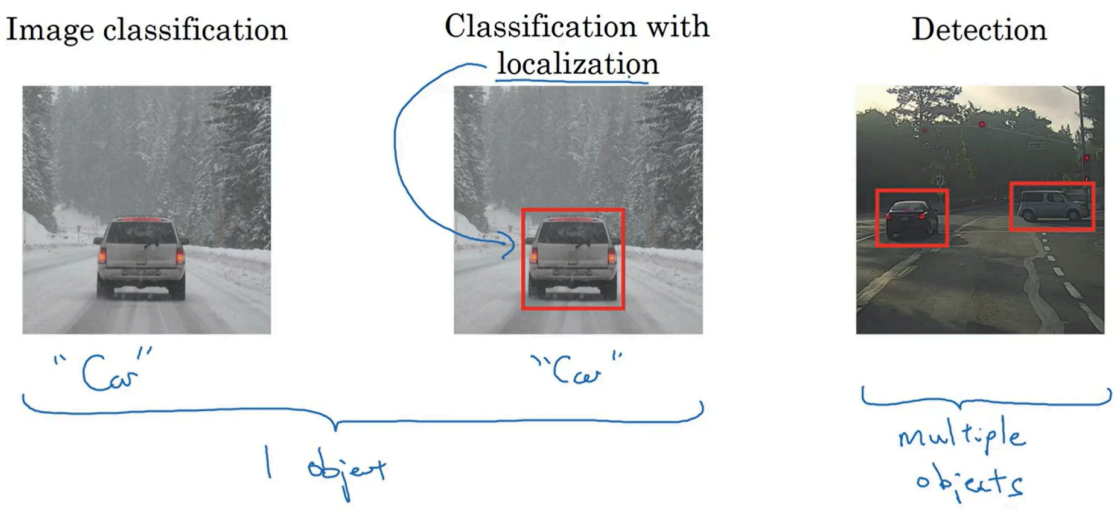
WEEK 3: Object Detection

Object Localization

1. Definitions & Problem Types



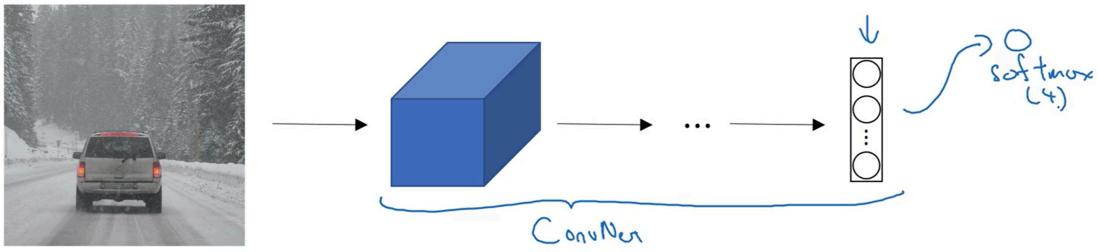
- **Image Classification:** Assigns a class label to the whole image (e.g., "car").
- **Object Localization:** Classifies the object **and** predicts its position via a bounding box.
- **Object Detection:** Finds and localizes **multiple objects** (possibly of different classes) in one image (covered later).

2. Assumptions (for this lecture)

- Only **one main object** per image (classification + localization).
- Detection (multiple objects) is a separate, more complex problem.

3. Neural Network Architecture for Localization

- **Standard CNN for classification:**

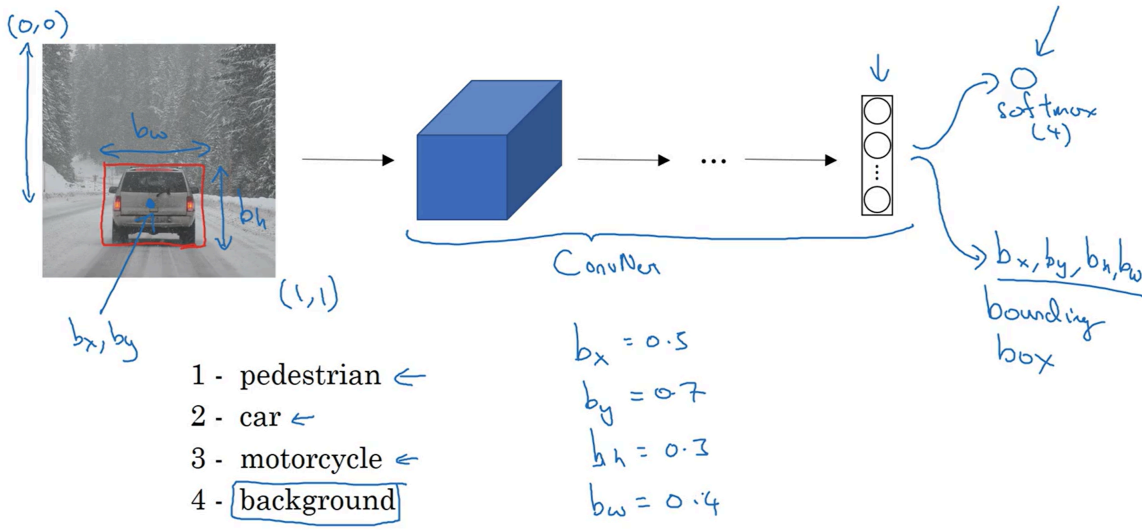


1 - pedestrian
2 - car
3 - motorcycle
4 - background

classification of 4 classes using softmax

- Input image → CNN layers → feature vector → softmax (class probabilities)
- Example classes: pedestrian, car, motorcycle, background

- **For localization:**

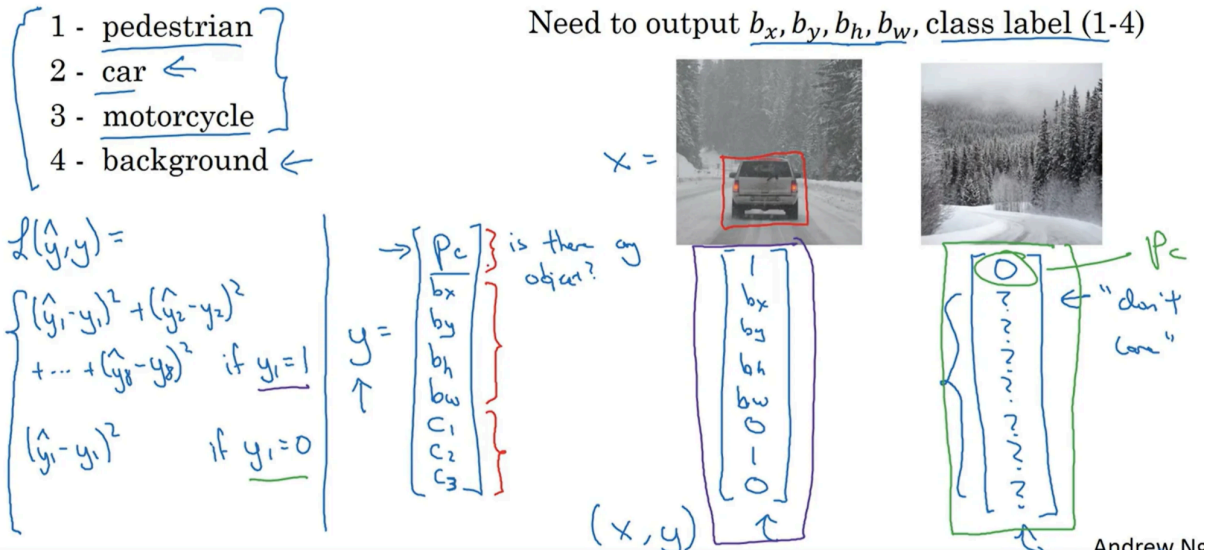


Classification with localization

we have softmax along with some other output units (bounding units for boundary)

- Add 4 outputs: b_x, b_y, b_h, b_w
  - b_x, b_y: Center coordinates of bounding box (normalized: (0,0) = top-left, (1,1) = bottom-right)
  - b_h: Height of box (relative to image height)
  - b_w: Width of box (relative to image width)

4. Label Vector (Target Format)



- $y = [p_c, b_x, b_y, b_h, b_w, c_1, c_2, c_3]$
  - p_c: 1 if object present, 0 if background
  - b_x, b_y, b_h, b_w: Bounding box (if object present)
  - c_1, c_2, c_3: One-hot class encoding (e.g., c1=pedestrian, c2=car, c3=motorcycle)
- **Example:**
  - Car present: $p_c = 1, b_x = 0.5, b_y = 0.7, b_h = 0.3, b_w = 0.4, c_1 = 0, c_2 = 1, c_3 = 0$
  - No object: $p_c = 0, b_x, b_y, b_h, b_w, c_1, c_2, c_3 =$?(don't care)

5. Training Data Requirements

- Each image must have:
  - Class label
  - Bounding box coordinates (if object present)
- For background images: only $p_c = 0$ matters; rest are ignored.

6. Loss Function

- **If object present ($p_c = 1$):**
  - Loss = sum of squared errors for all 8 outputs: $\text{Loss} = \sum_{i=1}^{8} (\hat{y}_i - y_i)^2$

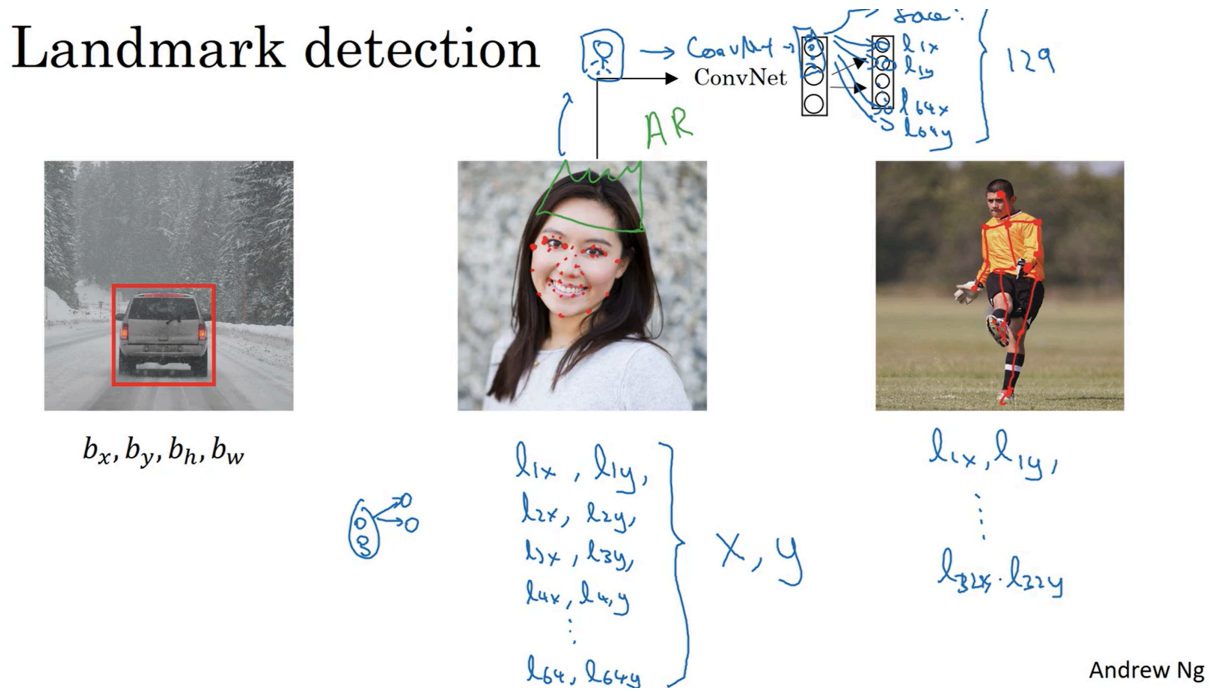
- **If no object ($p_c = 0$):**
  - Loss = squared error for p_c only: $\text{Loss} = (\hat{y}_1 - y_1)^2$
- **Advanced:**
  - Can use logistic loss for p_c, cross-entropy for class, squared error for bounding box.

7. Key Takeaways

- **Object localization = classification + bounding box regression.**
- **Label vector** encodes presence, location, and class.
- **Loss function** adapts based on object presence.
- **Neural network outputs real values** for bounding box (regression task).

Tip: This approach is foundational for more advanced detection systems (e.g., YOLO, SSD) that handle multiple objects per image.

Landmark Detection



1. Concept Overview

- **Landmark Detection:** Predicts the precise locations (x,y coordinates) of key points ("landmarks") on an object in an image.
- **Examples:**
  - Corners of the eyes, mouth, nose on a face
  - Joints (shoulder, elbow, wrist) for human pose
  - Key points on other objects (e.g., car headlights)

2. How It Works

- **Neural Network Output:**
  - For each landmark, network outputs two values: l_{ix}, l_{iy} (i = landmark index)
  - For N landmarks: total outputs = $2N$ (plus optional presence flag)
- **Example:**
  - 4 eye corners: outputs = $l_{1x}, l_{1y}, l_{2x}, l_{2y}, l_{3x}, l_{3y}, l_{4x}, l_{4y}$
  - 64 facial landmarks: outputs = $l_{1x}, l_{1y}, \ldots, l_{64x}, l_{64y}$ (128 outputs)
  - Often add a presence flag (e.g., is there a face?): total outputs = $2N + 1$

3. Labeling & Training

- **Training Data:**
  - Each image must be labeled with the exact (x, y) coordinates for every landmark.
  - Labels must be consistent: landmark 1 always means the same point (e.g., left eye corner) across all images.
- **Loss Function:**
  - Usually mean squared error (MSE) between predicted and true coordinates for each landmark.

