$  (why 28?)
- Stride captures how many pixels the filter moves each step

$$\begin{bmatrix} 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 30 & -30 & 0 \\ 0 & 30 & -30 & 0 \\ 0 & 30 & -30 & 0 \end{bmatrix}$$

Stride = 1

# Convolutional Neural Networks

## Stride and Padding



- Output Shape:  $28 \times 28 \times 16$  (why 28?)
- Stride captures how many pixels the filter moves each step

$$\begin{bmatrix} 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 30 & -30 & 0 \\ 0 & 30 & -30 & 0 \\ 0 & 30 & -30 & 0 \end{bmatrix}$$

Stride = 2

# Convolutional Neural Networks

## Stride and Padding

- Output Shape:  $28 \times 28 \times 16$  (why 28?)
- Stride captures how many pixels the filter moves each step

$$\begin{bmatrix} 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 30 & -30 & 0 \\ 0 & 30 & -30 & 0 \\ 0 & 30 & -30 & 0 \end{bmatrix}$$

Stride = 2

# Convolutional Neural Networks

## Stride and Padding

- Output Shape:  $28 \times 28 \times 16$  (why 28?)
- Stride captures how many pixels the filter moves each step

$$\begin{bmatrix} 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 30 & -30 & 0 \\ 0 & 30 & -30 & 0 \\ 0 & 30 & -30 & 0 \end{bmatrix}$$

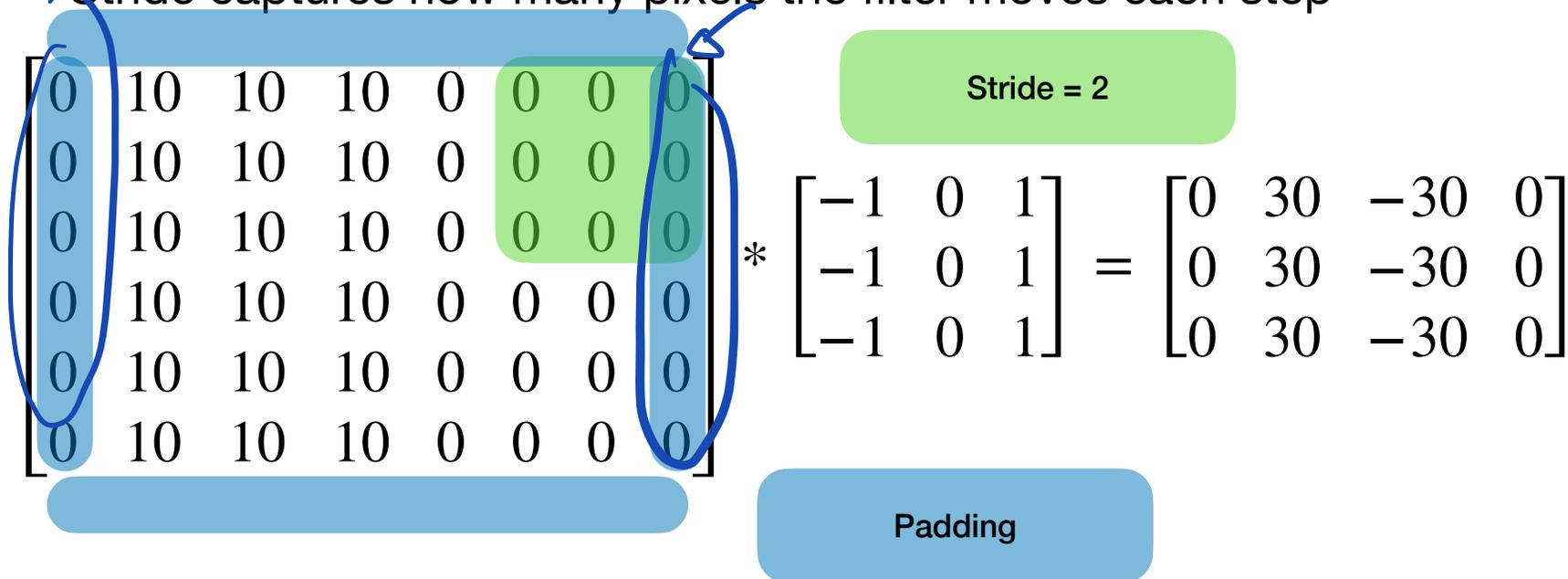
Stride = 2

Next Stride?

# Convolutional Neural Networks

## Stride and Padding

- Output Shape: 28 × 28 × 16 (why 28?)
- Stride captures how many pixels the filter moves each step



# Convolutional Neural Networks

## Stride and Padding

- Output Shape:  $28 \times 28 \times 16$  (why 28?)
- Stride captures how many pixels the filter moves each step
- Stride and padding change the shape of the output matrix
- With no padding, input size  $n$ , filter size  $k$  and stride  $s$  :

• Output Size =  $\lfloor \frac{n - k}{s} \rfloor + 1$  —

- With padding  $p$ :

• Output Size =  $\lfloor \frac{n + 2p - k}{s} \rfloor + 1$

$n$  = input dim  
 $k$  = filter size  
 $s$  = stride

$32 \times 32$   
 $s = 5$   
↓

$\lfloor \frac{32 - 5}{1} \rfloor + 1 = 28$

# Convolutional Neural Networks

## Stride and Padding

$$\text{Output Size} = \left\lfloor \frac{n + 2p - k}{s} \right\rfloor + 1$$


Input (s)	Filter (k)	Padding (p)	Stride (s)	Output
32x32	<u>3</u>	0	<u>1</u>	<u>?</u>
32x32	3	1	1	?
32x32	3	2	2	?

# Convolutional Neural Networks

## Stride and Padding

$$\text{Output Size} = \left\lfloor \frac{n + 2p - k}{s} \right\rfloor + 1$$


Input (s)	Filter (k)	Padding (p)	Stride (s)	Output
32x32	3	0	1	30
32x32	3	1	1	32
32x32	3	2	2	17

# Convolutional Neural Networks

## Stride and Padding

$\lfloor \frac{3}{2} \rfloor = 1$

$$\text{Output Size} = \left\lfloor \frac{n + 2p - k}{s} \right\rfloor + 1$$

Input (s)	Filter (k)	Padding (p)	Stride (s)	Output
32x32	3	0	1	30
32x32	<u>3</u>	1	1	32
32x32	3	2	2	17

To keep same output size,  
pad by  $\lfloor \frac{k}{2} \rfloor$  on each side

# Convolutional Neural Networks

## Pooling

- Pooling reduces spatial dimensions, providing:
  - Translation invariance
  - Reduced computation
  - Larger receptive field
- Example - **Max** Pool = (3x3) with stride = 3

$$\begin{bmatrix} 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 10 & 0 \\ 10 & 0 \end{bmatrix}$$

