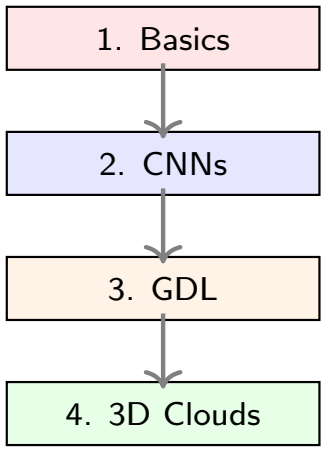
Geometric Deep Learning

Sotirios Nikoloutsopoulos

Presentation Outline

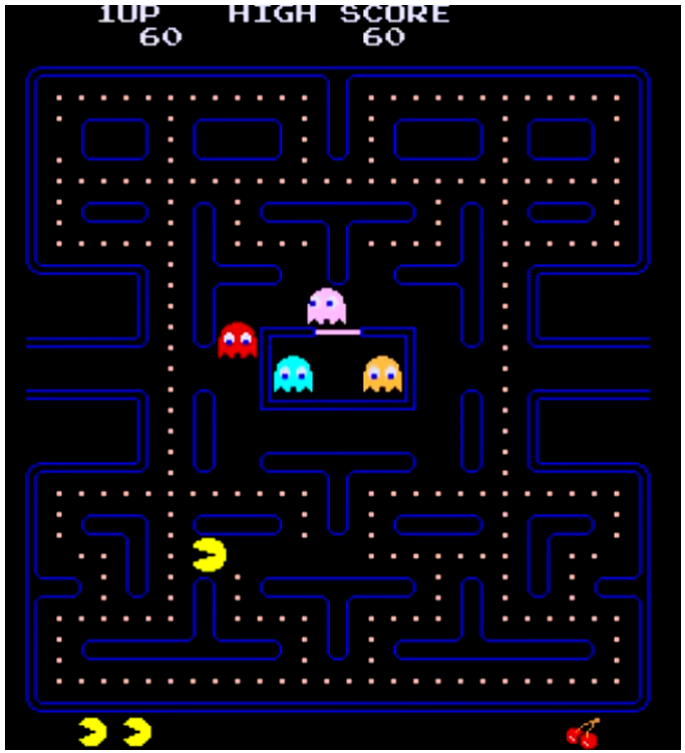
1. **The Big Picture**
   - What is a Neural Network?
   - Memorization: Why raw Pixels are not enough
2. **Classic Vision (CNNs)**
   - How Convolutions work
   - Rotation & Scaling challenges
3. **Geometric Deep Learning (GDL)**
   - Symmetry & Equivariance
   - Steerable Filters
4. **3D Point Clouds**
   - Relational Reasoning (Attention)
   - The Encoder-Decoder Logic



The Big Picture: What is a Neural Network?

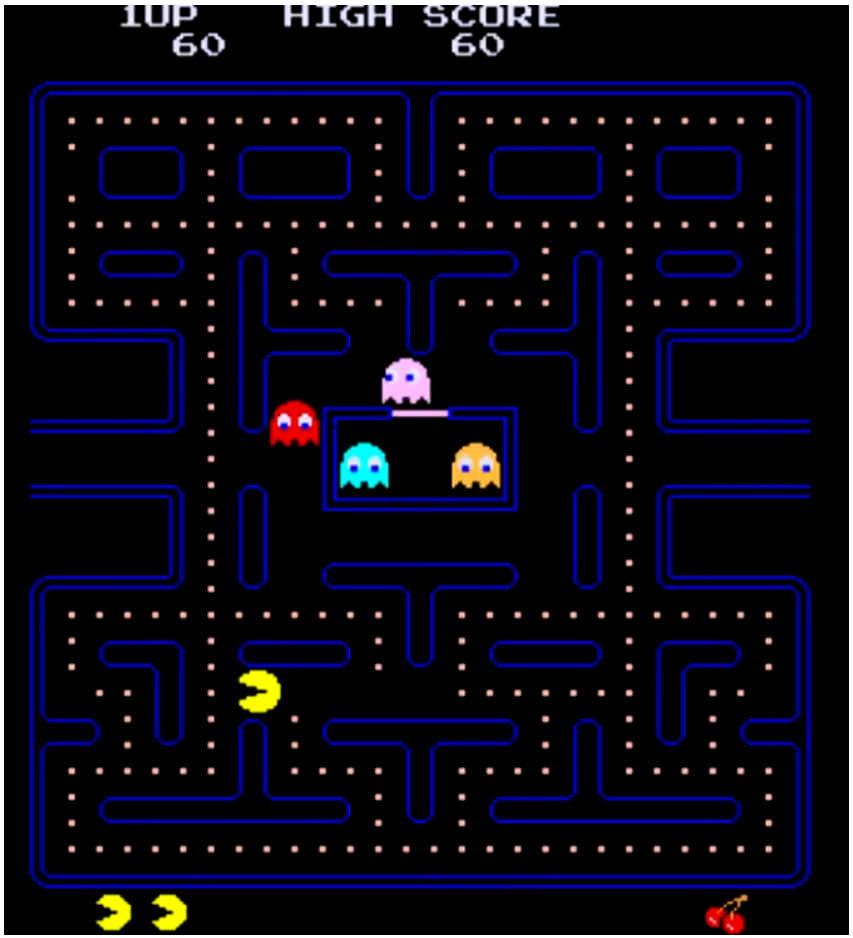
Think of a **Neural Network** as a **function** that takes an **input** and makes a **prediction**.

- **Input:**
  An image (like Pac-Man).
- **The Task:**
  The function should tell us what to do next.



What does the Function learn?

