YOLOE (Real-Time Seeing Anything)

By Roman Suksumit

What is YOLOE? Why should we use the model?

YOLOE, as mentioned in the first page of this report, is used for real-time seeing anything. Its ability to detect any object that you describe thanks to its built-in visual-language understanding is very revolutionary. It works with or without prompts, supports segmentation and detection, and remains fast enough for real-time applications (like robotics, CCTV, drones, and automation). Unlike previous YOLO models limited to fixed categories, YOLOE uses text, image, or internal vocabulary prompts, allowing detection of any object class in real-time. Built upon YOLOv10 and inspired by YOLO-World, YOLOE achieves state-of-the-art zero-shot performance, barely impacting speed or accuracy.

YOLOE's base is still the same as regular YOLO models like YOLOv8 or YOLO11. Its backbone (CSP-Darknet) extracts features from the image like edges, shapes, textures, or colors. Its neck (PAN-FPN) combines features from multiple scales so it can detect objects of any size. Lastly its detection head predicts whether something is an object, what class it is (based on class scores), and where it is (using bounding box coordinates). This is what makes YOLOE fast and efficient like original YOLO models.

Where YOLOE becomes different from earlier models is that it introduces three new modules that allow it to recognize any object, even those it wasn't trained on, or to put short, its addition of the open-vocabulary part. One of the modules is called **RepRTA** (Re-parameterizable Region-Text Alignment). This module lets YOLOE understand text prompts, essentially any word you type, uses CLIP embeddings (a model trained to link text and images) and aligns the text meanings with visual features from a given image. The "re-parameterizable" part means after training, the extra module is "folded" into the main model so there is no runtime penalties when trying to detect arbitrary text-labeled objects. Another module YOLOE uses is **SAVPE** (Semantic—Activated Visual Prompt Encoder). This module allows for image descriptions to be possible. Instead of a text prompt, you can give YOLOE an image of what you want it to find and SAVPE will turn that example image into a feature embedding and basically tells YOLOE to look for things that look like the example image. This can be quite useful when trying to detect logos, parts, or brain-specific items from a single example. The last module it uses is **LRPC** (Lazy Region-Prompt Contrast). This comes in handy if you don't give YOLOE any text or image prompts, as it still works by using its internal vocabulary trained on thousands of object categories from LVIS and Objects365. When it sees an object, it compares the image region's embedding with its large internal library of object embeddings allowing it to recognize rare or unseen objects, kind of like semantic similarity search. Additionally, YOLOE has some other extra features such as Instance Segmentation like predicting pixel-accurate masks around objects, like cutting out each detected item precisely. Not to mention the fact that none of the open-world modules introduce any inference cost. The modules mentioned are only used during training. Once trained, they are merged into the normal YOLO layers, so YOLOE runs as fast as YOLOEv8 or YOLO11.

How did I use the pre-trained YOLOE model?

Firstly, install ultralytics by running this in the command prompt

```
pip install ultralytics
```

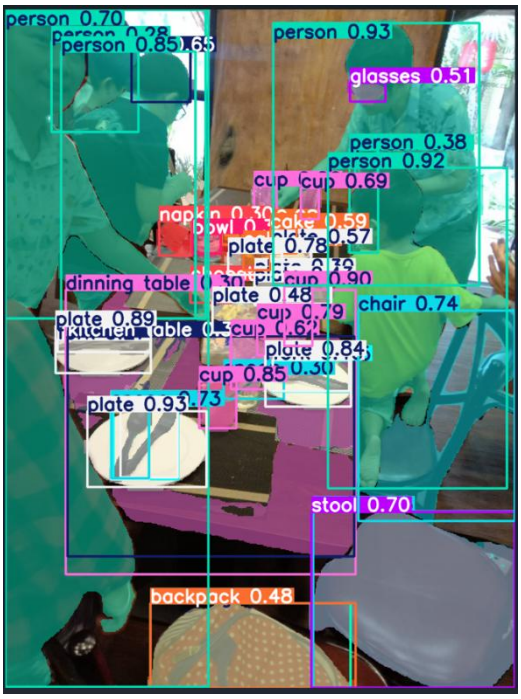
Then run this code in a notebook to run different portions of code. This portion of code should import the YOLOE model and its functions.