# Convolutional Neural Networks

## Pooling

- Pooling reduces spatial dimensions, providing:
  - Translation invariance
  - Reduced computation
  - Larger receptive field
- Example - **Max Pool** = (3x3) with stride = 3

$$\begin{bmatrix} 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 10 & 0 \\ 10 & 0 \end{bmatrix}$$

# Convolutional Neural Networks

## Pooling

- Pooling reduces spatial dimensions, providing:
  - Translation invariance
  - Reduced computation
  - Larger receptive field
- Example - **Max** Pool = (3x3) with stride = 3

$$\begin{bmatrix} 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 10 & 0 \\ 10 & 0 \end{bmatrix}$$

# Convolutional Neural Networks

## Pooling

- Pooling reduces spatial dimensions, providing:
  - Translation invariance
  - Reduced computation
  - Larger receptive field
- Example - **Max** Pool = (3x3) with stride = 3

$$\begin{bmatrix} 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 10 & 0 \\ 10 & 0 \end{bmatrix}$$

# Convolutional Neural Networks

## Pooling

- Pooling reduces spatial dimensions, providing:
  - Translation invariance
  - Reduced computation
  - Larger receptive field
- Example - **Max** Pool = (3x3) with stride = 3

$$\begin{bmatrix} 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 10 & 0 \\ 10 & 0 \end{bmatrix}$$

# Convolutional Neural Networks

## Pooling

- Pooling reduces spatial dimensions, providing:
  - Translation invariance
  - Reduced computation
  - Larger receptive field
- Example - **Average** Pool = (3x3) with stride = 3

$$\begin{bmatrix} 1 & 10 & 1 & 0 & 1 & 0 \\ 1 & 10 & 1 & 0 & 1 & 0 \\ 1 & 10 & 1 & 0 & 1 & 0 \\ 10 & 0 & 0 & 1 & 0 & 0 \\ 10 & 0 & 0 & 1 & 0 & 0 \\ 10 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 4 & 0.3 \\ 3.3 & 0.3 \end{bmatrix}$$

# Convolutional Neural Networks

## Pooling

- Pooling reduces spatial dimensions, providing:
  - Translation invariance
  - Reduced computation
  - Larger receptive field
- Example - **Average** Pool = (3x3) with stride = 3

$$\begin{bmatrix} 1 & 10 & 1 & 0 & 1 & 0 \\ 1 & 10 & 1 & 0 & 1 & 0 \\ 1 & 10 & 1 & 0 & 1 & 0 \\ 10 & 0 & 0 & 1 & 0 & 0 \\ 10 & 0 & 0 & 1 & 0 & 0 \\ 10 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 4 & 0.3 \\ 3.3 & 0.3 \end{bmatrix}$$

# Convolutional Neural Networks

## Pooling

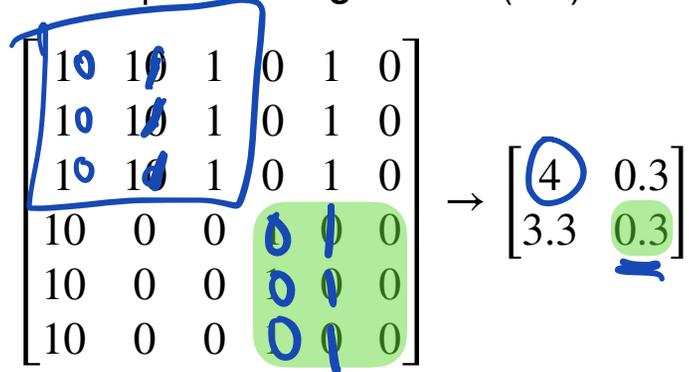
- Pooling reduces spatial dimensions, providing:
  - Translation invariance
  - Reduced computation
  - Larger receptive field
- Example - **Average** Pool = (3x3) with stride = 3

$$\begin{bmatrix} 1 & 10 & 1 & 0 & 1 & 0 \\ 1 & 10 & 1 & 0 & 1 & 0 \\ 1 & 10 & 1 & 0 & 1 & 0 \\ 10 & 0 & 0 & 1 & 0 & 0 \\ 10 & 0 & 0 & 1 & 0 & 0 \\ 10 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 4 & 0.3 \\ 3.3 & 0.3 \end{bmatrix}$$

# Convolutional Neural Networks

## Pooling

- Pooling reduces spatial dimensions, providing:
  - Translation invariance
  - Reduced computation
  - **Larger receptive field**
- Example - **Average Pool** = (3x3) with stride = 3



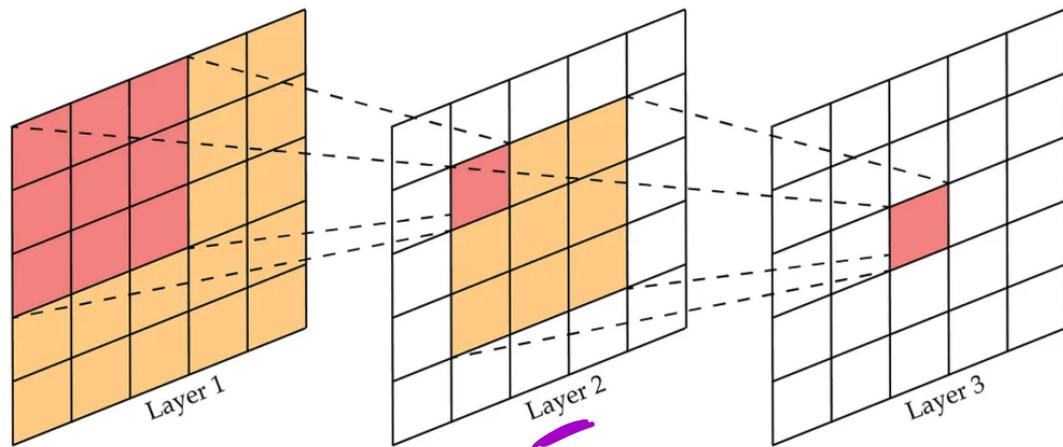
# Convolutional Neural Networks

## Pooling

- The receptive field is the **region of the input** that influences a particular output neuron.
- With stacking, receptive fields grow:
  - Layer 1 (3×3 filter): Each filter sees 3×3 input region
  - Layer 2 (3×3 filter): Each filter sees 5×5 input region
  - Layer 3 (3×3 filter): Each filter sees 7×7 input region

$$\begin{bmatrix} 1 & 10 & 1 & 0 & 1 & 0 \\ 1 & 10 & 1 & 0 & 1 & 0 \\ 1 & 10 & 1 & 0 & 1 & 0 \\ 10 & 0 & 0 & 1 & 0 & 0 \\ 10 & 0 & 0 & 1 & 0 & 0 \\ 10 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 4 & 0.3 \\ 3.3 & 0.3 \end{bmatrix}$$

