INFO8010: Ore Finder with one-stage detection algorithms

Simon Gardier,[1] **Lei Yang**,[2] and **Camille Trinh**[3]

[1] *s.gardier@student.uliege.be (s192580)*
[2] *Lei.Yang@student.uliege.be (s201670)*
[3] *camille.trinh@student.uliege.be (s192024)*

This project aims to analyse the mechanics and to reimplement the YOLOv5 object detection model using a Minecraft ores dataset with the objective of being able to detect whether or not ores are present in images. The results are satisfactory and demonstrate that the model is able to identify the presence of the ores. However, some issues, such as a lack of confidence in the prediction and a poor calibration of the bounding boxes, were observed. These limitations suggest that there are still many improvements that can be made to obtain better results.

I. INTRODUCTION

If you have already mined in Minecraft, you may have found yourself stressing out of fear of walking past valuable ores. With OreFinder, this time is over! OreFinder implements the one-stage object detector YOLOV5(m) v6 [1], trained on a labelled dataset of 3k Minecraft gameplay screenshots.

II. RELATED WORK

Previous models like R-CNN[2] were two-stage detection algorithms. The idea was the following: in a first model, the two-stage algorithms generate proposals, then, in a second model, these proposals are classified and the associated bounding boxes are generated. R-CNN and its successors (Fast-RCNN, Faster-RCNN,...) improved the accuracy of this type of model but added a high computational cost, making them less suitable for our end product idea (real-time detection of ores in Minecraft).
YOLO[3] is a one-stage detection algorithm. These algorithms perform localization and classification in 1 forward pass, over a one-step architecture. Instead of generating proposals, YOLO directly predicts class probabilities and bounding boxes from the full image in a single pass.

III. METHODS

A first study of the state-of-the-art object detection model was made to select the appropriate architecture to implement. We selected YOLO for its good trade-off between speed and accuracy. More specifically, we selected YOLOV5 (m). Even though YOLOv5 was first released in 2020, it received multiple improvements throughout the years, going from v1.0 to v6.0, which is the one that we based our implementation on. We selected the "m" size of the model; the medium size seemed like a reasonable choice given our small dataset (1.3k images, 3k augmented dataset) and our limited compute power (NVIDIA RTX3070).
We mainly developed our implementation based on the architecture description provided by Ultralytics [4].
To collaborate, we used GitHub as a code repository and WandB for training supervision.

A. Dataset collection

For this project, we selected the "minecraft-ore" dataset for Roboflow[5]. This dataset is composed of 1.3k images, from which a dataset of 3070 images is created using +- 11° shears on the horizontal and vertical axes. The dataset is split as follows:

- Training: 2686 images (86%)
- Validation: 254 images (8%)
- Test: 130 images (4%)

1. *Data augmentation*

Additionally, we applied the following transformations to the training set to give some differences to the initial images:

- Color variation (brightness, contrast, saturation, and hue)
- Small gaussian noise

B. Feasibility study

To get an idea of the results we could produce with our implementation, we trained for 20 epochs the reference model from Utralytics (see `train_reference_model.ipynb`).

As you can see in the metrics results (IV C) and in the results examples (1), the reference model performs well on the dataset we selected for this project.

Class Name	Precision	Recall	F1 Score	mAP50
gold_ore	0.784	0.800	0.859	0.494
iron_ore	0.793	0.609	0.749	0.402
diamond_ore	0.833	0.834	0.872	0.513
redstone_ore	0.626	0.660	0.694	0.386
deepslate_iron_ore	0.797	0.809	0.854	0.444

TABLE I. Performance Metrics of the reference model for YOLOv5(m) 6.0 on the project dataset[5]

FIG. 1. Results on test set

C. Architecture

The full architecture can be seen in Figure 8 and is divided into 3 main parts: Backbone, Neck, and Head.

1. Backbone: **Extracts** the **features** of the input image through convolutional layers, producing feature maps of different sizes.