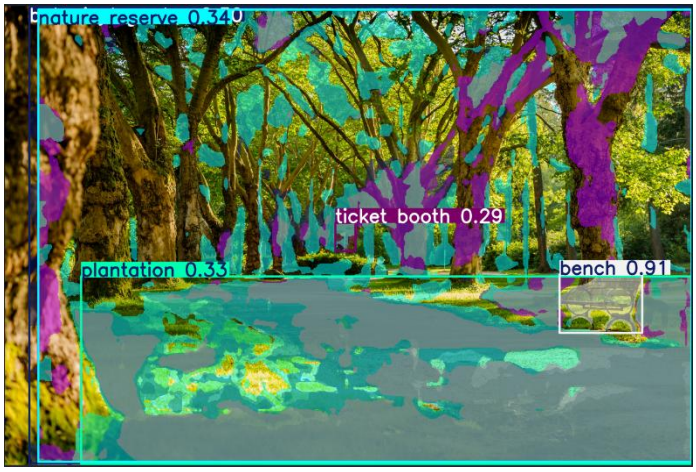
```
from ultralytics import YOLOE

model = YOLOE('yoloe-11l-seg-pf.pt')

model.info()
```

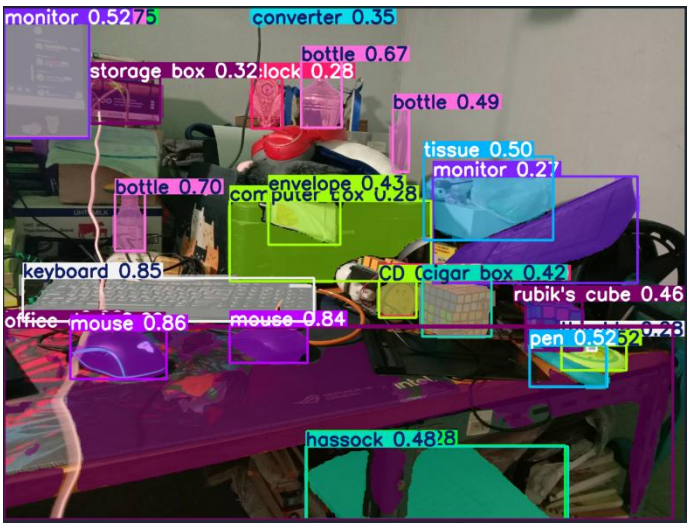
Then run this portion of code to use the model to detect anything

```
# run inference on an image

results = model(r'[insert path of image file here].jpg')

results[0].show()
```

Here are some of the test runs I did







As seen here, its predictions are sometimes incorrect but for the most part its predictions are right, and they can be very helpful in many situations

How did I train the model on my dataset?

First, import YOLOEPESegTrainer by running this line.

```
from ultralytics.models.yolo.yoloe import YOLOEPESegTrainer
```

Then I obtained a dataset called “chess pieces” from the roboflow website: https://universe.roboflow.com/yolo-ja0zo/chess-pieces-kaid5/dataset/3. This dataset contains images of chess pieces in different environments like a chess board, on a table, alone, etc.



The dataset includes 12 classes of *chess pieces: white pawn, white rook, white knight, white bishop, white queen, white king, black pawn, black rook, black knight, black bishop, black queen, black king*. All classes are denoted as acronyms e.g. white rook becomes WR, black knight becomes BKN (to remove the ambiguity between knight and king as they are denoted as KN and K respectively). It contains 304 training images, 35 testing images, and 30 validation images with some training images being orientations of others.

The images were pre-labeled using Roboflow’s built-in annotation tool, where each chess piece was manually outlined using bounding boxes to indicate its location and class. After annotation, the dataset was exported in YOLOv11-compatible format (also compatible for training with YOLOE) for training in Ultralytics.