Receptive Field in Convolutional Networks



# Convolutional Neural Networks

## Pooling

**Why it matters:** Deeper layers have larger receptive fields, allowing them to detect larger, more complex patterns.

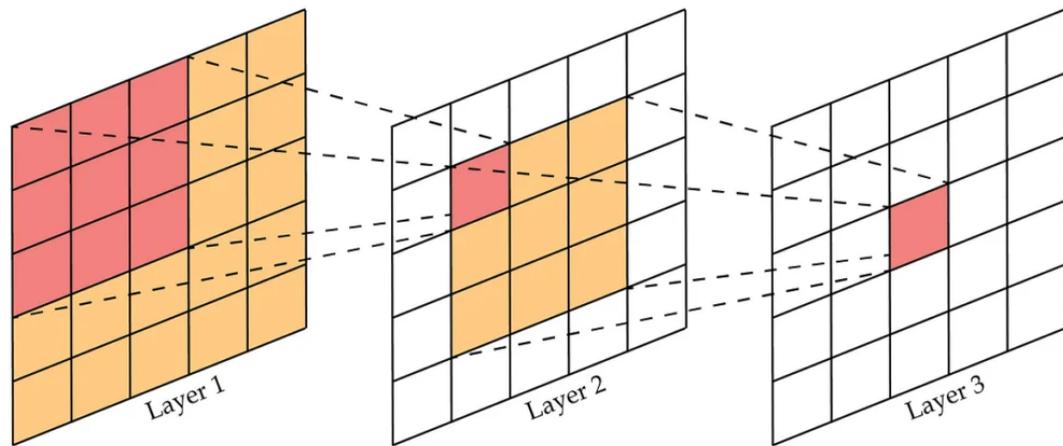
- The receptive field is the **region of the input** that influences a particular output neuron.
- With stacking, receptive fields grow:
  - Layer 1 (3×3 filter): Each filter sees 3×3 input region
  - Layer 2 (3×3 filter): Each filter sees 5×5 input region
  - Layer 3 (3×3 filter): Each filter sees 7×7 input region

$$\begin{bmatrix} 1 & 10 & 1 & 0 & 1 & 0 \\ 1 & 10 & 1 & 0 & 1 & 0 \\ 1 & 10 & 1 & 0 & 1 & 0 \\ 10 & 0 & 0 & 1 & 0 & 0 \\ 10 & 0 & 0 & 1 & 0 & 0 \\ 10 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 4 & 0.3 \\ 3.3 & 0.3 \end{bmatrix}$$

Formula for  $L$  layers of  $k \times k$  filters with stride 1:

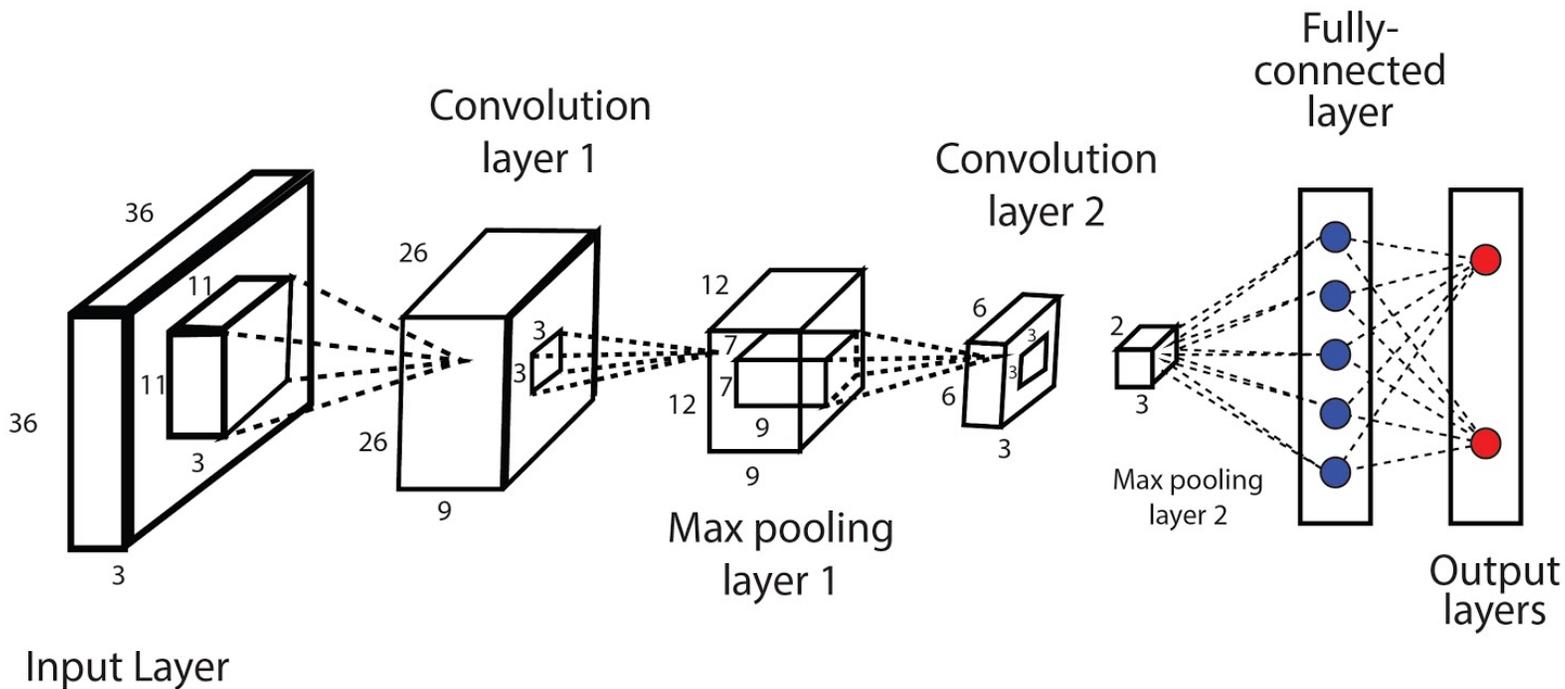
$$RF = L \times (k - 1) + 1$$

Receptive Field in Convolutional Networks



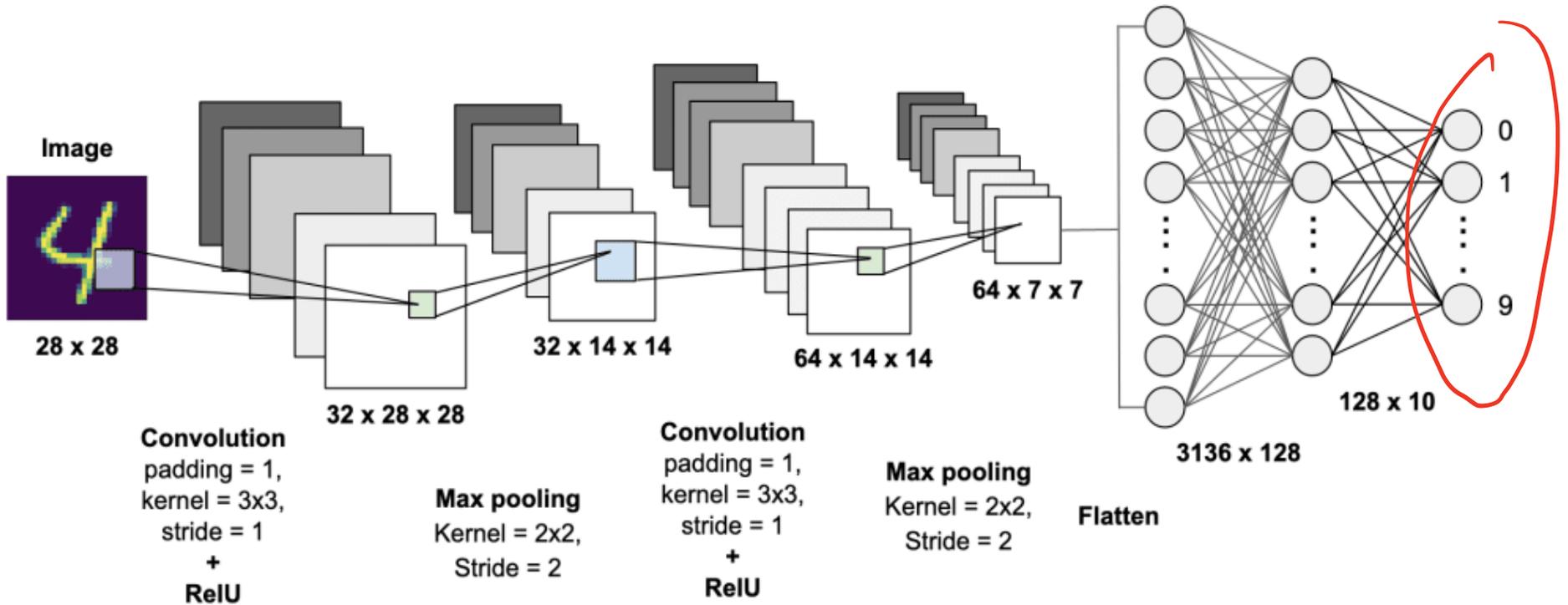
# Convolutional Neural Networks

## Overall Architecture