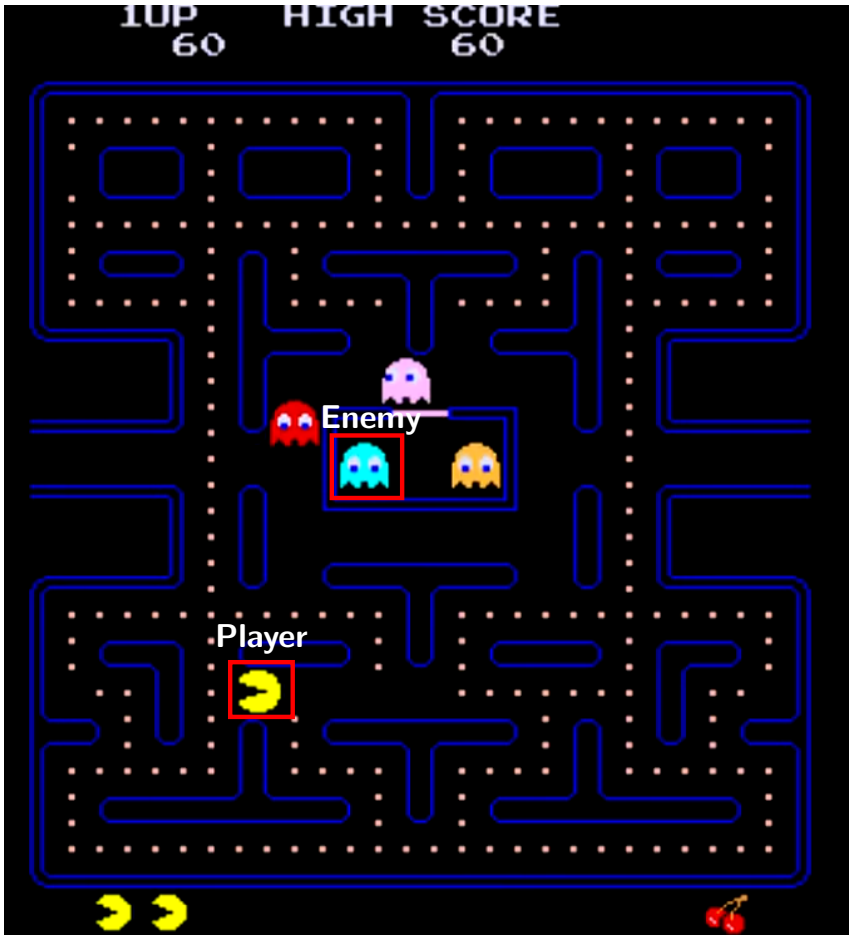
To make a prediction, the function must **see** and **understand** the components of the game.



What does the Function learn?

To make a prediction, the function must **see** and **understand** the components of the game.

- **Classification:**
  Identifying objects. *"Is this a Ghost or a Wall?"*

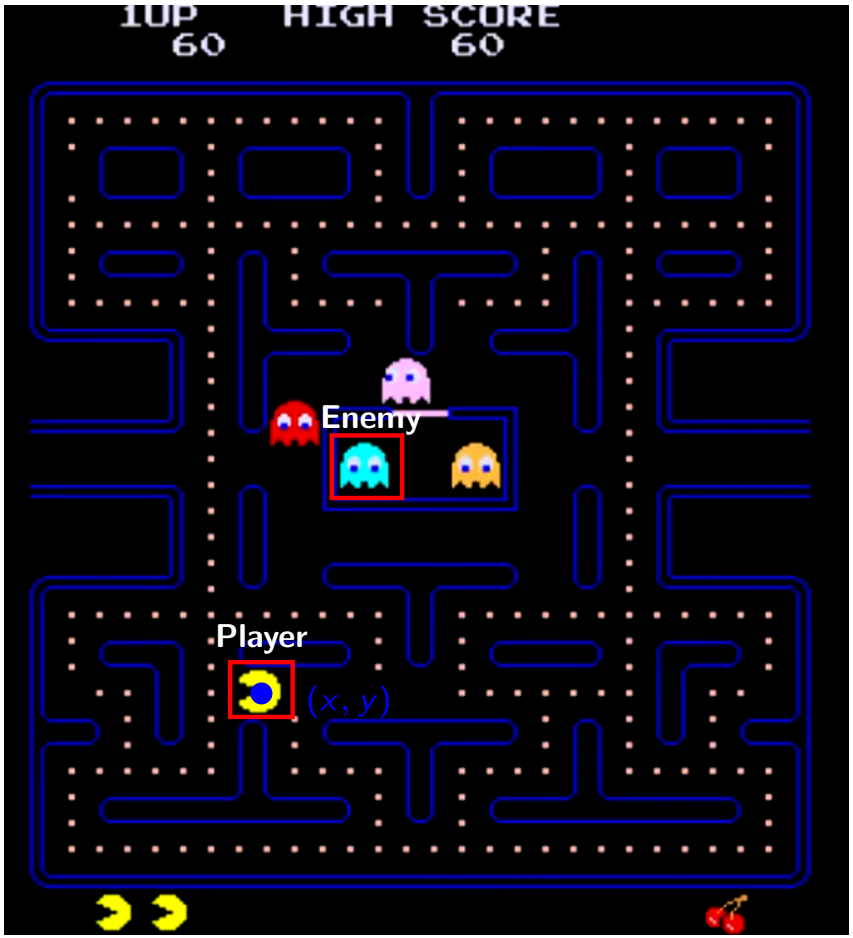


What does the Function learn?

To make a prediction, the function must **see** and **understand** the components of the game.