2. Neck: **Fuses** the different level mappings from the Backbone for better object detection. It provides local and global details of the images which are outputted and transferred to the Head.

3. Head: Applies **prediction** layers to each combination to get the bounding boxes (the coordinates of the center, height, and width), object class, and class confidence.

1. Backbone

The structure of the **Backbone**[6], implemented as CSPDarknet53, is composed of multiple CBS blocks (Convolution + Batch Norm + SiLu) and C3 blocks as well as one SPPF. The CBS and C3 modules are responsible for the feature extraction, where each convolution is followed by a normalization and an activation function for stable learning.

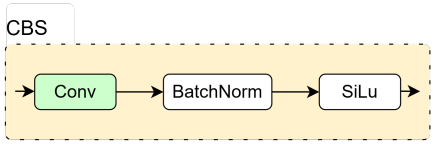


FIG. 2. CBS block architecture

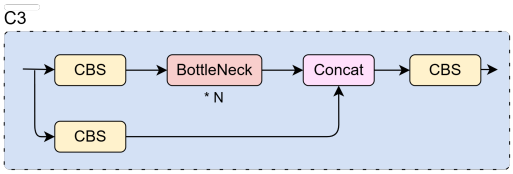


FIG. 3. C3 block architecture

Specifically, the goal of the bottleneck module, shown in Figure 4, in the C3 block is to reduce dimensionality while preserving essential features. To do so, the feature map first goes through a 1x1 CBS, which reduces the number of channels, then the second 3x3 CBS learns the spatial pattern from the compressed output of the first CBS. The bottleneck module can be repeated N times within a C3 block, the repetitions take the values [2,4,6,2], respectively, in the backbone.

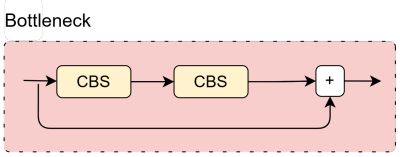


FIG. 4. Bottleneck architecture

The SPPF take as input the last feature map and apply sequentially max pooling three times in a stack then concatenate all of the max pooling output as well as the initial input and finally fuse the channels by using 1x1 CBS.

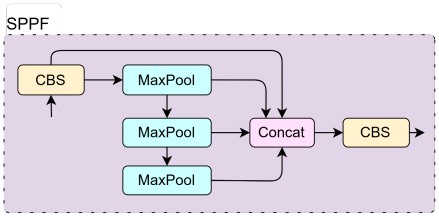


FIG. 5. SPPF block architecture

2. *Neck*

The **Neck**, implemented as a PANet, has a structure composed of CBS blocks, C3 blocks as well as upsampling and concatenation operations. A simplified version of the architecture is shown in Figure 6, where X3, X4, and X5 represent the feature maps from the backbone from the most precise (high resolution) to the most abstract (low resolution). After the fusion, the generated feature maps P3, P4, and P5 will serve as input for the detection Head.

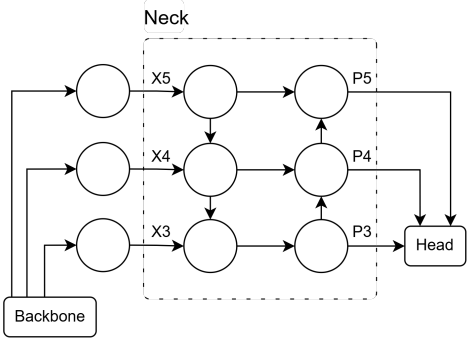


FIG. 6. Neck simplified architecture

3. *Head*

The purpose of the **Head** is to receive features from the **Neck**. Based on predefined small, medium, and large size boxes, also known as anchors, it then predicts where the object is located with the class it has among the 5 and the certainty that it is the predicted class.

D. Project implementation

The project is implemented using Python and the library PyTorch in the file `model.ipynb`. The architecture was compared with the model from ultralytics using Netron and the training was monitored using WandB. The model weights can be downloaded from Google Drive.

