Gemini Chat

ยินดีครับ ผมจะสอนเนื้อหาเรื่อง **Object Detection** จากเอกสารที่คุณแนบมาให้อย่างละเอียด โดยจะเจาะลึกไปทีละส่วนเพื่อให้เห็นวิวัฒนาการและเหตุผลเบื้องหลังของแต่ละเทคนิคครับ

เราจะเริ่มกันที่ **Part 1: พื้นฐานและปัญหาของ Object Detection** ครับ

Part 1: พื้นฐานและปัญหาของ Object Detection (Fundamentals & Challenges)

ก่อนที่เราจะไปดูโมเดล AI ล้ำๆ เราต้องเข้าใจก่อนว่า "โจทย์" ของเราคืออะไร และทำไมมันถึงยากกว่างานทั่วไปครับ

1. ความแตกต่างระหว่าง Classification และ Detection

ในงาน Computer Vision เรามีงานหลายประเภทที่คล้ายกันแต่ไม่เหมือนกันครับ:

- **Image Classification:** รับภาพเข้ามา ตอบว่า "ภาพนี้คืออะไร" (เช่น แมว) โดยไม่สนว่าแมวอยู่ตรงไหน
- **Object Detection:** ยากกว่ามาก เพราะต้องตอบ 2 อย่างคือ **"WHAT"** (วัตถุคืออะไร) และ **"WHERE"** (วัตถุอยู่ตรงไหน - โดยระบุเป็นกรอบสี่เหลี่ยม หรือ Bounding Box)

2. ความท้าทายหลัก (The Core Challenges)

ทำไมเราถึงใช้ Neural Network แบบเดิมๆ (Standard CNN) มาทำ Detection เลยไม่ได้?

1. **Multiple Objects:** ภาพหนึ่งภาพอาจมีวัตถุหลายชิ้น หรือไม่มีเลย เราไม่รู้ล่วงหน้าว่าต้อง Output กี่คำตอบ
2. **Variable Types:** เราต้องทำทั้งจำแนกประเภท (Classification) และหาพิกัด (Regression) พร้อมกัน

3. **High Resolution:** งาน Detection ต้องใช้ภาพความละเอียดสูงกว่างาน Classification ทั่วไป เพื่อให้เห็นวัตถุเล็กๆ ได้ชัดเจน (เช่น 800x600 เทียบกับ 224x224)

3. แนวคิดดั้งเดิมที่ "ล้มเหลว": Sliding Window

วิธีที่ตรงไปตรงมาที่สุดคือ **Sliding Window** หรือการเลื่อนกรอบสี่เหลี่ยมไปทั่วภาพครับ

- **วิธีการ:** ตัดภาพส่วนเล็กๆ (Crop) มาเข้า CNN เพื่อทายว่า "ใช่หมาไหม?", "ใช่แมวไหม?" ถ้าไม่ใช่ก็เลื่อนกรอบไปเรื่อยๆ
- **ปัญหา:**
  - เราไม่รู้ขนาดของวัตถุ จึงต้องลองกรอบหลายขนาด (Scales) และหลายสัดส่วน (Aspect Ratios)
  - ในภาพขนาด 800x600 หากเราลองทุกความเป็นไปได้ จะมีกรอบให้ตรวจสอบถึง **58 ล้านกรอบ!** (58M boxes)
  - **สรุป:** วิธีนี้ช้าเกินไปจนทำจริงไม่ได้ครับ (Computational Expensive)

4. ทางออกแรก: Region Proposals (Selective Search)

เพื่อแก้ปัญหาการกวาดทุกจุด นักวิจัยจึงเสนอให้ใช้ **Region Proposals**

- **หลักการ:** แทนที่จะสุ่มทุกจุด ให้ใช้อัลกอริทึม (ที่ไม่ใช่ AI แต่เป็น Heuristics) เพื่อหาบริเวณที่ "น่าจะเป็นวัตถุ" (Blob-like regions)

- **Selective Search:** เป็นอัลกอริทึมยอดนิยมในยุคนั้น โดยจะดู สี, พื้นผิว (Texture), และความต่อเนื่องของภาพ เพื่อเสนอกรอบสี่เหลี่ยมมาประมาณ **2,000 กรอบ** ต่อภาพ
- **ข้อดี:** ลดจำนวนพื้นที่ที่ต้องตรวจสอบจาก "58 ล้าน" เหลือแค่ "2 พัน" ทำให้การใช้ CNN เป็นไปได้มากขึ้น