- **Classification:**
  Identifying objects. *"Is this a Ghost or a Wall?"*

- **Regression:**
  Finding positions. *"Where exactly is Pac-Man?*
  $(x, y) = (12, 45)$



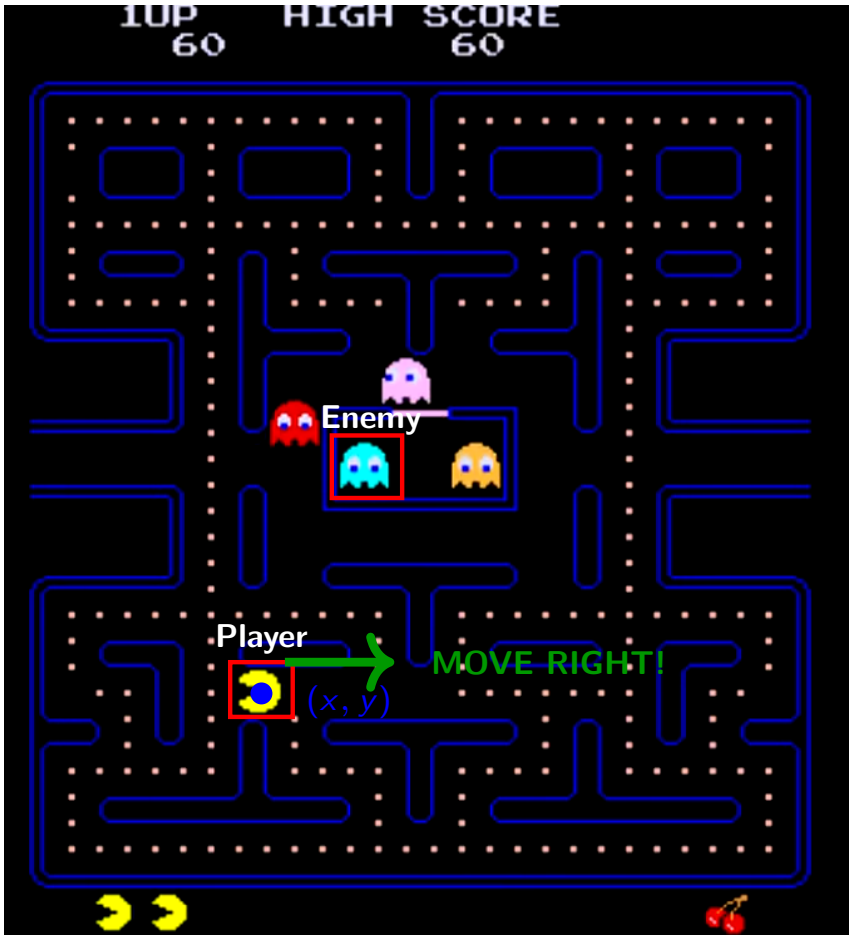
What does the Function learn?

To make a prediction, the function must **see** and **understand** the components of the game.

- **Classification:**
  Identifying objects. *"Is this a Ghost or a Wall?"*

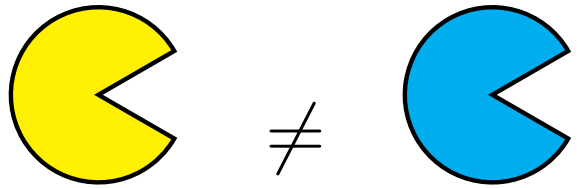
- **Regression:**
  Finding positions. *"Where exactly is Pac-Man?*
  $(x, y) = (12, 45)$

- **Control / Policy:**
  Making a decision. *"Ghost is near → Move Up!"*



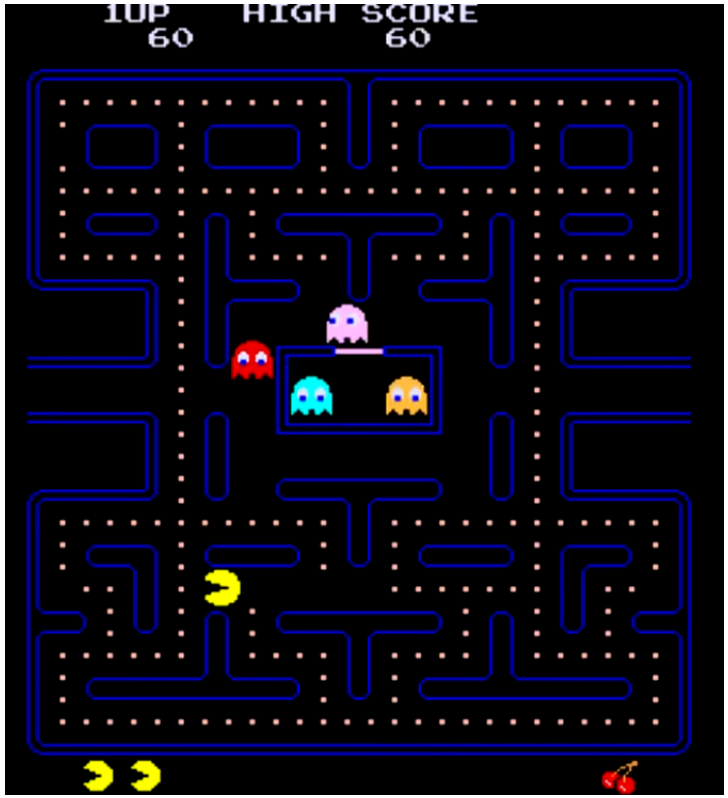
Memorization: Why raw Pixels are not enough

The Challenge:
Change the player's color from **Yellow** to **Blue**.