E. Hyperparameters and Training

Our model was trained for 100 epochs with the Adam optimizer and a Cosine annealing scheduler to control the learning rate.

1. *Optimizer*

For this problem, one of the most common optimizers was used for this project: Adam Optimizer, which does not have many parameters to define. The learning rate is important to define the value.

A high initial learning rate of 0.01 allows the model to rapidly explore a space of solutions at the start of training. Then, by using a scheduler, this rate can be progressively reduced, facilitating convergence towards a more stable minimum once the initial exploration is complete.

2. *Scheduler*

The scheduler used for this project is "**scheduler.CosineAnnealingLR**". Using the scheduler allows our model to learn quickly by exploring the possibilities and then reducing the learning rate so that it can converge towards a minimum to refine our predictions. The evolution of the learning rate according to the number of epochs is shown in Figure 7. At the beginning, the learning rate is high, but it quickly decreases to refine the predictions. At the end, the learning rate is close to zero.

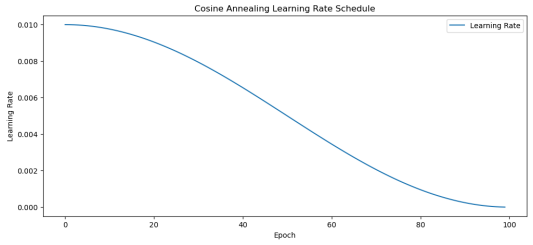


FIG. 7. The learning rate evolution

3. *Loss function*

The loss function used for the training of YOLOv5 is composed of 3 losses:

- The box regression loss: It will measure the error between the predicted box and the true box. It will compute the Complete Intersection over Union to get the value. This function will take into account the area intersection, the distance between the centers of the 2 boxes, and the ratio of the form of the boxes.

- The objectness loss: The main role of this loss is to predict if there is an object in the boxes or not. The loss used for this is the Binary Cross-Entropy.

- The classification loss: The main objective of this loss is to predict if the class predicted for the box is the correct one or not. This will be done by using the Binary Cross-Entropy.

And the sum of those 3 components will form the loss function to reduce.

F. Detection parameters

1. Strides

The "strides" define the feature map sizes at which we make predictions. The idea is to help the model detect objects of various sizes: small objects are more easily detected at high resolution, while big objects are more easily detected at low resolution.
Based on the work of Ultralytics [4], we chose the following scale size (= strides): 8, 16, and 32.

2. Anchors

YOLOv5 does not predict bounding boxes from scratch, instead, it predicts their center and an offset for a predetermined bounding box size, called an anchor. The anchors are of various sizes and shapes to detect objects of various shapes. The anchor size depends on the feature map size (i.e., depending on the stride). Based off the work of Ultralytics [4], we get the following table:

Scale	Size	Stride	Anchors
High resolution	80 x 80	8	(10x13), (16x30), (33x23)
Medium	40 x 40	16	(30x61), (62x45), (59x119)
Low resolution	20 x 20	32	(116x90), (156x198), (373x326)

TABLE II. Detection parameters combinations (strides and anchors).

IV. RESULTS

A. Training-Validation Loss

We tracked the training and validation loss during training (see Figure VI B) to monitor over-/under-fitting. Other than rare spikes, the two losses are satisfactory and do not show signs of over-/under-fitting.

B. Metrics choice

The metrics to evaluate the performance of the code are precision, recall, and F1 score. Those metrics are often used in the literature for the Yolo Model. The precision will measure the qualities of the detection. For example, if the model says there is redstone ore in the image, but this is a diamond ore. The recall will measure the quantity of objects detected in the image. For example, if there is an iron ore, but the model says no iron ore. And the F1 will be a good compromise between the recall and the precision that balances both recall and precision.. They are defined as (TP = True Positives, FP = False Positives, FN = False Negatives, TN = True Negatives):

Precision

Precision will measure the ratio of the true positives among all the positive predictions.