4. Applications & Fun Uses

- **Face Analysis:**
  - Emotion recognition (smile, frown, surprise)
  - Face alignment for recognition systems
  - Augmented reality (AR) filters (e.g., Snapchat crowns, hats, dog ears)

    - Face morphing and animation
- **Pose Estimation:**
    - Detecting body joints for fitness apps, dance analysis, gesture control
- **Medical Imaging:**
    - Detecting anatomical landmarks in X-rays or MRIs
- **Robotics & AR:**
    - Object manipulation, overlaying graphics on real-world objects

5. Project Ideas

- **1. Facial Landmark Detection App:**
    - Build a web or mobile app that detects facial landmarks in real time using webcam input.
    - Overlay AR effects (glasses, hats, animal faces) based on detected points.
- **2. Emotion Recognition from Landmarks:**
    - Use landmark positions to classify facial expressions (happy, sad, angry, surprised).
- **3. Human Pose Estimation:**
    - Detect body joints in images or video; visualize skeleton overlay.
    - Applications: yoga pose correction, dance move analysis, sports coaching.
- **4. Medical Landmark Detection:**
    - Detect key points in medical images (e.g., vertebrae in spine X-rays).
- **5. Fun with Filters:**
    - Create custom Snapchat-style filters that track and augment faces in real time.
- **6. Gesture-Controlled Games:**
    - Use hand or body landmarks to control simple games or interactive art.

6. Tips for Success

- **Data Quality:** High-quality, consistently labeled data is crucial.
- **Augmentation:** Use image flips, rotations, and scaling to improve robustness.
- **Visualization:** Always visualize predicted landmarks to debug and improve your model.

Summary:

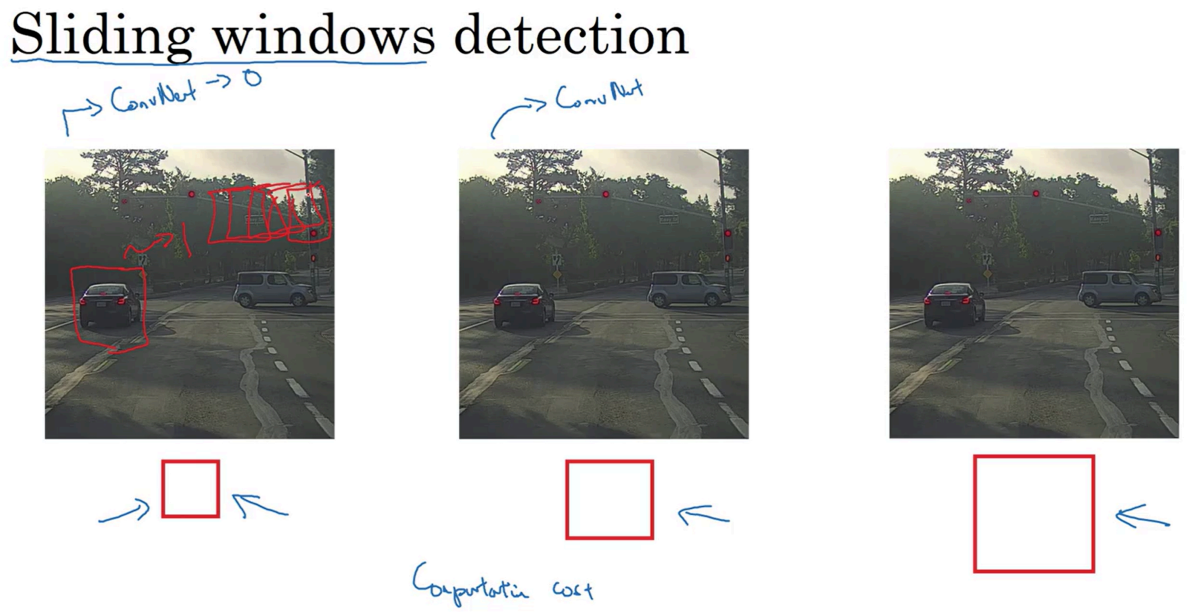
- Landmark detection extends object localization to multiple key points.
- Enables a wide range of applications from AR to medical imaging.
- Simple architectural change: just add more output units for each landmark's (x, y) coordinates.
- Consistent labeling and creative project ideas can make learning and applying this technique both practical and fun.

Object Detection

1. Goal & Context

- **Object Detection:** Find and localize objects (e.g., cars) in an image, possibly multiple per image.
- Builds on **object localization** (single object) and **landmark detection** (keypoints).

2. Sliding Windows Detection Algorithm



- **Step 1: Prepare Training Data**
- Collect many **closely cropped images** of the object (e.g., cars) and non-object images.

- Label: 1 (object present), 0 (not present).
- **Step 2: Train a ConvNet Classifier**
  - Input: Cropped image region
  - Output: 1 (object) or 0 (not object)
- **Step 3: Apply Sliding Window on Test Image**
  - Choose a window size (e.g., 64x64 pixels).
  - Slide this window across the image at various positions (using a stride/step size).
  - For each window:
    - Crop the region
    - Resize to ConvNet input size (if needed)
    - Run ConvNet to predict object presence
  - Repeat with **multiple window sizes** to detect objects at different scales.

3. Key Details & Insights

- **Stride:**
  - Large stride = fewer windows, faster, but may miss objects or be less precise.
  - Small stride = more windows, better localization, but much higher computational cost.
- **Multiple Scales:**
  - Use different window sizes to detect small and large objects.
- **Detection:**
  - If any window gets a positive prediction, object is detected at that location.

4. Limitations of Sliding Windows

- **Computationally expensive:**
  - Many windows per image, each requiring a ConvNet forward pass.
  - Fine stride and multiple scales make it even slower.
- **Accuracy vs. Speed Tradeoff:**
  - Coarse stride = faster but less accurate.

    - Fine stride = accurate but slow.
- **Historical Note:**
    - Sliding windows worked with simple classifiers (e.g., linear models) before ConvNets.
    - With ConvNets, this approach is often too slow for practical use.

5. Transition to Better Methods

- **Modern object detection** uses more efficient, convolutional approaches (covered in next videos).
- Key idea: avoid redundant computation by sharing features across overlapping windows.

---

6. Project Ideas & Fun Applications

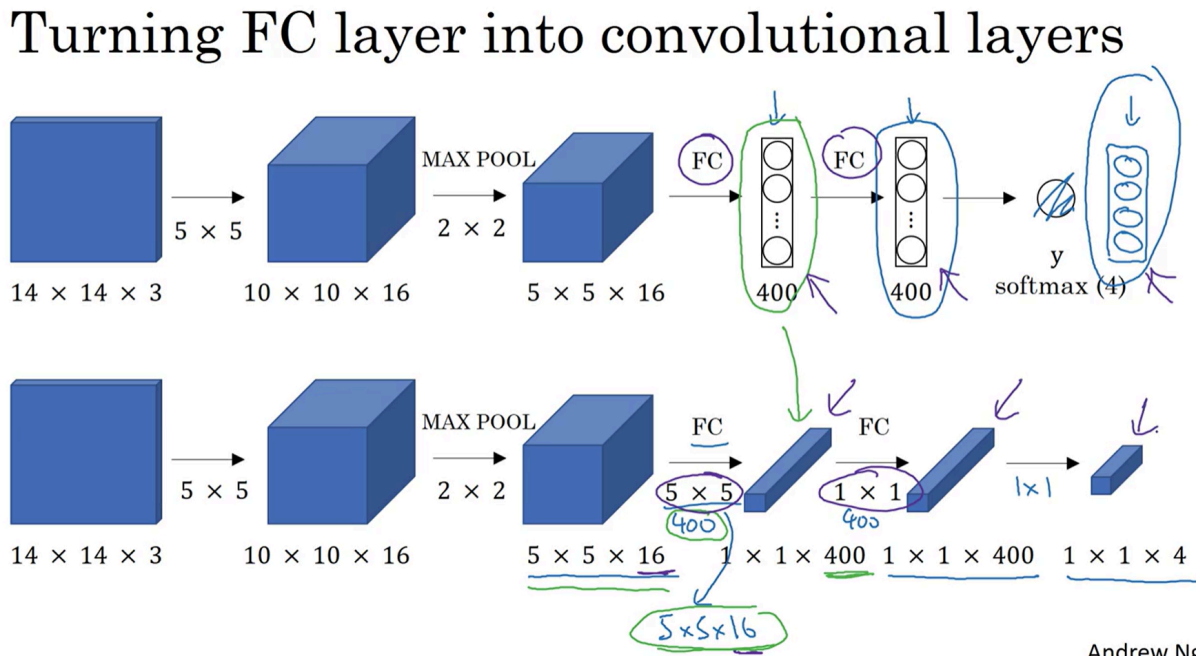
- **1. Real-Time Object Detector:**
    - Build a simple sliding window detector for faces, cars, or cats in images.
    - Visualize detected windows with bounding boxes.
- **2. Stride vs. Accuracy Experiment:**
    - Compare detection speed and accuracy for different stride values.
    - Plot tradeoff curves.
- **3. Multi-Scale Detection Demo:**
    - Detect objects of various sizes by running sliding windows at multiple scales.
    - Show how small objects are missed if only large windows are used.
- **4. "Where's Waldo?" Game:**
    - Use sliding window detection to find a specific character in crowded images.
- **5. Historical Comparison:**
    - Implement sliding window detection with both a simple linear classifier and a ConvNet.
    - Compare speed and accuracy.
- **6. Fun Visualization:**

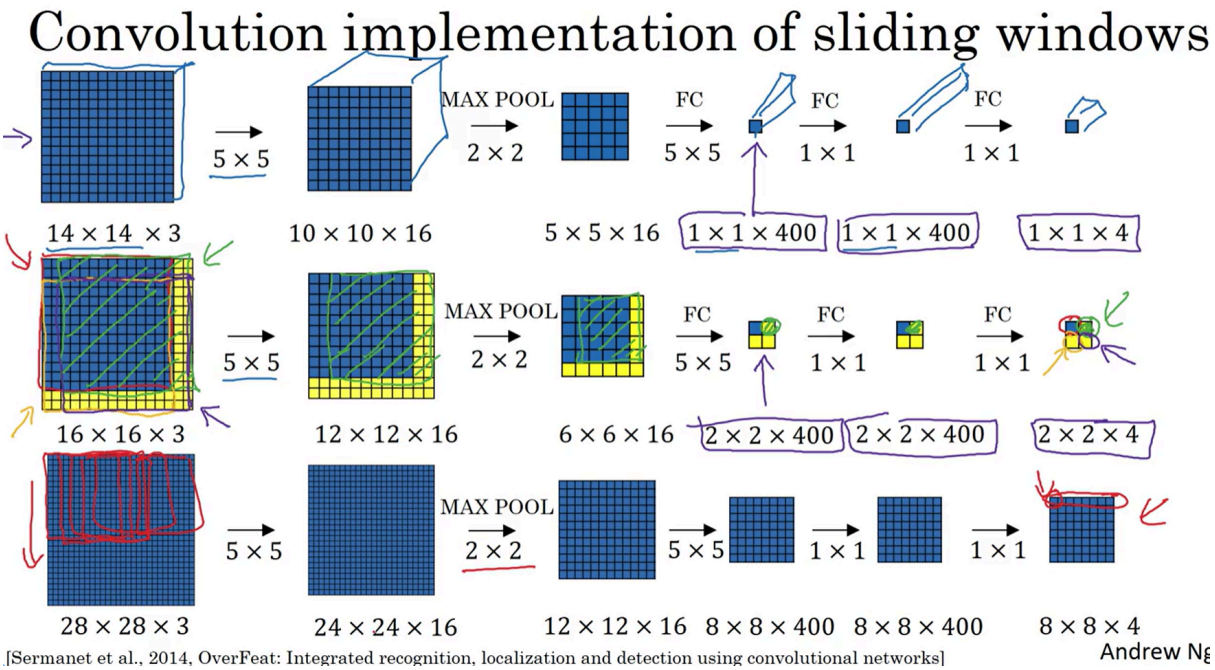
- Animate the sliding window process over an image to show how detection works step by step.

Summary:

- Sliding windows detection is a classic but computationally heavy approach for object detection.
- It laid the groundwork for modern, efficient detection algorithms.
- Great for learning and small projects, but not used in real-time systems today.
- Try building a simple detector and experiment with stride, window size, and classifier type for hands-on understanding!

Convolution Implementation of Sliding Windows





1. Background & Motivation

- **Sliding windows**: Classic method for object detection—run a classifier on every possible region of an image.
- **Problem**: Running a ConvNet separately on every window is very slow and repeats a lot of calculations.