สรุป Part 1: งาน Object Detection ยากเพราะต้องหาตำแหน่งวัตถุที่มีจำนวนไม่แน่นอน การใช้ Sliding Window นั้นเป็นไปไม่ได้ในทางปฏิบัติ จึงเกิดแนวคิด **Region Proposals (เช่น Selective Search)** ขึ้นมา

เพื่อคัดกรองพื้นที่ที่น่าสนใจก่อนส่งให้ AI ประมวลผล ซึ่งนี่คือกุญแจสำคัญที่นำไปสู่โมเดล **R-CNN** ครับ

หากคุณพร้อมสำหรับ **Part 2: R-CNN และวิวัฒนาการแรก** พิมพ์ **next** ได้เลยครับ

เยี่ยมครับ มาต่อกันที่ **Part 2: R-CNN และการวัดผล (Evaluation)** ครับ

ในส่วนนี้เราจะเจาะลึกโมเดลแรกที่นำ Deep Learning มาใช้กับ Object Detection ได้สำเร็จ นั่นคือ **R-CNN (Region-based CNN)** และเรียนรู้วิธีวัดความเก่งของโมเดลพวกนี้ครับ

Part 2: R-CNN (Region-based CNN)

R-CNN คือการเอาแนวคิด Region Proposals จาก Part 1 มาผสมกับพลังของ CNN ครับ

1. ขั้นตอนการทำงานของ R-CNN (The Pipeline)

กระบวนการทำงานแบ่งเป็น 4 ขั้นตอนหลักที่ทำต่อกันเป็นลูกโซ่:

1. **Input & Region Proposals:** รับภาพเข้ามา แล้วใช้ Selective Search หาบริเวณที่น่าสนใจ (Region of Interest - **RoI**) ประมาณ 2,000 จุด
2. **Warping (การบิดภาพ):**
    - **ปัญหา:** CNN ทั่วไป (เช่น AlexNet, VGG) ต้องการภาพ input ขนาดตายตัว (เช่น 224x224) แต่ RoI ที่ตัดมามีขนาดไม่เท่ากัน
    - **วิธีแก้:** บังคับยืด/หดภาพ (Warp) จาก RoI ให้กลายเป็นขนาด 224x224 เท่ากันหมด เพื่อส่งเข้า CNN
3. **Feature Extraction:** ส่งภาพ 2,000 ภาพที่ Warp แล้ว เข้าไปใน CNN เพื่อดึง Feature ออกมา (ได้เวกเตอร์ขนาด 4096)
4. **Classification & Regression:** นำ Feature ที่ได้ไปทำ 2 อย่างพร้อมกัน:
    - **Classify:** ทำนายว่า "กล่องนี้คืออะไร" (ใช้ SVM ในยุคแรก)

- **Bbox Regression:** ทำนาย "ค่าปรับแก้พิกัด" (Offset) เพื่อขยับกรอบให้แม่นยำขึ้น

2. การปรับแก้กรอบ (Bounding Box Regression)

ทำไมต้องปรับกรอบ? เพราะ Region Proposal ที่ได้มาอาจจะแค่ "เฉียดๆ" วัตถุ แต่ไม่เป๊ะ

- โมเดลจะไม่ทายพิกัด (x, y, w, h) ตรงๆ แต่จะทาย **ค่าการแปลง (Transform values)** 4 ตัว คือ (t_x, t_y, t_w, t_h)
- **Translation:** t_x, t_y ใช้เลื่อนจุดศูนย์กลางกล่อง
- **Scale:** t_w, t_h ใช้ยืดหดความกว้างและสูง (ผ่านฟังก์ชัน exponent เพื่อกันค่าติดลบ)

3. ตัวชี้วัดความแม่นยำ (Metrics: IoU & mAP)

ในงาน Detection เราวัดแค่ "ถูก/ผิด" ไม่ได้ เพราะกรอบที่ทายอาจจะเหลื่อมกับของจริง เราจึงใช้:

- **IoU (Intersection over Union):** วัดว่ากรอบที่ทาย (Prediction) ซ้อนทับกับเฉลย (Ground Truth) มากแค่ไหน
  - สูตร: $\frac{\text{พื้นที่ที่ซ้อนทับกัน}}{\text{พื้นที่รวมของทั้งสองกล่อง}}$
  - **เกณฑ์:** โดยทั่วไป IoU > 0.5 ถือว่า "เจอวัตถุ" (Decent), IoU > 0.7 ถือว่า "ดีมาก", IoU > 0.9 คือ "เกือบสมบูรณ์แบบ"
- **mAP (Mean Average Precision):** เป็นเกรดเฉลี่ยของโมเดล
  1. คำนวณกราฟ Precision-Recall ของแต่ละคลาส (เช่น กราฟของแมว, กราฟของหมา) โดยดูจากค่าความเชื่อมั่น (Confidence Score)
  2. หาพื้นที่ใต้กราฟ (Area Under Curve) ของแต่ละคลาส จะได้ค่า **AP (Average Precision)**
  3. นำ AP ของทุกคลาสมารวมกันแล้วหาค่าเฉลี่ย จะได้ **mAP**

จุดตายของ R-CNN (The Bottleneck)

ถึงแม้ R-CNN จะแม่นยำ แต่มีปัญหาใหญ่หลวงที่ทำให้ **"ช้ามาก"**:

- เราต้องรัน CNN ถึง **2,000 ครั้งต่อ 1 ภาพ!** (เพราะมี 2,000 Regions ต่างคนต่างรัน)
- การ Process ภาพเดียวอาจใช้เวลาเกือบนาที ซึ่งใช้จริงแบบ Real-time ไม่ได้เลย

นักวิจัยจึงคิดค้นวิธีแก้ปัญหาโดยการ **"รัน CNN แค่ครั้งเดียวพอ"** ซึ่งเป็นที่มาของ **Fast R-CNN**

ถ้าพร้อมที่จะไปดูวิธีแก้ปัญหานี้ใน **Part 3: Fast R-CNN และ RoI Pooling** พิมพ์ **next** ได้เลยครับ

มาต่อกันที่ **Part 3: Fast R-CNN และ RoI Pooling (การแก้ปัญหาความช้า)** ครับ

ในส่วนนี้เราจะมาดูว่านักวิจัยแก้ปัญหา "คอขวด" ของ R-CNN ที่ต้องรัน CNN ซ้ำๆ 2,000 ครั้งได้อย่างไร จนเกิดเป็นโมเดลที่เร็วกว่าเดิมแบบก้าวกระโดดครับ

Part 3: Fast R-CNN

หัวใจสำคัญของ **Fast R-CNN** คือแนวคิดที่ว่า **"อย่าทำซ้ำในสิ่งที่ทำไปแล้ว"** ครับ

1. การแชร์ฟีเจอร์ (Shared Features)

ใน R-CNN เดิม เราตัดภาพ 2,000 ชิ้นแล้วส่งเข้า CNN ทีละชิ้น ซึ่งส่วนใหญ่เป็นภาพที่ซ้อนทับกัน (Overlapping) ทำให้เราคำนวณซ้ำซ้อนมหาศาล

- **วิธีใหม่:** เราจะส่ง **ภาพทั้งภาพ (Whole Image)** เข้า CNN แค่ **ครั้งเดียว** ไปเลย
- **ผลลัพธ์:** เราจะได้ **Feature Map** ขนาดใหญ่ที่เป็นตัวแทนของภาพทั้งภาพ ออกมาหนึ่งอัน

2. การฉายภาพกลับ (Projection)

เมื่อเรามี Feature Map ของทั้งภาพแล้ว แต่เรายังต้องการฟีเจอร์เฉพาะจุด (RoI) จาก Selective Search อยู่ เราจะทำอย่างไร?

- เราใช้วิธี **Project** (ฉายภาพ) กรอบสี่เหลี่ยมจากภาพต้นฉบับ ลงไปบน Feature Map โดยตรงครับ
- เนื่องจาก CNN ย่อขนาดภาพลง (Subsampling) เราจึงสามารถคำนวณพิกัดที่สัมพันธ์กันได้ (เช่น ถ้าภาพถูกย่อลง 32 เท่า พิกัด (x, y) บนภาพจริง ก็จะอยู่ที่ $(x/32, y/32)$ บน Feature Map)

3. นวัตกรรมสำคัญ: RoI Pooling

นี่คือพระเอกของ Fast R-CNN ครับ เพราะหลังจาก Project กรอบลงมาบน Feature Map แล้ว กรอบแต่ละอันจะมีขนาด **ไม่เท่ากัน** แต่เลเยอร์ถัดไป (Fully Connected Layer) ต้องการ input ขนาด **คงที่** (เช่น 2×2 หรือ 7×7)

RoI Pooling คือเทคนิคการแปลง Feature ขนาดเท่าไหร่ก็ได้ ให้กลายเป็นขนาดคงที่:

1. **Snap to Grid:** ปัดเศษพิกัดของกรอบให้ลงล็อกกับตาราง (Grid) ของ Feature Map
2. **Divide:** แบ่งพื้นที่ในกรอบออกเป็นตารางย่อยๆ ตามขนาดที่ต้องการ (เช่น ต้องการ output 2×2 ก็แบ่งเป็น 4 ช่อง)
3. **Max Pool:** หาค่าสูงสุดในแต่ละช่องย่อย เพื่อให้ได้ค่าตัวแทน
    - **ผลลัพธ์:** ไม่ว่ากรอบต้นฉบับจะกว้างยาวแค่ไหน สุดท้ายจะได้เวกเตอร์ขนาดเท่ากันเป๊ะ พร้อมส่งไปจำแนกประเภทต่อได้เลย

4. ข้อเสียของ RoI Pooling และทางแก้ (RoI Align)

แม้ RoI Pooling จะดี แต่การ "ปัดเศษ" (Snap to Grid) ทำให้ข้อมูลคลาดเคลื่อนเล็กน้อย (Misalignment) ซึ่งส่งผลกระทบมากถ้าวัดความแม่นยำระดับพิกเซล

- **RoI Align:** เป็นเวอร์ชันอัปเกรด (มักใช้ใน Mask R-CNN) โดยเลิกใช้การปัดเศษ แต่ใช้คณิตศาสตร์เรื่อง **Bilinear Interpolation** (การประมาณค่าในช่วง) เพื่อคำนวณค่าจุดทศนิยมแทน ทำให้ได้ฟีเจอร์ที่แม่นยำกว่ามาก

สรุปความเร็ว (Speedup)

Fast R-CNN เร็วกว่า R-CNN อย่างมหาศาล:

- **เวลาเทรน:** จาก 84 ชั่วโมง เหลือ 8.75 ชั่วโมง
- **เวลาใช้งานจริง (Test time):** เร็วกว่าเดิมมาก แต่... **ยังติดปัญหาเดิม**

ปัญหาใหม่ (The New Bottleneck): แม้ตัว AI จะรันเร็วมาก แต่ขั้นตอนการหา **Region Proposals (Selective Search)** ยังเป็นอัลกอริทึมเก่าที่รันบน CPU ซึ่งกินเวลาไปเกือบทั้งกระบวนการ (ประมาณ 2 วินาทีต่อภาพ ในขณะที่ AI ใช้เวลาเสี้ยววินาที)

เพื่อให้เร็วระดับ Real-time เราต้องกำจัด Selective Search ทิ้งไปครับ

ถ้าพร้อมที่จะดูว่าเขาทำอย่างไรใน **Part 4: Faster R-CNN และ Region Proposal Network** พิมพ์ **next** ได้เลยครับ

เยี่ยมเลยครับ เรามาต่อกันที่ **Part 4: Faster R-CNN และ Region Proposal Network (RPN)** ซึ่งเป็นจุดเปลี่ยนสำคัญที่ทำให้ Object Detection เร็วพอที่จะใช้งานจริงได้ครับ

Part 4: Faster R-CNN

ใน Part 3 เราเห็นแล้วว่า Fast R-CNN แก้ปัญหาการรันโมเดลซ้ำซ้อนได้ แต่ยังติดปัญหาที่ **Selective Search** (การหา Region Proposals) นั้นช้าและรันบน CPU ไม่ได้เรียนรู้ไปพร้อมกับโมเดล (Not Learnable)

Faster R-CNN จึงเกิดมาเพื่อตอบโจทย์นี้โดยเฉพาะ ด้วยแนวคิด **"ใช้ CNN หา Region Proposals เองซะเลย"**

1. Region Proposal Network (RPN)

แทนที่จะใช้อัลกอริทึมภายนอก (Selective Search) เราจะสร้าง Neural Network เล็กๆ ขึ้นมาตัวหนึ่งชื่อว่า **RPN** แล้วแทรกมันเข้าไปในโมเดลครับ

- **ตำแหน่ง:** RPN จะอยู่ต่อจาก "Backbone CNN" (หลังจากได้ Feature Map แล้ว) และอยู่ก่อน "Classifier"
- **หน้าที่:** รับ Feature Map เข้ามา แล้วพ่น "กรอบสี่เหลี่ยมที่น่าจะเป็นวัตถุ" (Proposals) ออกมาให้เรา

2. หัวใจสำคัญ: Anchor Boxes

RPN ทำงานอย่างไร? มันจะเลื่อนหน้าต่างเล็กๆ (Sliding Window) ไปทั่ว Feature Map แต่ละจุด