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

Recall

Or sensitivity will measure the ratio of the true positives to the sum of true positives and false negatives

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

F1 Score

The F1 score is a good compromise between the Recall and the precision that balances both recall and precision.

$$\text{F1 Score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

C. Evaluation metrics

The model performs correctly on the classification task after 100 epochs (see Table IV C). However, it performs

poorly on the bounding box size prediction. Indeed, as shown in Figure VI B, the model correctly predicts the position of the bounding box and the associated class, but **not the bounding box dimensions**. At this stage, it is not clear if the issue is simply a problem of decoding at inference or if the issue comes from the training itself.

Class Name	Precision	Recall	F1 Score
gold_ore	0.857	0.837	0.847
iron_ore	0.800	0.545	0.649
diamond_ore	0.725	0.806	0.763
redstone_ore	0.939	0.756	0.838
deepslate_iron_ore	0.744	0.763	0.753

TABLE III. Performance Metrics of our YOLOv5(m) 6.0 model on Minecraft Ore dataset, 100 epochs

V. DISCUSSION

A. Limitations

One of the biggest limitations of our work is the size of the initial dataset. It is about 3000 images, however, among those, there are not all unique images. Most of the images have 3 versions with a different tilt, this limits the learning process. With a larger dataset, the results would be positively impacted. One of the other major limitations of our work is the quality of the dataset we have to train the model on, which is inconsistent at times, it can be seen in Figure 10. The labels are sometimes not the right ones or the ore framework is not consistent, e.g, there is a grouping of 4 ores for 1 label or 5 labels for 3 ores. These inconsistencies have an impact on the model's performance.

B. Future work

As for hyperparameter tuning, this has not been carried out on the project, so better results can be expected with the correct choice of parameters, such as the start of the learning rate, the choice of scheduler, or the optimizer. As discussed in Section IV, our implementation does not correctly size the bounding boxes around the ores. Furthermore, the selected dataset [5] does not provide data about all the Minecraft ores (Emerald, Copper, and Lapis-Lazuli are missing from the dataset), thus one should look into a more complete dataset (e.g. mineguide/minecraft-ore-detection). With these improvements, one could implement a complete Minecraft mod for real-time ore detection.