Training Sample
RGB: (255, 255, 0)

Testing Sample
RGB: (0, 255, 255)



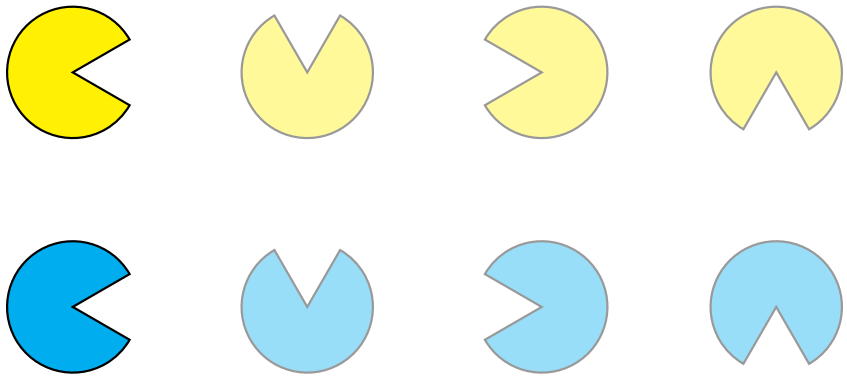
- **Human:** It's just Pac-Man in blue!
- **Model:** `Wait 🧐 ... blue pixels? I know this!` `Prediction: GHOST (99% Confidence)`

The Logic Gap

The model hasn't learned the **geometry** (shape); it has only memorized a specific **color pattern**.

The Cost of Orientation

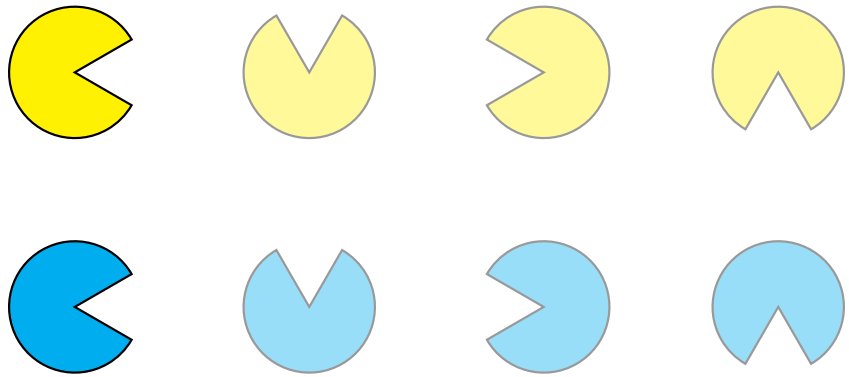
Normally, we must learn every direction for every color individually.



But do we really need to learn every possible angle from scratch?

The Cost of Orientation

Normally, we must learn every direction for every color individually.



But do we really need to learn every possible angle from scratch?

Option A: Augmentation

- Training with multiples of 90° rotations to **augment** the dataset.

Option B: Geometric Prior

- **Hard-code rotation** laws directly into the **model**.
  $\Rightarrow$ **Data efficiency!**.

Model Architecture

In the next slides, we will analyze the visual component of the model.

Convolutional Neural Networks (CNNs)

From pixels to prediction:

$\Longrightarrow$

Input Image (I)

→ **Conv Layer** → Pattern Detection (output pixels)

→ **Pooling Layer** → Aggregation op (Max, Avg, Min)

→ **Flatten** → Vectorization 2D to 1D

→ **Prediction** → Final Decision

- **Convolution:** **Detect patterns** (edges, shapes)
  - Scan small local pixel **neighborhoods**.
- **Pooling:** **Reduce dimensions**
  - Keep only the most **important information**.
- **Flatten:** **Vectorize features**
  - Transform the 2D grid into a **1D vector**.

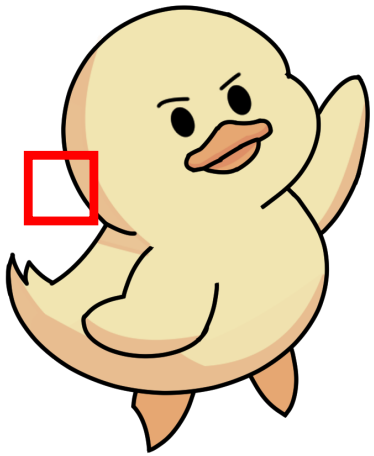
Convolution: The Visual Intuition

The Process: A **Kernel** (Filter) scans the image to find features.

Feature Map (M)

Score		
		...
	...	

Input Image (I)



Scan →

Kernel (K)

3×3		

What's happening?

For every position, we calculate a **Similarity Score**:

1. Overlay the Kernel on a part of the Image.
2. Multiply matching pixels.
3. Sum them up → Put number in M.

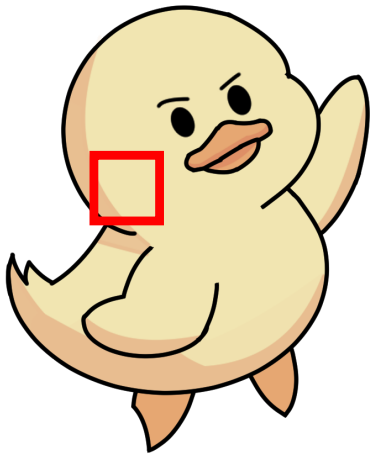
Convolution: The Visual Intuition

The Process: A **Kernel** (Filter) scans the image to find features.

Feature Map (M)

Score	Score	
		...
	...	

Input Image (I)



Scan →

Kernel (K)

3×3		

What's happening?