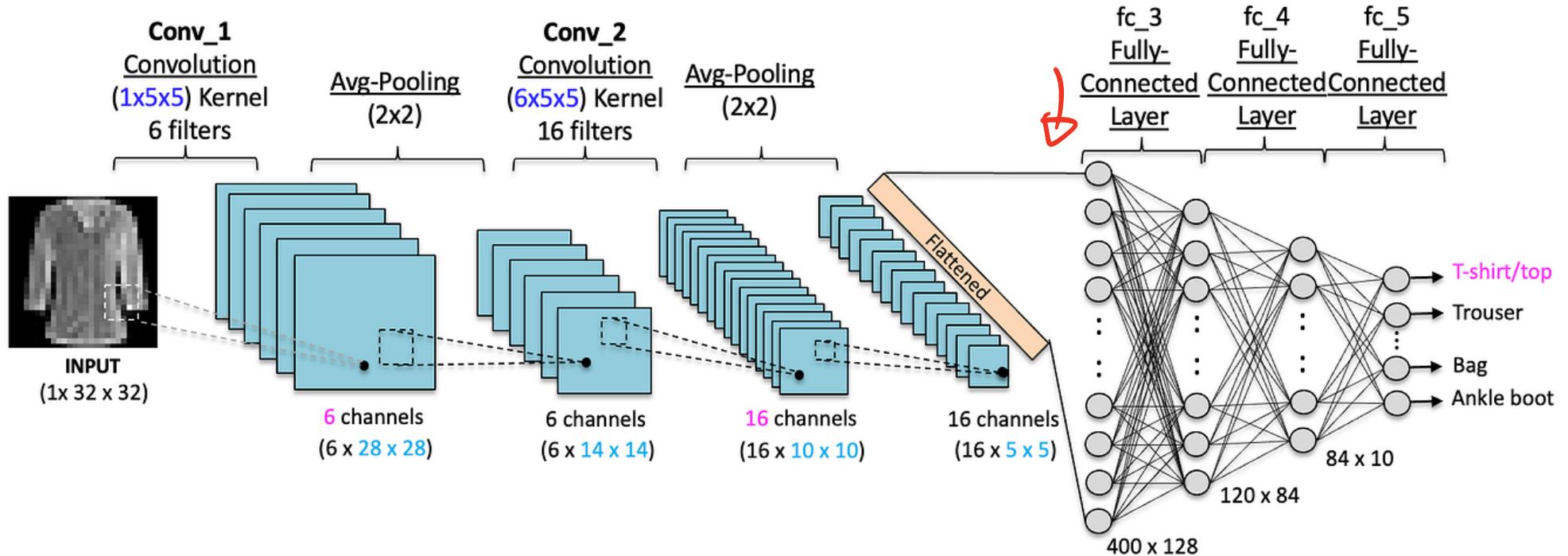
# Convolutional Neural Networks

## Overall Architecture



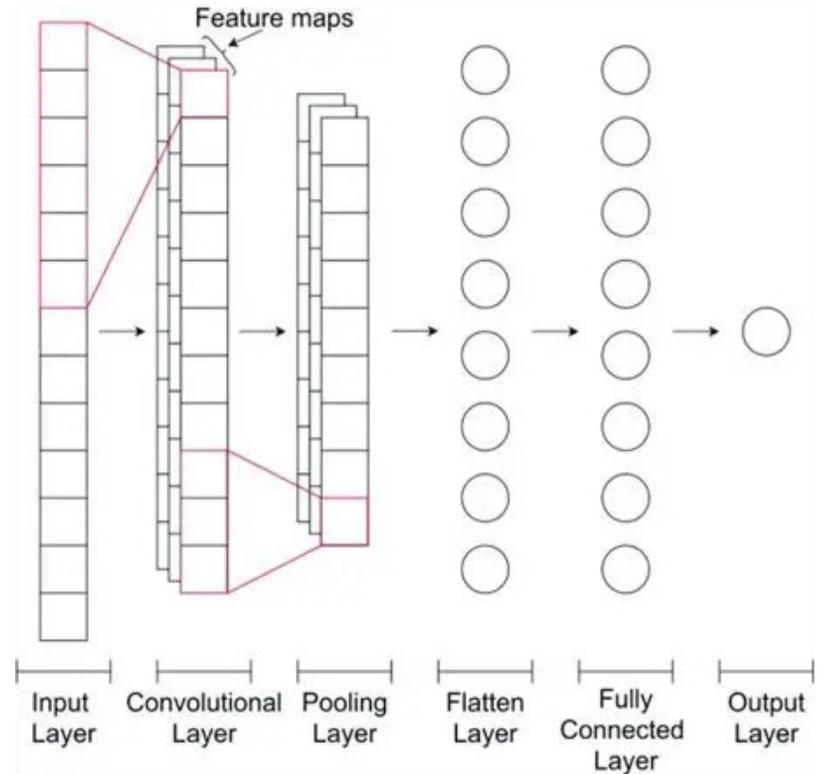
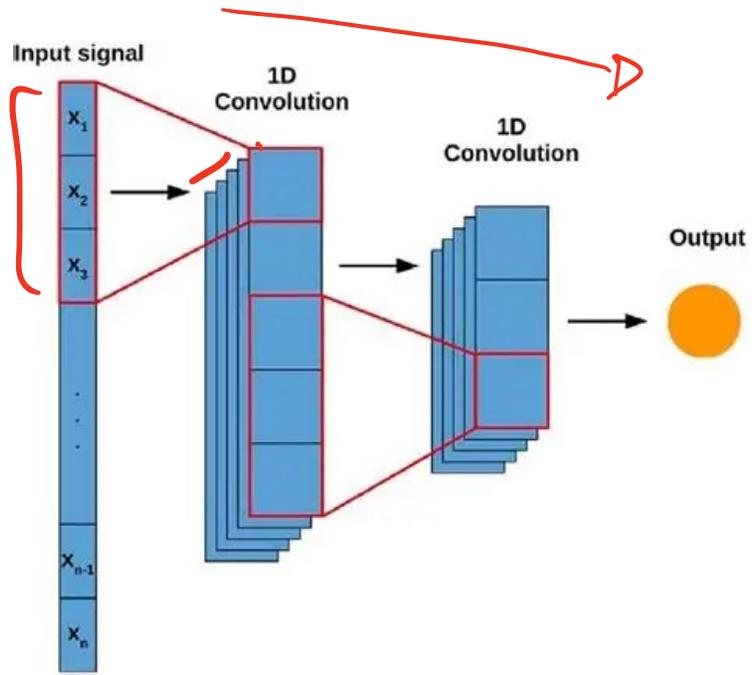
# Convolutional Neural Networks