VI. APPENDIX

A. Complete architecture

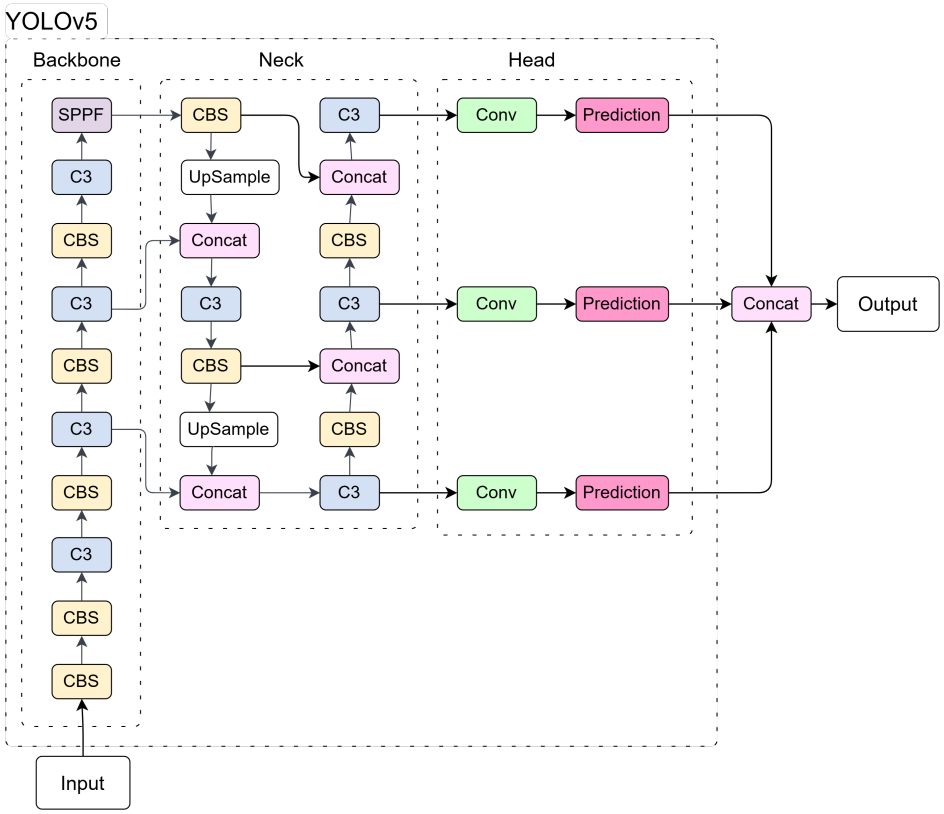
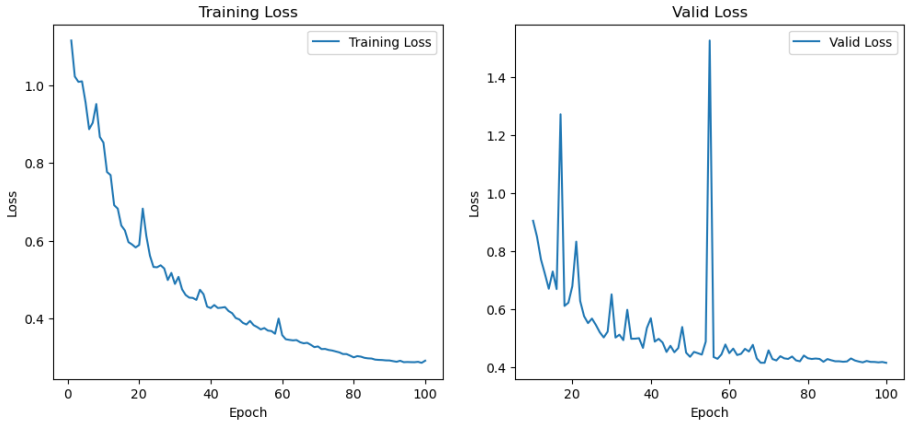