For every position, we calculate a **Similarity Score**:

1. Overlay the Kernel on a part of the Image.
2. Multiply matching pixels.
3. Sum them up → Put number in M.

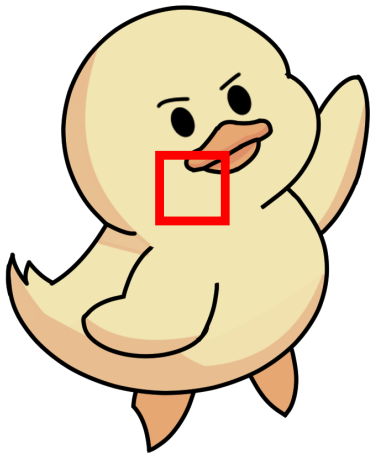
Convolution: The Visual Intuition

The Process: A **Kernel** (Filter) scans the image to find features.

Feature Map (M)

Score	Score	Score
		...
	...	

Input Image (I)



Scan →

Kernel (K)

3	×	3

What's happening?

For every position, we calculate a **Similarity Score**:

1. Overlay the Kernel on a part of the Image.
2. Multiply matching pixels.
3. Sum them up → Put number in M.

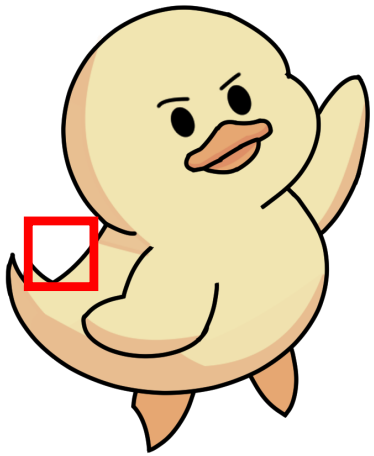
Convolution: The Visual Intuition

The Process: A **Kernel** (Filter) scans the image to find features.

Feature Map (M)

Score	Score	Score
Score		...
	...	

Input Image (I)



Scan →

Kernel (K)

3×3		

What's happening?

For every position, we calculate a **Similarity Score**:

1. Overlay the Kernel on a part of the Image.
2. Multiply matching pixels.
3. Sum them up → Put number in M.

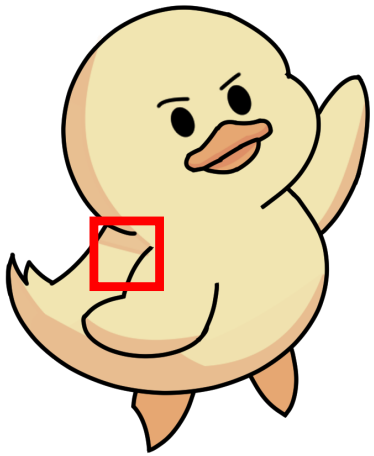
Convolution: The Visual Intuition

The Process: A **Kernel** (Filter) scans the image to find features.

Feature Map (M)

Score	Score	Score
Score	Score	$\cdots$
	$\cdots$	

Input Image (I)



Scan $\longrightarrow$

Kernel (K)

3×3		

What's happening?

For every position, we calculate a **Similarity Score**:

1. Overlay the Kernel on a part of the Image.
2. Multiply matching pixels.
3. Sum them up $\rightarrow$ Put number in M.

The Architecture Solution: Convolution

The Process: The Kernel K slides over Image I to compute the Feature Map M.

Input I (with 3×3 sliding window)

1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1

Window

$*$

Kernel K (3×3)

1	2	3
4	5	6
7	8	9

Full Feature Map M

45	...	...
	...	

The Calculation

Dot Product (Element-wise)

$$M_{1,1} = \sum (I_{window} \odot K)$$

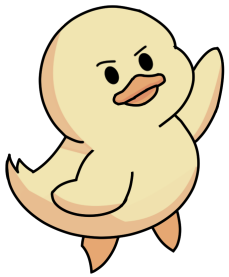
$$(1 \cdot 1) + (1 \cdot 2) + \cdots + (1 \cdot 9) = 45$$

A few kernel examples

1. Blurred (Smoothing)

$$K_{blur} = \frac{1}{15^2} \underbrace{\begin{pmatrix} 1 & \dots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \dots & 1 \end{pmatrix}}_{15 \times 15}$$

2. Original Data

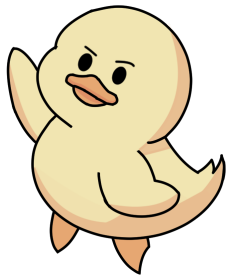


$$\text{Identity } I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

3. Rotated 90°

$$M_{rot} = \begin{pmatrix} \cos 90^\circ & \sin 90^\circ \\ -\sin 90^\circ & \cos 90^\circ \end{pmatrix}$$

4. Flipped (Reflection)



$$M_{flip} = \begin{pmatrix} -1 & 0 & \text{cols} \\ 0 & 1 & 0 \end{pmatrix}$$

$$\Downarrow$$

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} -x + \text{cols} \\ y \end{pmatrix}$$

The Bridge: From CNNs to Geometry

CNNs learn filters, but are only **translation invariant**.

