DS677 Group 9 Report

Andy Lanchipa, Daniel Meltzer, Jordan Davis

Github/Demo: https://github.com/jgd45/DS677_Project
Video Presentation:https://www.youtube.com/watch?v=ZSwIJrvrkrg

A. YOLOv10 Model Description

The YOLOv10 model, short for You Only Look Once, represents one of the most advanced architectures in real-time object detection. Designed to operate efficiently and accurately, YOLOv10 builds upon the foundational principles of the YOLO family while introducing key improvements in architectural design and post processing. The variety of models YOLO supports, fits the needs of users in an array of different computing environments.

The YOLOv10 architecture comes in multiple sizes and complexities; YOLO-N (Nano), YOLO-S (Small), YOLO-M (Medium), YOLO-B (Balanced), YOLO-L (Large), and YOLO-X (X-Large), each designed to accommodate varying computational resources and application requirements. For our implementation, we selected YOLO-M for object detection with classification tasks, and YOLO-B for standalone classification. This combination provided a balanced trade off between inference speed and detection accuracy on local machines.

At the heart of YOLOv10's adaptability is the scaling module located at the top of the architecture definition. This module allows users to control both the depth and width of the model via multipliers applied to a base design. The depth multiplier determines how many times certain network blocks are repeated, effectively controlling how deep the model becomes. Deeper models are capable of capturing more complex, hierarchical features, albeit at increased computational cost. The width multiplier, on the other hand, adjusts the number of channels in each convolutional layer. Wider models can process more diverse information per layer, improving the model's ability to distinguish fine-grained features in images. In our setup, we increased the width to 1024 channels from the default 728 to enhance the model's representational capacity and improve performance on a feature-rich dataset.

The backbone of YOLOv10 serves as the feature extraction pipeline. This component processes the raw input image through a series of convolutional layers and custom modules to extract meaningful feature maps. Each layer is defined in the YAML configuration using the format [from, repeats, module, args], which specifies its position, depth, type, and parameters. The backbone begins with a pair of 3×3 convolutions, each using a stride of 2 to downsample the image and increase channel depth. This early

downsampling reduces spatial resolution while retaining essential details, setting the stage for more abstract feature learning.

Following these initial layers, the architecture introduces C2f blocks, short for Cross Stage Partial fused. These modules are lighter versions of traditional CSP bottlenecks and help maintain efficient gradient flow while encouraging feature reuse, ultimately reducing the number of parameters without sacrificing learning capacity. One such C2f block consists of five internal layers and plays a key role in extracting mid-level features from the input. As the image continues to pass through deeper layers, additional convolutions further reduce its spatial resolution, bringing it to one-eighth of the original size. Another C2f block, wider and deeper, is then applied to capture more nuanced mid-level patterns.

To prevent overfitting, especially when training on reduced datasets for speed, we included a Dropout2D layer with a 10% drop rate. This regularization technique randomly zeroes out entire feature maps during training, forcing the model to generalize and not rely too heavily on specific channels.

As the architecture progresses, it introduces SCDown modules, or Selective Context Downsampling. Unlike simple strided convolutions, SCDown layers incorporate contextual cues during spatial resolution reduction, preserving more semantic information, especially important for detecting small or partially occluded objects. These are followed by deeper C2f modules that continue to build on higher-level representations.

Toward the deepest layers of the backbone, we employ C2fCIB blocks, Context-Injected Bottlenecks, which reintroduce global context into the compressed feature maps. These are critical for restoring semantic richness lost during aggressive downsampling. Next, Position Sensitive Attention (PSA) modules are used to refine both spatial and channel-wise focus, allowing the model to prioritize the most relevant regions in the feature maps. The backbone also integrates an SPPF (Spatial Pyramid Pooling Fast) layer, which captures multi-scale features using multiple receptive field sizes. A second PSA layer further sharpens attention, followed by a CBAM (Convolutional Block Attention Module) that applies sequential channel and spatial attention. CBAM helps the model emphasize important features and improves the handling of occluded objects.

Once the backbone has distilled the input into refined, multi-scale feature maps, the head of the architecture takes over. This section is responsible for assembling these features into the final object detection predictions. It begins by upsampling high-resolution features and concatenating them with mid-level feature maps. These fused features are then passed through a C2f block to improve medium-scale object

detection. A similar process is repeated by upsampling mid-level features and combining them with low-resolution features to better detect small objects.

The head then reverses the flow by downsampling fused low-resolution features and combining them with mid-resolution features. This bottom-up path helps reinforce context and semantic consistency between scales. A final downsampling step merges these enriched features with high-level representations from earlier in the backbone, ensuring that even large-scale object detection benefits from context-aware refinement.