2. Turning Fully Connected Layers into Convolutional Layers

- **Standard ConvNet**: Input (e.g., 14x14x3) → Conv layers → Max pool → Fully connected (FC) layers → Output (e.g., softmax for 4 classes).
- **Key idea**: FC layers can be replaced by convolutional layers with appropriate filter sizes.
  - Example: FC layer with 400 units can be implemented as 400 filters of size 5x5 (if previous layer is 5x5x16), giving output 1x1x400.
  - 1x1 convolutions can replace FC layers for further processing.

Layer	Input Size	Output Size
Input	14x14x3	14x14x3
Conv (5x5, 16)	14x14x3	10x10x16
Max Pool (2x2, s=2)	10x10x16	5x5x16
Conv (5x5, 400)	5x5x16	1x1x400

Layer	Input Size	Output Size
1x1 Conv (400)	1x1x400	1x1x400
1x1 Conv (4)	1x1x400	1x1x4

- **Result**: The network becomes fully convolutional—can process images of different sizes and make predictions at multiple locations in one pass.

3. Convolutional Sliding Windows – How It Works

- **Old way**: For a 16x16 image, run the ConvNet 4 times (once for each 14x14 window, with stride 2).
- **New way**: Run the ConvNet once on the whole image, using convolutional layers instead of FC layers.
  - Shared computation: Overlapping regions only need to be computed once.
  - Output: Instead of a single prediction, get a grid of predictions (e.g., 2x2x4 for 4 classes at 4 locations).
- **Example**:
  - Input: 16x16x3 image
  - After convolutions and pooling: 6x6x16
  - After 5x5 conv with 400 filters: 2x2x400
  - After 1x1 conv and softmax: 2x2x4 (each 1x1x4 cell = prediction for a window)

Layer	Input Size	Output Size
Conv (5x5, 16)	16x16x3	12x12x16
Max Pool (2x2, s=2)	12x12x16	6x6x16
Conv (5x5, 400)	6x6x16	2x2x400
1x1 Conv (400)	2x2x400	2x2x400
1x1 Conv (4)	2x2x400	2x2x4

4. Benefits

- **Much faster**: One forward pass gives predictions for all windows.
- **Efficient**: No repeated computation for overlapping regions.

- **Flexible**: Works for images of different sizes; output grid size depends on input size and stride.

5. Limitations

- **Bounding box precision**: Window stride limits how precisely you can localize objects (e.g., stride 2 means you can only detect at every 2 pixels).
- **Next step**: Later methods (like YOLO, SSD) improve localization and speed even more.

6. Summary Table

Step	Old Sliding Window	Convolutional Implementation
Run ConvNet	On each window	Once on whole image
Computation	Repeated	Shared
Output	One label per window	Grid of labels
Speed	Slow	Fast

7. Simple English Recap

- Instead of checking every part of the image one by one, you can check all parts at once using convolutional layers.
- This saves time and makes object detection much faster.

8. Project Ideas & Fun Applications

- **Build a simple object detector**: Use a small ConvNet and try both the old and new methods—compare speed and output.
- **Visualize sliding windows**: Show how the output grid matches different parts of the image.
- **Experiment with stride**: See how changing stride affects detection accuracy and speed.
- **Try on your own images**: Use the convolutional sliding window approach to find faces, cars, or other objects in photos.
- **Animation project**: Animate the difference between running the ConvNet on each window vs. the whole image at once.

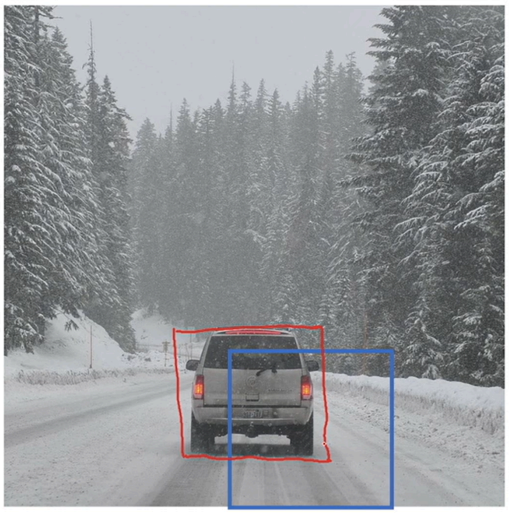
Key takeaway:

> Convolutional implementation of sliding windows lets you detect objects much faster by sharing computation, making it practical for real-world applications.

Bounding Box Predictions

1. Background & Motivation

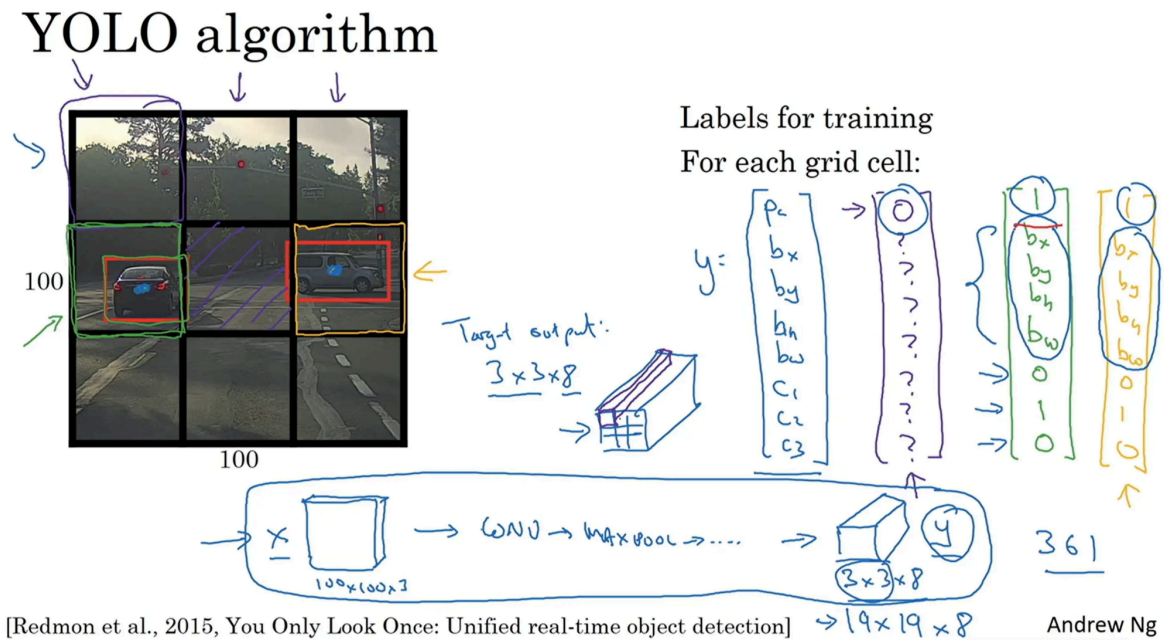
Output accurate bounding boxes



- Traditional *sliding window* convolutional methods for object localization are limited:
  - Output bounding boxes are *discrete* and often do not match object positions/shape accurately.
  - The ground truth bounding boxes are often not perfectly square; real-world objects may need *asymmetric* boxes.

---

2. YOLO Algorithm ("You Only Look Once"): Detailed Structure



Definition:

- YOLO reframes object detection as a *regression problem* for spatially separated bounding boxes and associated class probabilities.

Core Concept:

- Place a *grid* (e.g., 3×3, in actual implementation typically 19×19) over the input image.
- Each grid cell is responsible for detecting objects whose *midpoint* falls inside the cell.

Label Vector Format (Y):

- For every grid cell, output an 8-dimensional vector:
  - p_c — Is there an object? (Objectness score, binary)
  - b_x, b_y — Bounding box center offset (relative to grid cell)
  - b_h, b_w — Height and width of the bounding box (fractional, relative to grid cell size)
  - c_1, c_2, c_3 — Class probabilities (one-hot encoding; e.g., pedestrian, car, motorcycle)

Training Data Mapping:

- For each grid cell:
  - If no object, vector is: [0,–,–,–,–,–,–,–]
  - If object midpoint inside cell, vector is: $[1, b_x, b_y, b_h, b_w, c_1, c_2, c_3]$

- Objects assigned to the **single grid cell** containing their midpoint (even if spanning multiple cells).

Target Output Shape:

- For an image with a grid of n×n, the neural network outputs a n×n×8 volume.

---

3. Architecture & Training

Input:

- Typical: Image size 100×100×3.

Network Design:

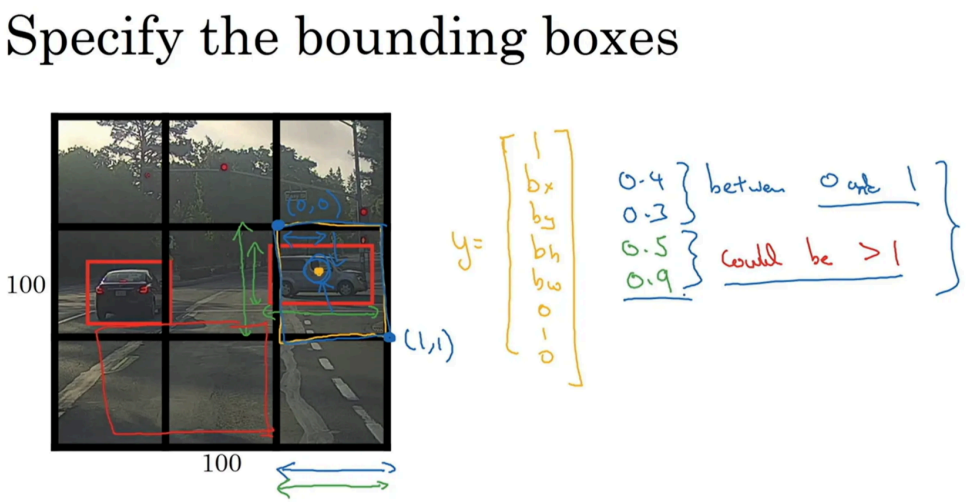
- Standard convolutional layers and max-pooling, culminating in final output compatible with grid structure (e.g., 3×3×8).
- Backpropagation used to train for mapping input image X to target volume Y.

Inference/Test Time:

- Feed image X, network outputs Y (volume of probability and bounding box predictions).
- For each grid cell, retrieve:
  - Objectness score (is there an object?)
  - Class prediction
  - Bounding box parameters (only for cells with object score = 1)
- Assumes *maximum one* object per cell (problem discussed in extensions).

---

4. Bounding Box Parameterization



Relative Coordinates:

- All bounding box values (b_x, b_y, b_h, b_w) are normalized relative to grid cell:
  - Top-left corner = (0,0)
  - Bottom-right corner = (1,1)
  - Box center (b_x, b_y) must be within since assigned based on midpoint.
- Heights/widths (b_h, b_w) can **exceed 1** if object overlaps grid cell boundaries.

Advanced Parameterization:

- YOLO papers may use sigmoid functions to constrain (b_x, b_y) between 0 and 1, exponentials for
  (b_h, b_w) to ensure non-negativity.

---

5. Grid Structure & Practical Notes

- Real implementations use finer grids (e.g., 19×19), reducing the chance of >1 object per cell.
- Assignment algorithm: Assign object to cell containing its midpoint, not multiple cells.

Efficiency:

- Computationally *efficient* via a single convolutional network run (not independent runs per cell).

- Scales to real-time detection due to shared computation.

---

6. Comparison with Sliding Windows

Aspect	Sliding Window	YOLO
Bounding Box Precision	Discrete, stride-bound	Continuous, spatial regression
Computation	Many separate runs	Single, shared convolutional net
Output Representation	Binary classification	Multi-parametric regression
Speed	Slower	Much faster; real-time capable

---

7. Research Extensions & Paper Advice

- YOLO research papers present more advanced parametrizations; can be harder to understand for beginners, even senior researchers may need to refer to code or contact authors.
- It is normal to find original papers challenging; focus on understanding the *core logic*, *vector structure*, and *assignment algorithm*.

