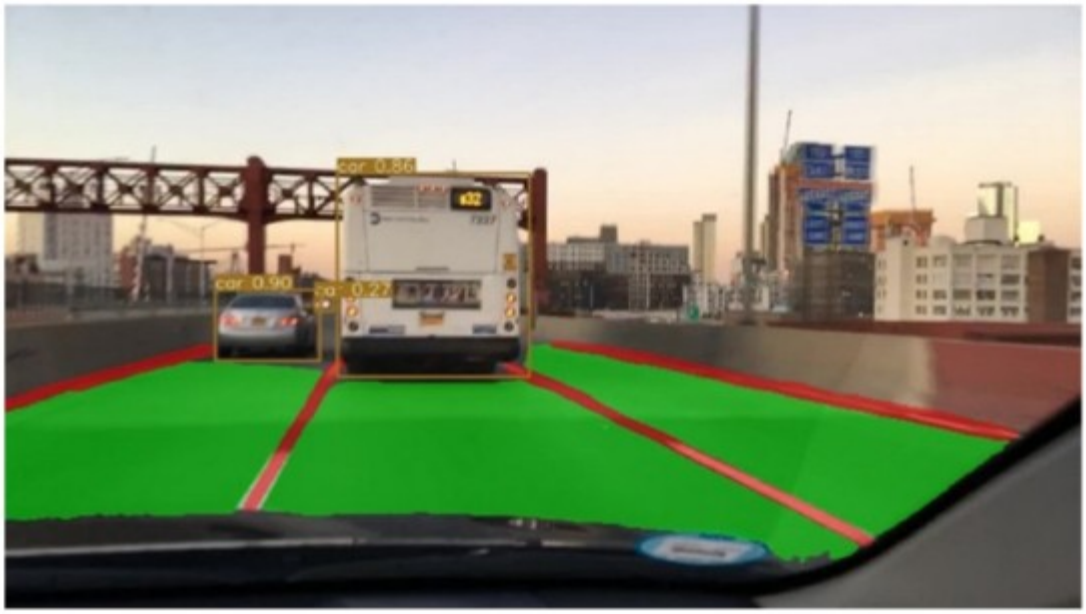
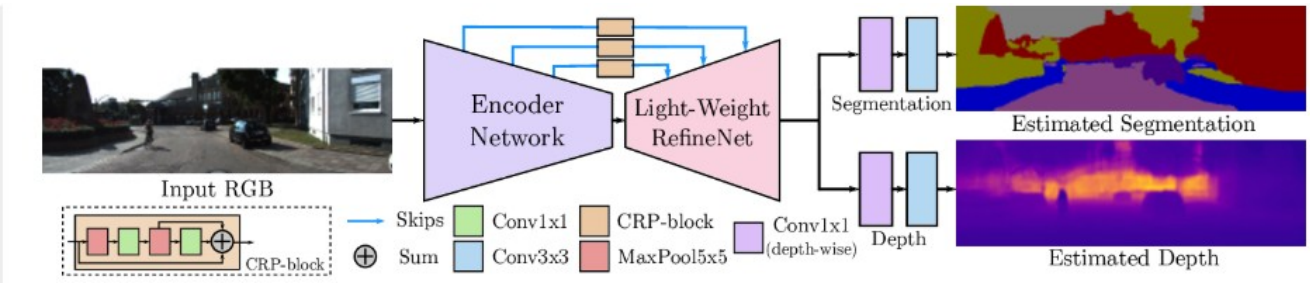
Problem domain + Intro to architecture - Minh

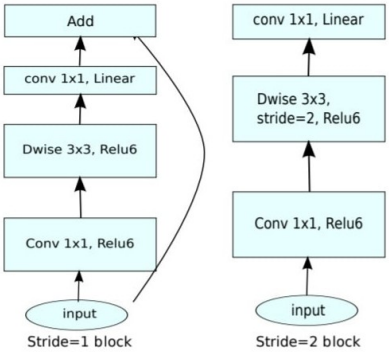
Slide 1. - Problem Domain



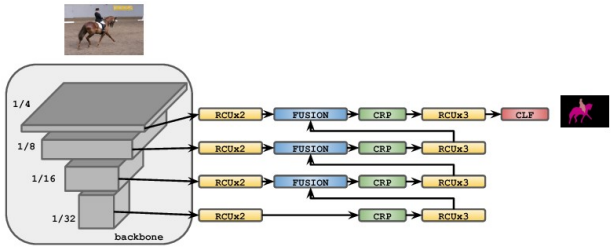
A single task like semantic segmentation is already quite heavy on its own, but autonomous vehicles need much more than that to make safe decisions like object detection, depth estimation, and more. Running separate models for each of these tasks slows down the system, dropping the FPS and causing delays in decision-making. In real-time driving, even a small delay can be dangerous or fatal. So we need a solution that can handle multiple perception tasks together without losing speed or accuracy. HydraNet was introduced to solve this it uses a single shared encoder to process the image once, and then branches into separate decoders for each task, saving time and improving efficiency.