The last step is the v10Detect layer, which takes three refined feature maps corresponding to small (P3/8), medium (P4/16), and large (P5/32) object scales and produces bounding box predictions. Each bounding box includes:
x,y: coordinates of the box center, w,h: width and height of the box , an objectness score, indicating the confidence that an object is present, and class probabilities for identifying the object category. During training, the model minimizes a composite loss function: Localization loss, using metrics like CIoU, adjusts bounding box coordinates, Objectness loss evaluates confidence prediction, and Classification loss helps the model accurately label objects. These predictions are matched to ground truth boxes using Intersection over Union (IoU), and the network updates its weights accordingly via backpropagation.

Once architectural changes are made in the YAML file and saved within the Ultralytics module, the new model configuration is automatically applied when the model is instantiated. This streamlined integration makes it easy to experiment with different architectural choices and tailor the model to specific datasets or hardware environments.

```yaml

# Parameters
# ------------------------------------------------------------
#  YOLO-v10 m  •  2 classes  •  CBAM after backbone
# ------------------------------------------------------------
#nc: 2
scales:
  m: [0.67, 0.75, 1024]        # depth, width, max_ch

backbone:
  # [from, repeats, module, args]
  - [-1, 1, Conv,        [64,   3, 2]]         # 0  P1/2
  - [-1, 1, Conv,        [128,  3, 2]]         # 1  P2/4
  - [-1, 5, C2f,         [128, True]]          # 2
  - [-1, 1, Conv,        [256,  3, 2]]         # 3  P3/8
  - [-1, 8, C2f,         [256, True]]          # 4
  - [-1, 1, nn.Dropout2d,[0.1]]                # 5
  - [-1, 1, SCDown,      [512,  3, 2]]         # 6  P4/16
  - [-1, 8, C2f,         [512, True]]          # 7
  - [-1, 1, SCDown,      [1024, 3, 2]]         # 8  P5/32
  - [-1, 5, C2fCIB,      [1024, True]]         # 9
  - [-1, 1, PSA,         [1024]]               # 10
  - [-1, 1, SPPF,        [1024, 5]]            # 11
  - [-1, 1, PSA,         [1024]]               # 12
  # ---------- NEW: attention block ----------
  # 1024×0.75 = 768 real channels, so pass 768
  - [-1, 1, CBAM,        [768]]                # 13
  # ------------------------------------------
```

```yaml
head:
  # P5 → P4
  - [-1, 1, nn.Upsample, [None, 2, "nearest"]] # 14
  - [[-1,  6], 1, Concat, [1]]                 # 15  (768 + 384)  P4
  - [-1, 3, C2f,         [512]]                # 16

  # P4 → P3
  - [-1, 1, nn.Upsample, [None, 2, "nearest"]] # 17
  - [[-1, 4], 1, Concat, [1]]                  # 18  (512 + 192)  P3
  - [-1, 3, C2f,         [256]]                # 19

  # small head → medium
  - [-1, 1, Conv,        [256, 3, 2]]          # 20
  - [[-1, 16], 1, Concat, [1]]                 # 21
  - [-1, 3, C2fCIB,      [512, True]]          # 22  (P4/16)

  # medium → large
  - [-1, 1, SCDown,      [512, 3, 2]]          # 23
  - [[-1, 10], 1, Concat, [1]]                 # 24
  - [-1, 3, C2fCIB,      [1024, True]]         # 25  (P5/32)

  # Detect heads
  - [[19, 22, 25], 1, v10Detect, [nc]]         # 26
```

B. Innovations of YOLOv10

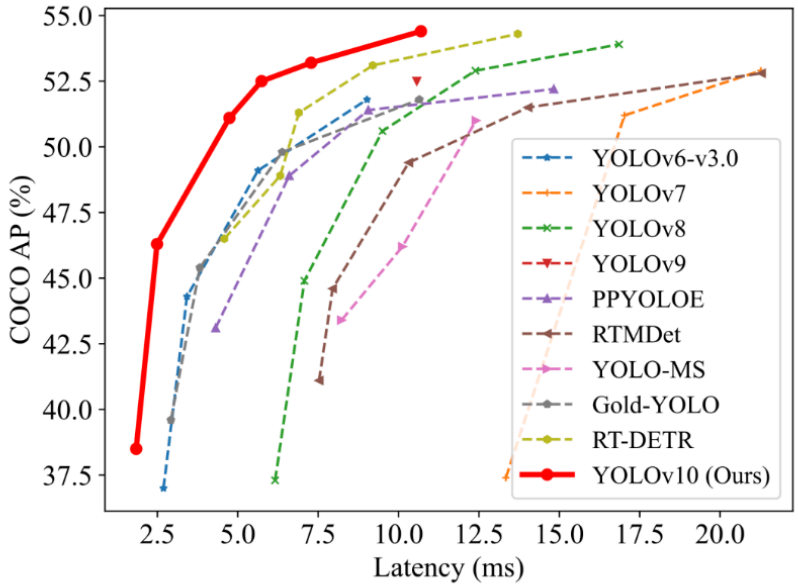
As previously mentioned, when first introduced, YOLO (You Only Look Once) marked a significant innovation in the field of object detection. Models preceding YOLO commonly relied on traditional classifiers applied across an entire image. One such methodology was the *Deformable Parts Model* (DPM), which employed a sliding window technique. In this approach, a classifier was applied within the window bounds as it moved systematically across the image at evenly spaced intervals (Felzenszwalb et al., 2010). Another notable method was the *Region-based Convolutional Neural Network* (R-CNN), which generated candidate bounding boxes and then applied a classifier to each proposed region. These models also required additional processing to refine bounding box coordinates. Although accurate, both DPMs and R-CNNs were complex, slow, and difficult to optimize for real-time applications (Girshick et al., 2014).