The Challenge:

- **Rotation (θ):** An object can be at any angle.
- **Scale (s):** In cameras, objects change size based on distance.
- *So, how do we actually design filters to handle every possible size and angle?*

The solution is **Geometric Deep Learning (GDL)**:

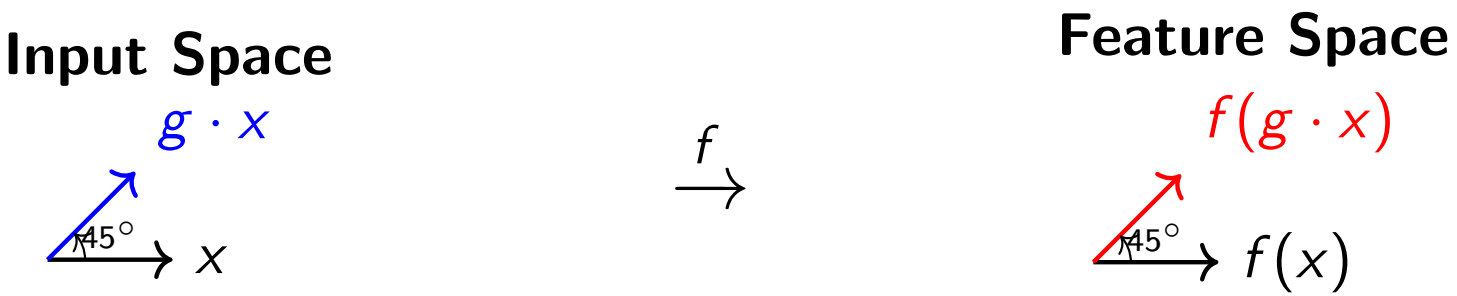
- It's our **Rulebook**.
- It allows the model to use **actions** of geometry
  - Rotation, Scale, Symmetry.

The Formal Language: Equivariance

In **Geometric Deep Learning**, the actions $\{g\}$ are **Equivariant**.

What does $f(g \cdot x) = g \cdot f(x)$ mean?

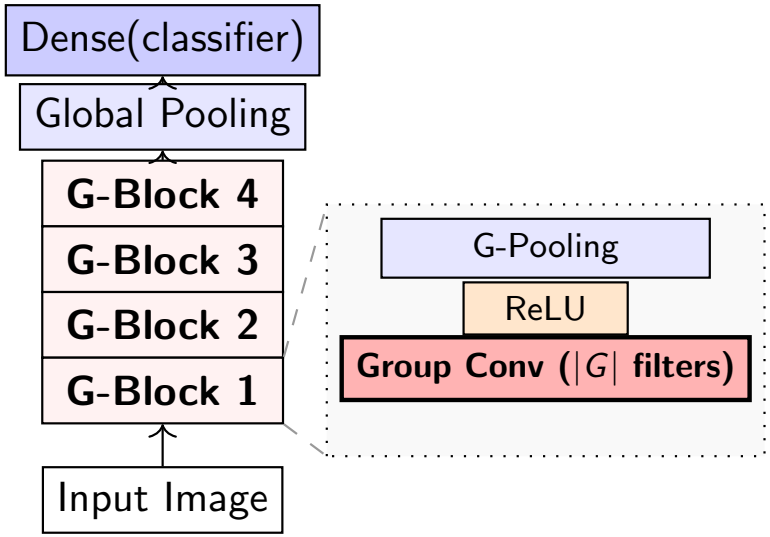
If we **rotate** the input (x) by an angle (g), the output features should **rotate** by the same angle.



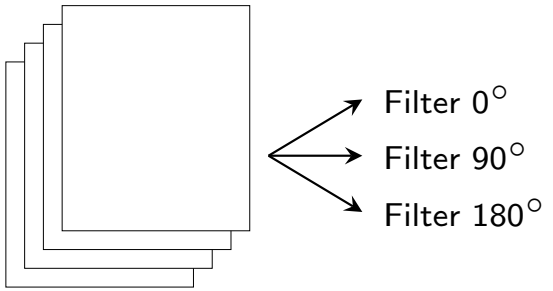
Group Equivariant CNNs (G-CNNs): Stacked Filters

CNN Architecture

How do we build an equivariant layer? $f(g \cdot x) = g \cdot f(x)$

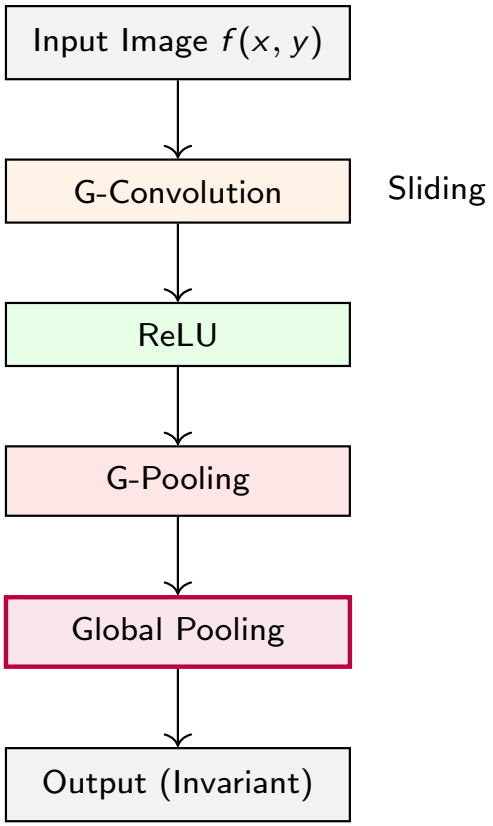


Group of Filters



$$\mathcal{F}_{\mathrm{G}} = \{\mathrm{Filter}_\theta \mid \theta \in \{0^\circ, 90^\circ, \dots\}\}$$

G-CNNs: Pixel-to-Pixel Logic



G-Convolution (Sliding + Rotation)

- Standard **sliding window** logic.
- Computes features for all rotations $s \in G$.

Maps $f(x, y)$ to a 3D-map $f(x, y, s)$.

G-Pooling (Pose Invariance)

- At each pixel (x, y), picks the best rotation s:

$$\text{Output}(x, y) = \max_{s \in G} f(x, y, s).$$

- It **forgets** the **rotation** but keeps the feature.