## Overall Architecture



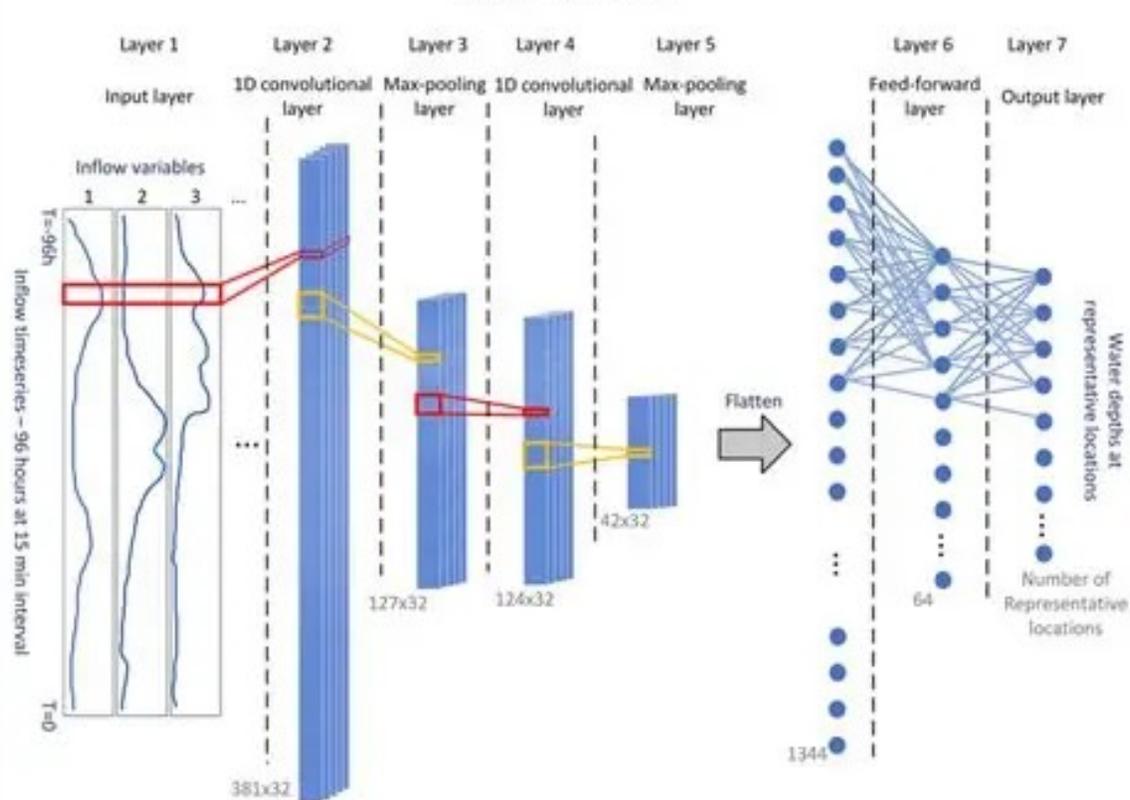
# Convolutional Neural Networks

## Overall Architecture - 1D Example



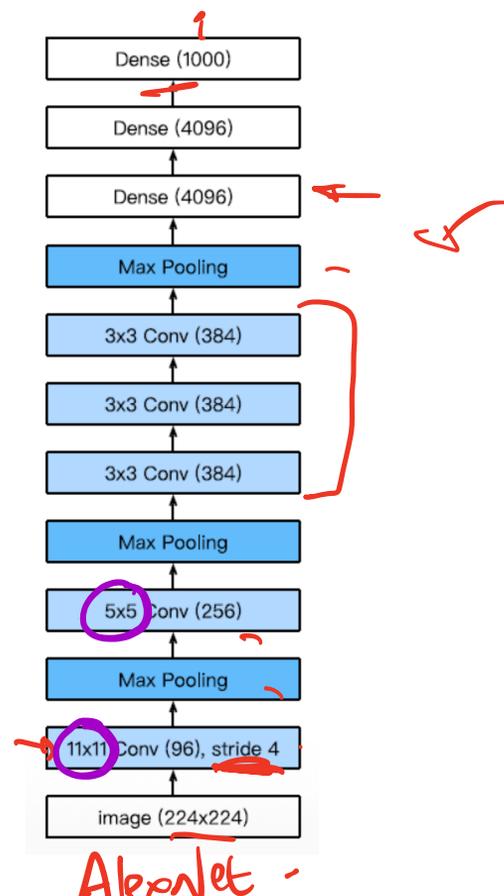
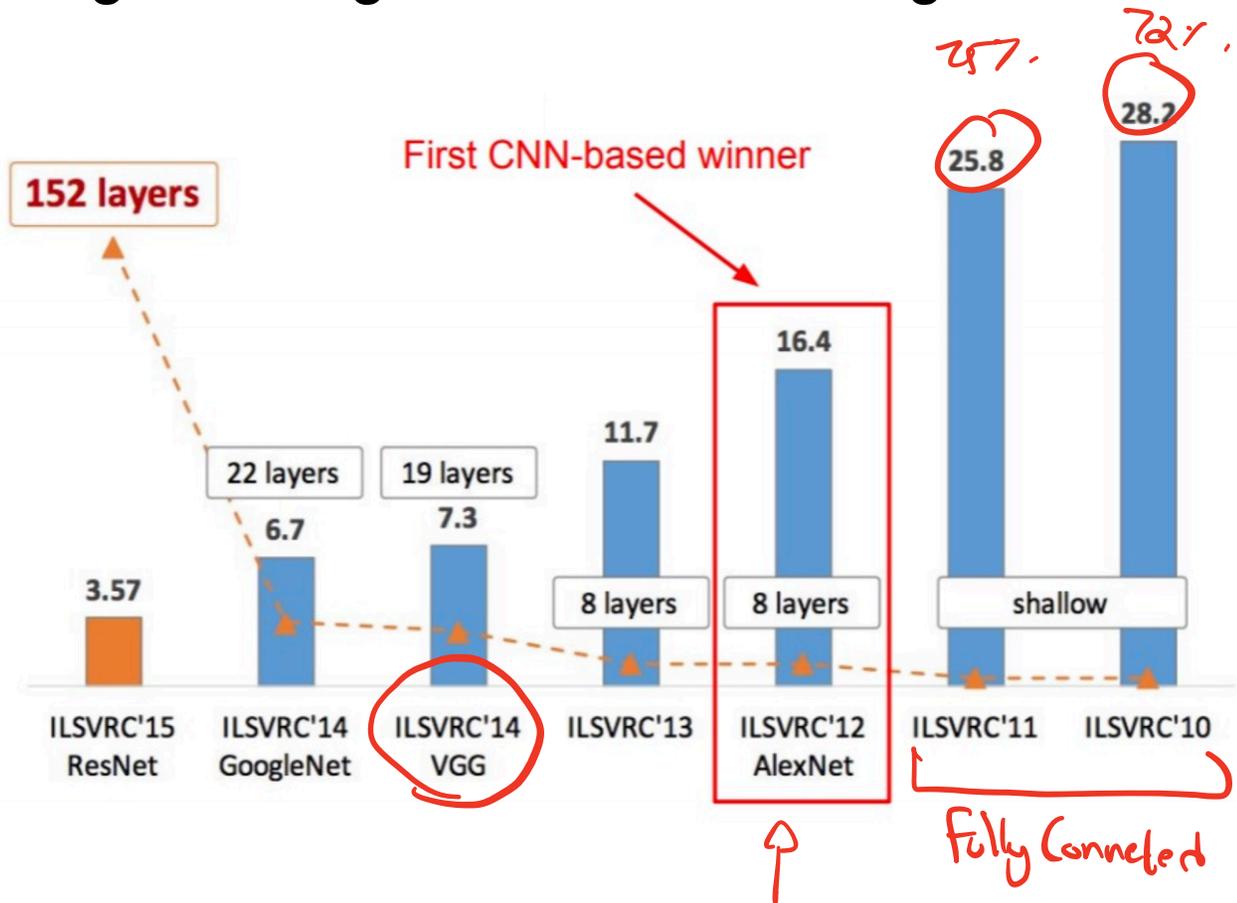
# Convolutional Neural Networks

## Overall Architecture - 1D Example



# Historical Architectures

## ImageNet Large Scale Visual Recognition Challenge (ILSVRC) Winners





# Data Augmentation for Images

## Geometric Transforms

- **Random crop:** Crop random regions from larger image
- **Random horizontal flip:**  $P=0.5$  usually
- **Random rotation:** Small angles ( $\pm 15^\circ$ )
- Random scale/zoom
- Random affine transforms

# Data Augmentation for Images

## Color Transforms

- **Color jitter:** Randomly adjust brightness, contrast, saturation, hue
- **Random grayscale:** Sometimes convert to grayscale
- Channel shuffle/drop