---

8. Summary of Key Equations and Concepts

- **Label Vector (per cell):** $Y = [p_c, b_x, b_y, b_h, b_w, c_1, c_2, c_3]$
- **Network Output Shape:** $\text{Output Volume} = n \times n \times 8$
- **Bounding box parametrization:** (relative to grid cell, can use advanced parametrizations like sigmoids/exponentials for constraints)

---

9. References

- YOLO algorithm: Redmon et al. (Joseph Redmon, Santosh Divvala, Rajesh B., Ali Farhadi)

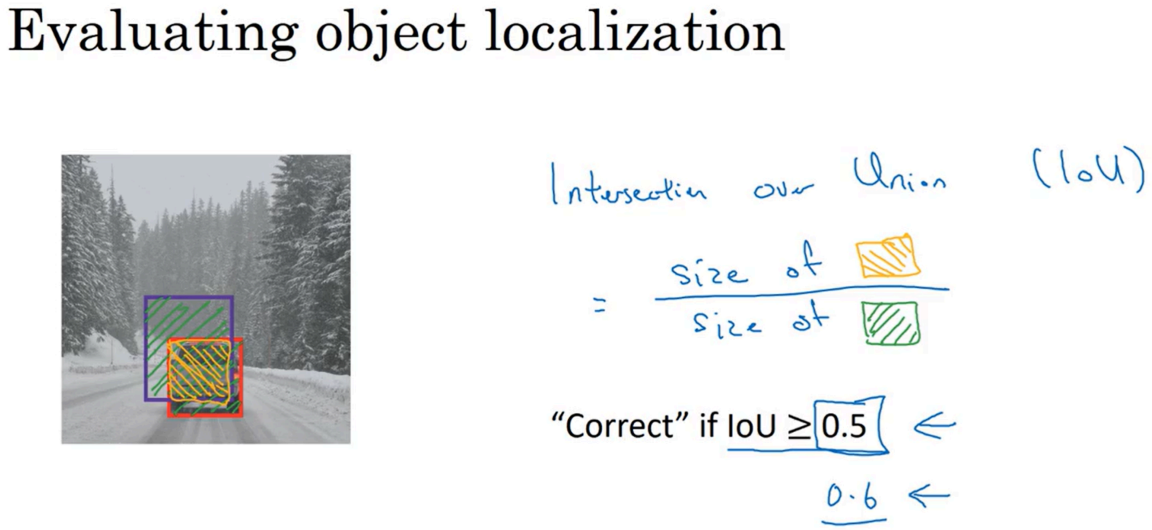
  https://arxiv.org/pdf/1506.02640.pdf

- Further reading: YOLO original paper, codebase, open-source projects for implementation details.

Tip for Researchers: If facing confusion in implementation details, verify against open source code and reach out to authors/community as needed. Concept clarity comes first;

parametrization details can be iterated with practical experimentation.

Intersection Over Union



Definition and Purpose

- *Intersection Over Union (IoU)* is a key metric for evaluating *object detection algorithms*, specifically their ability to localize objects with bounding boxes.
- *Objective*: Quantify how much the predicted bounding box overlaps the ground-truth bounding box.

Mathematical Formula

$$\text{IoU} = \frac{\text{Area of Intersection}}{\text{Area of Union}}$$

- **Intersection**: Region shared by both predicted and ground-truth bounding boxes.
- **Union**: Total area covered by both boxes, i.e., area contained in either box.

Properties & Interpretation

- $\text{IoU} = 1$: *Perfect overlap*; both intersection and union areas are equal.

- $\text{IoU} \geq 0.5$: By *convention*, often considered a *correct prediction* in object detection tasks.
- *Threshold*: The 0.5 criterion is *arbitrary*—can be increased (e.g., 0.6, 0.7) for stricter evaluation.
- *Higher IoU → More accurate localization*.

Practical Notes

- Used in object detection to *map localization accuracy*.
- Count predictions as “correct” if IoU meets or exceeds the chosen threshold.
- Many research papers and competitions use the 0.5 threshold, though some apply stricter measures.

Generalization

- IoU can be applied to *measure similarity* between any two bounding boxes.
- Useful in *non-max suppression* (NMS), where IoU is used to filter overlapping boxes.

Miscellaneous

- Not related to "I owe you" (promissory notes, also abbreviated IOU).

Summary Table

Metric	Formula	Typical Threshold	Interpretation
IoU	$\frac{\text{Area of Intersection}}{\text{Area of Union}}$	0.5–0.7	Overlap quality between boxes

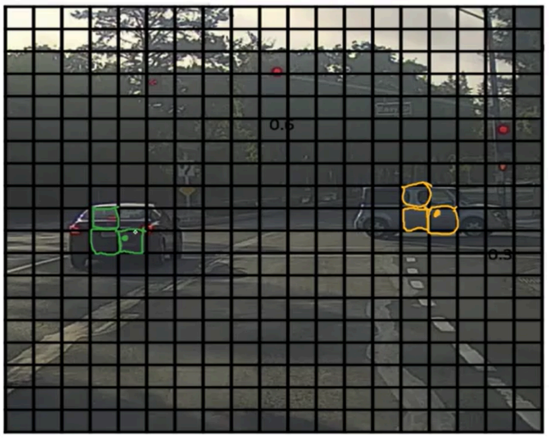
Conclusion

- IoU is foundational for robust evaluation and improvement of object detection models. It offers a quantifiable way to judge bounding box predictions and is widely adopted in computer vision research and practice.

Non - max Suppression

Introduction & Motivation

Non-max suppression example



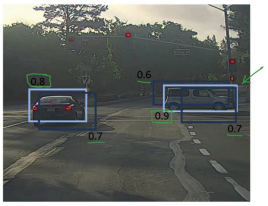
- In object detection, the algorithm may output *multiple bounding boxes* for a single object (e.g., the same car detected several times).
- **Non-max suppression (NMS)** ensures that *each object is detected only once* by suppressing redundant bounding boxes.

Why Needed?

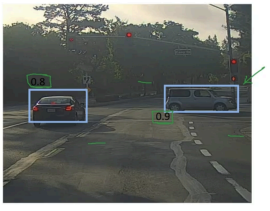
- Grid-based detection (e.g., 19×19 grid on an image) leads to *multiple, overlapping predictions* from nearby cells about the same object.
- *Without NMS*: The output contains several high-confidence detections for the same object.

Core Principle

Non-max suppression example



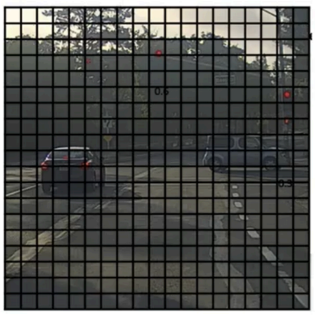
Non-max suppression example



- *NMS selects the best bounding boxes (highest probability)* and *suppresses all other boxes* with a high overlap (measured by IoU) with those selected.

Algorithm Steps

Non-max suppression algorithm



19× 19

Each output prediction is: p_c, b_x, b_y, b_h, b_w, c_1, c_2, c_3

Discard all boxes with $p_c \leq 0.6$

While there are any remaining boxes:

- Pick the box with the largest p_c Output that as a prediction.
- Discard any remaining box with IoU ≥ 0.5 with the box output in the previous step

Andrew Ng

1. **Discard Low-Probability Boxes**:
   - Remove all bounding boxes with confidence $P_c \leq$ threshold (e.g., 0.6).
2. **Iterative Selection**:
   - While any boxes remain:
     - Pick the box with the *highest confidence score* that is not yet processed.
     - *Output* this box as a *final prediction*.
     - *Suppress* all remaining boxes with *high IoU overlap* (IoU above threshold) with the selected box.
     - Repeat until all boxes are either selected or suppressed.

Pseudocode Outline

```
while boxes remain:
    select box with highest Pc
    output the box as final prediction
    suppress all remaining boxes with IoU > threshold with selected box
```

- *Result*: Only non-overlapping, highest-confidence detections remain.

Thresholds & Settings

- *Confidence Threshold*: Removes boxes unlikely to contain objects.
- *IoU Threshold*: Controls how much overlap triggers suppression (typically 0.5, but tunable).

Extension to Multiple Classes

- For multi-class problems (e.g., pedestrian, car, motorcycle), NMS is applied *separately for each class*.
- Output vector has multiple class probabilities; independently run NMS per class.

Visualization & Effect

- After NMS, only one bounding box per object (the one with the highest confidence) is kept; overlapping lower-confidence boxes are suppressed.

Summary Table

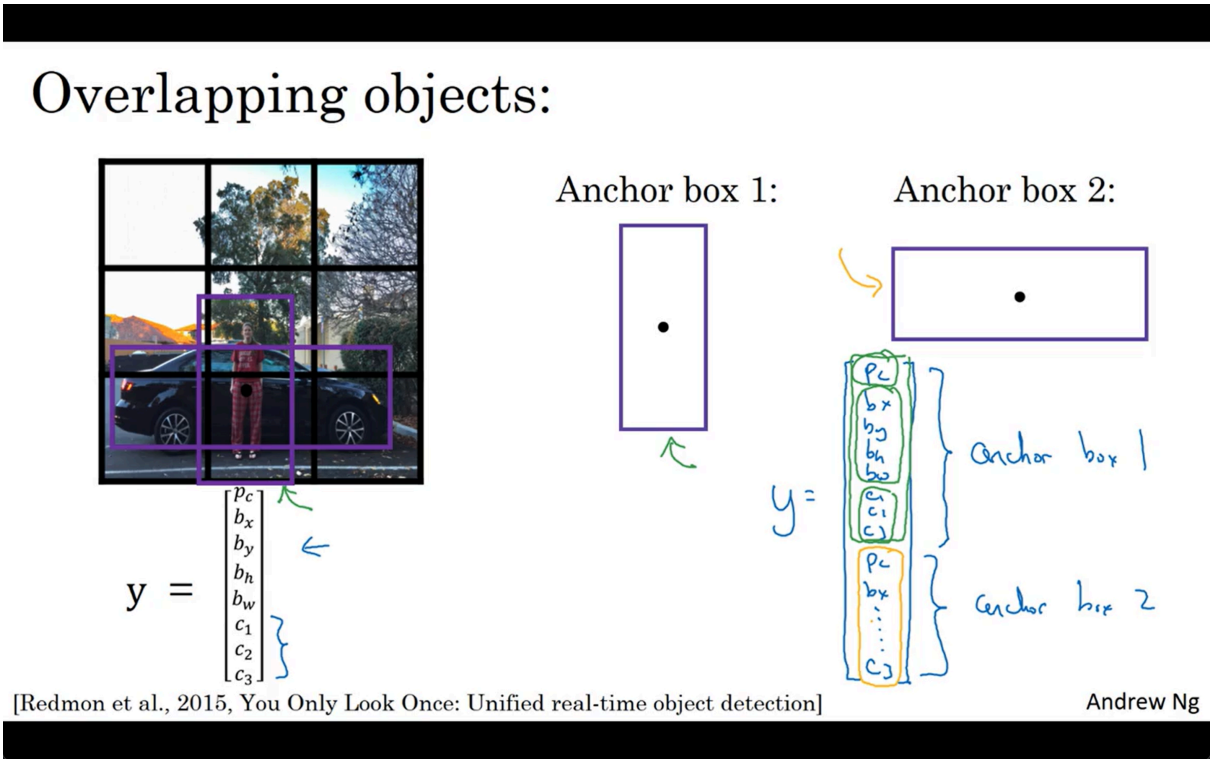
Step	Action
Discard low confidence boxes	Remove boxes with $P_c \leq$ threshold
Select most confident box	Output as prediction
Suppress overlaps	Remove boxes with high IoU with the selected box
Repeat	Until no boxes remain

Conclusion

- Non-max suppression is *critical for correct detection counts* and *robust object localization* in deep learning-based object detectors. It prevents double/multiple counts of the same object and results in clear, non-redundant predictions.

Anchor Boxes

Problem Context



- In grid-based object detection (e.g., YOLO), each grid cell is restricted to predicting *one object*.
- **Challenge**: When multiple objects' centers fall into one grid cell, standard methods fail to assign distinct detections for each object.

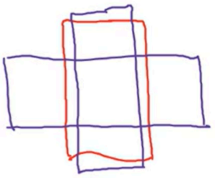
Anchor Boxes: Concept

Anchor box algorithm

Previously:

Each object in training image is assigned to grid cell that contains that object's midpoint.

Output y: $3 \times 3 \times 8$



With two anchor boxes:

Each object in training image is assigned to grid cell that contains object's midpoint and anchor box for the grid cell with highest IoU.

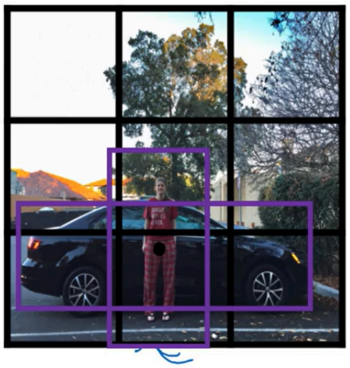
(grid cell, anchor box)

Output y: $3 \times 3 \times 16$ or $3 \times 3 \times 2 \times 8$

Andrew Ng

- **Anchor boxes (a.k.a. prior boxes)**: Pre-defined shapes or templates (rectangles of various aspect ratios/sizes).
- Each grid cell predicts multiple bounding boxes, one for each anchor box shape.
- *Goal:* Allow each cell to detect and distinguish multiple objects.

Anchor box example



Anchor box 1: Anchor box 2:

$$y = \begin{bmatrix} p_c \\ b_x \\ b_y \\ b_h \\ b_w \\ c_1 \\ c_2 \\ c_3 \\ p_c \\ b_x \\ b_y \\ b_h \\ b_w \\ c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} 1 \\ b_x \\ b_y \\ b_h \\ b_w \\ 1 \\ 0 \\ 0 \\ 1 \\ b_x \\ b_y \\ b_h \\ b_w \\ 0 \\ 1 \\ 0 \end{bmatrix} \quad \begin{matrix} \text{car only?} \\ \begin{bmatrix} 0 \\ ? \\ ? \\ ? \\ ? \\ ? \\ ? \\ ? \\ 1 \\ b_x \\ b_y \\ b_h \\ b_w \\ 0 \\ 1 \\ 0 \end{bmatrix} \end{matrix} \quad \begin{matrix} \text{anchor box 1} \\ \\ \\ \\ \\ \\ \\ \\ \\ \text{anchor box 2} \\ \\ \\ \\ \\ \\ \end{matrix}$$

Andrew Ng

Mathematical Structure

- For each grid cell, instead of one output vector per cell, *repeat the detection vector* for each anchor box (e.g., if 2 anchor boxes: two 8-element vectors per cell).
    - **Each detection vector for anchor box:**
        - P_c : Objectness/confidence score (is there an object?)
        - b_x, b_y, b_h, b_w: Bounding box coordinates relative to cell
        - $c_1, c_2, \ldots, c_n$: Class probabilities (e.g., pedestrian, car)
- Output tensor becomes $(\text{grid size}) \times (\text{grid size}) \times (\text{number of anchor boxes} \times \text{vector length})$.
    - Example: 3×3×2×8=3×3×16 for 2 anchor boxes, 3 classes.