Global Pooling (Final Decision)

- Aggregates **all** positions (x, y) into **a single scalar value**.
- **Classification Tip:** In the final layer omit G-pooling. Use N filters and perform Global Average to get one number per **class**.

Steerable filters

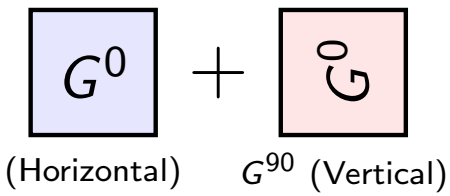
We learn a set of basis:

- E.g. **two basis kernels**: G^0 (horizontal) and G^{90} (vertical).

The Steerable Equation

Any orientation θ can be synthesized as a linear combination:

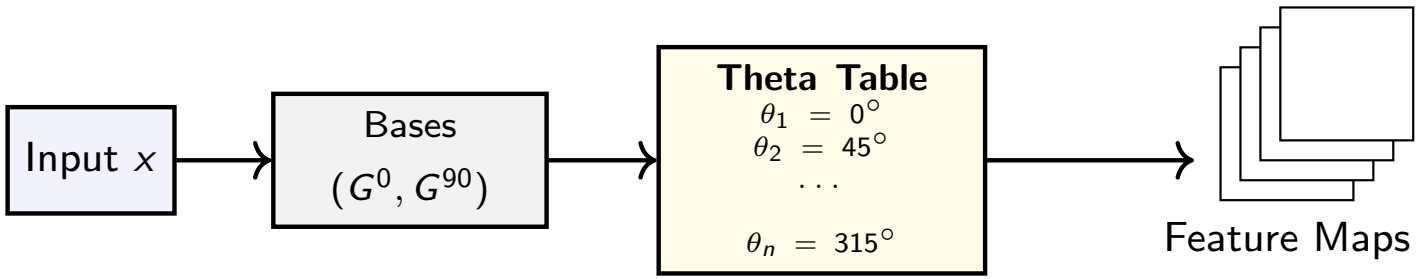
$$G^{\theta} = \cos(\theta) \cdot G^0 + \sin(\theta) \cdot G^{90}$$



- **Parameters:** We only store the weights for G^0 and G^{90}.
- **Inductive Bias:** The model is **forced** to follow the laws of rotation.

The Orientation Table: Learning Angles

Instead of a fixed grid, we maintain a **Table of Orientations** $\Theta = \{\theta_1, \theta_2, \ldots, \theta_n\}$ as trainable parameters.



Gradient-Based Optimization

Since the synthesis $G^\theta = \cos(\theta)G^0 + \sin(\theta)G^{90}$ is **differentiable**, we can update the entire table via gradient-based optimizers (e.g. compute derivatives):

$$\theta_{new} \leftarrow \theta - \eta \frac{\partial \mathcal{L}}{\partial \theta} \quad \Longrightarrow \quad \text{Optimizes the } \textbf{Looking Angle}$$

How to Scale: The Inverse Mapping

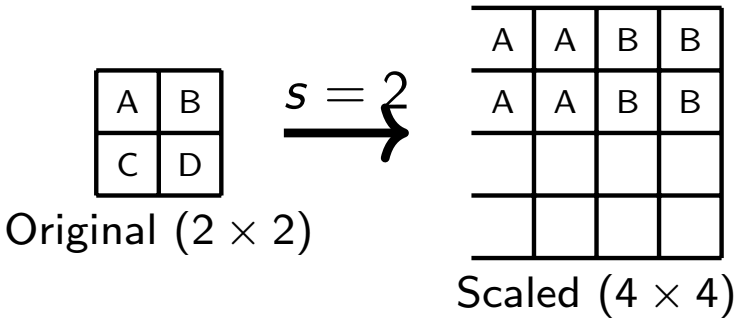
To scale a filter by a factor s, we calculate the new value for every pixel (x, y) using the **Inverse Map**:

The Rule

$$G_{new}(x, y) = G_{old}\left(\frac{x}{s}, \frac{y}{s}\right)$$

Step-by-Step Example ($s = 2$):

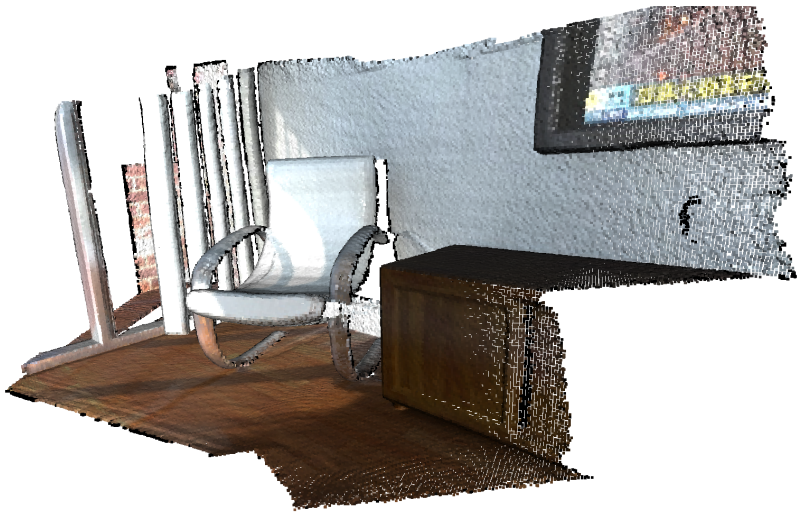
1. We want to fill the pixel at $(x', y') = (2, 2)$ in the **large** filter.
2. We look at the **original** filter at position $(\frac{2}{2}, \frac{2}{2}) = (1, 1)$.
3. We copy that value to the new position.