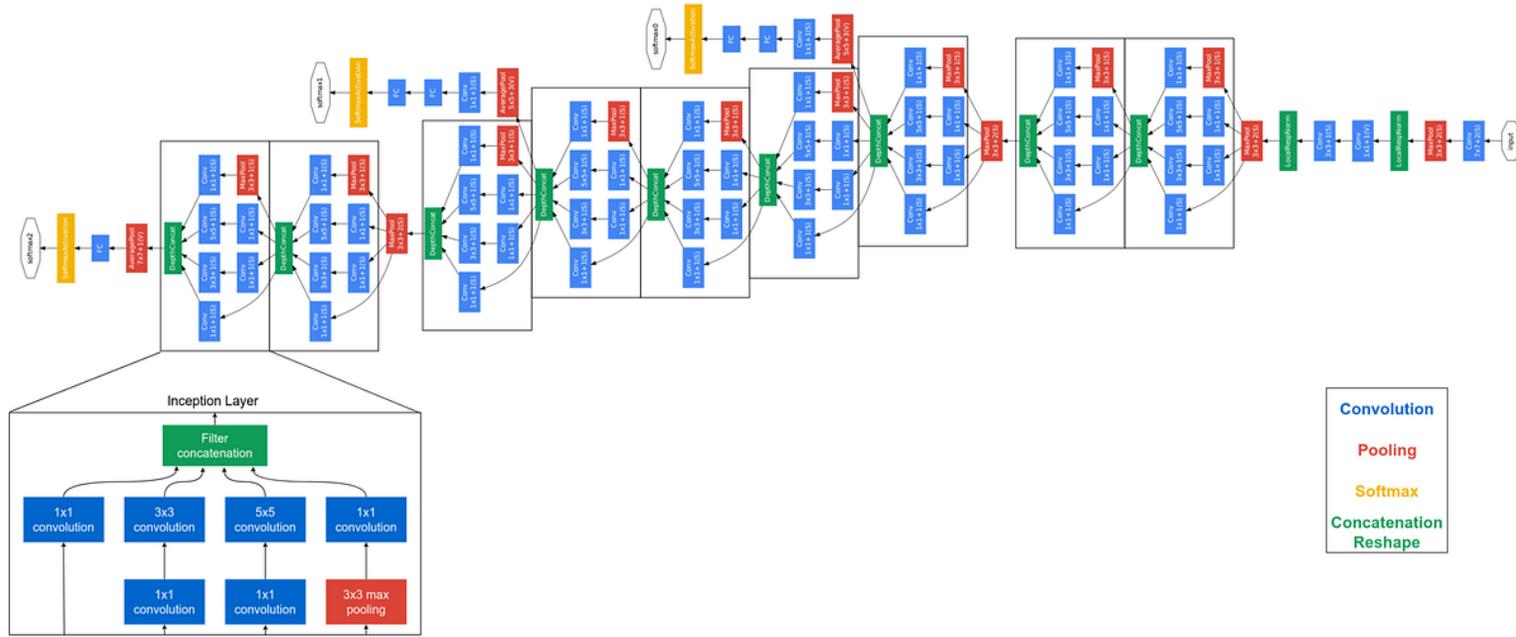
# Data Augmentation for Images

## Advanced Augmentations

- **Cutout:** Randomly mask square regions
- **Mixup:** Blend two images and their labels
- **CutMix:** Cut and paste patches between images

# Transfer Learning

- Training from scratch requires:
  - Millions of labeled images
  - Days of GPU time
  - Expertise in architecture design



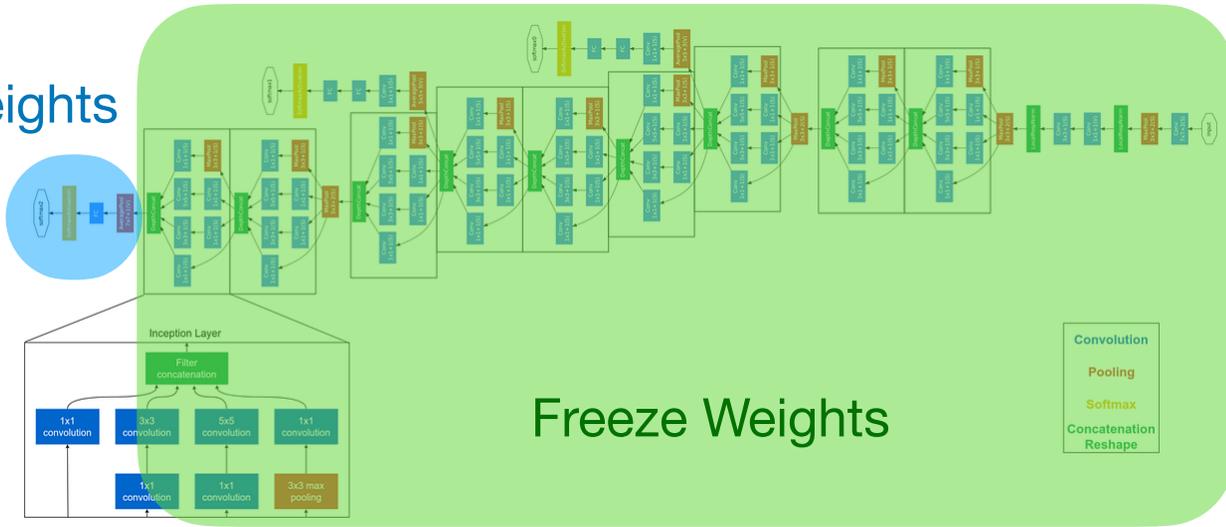
# Transfer Learning

- GoogLeNet pre-trained models have learned:
  - Low-level features (edges, textures) - Layer 1-2
  - Mid-level features (shapes, parts) - Layer 3-4
  - High-level features (object parts) - Layer 5+
- These features are **transferable** to many vision tasks

# Transfer Learning

- These features are **transferable** to many vision tasks
- Key Idea:
  - Freeze pretrained layers, train only new classifier

Re-train Weights



# Transfer Learning

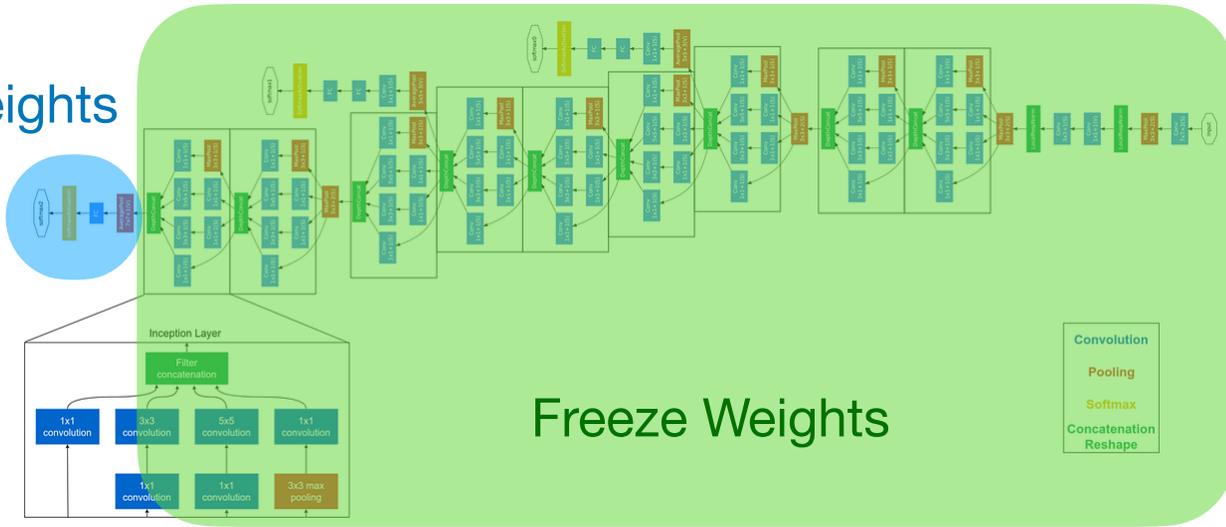
- These features are **transferable** to many vision tasks