Slide 2 -





Encoder – MobilenetV2



Decoder – Refinenet

Short explanantion while presenting

The model takes a regular RGB image as input and produces two different outputs: a semantic segmentation map (which shows what each pixel belongs to like road, car, person) and a depth map. It does this all in one go, using a shared encoder-decoder architecture.

Encoder function:
The encoder is like the brain of the model that tries to understand what's in the image. It takes the original image and passes it through a series of filters (convolutional layers) that gradually reduce its size but increase its meaning by capturing patterns like edges, textures, shapes, and objects. Think of it like compressing the image into a smaller version that still holds all the important information. In this project, the encoder is MobileNetV2, which is fast and efficient, and already trained to understand basic visual features.

Decoder function:
The decoder takes this compressed, meaningful information and rebuilds it into something useful. But instead of reconstructing the original image, it reconstructs things like Segmentation and Depth. To do this, the decoder uses techniques like **upsampling** and **skip connections** to recover details and sharpen the predictions. In this project, that's handled by Lightweight RefineNet, which is good at combining low-level and high-level features.

Upsampling:
When the image goes through the encoder, its size is reduced (e.g., from 640×480 to 40×30). But we want to predict something for each pixel (e.g., in segmentation or depth). So the decoder needs to bring it back up to the original size. Similar to what we learned in class for image sticthing using bilinear interpolation etc.

Skip Connections:
As the encoder compresses the image, it loses fine spatial details like edges, boundaries, and textures. These details are stored in earlier encoder layers (the shallow ones). Skip connections "skip" over to copy those fine details into the decoder, so that it can reconstruct sharper, more accurate outputs.
The RefineNet in above architecture receives multi-resolution features (details from various encoder layers) from the encoder via skip connections (3 blocks above encoder decoder). This helps improve both segmentation and depth quality, especially around object boundaries.

MobilenetV2 (some basic information)
Pretrained on ImageNet, so it already captures general visual patterns like edges, corners, and textures. Built using depthwise separable convolutions, which significantly reduce the number of parameters and computations without sacrificing much accuracy. Provides intermediate feature maps from different layers, which are used in skip connections to the decoder. These feature maps come from different resolutions (think of it as scale space in SIFT), capturing both local and global context. The encoder processes the RGB image once, and the same feature representation is used by both tasks (segmentation and depth), ensuring computational efficiency.

Refinenet (Some basic Information)
Uses **CRP** blocks (Chained Residual Pooling) to capture multi-scale context without heavy computation. Combines low-level features (from early encoder layers) and high-level features (from deeper layers) through skip connections. This decoder is shared by both tasks up to a point, and then it splits into two distinct heads.

CRP:
When we do tasks like segmentation or depth estimation, the model needs to understand both local details (edges, corners, object boundaries), global context (like this whole area is a road and not just an edge) But increasing the model's field of view (how much of the image it sees at once) usually means reducing resolution which we don’t want (basically we need to know that this edge is actually a part of the road). CRP captures a large context without downsampling the image using MAX Pooling

After Decoder, How are the two heads created? Very basic
Seg Head: Takes the refined feature map. Applies a few more convolutional layers. Outputs a per-pixel class probability for each semantic class(similar to assignment 1). Each pixel gets assigned a specific class. This project only used 6 classes
Depth Head: Takes the same shared feature map. Uses 1×1 convolutions. Outputs a single-channel image where each pixel is a depth value.

Slide 3 – Project Idea

Extend the above architecture to add a third task(Object Detection). We make use of YOLOv8 and SSD architecture to add a third object detection head
You already have this image in your slides. Highlight our addition.