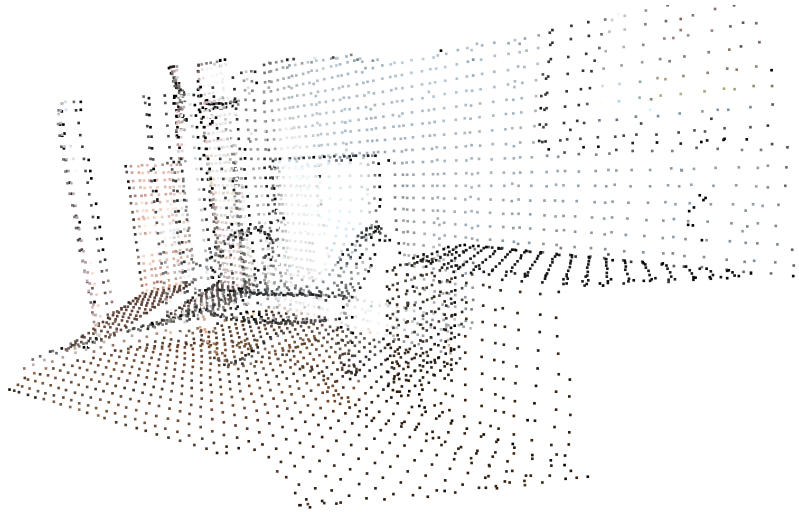
Original (2×2) Scaled (4×4)

Practical Use: 3D Point Clouds

A 3D Point Cloud is just a collection of coordinates (x, y, z) from a LiDAR or 3D scanner.



Point Cloud with RGB

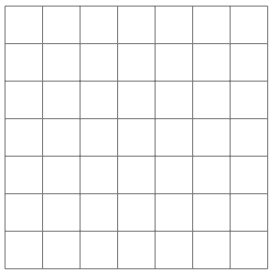


Geometry only

- **Unordered Set:** Unlike pixels, there is no 1st, 2nd, . . . , point.
- **Permutation Invariance:** The model must produce the same result regardless of point ordering.
- **Continuous:** Points can be anywhere, not just on a grid.

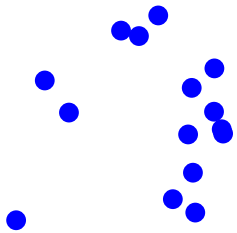
Why CNNs Fail on 3D Point Clouds

- **CNNs** need a **Grid** (like pixels in a 2D image).
- **Point Clouds** have no grid, no rows, and no columns.



CNN: Fixed Grid

$\neq$



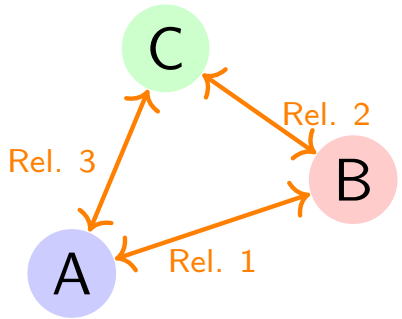
Point Cloud: Floating Points

Solution: Instead of a **sliding window**, we use **Attention**. Points relate to one another directly to identify geometric structures.

Attention: A Tool for Learning Relations

In our 3D world, we use **attention** to achieve **Relational Reasoning**.

- **Standard AI:** Treats points as **isolated coordinates**.
- **Relational AI (attention):** Analyzes the **interactions** between points.

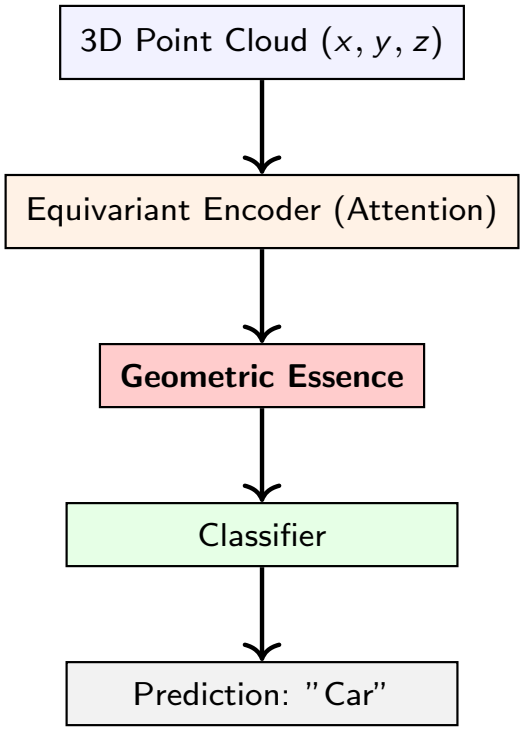


What is a Relation?

- **Distance** between points.
- **Relative Angles**.

Key Idea: Instead of learning (x, y, z), the model learns the **distances** between points. Since distances don't change when things rotate, the model is **invariant**!

Putting it all together: Encoder-Decoder



How it works:

1. **Encoder:** Uses Attention to analyze through points and find relations.
2. **Bottleneck:** Only the **Geometric Essence** (the shape) survives.
3. **Invariance:** Since the model focuses on **distances** (which never change) rather than **coordinates** (which change during rotation), the final prediction remains the same!

Summary Takeaways

- **Geometry is a Prior:** By encoding symmetry, we give the network common sense about the physical world.
- **Data Efficiency:** Mathematical constraints reduce the need for massive datasets.
- **Universal Framework:** GDL provides a unified language for CNNs, Graphs, and 3D data.

The End

The End

Useful links

- Group Equivariant Convolutional Networks
- Learning Steerable Filters for Rotation Equivariant CNNs
- Group Invariant Global Pooling