FIG. 8. Full architecture diagram

B. Training-Validation loss



C. Prediction results (ours)

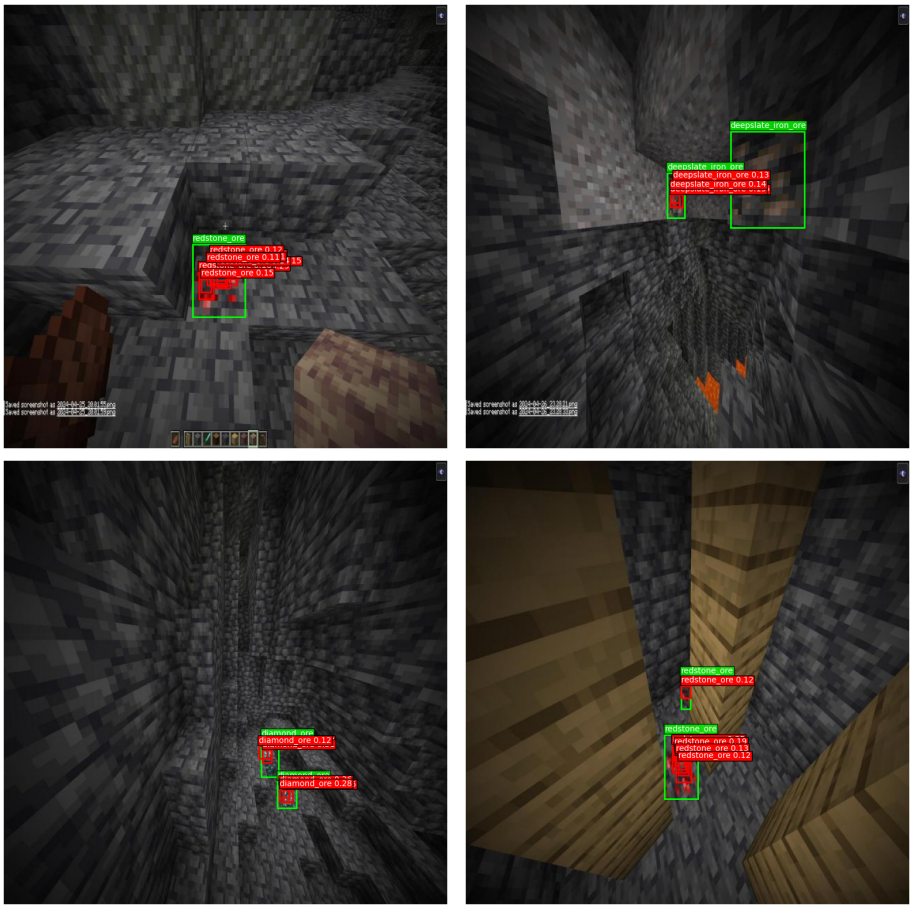


FIG. 9. Example of results of our models on the test set

D. Quality of the Dataset

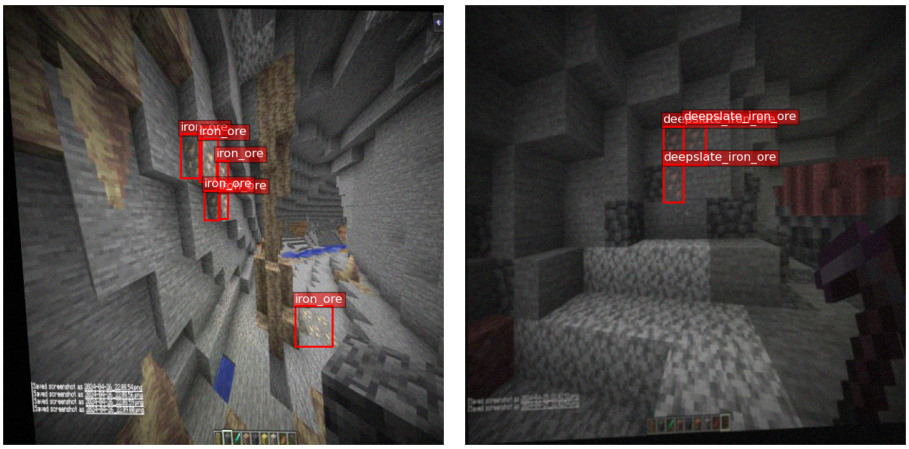


FIG. 10. Example of dataset images with their labels

VII. REFERENCES

[1] G. Jocher, "Yolov5," (2020), object detection model.

[2] R. Girshick, J. Donahue, T. Darrell, and J. Malik, *Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation*, Tech. Rep. (University of California, Berkeley, Berkeley, CA, USA, 2013) arXiv:1311.2524.

[3] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, arXiv preprint arXiv:1506.02640 (2016), 10.48550/arXiv.1506.02640.

[4] Ultralytics, "Ultralytics YOLOv5 Architecture Description," `https://docs.ultralytics.com/yolov5/tutorials/architecture_description/` (2021), accessed: 2025-05-22.

[5] oblig10, "minecraft-ore dataset," `https://universe.roboflow.com/oblig10/minecraft-ore` (2024), accessed: 2025-05-18.

[6] H. Liu, F. Sun, J. Gu, and L. Deng, Sensors **22** (2022), 10.3390/s22155817.

[7] Ultralytics, "Yolov5 architecture description," (2025), accessed: 2025-05-18.

[8] S. Tsang, "Brief review: Yolov5 for object detection," `https://sh-tsang.medium.com/brief-review-yolov5-for-object-detection-84cc6c6a0e3a` (2020), accessed: 2025-05-18.

[9] G. K. Gilbert, "Jaccard index (iou measure)," .