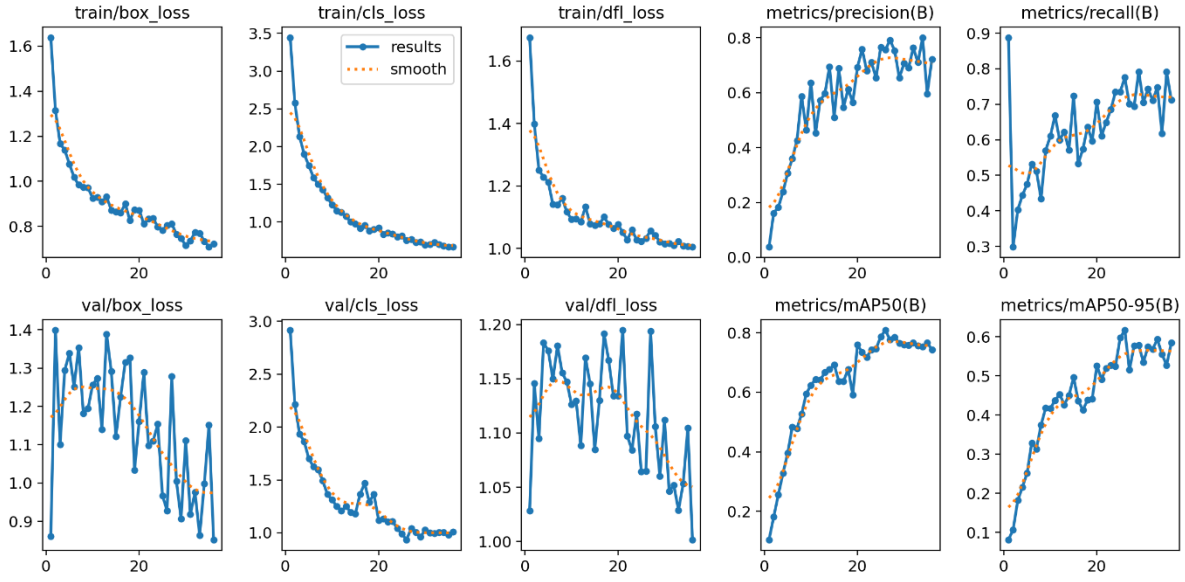
Then I used YOLOE’s built-in detection trainer called YOLOEPETrainer to train the YOLOE-11s model and ran this block of code:

```
model_train = YOLOE("yoloe-11s.yaml")
model_train.load("yoloe-11s-seg.pt")
results = model_train.train(
    data=r"[insert path to yaml file here].yaml",  # Detection dataset
    epochs=80,
    patience=10,
    trainer=YOLOEPETrainer,  # <- Important: use detection trainer
)
```

Here is a table on the parameters I set to train the model:

Parameter	Value / Description
Model	yoloe-11s
Pretrained Weights	yoloe-11s-seg.pt
Trainer	YOLOEPETrainer
Epochs	80
Patience	10 (early stopping if validation loss stops improving)
Image Size	640 × 640 pixels
Batch Size	16 (default for YOLOE small model)
Optimizer	AdamW (default YOLOE optimizer)
Learning Rate	0.01 initial, cosine decay
Device	NVIDIA RTX 3050 Laptop GPU (CUDA enabled)
Framework	Python 3.10, PyTorch 2.9.0, Ultralytics 8.3.217

Training the model took approximately 15 minutes and 35 epochs. While training, the model's performance (e.g., loss, precision, recall, and mAP AKA mean Average Precision) was tracked per epoch. Below are graphs recorded automatically by Ultralytics that display different values during the 35-ish epochs:



I then validated the model that is autosaved by Ultralytics by running this:

```
model_load = YOLOE(r"C:\Users\roman\Desktop\promethean\som ting wong\AI\runs\detect\train11\weights\best.pt")

validation = model_load.val(data=r"[insert data path here].yaml")
```

This gave me data like the mAP50 and mAP50-95 values, its Precision or P, and its Recall or R for all classes after training the model.