- Key Idea:

Fine-tuning Process

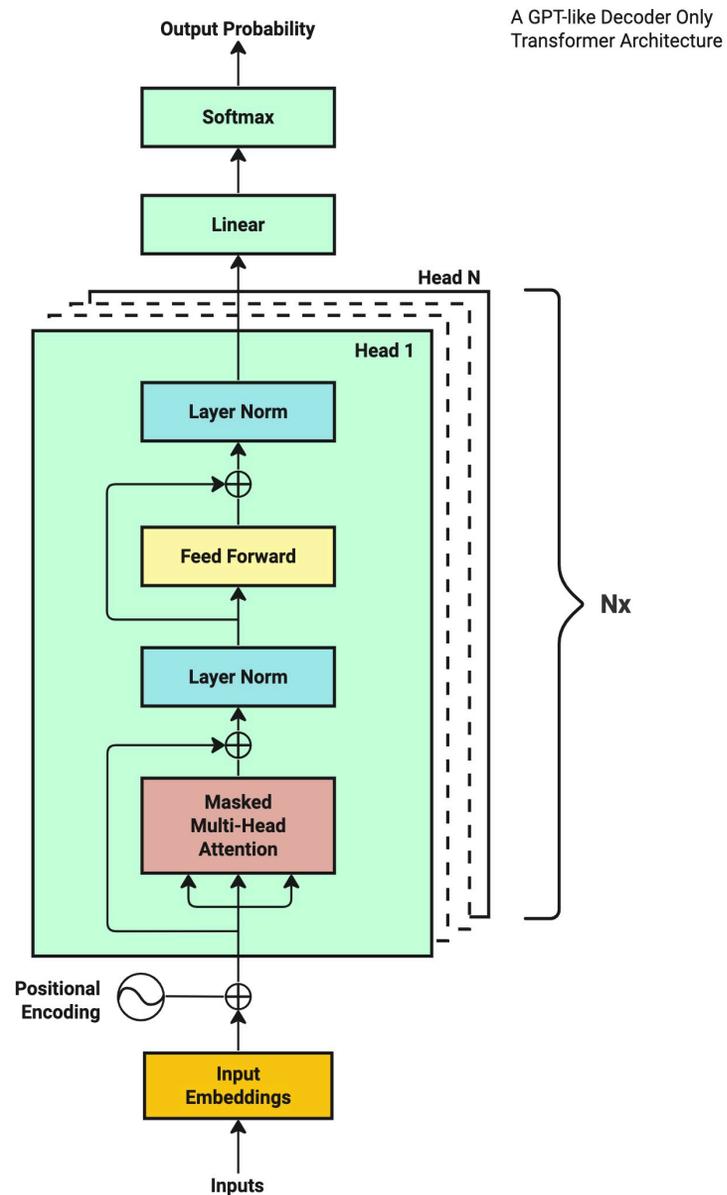
- Freeze pretrained layers, train only new classifier

Re-train Weights



# Transfer Learning

Current LLM Architecture

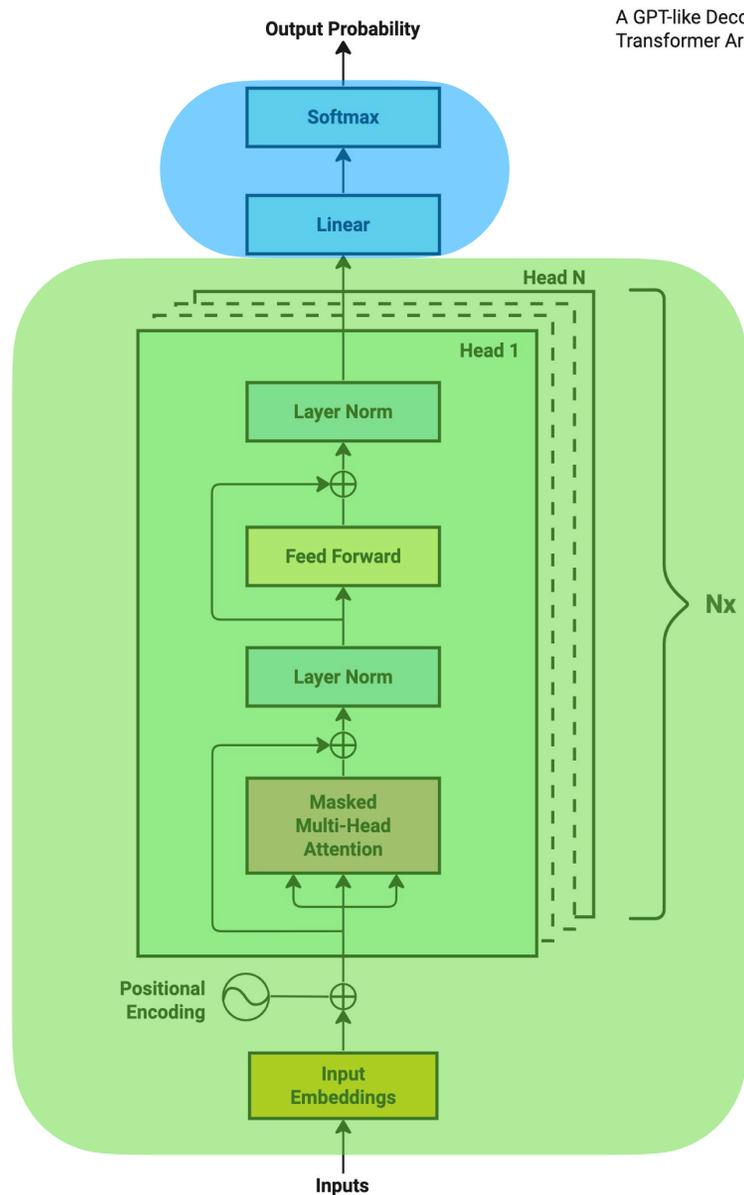


# Transfer Learning

## Current LLM Architecture

A very simplified fine-tuning approach:

1. Keep **initial layers** fixed
2. Retrain **final prediction layers**



# Convolutional Neural Networks

- CNNs exploit these insights through:
  - **Local receptive fields:** Each neuron sees only a small region
  - **Weight sharing:** Same detector (filter) applied across entire image
  - **Pooling:** Reduces spatial dimensions, adds invariance

# Summary

- Applicable to data with natural grid topology
  - Time series
  - Images
- Convolution is a linear operation that uses local information
- Used for dimensionality reduction and learning hierarchical feature representations
- Much fewer parameters relative to Feed-Forward Neural Networks
- Reached human-level performance in ImageNet in 2014