YOLO revolutionized object detection by framing the task as a single regression problem, allowing the model to simultaneously predict bounding box coordinates and class probabilities from the entire image in one unified pass (Redmon et al., 2016). The model's name is inspired by this efficiency it "only looks once" at the image, avoiding the need for sliding windows or region proposal networks. Furthermore, YOLO tends to produce fewer false positives in background regions compared to R-CNNs. This is largely due to its ability to process the entire image at once, enabling it to understand the contextual meaning of objects within their environment.

During training, YOLO learns to predict bounding box locations based on this contextual understanding and applies its object classifier to the most probable areas within the image. Each bounding box is assigned a confidence score, which reflects both the likelihood that an object is present and the precision of the bounding box's placement (Redmon et al., 2016).

Due to its architectural efficiency, the original YOLO model is capable of processing images at 45 frames per second (FPS), making it suitable for real-time detection tasks. This emphasis on speed and accuracy set the foundation for subsequent YOLO versions, which introduced increasingly lightweight and faster variants, further advancing the field of real-time object detection (Bochkovskiy et al., 2020; Wang et al., 2024).

YOLOv10 improved on previous iterations by eliminating several post-processing steps that earlier models relied upon. Most notably, it removed the need for Non-Maximum Suppression (NMS), a component traditionally used to filter overlapping bounding boxes. By replacing this step with a post-NMS distillation loss, YOLOv10 became fully end-to-end, simplifying deployment and reducing inference times (Wang et al., 2024). In addition to these improvements, YOLOv10 introduced a new backbone known as SCNet (Selective Context Network). This backbone architecture incorporates modules such as SCDown and C2fCIB, which allow the model to downsample input while preserving important contextual information. These changes helped YOLOv10 outperform previous versions in both speed and accuracy. Further enhancements include the use of Position-Sensitive Attention (PSA) layers, which guide the model to focus on the most relevant regions in the image, especially when objects are small or overlapping. The result is a model that not only performs better but does so more efficiently. YOLOv10 is offered in several sizes, ranging from Nano (N) to X-Large (X), with optimized scaling parameters to suit a wide range of computing environments, from edge devices to high-end servers (Viso.ai, 2024). Collectively, these updates make YOLOv10 one of the most streamlined and powerful real-time object detection models to date. YOLO model comparisons are visualized below.



C. Self-Contained and Easy-to-Understand Demo

These next commands are assuming you are running on a colab environment

1. **Input**
    a. The input for this model must contain images in .jpg format with corresponding labels in .txt format
        i. Label files should be in YOLO format which is <class ID> <bounding box coordinates and dimensions>
    b. The input structure which is your dataset should look be in the following format
        i. Dataset
            1. Train /
                a. Images
                b. Labels
            2. Valid /
                a. Images
                b. Labels
            3. Data.yaml
                a. This data.yaml file should be in a similar format as show below
                    i. path:<path to dataset>
                    ii. train: train/images
                    iii. val: valid/images
                    iv. nc: <# of classifiers>
                    v. names: ['bird', 'drone']
2. **Setup**
    a. This section is to download model dependencies
        i. Install dependencies
            1. !git clone https://github.com/THU-MIG/yolov10.git
            2. %cd yolov10
            3. !pip install -r requirements.txt
            4. !pip install -e .
        ii. Download model
        iii. !wget https://github.com/THU-MIG/yolov10/releases/download/v1.1/yolov10b.pt
3. **Train and evaluate model**
    a. Train model

i. By running the below code you set up your model to be ready for training

```
from ultralytics import YOLO
model = YOLO("yolov10b.pt")
results = model.train(
    data="/path/to/Dataset/data.yaml",
    imgsz=416,
    batch=8,
    epochs=100,
    device="cuda",
)
```

b. Model Evaluation
   i. Once the model has been trained, it stops and returns an object of metrics of the models performance, the most important ones include:
      1. map50
         a. Measures how accurate the predictions are
      2. Precision
         a. Measure how many predicted boxes were correct
      3. Recall
         a. Measures how many ground-truth objects were detected
      4. Box
         a. Measure box coordinates error
      5. ClS
         a. Measures classification loss, how well the model identifies classes