- ที่แต่ละจุด มันจะจินตนาการ **"สมอ" (Anchors)** หรือกรอบสี่เหลี่ยมมาตรฐานขึ้นมาหลายๆ แบบพร้อมกัน (เช่น 3 ขนาด x 3 สัดส่วน = 9 Anchors ต่อจุด)
- **ทำไมต้องมีหลายแบบ?:** เพื่อให้รองรับวัตถุที่หลากหลาย เช่น คน (ผอมสูง), รถ (กว้างเตี้ย), หรือสุนัข (จัตุรัส)

3. การทำงานของ RPN

สำหรับ Anchor แต่ละอันที่จุดหนึ่งๆ RPN จะทำนาย 2 อย่าง:

1. **Objectness Score:** คะแนนว่า "ในกรอบนี้มีวัตถุไหม?" (ใช่/ไม่ใช่)
2. **Box Regression:** ค่าปรับแก้พิกัด (t_x, t_y, t_w, t_h) เพื่อขยับ Anchor ให้ตรงกับวัตถุจริงมากขึ้น

4. โมเดลแบบ 2 ขั้นตอน (Two-Stage Detector)

Faster R-CNN ถือเป็นต้นแบบของ **Two-Stage Detector** ที่สมบูรณ์แบบ:

- **Stage 1 (RPN):** กวาดทั่วภาพอย่างรวดเร็วเพื่อคัดกรอง "ว่าที่วัตถุ" (Proposals) ออกมาจำนวนหนึ่ง (เช่น 300-2,000 กล่อง)
- **Stage 2 (Refinement):** นำกล่องเหล่านั้นไปเข้ากระบวนการ RoI Pooling (เหมือน Fast R-CNN) เพื่อจำแนกประเภทละเอียดและปรับกรอบครั้งสุดท้าย

5. ผลลัพธ์

- **ความเร็ว:** เร็วขึ้นจนเกือบเป็น Real-time (ประมาณ 5-7 fps ในยุคนั้น) เพราะทุกส่วนรันบน GPU ทั้งหมด
- **ความแม่นยำ:** สูงขึ้น เพราะตัวหา Region Proposals (RPN) ถูกเทรนมาเพื่อให้เก่งขึ้นเรื่อยๆ (Learnable) ไม่ได้ใช้กฎตายตัวเหมือน Selective Search

---

คำถามที่นำไปสู่อนาคต

Faster R-CNN ทำงานได้ยอดเยี่ยม แต่การมี 2 ขั้นตอน (Stage 1 หาของ -> Stage 2 จำแนกของ) ทำให้มันยังมีความซับซ้อนและช้าอยู่บ้าง

นักวิจัยจึงตั้งคำถามว่า **"เราจำเป็นต้องมี 2 ขั้นตอนจริงๆ เหรอ? ทำไมไม่ทำนายทุกอย่างออกมาทีเดียวเลยล่ะ?"**

นี่คือจุดเริ่มต้นของ **Single-Stage Detectors** ที่โด่งดังที่สุดอย่าง **YOLO (You Only Look Once)** ครับ

ถ้าพร้อมที่จะไปรู้จักกับ YOLO ที่เน้นความเร็วแบบสุดๆ ใน **Part 5: YOLO และ Single-Stage Detectors** พิมพ์ **next** ได้เลยครับ

มาถึงส่วนสุดท้ายและเป็นส่วนที่น่าตื่นเต้นที่สุดครับ **Part 5: YOLO และ Single-Stage Object Detectors**

ในขณะที่ตระกูล R-CNN พยายามทำให้กระบวนการ 2 ขั้นตอน (หาของ -> จำแนกของ) เร็วขึ้น แต่ **YOLO (You Only Look Once)** ฉีกตำราทิ้งด้วยการบอกว่า **"ทำทุกอย่างพร้อมกันในรอบเดียวไปเลยสิ"**

Part 5: YOLO (You Only Look Once)

YOLO เปลี่ยนมุมมองจากงาน Classification ที่ต้องทำซ้ำๆ ให้กลายเป็น **งาน Regression (การหาค่าต่อเนื่อง)** เพียงงานเดียว

1. แนวคิด: ตารางตารางเดียวครองพิภพ (The Grid)

YOLO ไม่ใช้ Region Proposals แต่จะแบ่งภาพออกเป็นตาราง (Grid) ขนาด $S \times S$ (เช่น 7×7)