YoloV8 archiutecture + How we connected YOLO to the above architecture – Yi

Slide 4 – YOLOv8 architecture

You already have the architecture in the slides. Just a simple explanation of classic YOLO about anchor box, grid cells, uses predefined anchor box uisng K mean and then trains offset values. Unlike YOLOv5, it isan anchor free based detector where it directly predicts the center of an object instead of the offset from a known anchor box It also uses decoupled head (heads of a head), where each head focuses only on respective task Bbox generation and Classification. Here mainly explain that it uses three feature scale representation which goes in the YOLO detection head (Because we are making use of that in connection of this head)

Slide 5– How did we connect yolov8 to the hydranet for training the object detection head?

Diagram explanation -
- As in the previous slide, YOLO uses a detection head that operates on three different feature scales, allowing it to detect small, medium, and large objects. We make use of the same idea to add the YOLO detection head to our HydraNet.
- Since we are working with a pretrained encoder-decoder architecture (HydraNet), we didn't need YOLOv8's own backbone and neck (which usually extract features).
So, we removed the YOLO backbone and neck from the architecture.
- YOLOv8 detection head needs three feature maps from different resolution levels for detection.
In HydraNet, we selected three high-level feature maps (L3, L5, and L7) from the decoder (RefineNet). These layers represent progressively smaller and more abstract visual features ideal for multi-scale object detection.
- To make these RefineNet feature maps compatible with the YOLO detection head
Each feature map was passed through a convolution layer to reduce the number of channels.
Then a max pooling layer was applied to slightly reduce spatial resolution and increase receptive field.
This step ensures the shape and content of the feature maps are aligned with what the YOLO detection head expects.
- Each processed feature map was passed into the YOLOv8 detection head, which consists of two parallel branches:
One branch for bounding box prediction
One branch for class prediction
Each branch predicts its part, and then the outputs are concatenated and reshaped.
After this:
The predictions from all three feature scales are also concatenated together.
Then again, the full output is split into two parts:
One for bounding box coordinates

One for object class scores

Training:

1. Freeze HydraNet
We used a pretrained HydraNet for segmentation and depth.
To keep those parts unchanged, we froze all of HydraNet's weights.
Only the new YOLO detection head was trained.

2. We used the standard YOLOv8 Loss Function, with slight modifications to match our input/output format:
Bounding Box Loss:
CIoU (Complete IoU Loss) for better box alignment
Distribution Focal Loss (DFL) to refine box edges
Classification Loss:
Binary Cross Entropy (BCE) for class prediction

SSD archiutecture + How we connected SSD to the above architecture – Yuhan

Slide 6 – SSD Architecture

- Brief explanation of how SSD is different from YOLO.

Slide 7 - How did we connect yolov8 to the hydranet for training the object detection head?

Diagram Explanation:
As in previous slide Unlike YOLO, SSD doesn't use a shared detection head over multiple feature maps. Instead, it processes each feature map separately, and it typically works with 2 to 6 feature maps for object detection. In our case, we used just two feature maps from HydraNet.

- YOLOv8 is anchor-free and works best with three standardized multi-scale features.
SSD is anchor-based, and has a different structure — it applies separate convolution layers on each selected feature map to make predictions at multiple scales.
So instead of copying YOLO's three-layer detection setup, we used SSD's typical structure:
Pick a few useful feature maps from different levels.
Run separate detection heads on them.

- We used two layers from HydraNet:
L4 from MobileNetV2 (Encoder): Captures low-level features like edges and textures.
Good for detecting small objects.
L7 from RefineNet (Decoder): Captures high-level abstract features like shapes and context.
Better for larger objects and general object recognition.
We used only two feature maps (instead of more) because:
HydraNet wasn't originally designed for detection, so we had fewer clean multi-scale layers.
The goal was to keep the model lightweight and efficient.

- Each feature map was passed into its own set of SSD detection layers, which included:
Convolution layers to predict:
Bounding box coordinates
Object class probabilities
These predictions were made at multiple default anchor boxes (with different sizes and aspect ratios) for each feature map location.
In total, about 8732 anchor boxes were generated (same as in the original SSD paper), covering various sizes across the image.

Training:
1. Step 1: Freeze HydraNet
Just like with YOLO, we froze the encoder and decoder weights of HydraNet.
We only trained the custom SSD detection head.
This way, segmentation and depth outputs remained the same, and only detection was learned.

2. SSD uses a loss function called MultiBox Loss, which includes two parts:
Localization Loss (Smooth L1 Loss):
Measures how close the predicted box is to the ground truth box.
Confidence Loss (Cross Entropy Loss):
Measures whether the model correctly predicts the object's class.
The loss is computed for all anchor boxes, and matched with ground truth boxes using IoU (Intersection over Union).