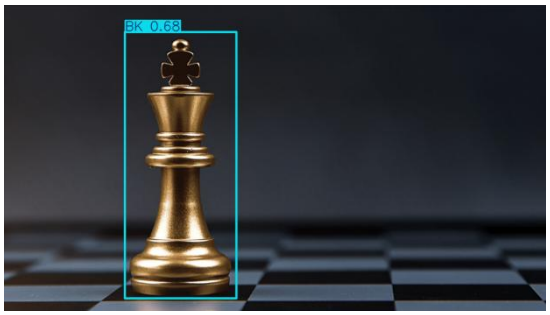
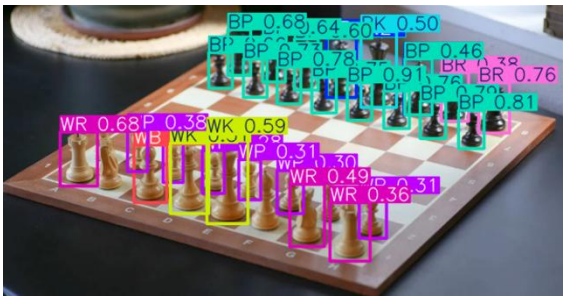
Class	Images	Instances	Box(P	R	mAP50	mAP50-95):
all	30	134	0.756	0.775	0.809	0.614
BB	6	9	1	0.623	0.803	0.659
BK	6	6	0.605	0.833	0.832	0.646
BKN	7	9	0.865	0.556	0.717	0.557
BP	7	19	0.636	0.919	0.914	0.674
BQ	6	7	0.647	0.714	0.743	0.619
BR	8	10	0.718	0.8	0.757	0.518
WB	9	12	0.894	0.702	0.818	0.584
WK	6	6	0.702	0.789	0.656	0.493
WKN	10	11	0.717	0.818	0.786	0.561
WP	7	20	0.69	1	0.981	0.666
WQ	9	10	0.679	0.8	0.864	0.708
WR	13	15	0.918	0.749	0.838	0.677

Moving on to how to use the already-trained model to predict images, I wrote this block of code to have the model detect correctly the chess pieces.

```
names = ['BB', 'BK', 'BKN', 'BP', 'BQ', 'BR', 'WB', 'WK', 'WKN', 'WP', 'WQ', 'WR']

model_load.set_classes(names, model_load.get_text_pe(names))

results = model_load.predict(r"[insert image path here].jpg")

results[0].show()
```

After this, I obtained some amount of images of chess pieces in different environments and tested them with the model I trained along with my description of how well the model did on each image. Here's how it went:

	For the most part, it got the chess pieces correctly. There are some hiccups though with the white queen being labeled as BQ instead of WQ at the bottom which could be due to lighting. There are also some pieces like the white king or white rook that didn't even get detected. To my surprise, it correctly detected all the black pieces.

	I expected the model to have detected every piece correctly in this image as it is the only thing in this image at all. So a perfect result was to be expected. I also wanted to use some easy images to see if my model could do perfectly in all of them
	I expected the model to accidentally detect white pieces as black for this image as the lighting can make it difficult for the model to tell whether a piece is white or black. To my surprise, it was able to tell if a piece was white or black. The only thing it failed to do was tell which piece was which. It denoted the black king as BB or black bishop, a white rook as WP or white pawn and it didn't detect the white queen in the middle of the board.
	Another freebie I gave the model as the king is the only piece on the board. Whether the king is black or white is still unknown so I only tested it on the piece, not the color
	Finally, I wanted to test it with a full chess board with all of the chess pieces. There are many things to look at. Firstly, it correctly detected all the black pawns, the white rooks, both colored kings, and the black queen. However, there are still some flaws like the model detecting the white queen as WK or white king instead, the model either incorrectly naming the white knight or not naming it at all. One of the white bishops weren't labeled, though the other one was correctly labeled. A black knight, 2 black bishops and a black rook were all called BP or black pawn. This could be due to occlusions, some pieces blocking others, or just bad orientation of the pieces.

To conclude this report, I would say I've learnt a lot from how image resolution can affect training time as a 640x640 image res can take up to 1 second to read which when having even only 1,000 something pictures with that high of a resolution to run through per epoch, it can take up to 20 minutes to finish a single epoch and with having to run 50 – 80 epochs to finish training, it can take up to days to finish training the model. However, having many images per epoch can have its benefits as more images can be run through and the model will be improved significantly. For the model that I ran, I only ran through 20 images per epoch to speed through the training process and I already yielded incredible results, though results could have been a lot greater if I had used a larger dataset. The model could also benefit from having some training images be occlusions such as a white bishop partially blocked by black rook and still have the model perfectly predict the position of the white bishop. But overall, this model does pretty well on images considering it only got trained for 35 something epochs.