4. Model prediction

a. To use models to make predictions on images you find the path to the best run/trained model. After creating an instance of this model you run the .predict() function which allows you to pass the source image you want to predict

```
model = YOLO("runs/detect/train/weights/best.pt")
results = model.predict(source="/path/to/test/images", save=True)
```

b. The return for the .predict() function is a result object similar to the .train(), however this time it returns predictions in the image, below are some of the most essential items of information for this output:

   i. boxes.xyxy

1. This is a bounding box coordinates in the format of [x1,y1,x2,y2] for each detection in the input image
2. The two sets of coordinates represents the top-left and bottom-right corners of box

ii. boxes.conf
1. Confidence score for each detection

iii. boxes.cls
1. Predicted class for each bounding box

iv. plot
1. Returns the image with bounding boxes,class labels and confidence scores.

D. Applications of Yolov10

Yolov10 is mainly used for object detection but can also be used for classification. Some applications for Yolov10 would be binary detection for abnormalities in an eye image dataset as well as multiclass detection. Dataset's that can be used for this are Microsoft Common Objects in Context for multiclass detection and Eye Disease Image Dataset posted to kaggle by Ruhul Amin Shari can be used for binary detection but requires some modifications.

For Microsoft Common Objects in Context (MS COCO) this dataset is used very commonly in detection tasks as a benchmark and is multiclass detection. Images in MS COCO have boundary boxes already so that will not be part of the preprocessing.

It would be best for the task to be binary detection for the Eye Disease Image Dataset with the labels being no abnormalities or an abnormality so to begin it would be best to change the labels for that purpose. The task being detection means that each image needs to have bounding boxes to find where the object that has to be detected is. This can be done manually, through the use of another model, or something like Class Activated Maps. After any necessary preprocessing steps depending on the images themselves we can use methods like raytune and optuna to find the best hyperparameters to train the model on and then training the model itself.

While there are more specialized models for detecting eye abnormalities Yolov10 has the potential to detect abnormalities in images at a much faster rate due to it being made with the purpose of detecting things in real time.

E. Future Applications

YOLO newer versions are usually upgrades over one of its predecessors or specialize in a particular aspect of computer vision. Some may have faster processing so that objects can be detected in fast environments like driving or possibly higher accuracy.

YOLO in general will be further improved for applications in self-driving cars, self-driving drones, detecting defects in products, and various other applications.

Due to how fast some YOLO versions are in object detection it is very suitable for applications like self-driving cars and drones. YOLO is primarily used to detect objects and it can be used for detecting other cars on the road, street lights, signs, people, animals and various other things in a driving environment to detect the state of the environment and respond accordingly.

For applications like detecting defects in a product a YOLO version that is optimized for high accuracy. YOLO can also be used for tasks like intruder detection and various other types of medical imaging.

References

Bochkovskiy, A., Wang, C.-Y., & Liao, H.-Y. M. (2020). YOLOv4: Optimal Speed and Accuracy of Object Detection. arXiv preprint arXiv:2004.10934.

Felzenszwalb, P. F., Girshick, R. B., McAllester, D., & Ramanan, D. (2010). Object Detection with Discriminatively Trained Part-Based Models. IEEE Transactions on Pattern Analysis and Machine Intelligence, 32(9), 1627–1645.

Girshick, R., Donahue, J., Darrell, T., & Malik, J. (2014). Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR).

Redmon, J., Divvala, S., Girshick, R., & Farhadi, A. (2016). You Only Look Once: Unified, Real-Time Object Detection. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR).

Wang, C.-Y., Liao, H.-Y. M., Yeh, I.-H., & Chen, Y.-H. (2024). YOLOv10: Real-Time End-to-End Object Detection. arXiv preprint arXiv:2503.07465.

Viso.ai. (2024). YOLOv10 Explained: A Leap in Real-Time Object Detection. Retrieved from https://viso.ai/deep-learning/yolov10

Ruhul Amin Shari (2025) Eye Disease Image Dataset
https://www.kaggle.com/datasets/ruhulaminsharif/eye-disease-image-dataset

Microsoft Common Objects in Context https://paperswithcode.com/dataset/coco