Assignment & Encoding

- During training, an object is assigned to the anchor box (among those in the relevant cell) with the *highest IoU* with the object's true bounding box.
- Each (cell, anchor box) pair either:
    - Encodes an object (confidence = 1, box, and class info), **or**
    - Encodes "no object" (confidence = 0, remaining values "don't care").

Example

- In grid cell where both a car and a pedestrian are centered:
    - Assign the car to the anchor box shape most similar to a car and the pedestrian to the shape most similar to a human.

Limits & Corner Cases

- *If more objects are centered in one cell than number of anchor boxes?* Not handled optimally (rare if grid is fine enough).
- If two objects match the same anchor box best: tie-breaking is used.
- Not ideal, but unlikely with typical settings (e.g., 19×19 grid).

Benefits

- Enables detection of *multiple overlapping objects* in the same grid cell.
- Allows the network to specialize:
    - Some anchor boxes detect tall, skinny objects (e.g., pedestrians)

- Others specialize in wide, short objects (e.g., cars)

Design and Selection

- Anchor box shapes can be selected:
  - By hand (using diverse sizes/aspect ratios) to cover expected object shapes.
  - Via automation (e.g., K-means clustering on training box shapes, as in advanced YOLO versions).

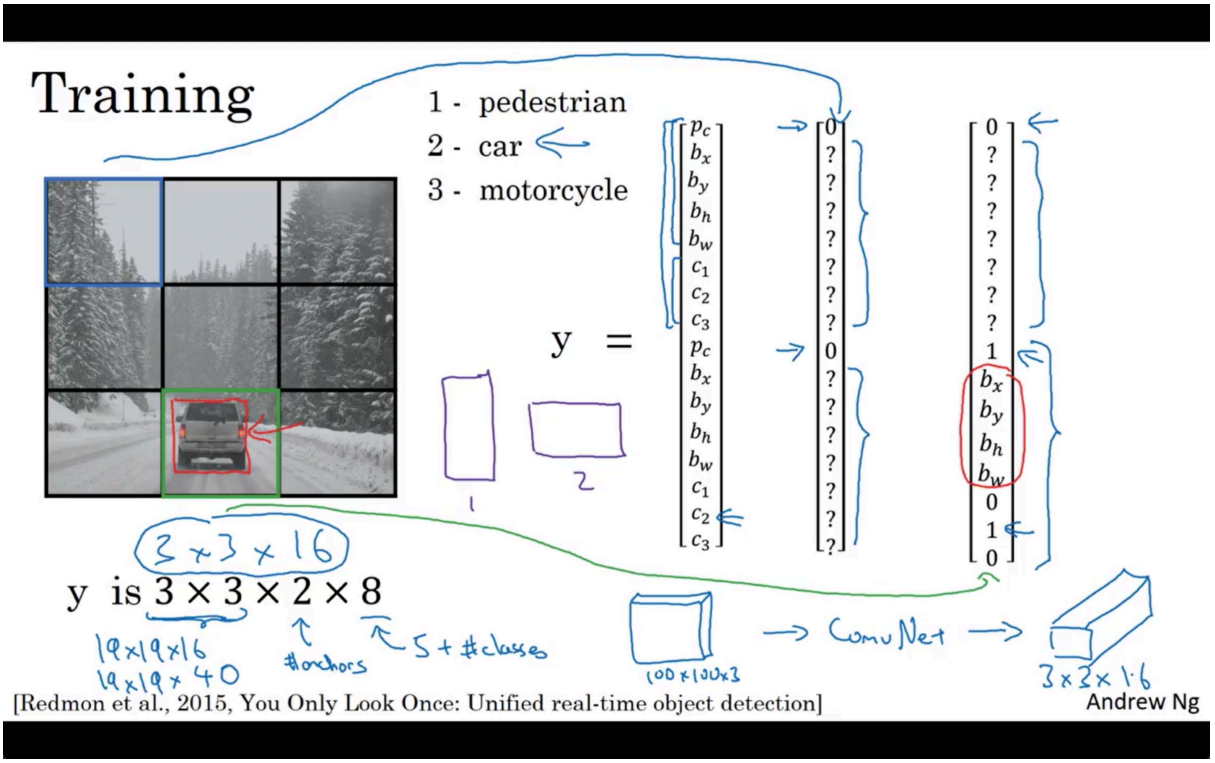
Summary Table

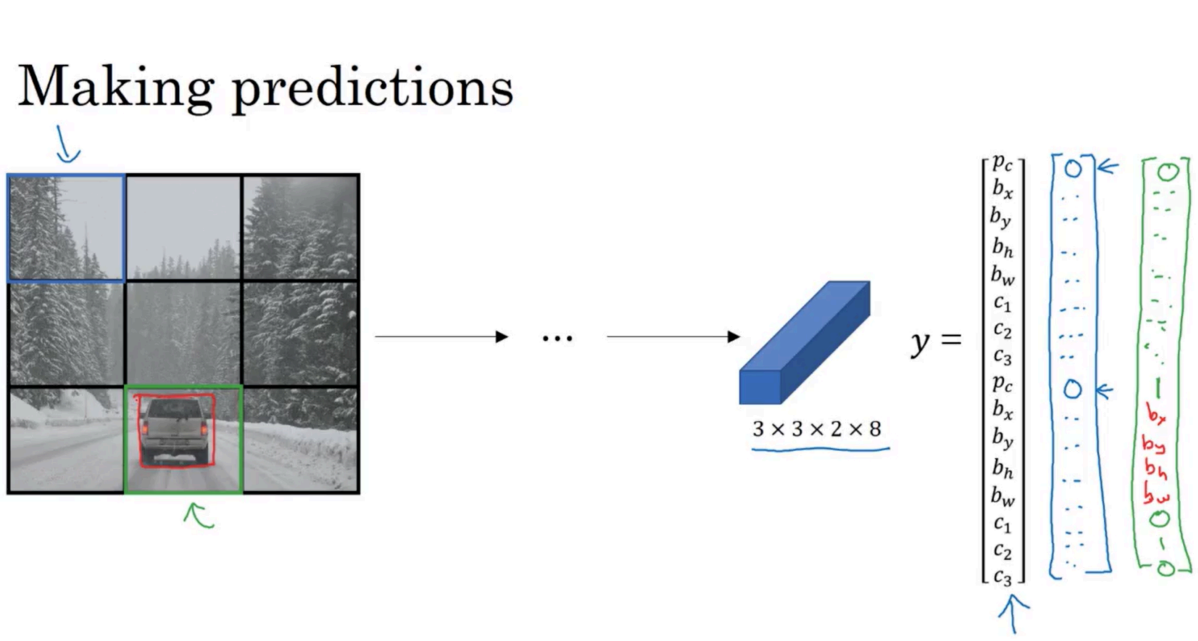
Concept	Description
Anchor Box	Pre-defined box template for prediction specialization
Tensor Output	$(\text{grid size}) \times (\text{grid size}) \times (\text{num anchors} \times \text{vector length})$
Object Assignment	Object ↔ anchor box–cell pair with highest IoU
Selection Strategies	Hand-picked or via K-means on bounding box shapes
Specialization	Each anchor can specialize for different shapes (car, pedestrian)
Limitation	More objects/cell than anchors: not optimal (rare in practice)

Conclusion

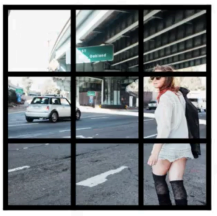
- *Anchor boxes* are essential for robust, multi-object detection in modern architectures (YOLOv2, SSD, etc.). They boost detection accuracy by enabling grid cells to output multiple predictions and by letting the model specialize detectors by object shape.

YOLO Algorithm

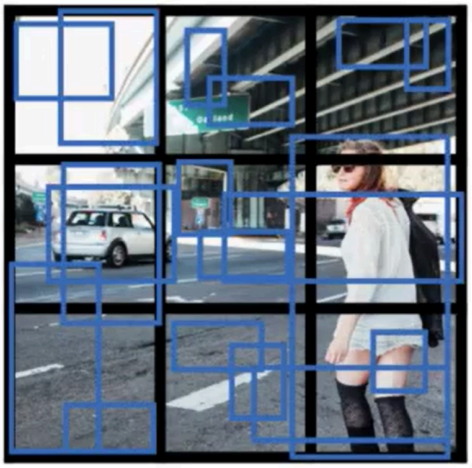

Training
1 - pedestrian
2 - car
3 - motorcycle
y =
y is 3 × 3 × 2 × 8
3 × 3 × 16
19 × 19 × 16
19 × 19 × 40
#anchors
5 + #classes
100 × 100 × 3
ConvNet
3 × 3 × 16
[Redmon et al., 2015, You Only Look Once: Unified real-time object detection]
Andrew Ng

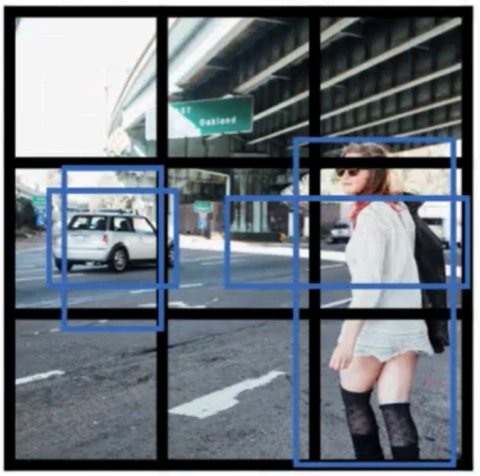

Making predictions
3 × 3 × 2 × 8
y =

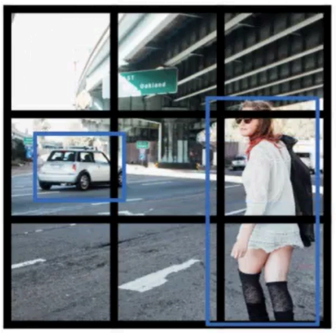
Outputting the non-max supressed outputs



- For each grid call, get 2 predicted bounding boxes.







- For each grid call, get 2 predicted bounding boxes.
- Get rid of low probability predictions.
- For each class (pedestrian, car, motorcycle) use non-max suppression to generate final predictions.

in the image it is “For each grid cell” not each grid call

1. Problem Setup & Output Structure

- **Goal:** Detect multiple object classes (e.g., pedestrian, car, motorcycle) in an image using a single neural network pass.

- **Grid Division:** Image is divided into an N×N grid (e.g., 3×3 for illustration, 19×19 in practice).
- **Anchor Boxes:** Each grid cell predicts multiple bounding boxes (one per anchor box shape). Example: 2 anchor boxes per cell.
- **Output Tensor:**
  - Shape: $N \times N \times (\text{num anchors} \times (5 + \text{num classes}))$
  - For 3 classes, 2 anchors, 3x3 grid: 3×3×2×8=3×3×163
  - Each anchor's vector: $[P_c, b_x, b_y, b_h, b_w, c_1, c_2, c_3]$
    - P_c: Probability object exists
    - b_x, b_y, b_h, b_w: Bounding box (relative to cell)
    - c_1, c_2, c_3: Class probabilities

2. Training Set Construction

- For each grid cell and anchor box:
  - If no object center falls in the cell: $P_c = 0$, rest are "don't care" (ignored in loss).
  - If an object center falls in the cell:
    - Assign object to anchor box with highest IoU to its true box.
    - For that anchor: $P_c = 1$, bounding box coordinates, correct class label;
      other anchors in cell: $P_c = 0$, rest "don't care".
- **Output volume:** For all grid cells, stack anchor box vectors to form the full output tensor.

3. Prediction (Inference) Pipeline

- **Input:** Image (e.g., 100×100×3).
- **Network Output:** Tensor as above (e.g., 3×3×16).
- For each grid cell and anchor box:
  - If P_c is low, ignore prediction (likely no object).
  - If P_c is high, use predicted bounding box and class probabilities.
- **Note:** Network always outputs numbers for all positions, but only high-P_c outputs are meaningful.

4. Post-processing: Non-max Suppression (NMS)

- **Step 1:** Discard all boxes with low P_c (below threshold).
- **Step 2:** For each class (e.g., pedestrian, car, motorcycle):
  - Run NMS independently:
    - Select highest-P_c box for the class.
    - Suppress (remove) all other boxes for that class with high IoU overlap (above threshold) with the selected box.
    - Repeat until no boxes remain for that class.
- **Result:** For each class, only the best, non-overlapping boxes are kept as final predictions.