- **กฎเหล็ก:** ถ้า **จุดกึ่งกลาง (Center)** ของวัตถุตกลงในช่องไหน ช่องนั้นต้องรับผิดชอบในการตรวจจับวัตถุชิ้นนั้น

2. สิ่งที่แต่ละช่องต้องทำนาย (Predictions per Cell)

แต่ละช่องในตารางจะทำนายข้อมูลออกมาเป็นชุดเวกเตอร์ชุดเดียว ประกอบด้วย:

1. **Bounding Boxes (B กล่อง):** แต่ละช่องจะเดากรอบสี่เหลี่ยมออกมา B แบบ (ปกติ $B = 2$) โดยแต่ละกล่องมีค่า 5 ตัวคือ:
   - x, y: พิกัดจุดกึ่งกลาง (เทียบกับขอบของช่องนั้นๆ)
   - w, h: ความกว้างและสูง (เทียบกับขนาดภาพทั้งภาพ)
   - **Confidence:** ความมั่นใจว่า "มีวัตถุไหม" และ "กรอบแม่นแค่ไหน" (คำนวณจาก $P(\text{Object}) \times \text{IoU}$)
2. **Class Probabilities (C คลาส):** ความน่าจะเป็นว่าถ้ามีวัตถุ มันคืออะไร (เช่น หมา, แมว, รถ)

สรุป Output: ผลลัพธ์สุดท้ายจะเป็น Tensor ขนาด $S \times S \times (B \times 5 + C)$

3. การคำนวณ Loss (Loss Function)

YOLO ใช้ **Sum-Squared Error** ง่ายๆ รวมกัน 3 ส่วน:

1. **Localization Loss:** ความคลาดเคลื่อนของพิกัด (x, y, w, h)
   - *เกร็ดน่าสนใจ:* YOLO ถอดรากที่สอง ($\sqrt{w}, \sqrt{h}$) ก่อนคำนวณ เพื่อให้ความผิดพลาดในกล่องเล็ก มีความสำคัญพอๆ กับกล่องใหญ่ (ไม่งั้นผิด 5px ในกล่องใหญ่จะดูเล็กน้อย แต่ผิด 5px ในกล่องเล็กจะดูมหาศาล)
2. **Confidence Loss:** ความคลาดเคลื่อนของคะแนนความมั่นใจ (แยกคิดกรณีเจอกับไม่เจอของ)
3. **Classification Loss:** ความคลาดเคลื่อนของการทายชนิดวัตถุ

4. ข้อจำกัดของ YOLO (Limitations)

ถึงจะเร็วปานสายฟ้าแลบ แต่ YOLO รุ่นแรกมีจุดอ่อนที่ชัดเจน:

1. **Crowded Objects:** เนื่องจาก 1 ช่องตาราง รับผิดชอบวัตถุได้แค่ 1 ชนิด ถ้ามีนกฝูงใหญ่บินซ้อนกันในช่องเล็กๆ YOLO จะจับได้แค่ตัวเดียว
2. **Aspect Ratio:** มีปัญหากับวัตถุรูปทรงประหลาดๆ ที่ไม่ได้เห็นบ่อยตอนเทรน
3. **Small Objects:** ยังทำได้ไม่ดีเท่า Faster R-CNN ในการจับวัตถุขนาดเล็ก

บทสรุปวิวัฒนาการ (The Grand Summary)

เราได้เดินทางผ่านประวัติศาสตร์ของ Object Detection มาอย่างยาวนาน สรุปได้ดังนี้ครับ:

1. **R-CNN:** แม่นยำแต่ช้ามาก เพราะต้องรัน CNN 2,000 ครั้ง
2. **Fast R-CNN:** เร็วขึ้น เพราะแชร์ฟีเจอร์กัน แต่ยังติดคอขวดที่การหา Region Proposals (Selective Search)
3. **Faster R-CNN:** เป็น Two-stage ที่สมบูรณ์แบบ ใช้ RPN หา Proposals เอง แม่นยำและเร็วพอใช้งานจริง
4. **YOLO / SSD:** เป็น Single-stage ที่เน้นความเร็วสูงสุด รันรอบเดียวจบ เหมาะกับงาน Real-time บนอุปกรณ์ที่ไม่แรงมาก

หวังว่าคำอธิบายนี้จะช่วยให้คุณเห็นภาพรวมและเข้าใจรายละเอียดเชิงลึกของ Object Detection ได้ชัดเจนนะครับ! หากมีส่วนไหนอยากให้ขยายความเพิ่ม บอกได้เลยครับ