5. Scalability & Practical Details

- **Grid Size:** In practice, use larger grids (e.g., 19×19) for finer localization.
- **Anchor Boxes:** More anchor boxes (e.g., 5) allow detection of more object shapes per cell; output tensor grows accordingly (e.g., 19×19×40 for 5 anchors, 3 classes).
- **Generalization:** YOLO can be extended to more classes and more complex images by adjusting grid size and anchor box count.

6. Summary Table

Component	Description
Grid	Divide image into N×N cells
Anchor Boxes	Multiple per cell, each predicts a bounding box
Output Tensor	$N \times N \times (\text{anchors} \times (5 + \text{classes}))$
Training Target	Assign object to anchor with highest IoU in cell
Prediction	Use high-P_c outputs, ignore low-P_c
NMS	Remove redundant boxes, keep best per class

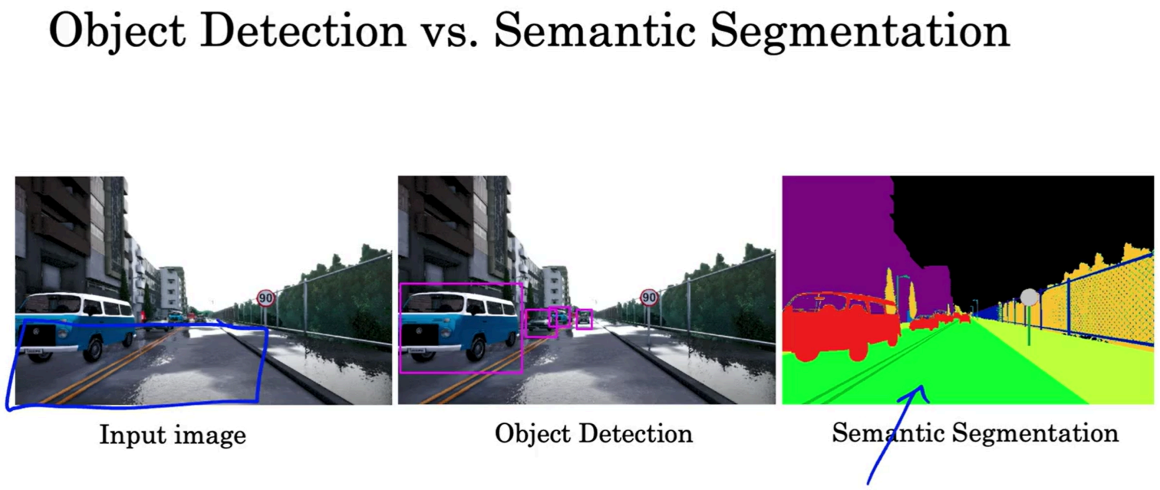
7. Key Takeaways

- YOLO is a *single-pass* object detector: fast and efficient.
- Uses grid and anchor boxes to handle multiple objects and classes.
- Post-processing (NMS) is essential to avoid duplicate detections.

- Output tensor structure is flexible for different grid/anchor/class configurations.
- Widely used in real-time applications due to speed and accuracy.

Semantic Segmentation with U-Net

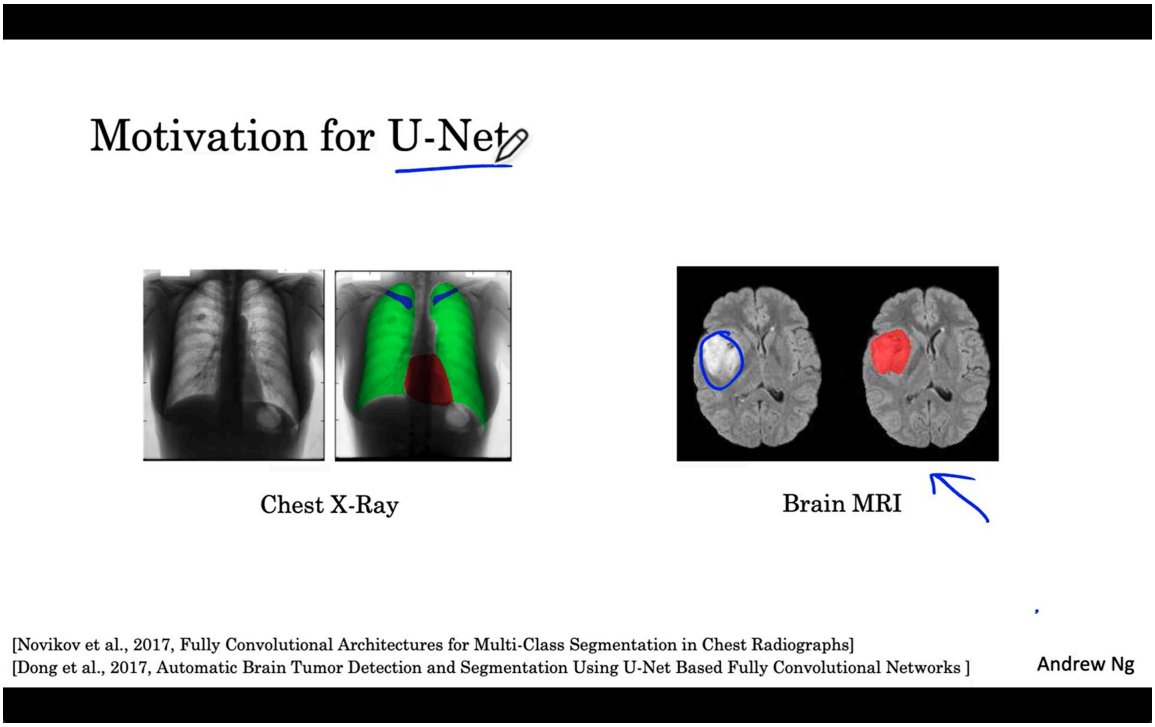
1. Semantic Segmentation: What & Why?



- **Definition:** Assigns a class label to **every pixel** in an image, creating a detailed outline of objects.
- **Comparison:**
  - *Object Recognition:* Classifies the whole image (e.g., "cat" or "not cat").
  - *Object Detection:* Draws bounding boxes around objects.
  - *Semantic Segmentation:* Labels each pixel (e.g., which pixels are "cat").

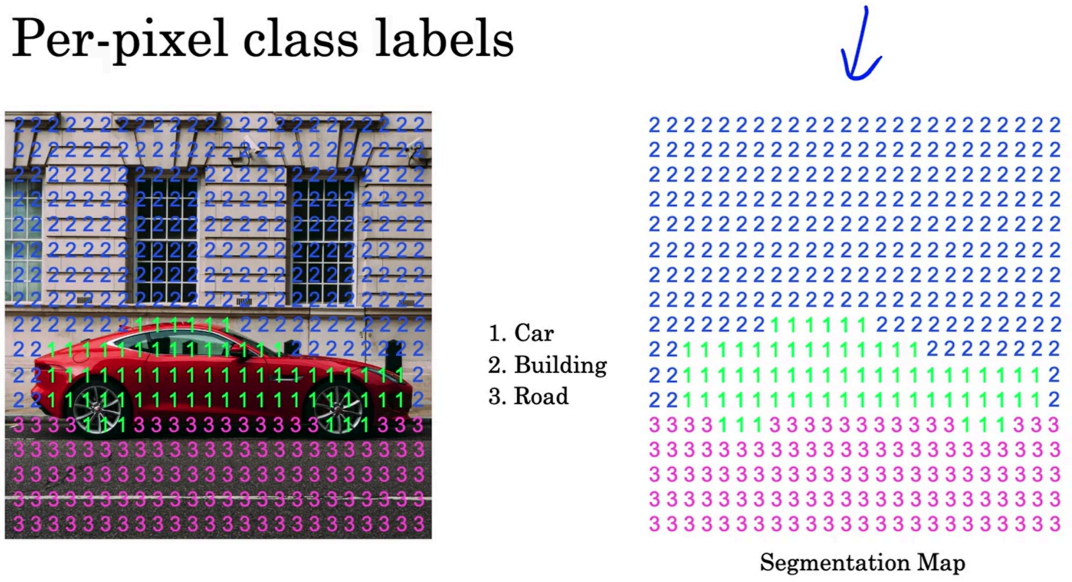
2. Key Applications

- **Self-Driving Cars:**
  - *Detection:* Finds cars with bounding boxes.
  - *Segmentation:* Labels each pixel as "drivable road" or not, helping the car know exactly where it can drive.
- **Medical Imaging:**



- *X-rays:* Segment lungs, heart, clavicle for easier diagnosis and surgical planning.
- *MRI scans:* Automatically segment tumors, saving radiologists time and aiding surgery planning.

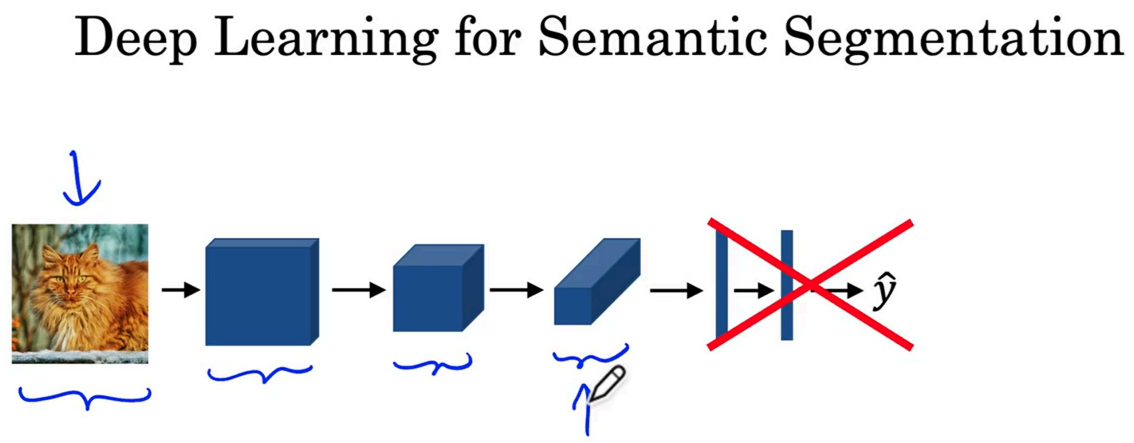
3. How Semantic Segmentation Works



- **Pixel-wise Classification:**
  - For binary segmentation (e.g., car vs. not car):

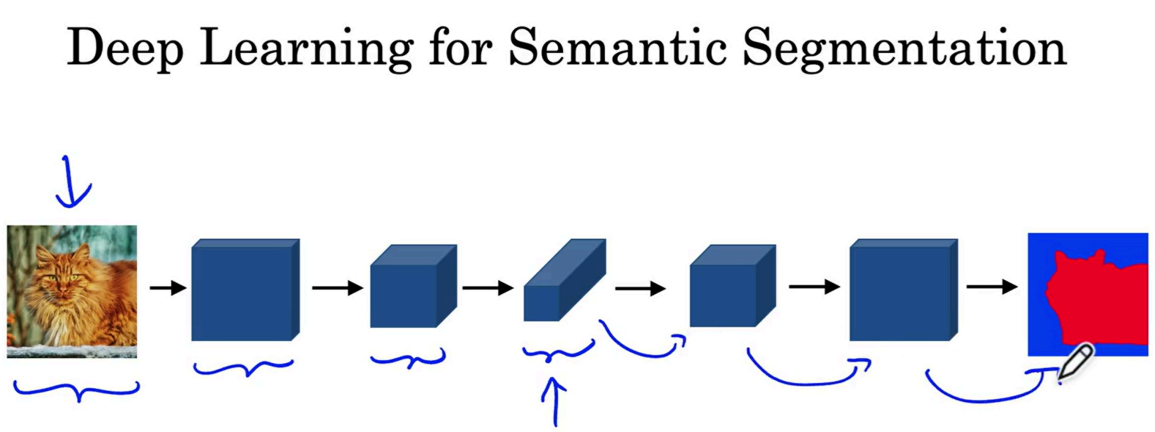
        - Label each pixel as 1 (car) or 0 (not car).
    - For multi-class segmentation (e.g., car, building, road):
        - Assign a unique label to each class (e.g., 1 = car, 2 = building, 3 = road).
- **Output:**
    - The model produces a **matrix** of class labels, one for each pixel.

4. Neural Network Architecture for Segmentation



- **Standard CNNs:**
    - Input image passes through layers, outputting a single class label (for recognition) or a few numbers (for detection).
- **Modification for Segmentation:**
    - Remove the final layers that shrink the output to a single label.
    - The network must "blow up" the activations back to the original image size, so each pixel gets a label.

5. U-Net Architecture



- **Key Idea:**
  - As you go deeper, the spatial dimensions (height, width) shrink, but the number of channels increases.
  - In the "expanding path," the network upsamples (increases) the spatial dimensions back to the original size, while reducing the number of channels.
- **Output:**
  - A segmentation map (same size as input) with a class label for each pixel.

6. Transpose Convolution (Deconvolution)

- **Purpose:**
  - Used to upsample feature maps (make them bigger) in the expanding path of U-Net.
- **Importance:**
  - Essential for reconstructing the full-resolution segmentation map from compressed features.

7. Summary Table: Task Comparison

Task	Output Type	Example Output
Object Recognition	Single class label	"cat"
Object Detection	Class + bounding box	"cat" + box coordinates
Semantic Segmentation	Per-pixel class labels	Matrix: each pixel labeled

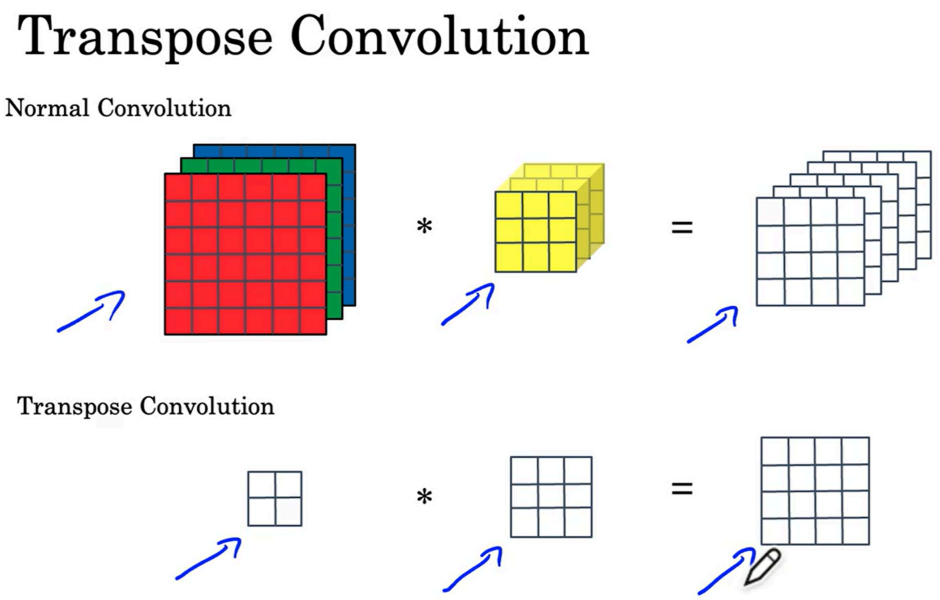
8. Key Takeaways

- Semantic segmentation is crucial for tasks needing precise object boundaries (e.g., self-driving, medical imaging).
- U-Net is a popular architecture for this, using transpose convolutions to upsample features.
- The next step is to understand how transpose convolutions work in detail.

Note: If your task needs pixel-level precision, use segmentation. U-Net is especially effective when you have limited data and need accurate segmentation (e.g., medical images).

Transpose Convolutions

What Problem Does Transpose Convolution Solve?



- **Goal:** Take a small input (e.g., 2×2) and "blow it up" to a larger output (e.g., 4×4).
- **Use case:** Needed in tasks like semantic segmentation, where you want to upsample feature maps.

Key Parameters

- **Filter size (f):** e.g., 3×3
- **Padding (p):** e.g., 1
- **Stride (s):** e.g., 2

How Does Transpose Convolution Work?

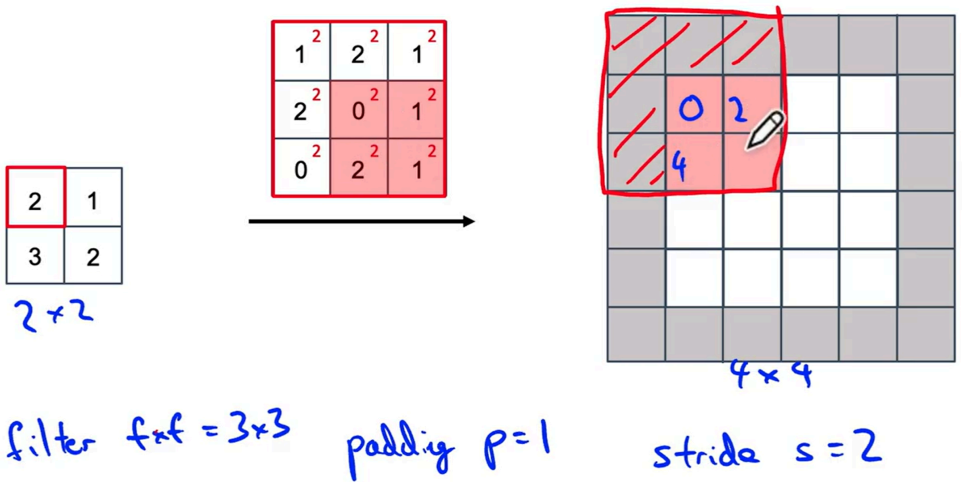
1. **For each input value:**
    - Multiply it by every value in the filter.
    - Place the resulting block into the output at the correct location (based on stride and padding).
2. **Overlap:**
    - If two blocks overlap in the output, **add** their values together.
3. **Repeat for all input values.**

Example Walkthrough

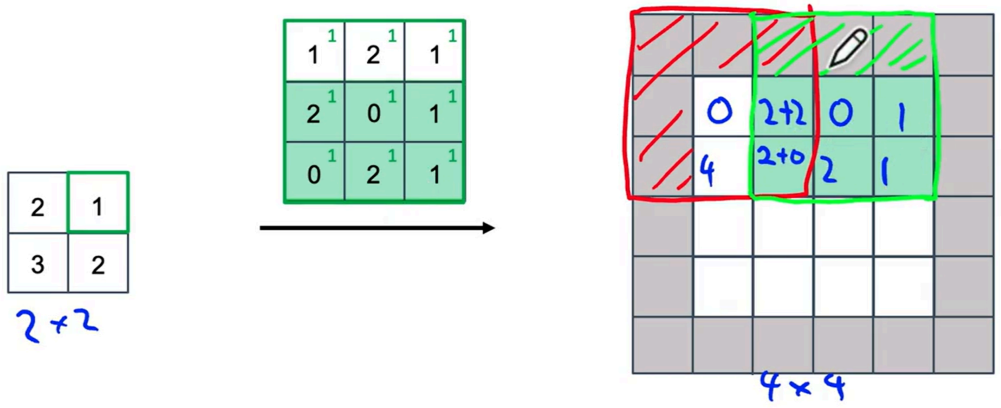
- **Input:** 2×2 matrix
- **Filter:** 3×3
- **Padding:** 1
- **Stride:** 2
- **Output:** 4×4 matrix

Steps:

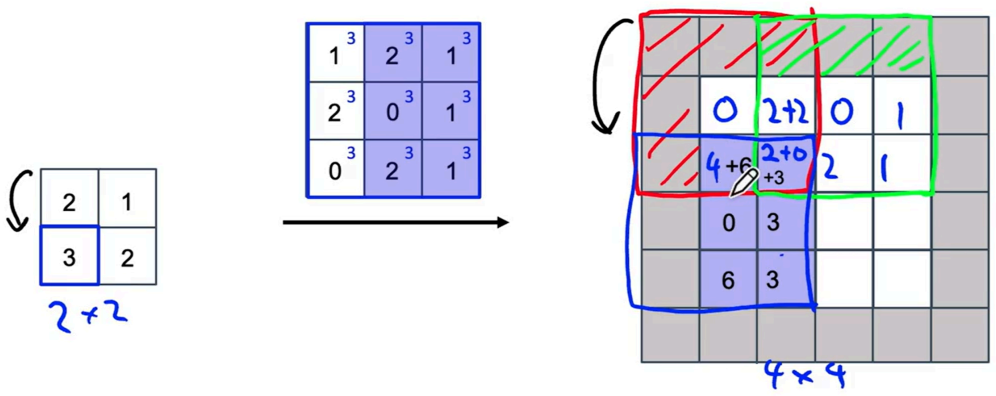
Transpose Convolution



Transpose Convolution

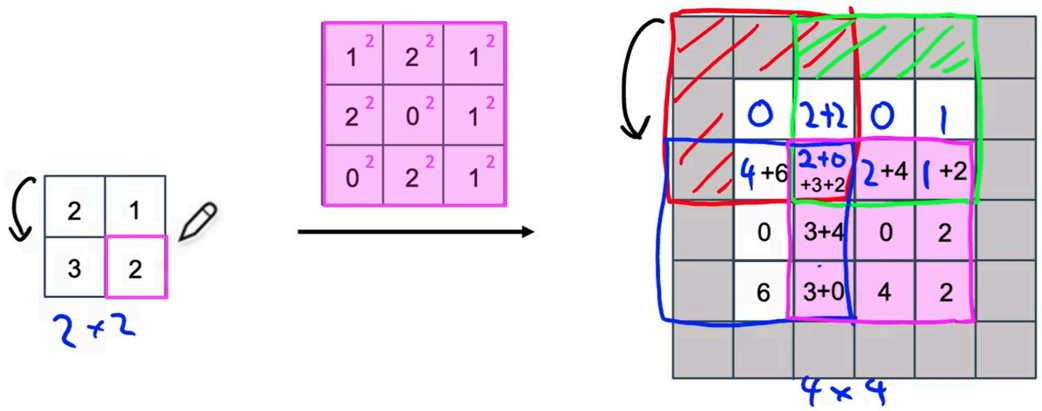


Transpose Convolution

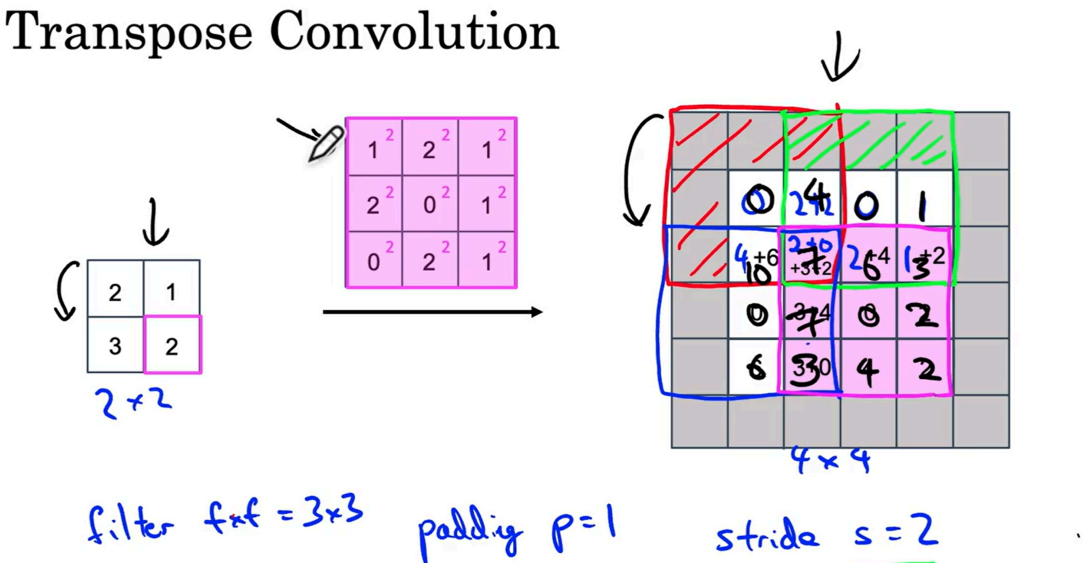


filter $f \times f = 3 \times 3$ padding $p = 1$ stride $s = 2$

Transpose Convolution



filter $f \times f = 3 \times 3$ padding $p = 1$ stride $s = 2$



- Start with the upper-left input value. Multiply by the filter, place the result in the output.
- Move to the next input value, shift the placement by stride, multiply by the filter, and add to overlapping output regions.
- Continue for all input values.
- **Final output:** Add up all overlapping values to get the 4×4result.

Why Add Overlapping Values?

- Overlaps occur because each input value "spreads" its influence over a region in the output.
- Adding ensures all contributions are included, allowing the network to learn effective upsampling.

💡 General Formula

If we have

- Input size: H_{in}, W_{in}
- Kernel size: k
- Stride: s
- Padding: p
- Output padding: op (extra adjustment, usually 0 or 1)

then the **transpose convolution output size** is:

$$H_{out} = (H_{in} - 1) \cdot s - 2p + k + op$$
$$W_{out} = (W_{in} - 1) \cdot s - 2p + k + op$$

Explanation of Each Term

- $(H_{in} - 1) \cdot s (Hin - 1) \cdot s$: expands by stride (inserting zeros).
- −2p: accounts for padding in the forward conv (removes border).
- +k: adds back the filter size.
- +op: optional output padding (to match desired shape, usually 0 or 1).

Example (your case from Andrew Ng)

- $H_{in} = 2$
- k=3
- s=2
- p=1
- op=1

$$H_{out} = (2 - 1) \cdot 2 - 2(1) + 3 + 1 = 4$$

So final output is **4×4**

👉 So the **correct universal formula** is:

$$\boxed{H_{out} = (H_{in} - 1) \cdot s - 2p + k + op}$$
$$\boxed{W_{out} = (W_{in} - 1) \cdot s - 2p + k + op}$$

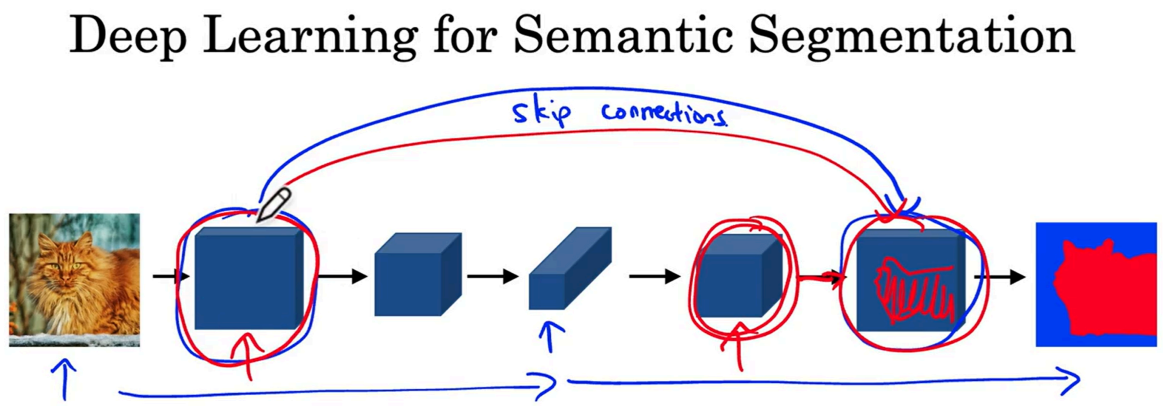
Summary Table

Parameter	Example Value
Input size	2×2
Filter size	3×3
Padding	1
Stride	2
Output size	4×4

Quick Review

- **Transpose convolution** lets you upsample feature maps in a trainable way.
- Each input value is multiplied by the filter and "spread" into the output.
- Overlapping regions are **added** together.
- Used in architectures like **U-Net** for semantic segmentation.

U-Net Architecture Intuition



1. Purpose of U-Net

- **U-Net** is a neural network architecture designed for **semantic segmentation** — assigning a class label to each pixel in an image.
- It is especially useful in fields like **medical imaging** and any task where precise localization is needed.

2. Overall Structure

- U-Net has a **"U"-shaped** architecture with two main parts:
  - **Contracting path (encoder):** Compresses the input image, reducing spatial dimensions but increasing feature depth.
  - **Expanding path (decoder):** Upsamples the compressed representation back to the original image size.

3. Contracting Path (Encoder)

- Uses **regular convolutions** and pooling layers.
- **Effect:**
  - Reduces the height and width of the feature maps (spatial resolution decreases).
  - Increases the number of channels (feature depth increases).
- **Result:**
  - The network captures high-level, contextual information (e.g., "there is a cat in the lower right").
  - **Fine spatial details are lost** due to downsampling.

4. Expanding Path (Decoder)

- Uses **transpose convolutions** (also called up-convolutions) to increase the spatial dimensions.
- **Effect:**
  - "Blows up" the compressed feature maps back to the original image size.
  - Alone, this would not recover fine details lost during downsampling.

5. Skip Connections — The Key U-Net Innovation

- **Skip connections** copy feature maps from the encoder directly to the decoder at corresponding levels.
- **Why?**
  - The decoder needs both:
    - **High-level context** (from the compressed, deep features)
    - **Fine-grained spatial details** (from early, high-resolution layers)
- **How?**
  - For each decoder layer, concatenate (or add) the corresponding encoder feature map.
  - This gives the decoder access to both detailed and contextual information.

6. Intuition Behind Skip Connections

- **High-level features:** Help the network know *where* objects are (e.g., "cat in lower right").
- **Low-level features:** Provide pixel-level details (e.g., "how much fur is in this pixel?").
- **Combining both:** Allows the network to make accurate, pixel-wise decisions (e.g., "is this pixel part of a cat?").

7. Summary Table

Part	Operation	Effect
Encoder	Convolution + Pooling	Downsample, capture context
Decoder	Transpose Convolution	Upsample, recover spatial size
Skip Connections	Copy encoder features	Restore fine details, improve accuracy

8. Key Takeaways

- U-Net combines **context** and **detail** for precise segmentation.
- **Skip connections** are essential for recovering spatial information lost during downsampling.
- Widely used in applications needing accurate, pixel-level predictions.

Quick Review:

- Can you explain why skip connections are important in U-Net?

- How does the encoder-decoder structure help with semantic segmentation?

U-Net Architecture

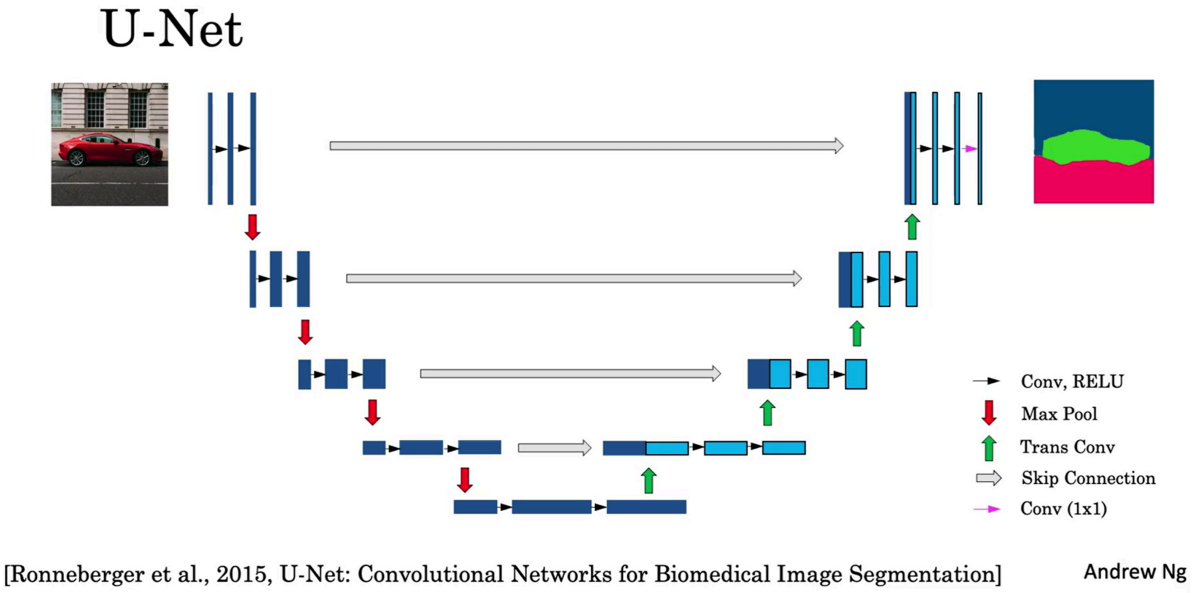
1. Introduction & Motivation

- **U-Net** is a foundational neural network architecture for **semantic segmentation** — assigning a class label to each pixel in an image.
- Originally designed for **biomedical image segmentation**, but now widely used in many computer vision tasks requiring precise localization.
- Named "U-Net" because its architecture diagram forms a U-shape.

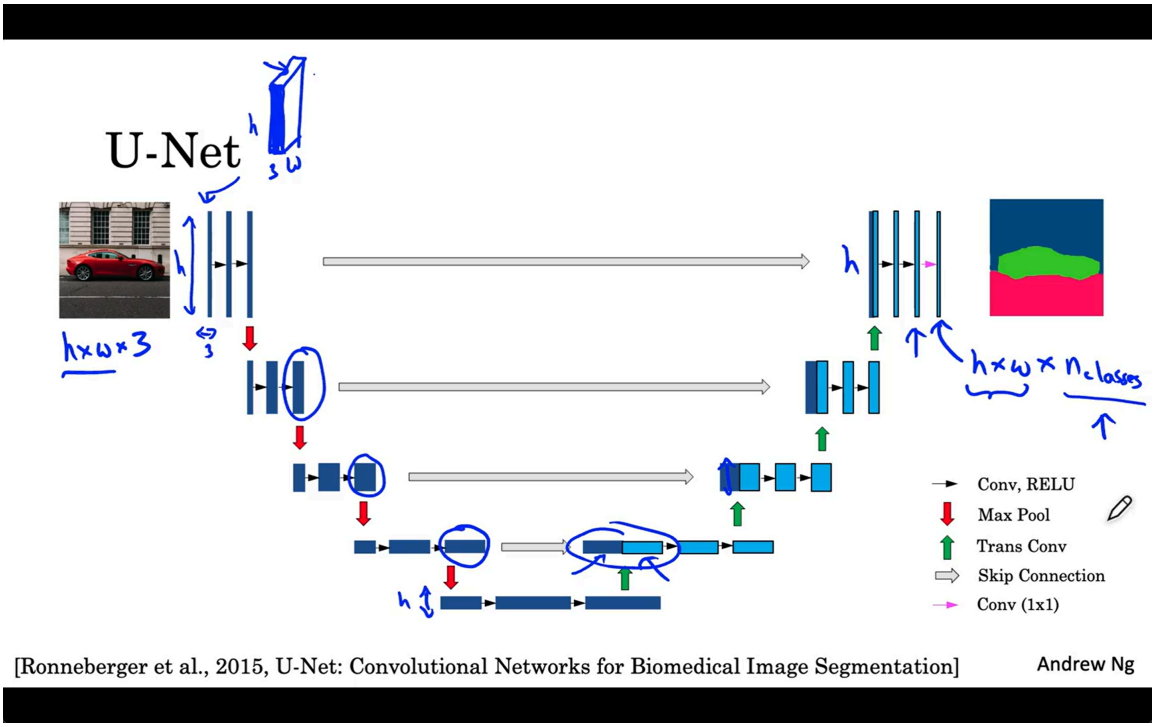
2. Input and Output

- **Input:** An image of size h×w×3 (for RGB images).
- **Output:** A segmentation map of size $h \times w \times n_{\text{classes}}$, where each pixel has a vector of class probabilities.
  - For each pixel, the network predicts the likelihood of belonging to each class.
  - The final class for each pixel is chosen by taking the $\arg\max$ over the class probabilities.

3. U-Net Structure Overview



- The architecture consists of two main paths:
  - **Contracting Path (Encoder):** Downsamples the input, capturing context.
  - **Expanding Path (Decoder):** Upsamples to recover spatial details and original resolution.
- **Skip Connections:** Directly connect corresponding layers in the encoder and decoder to preserve fine-grained information.



4. Contracting Path (Encoder)

- **Sequence of operations:**
    1. **Convolutional layers** (with ReLU activation): Extract features and increase the number of channels.
    2. **Max pooling:** Reduces spatial dimensions (height and width), increases feature abstraction.
    3. **Repeat:** Multiple blocks, each time reducing spatial size and increasing depth.
- **Effect:**
    - The feature maps become smaller in height and width but deeper (more channels).
    - Captures high-level, contextual information but loses some spatial detail.

5. Expanding Path (Decoder)

- **Sequence of operations:**
    1. **Transpose convolution (up-convolution):** Increases spatial dimensions (height and width), reduces the number of channels.
    2. **Skip connection:** Concatenate (or add) the corresponding feature map from the encoder.
    3. **Convolutional layers (with ReLU):** Refine the upsampled features.
    4. **Repeat:** Multiple blocks, each time increasing spatial size and reducing depth.
- **Effect:**
    - Recovers spatial resolution and combines context with fine details.

6. Skip Connections — The U-Net Innovation

- **Purpose:**
    - Restore fine-grained spatial information lost during downsampling.
    - Allow the decoder to access both high-level context and low-level details.
- **How:**
    - At each decoder stage, copy the corresponding encoder feature map and concatenate it with the upsampled output.
    - This is visualized as a "grey arrow" in the diagram.

7. Final Output Layer

- After the last upsampling and skip connection, apply a few more convolutional layers.
- **1x1 Convolution:**
  - Maps the final feature map to the desired number of classes per pixel.
  - Output shape: $h \times w \times n_{\text{classes}}$.
- **Pixel-wise Classification:**
  - For each pixel, the network outputs a vector of class probabilities.
  - The class with the highest probability is assigned to that pixel.

8. Summary Table: U-Net Flow

Stage	Operation	Effect
Input	Image $(h \times w \times 3)$	Start of pipeline
Encoder	Conv + ReLU, Max Pooling	Downsample, capture context
Bottleneck	Deepest features	Most abstract representation
Decoder	Transpose Conv, Skip, Conv + ReLU	Upsample, restore details
Output	1x1 Conv, Softmax	Segmentation map $(h \times w \times n_{\text{classes}})$

9. Key Insights

- **U-Net is powerful for pixel-level prediction tasks** due to its ability to combine context and detail.
- **Skip connections** are essential for recovering spatial information lost during downsampling.
- **Transpose convolutions** allow the network to learn how to upsample feature maps effectively.
- **1x1 convolutions** at the end enable per-pixel classification for any number of classes.

10. Practical Tips

- U-Net can be adapted for different input sizes and numbers of classes.
- Works well even with limited training data, especially in medical imaging.

- The architecture is modular: you can adjust the number of encoder/decoder blocks, channels, and skip connection strategies.

Quick Review Questions:

- Why are skip connections important in U-Net?
- What is the role of transpose convolutions in the decoder?
- How does the network produce a segmentation map from the final feature map?