Global Wheat Detection Project Report

Patrick Di Rita & Ricardo Qiu & Zekun Wu

Dec. 2020

1 Introduction

1.1 Global Wheat Detection

It is an important task to detect the wheat heads in plant image when estimating the pertinent wheat traits. These traits include the head population density and head characteristics. The rapid progresses in the deep neural network have led the possibility of quantifying the wheat head from RGB high-resolution images. However, most relative image datasets are limited in terms of genotypes, geographic area and observational conditions. To tackle this limit, we used the Global Wheat Head Detection (GWHD) dataset for our study [1, 2]. As for now, the GWHD is the largest and most diverse wheat head dataset with consistent label on wheat heads detection task. It actually results from the several datasets collected from nine different institutions across seven countries and three continents. Based on the GWHD dataset, the Global Wheat Detection was a Kaggle competition which ran from May 4, 2020 to August 4, 2020 [2]. In consistent with goal of the competition, the motivation for our project is to detect wheat heads from outdoor natural images of wheat plants and we are focusing on the a generalized solution to estimate the number and size of wheat heads.

1.2 Data

The training data and testing data are close-up images of wheat fields from a diverse range of worldwide locations. Each image is in the format [*image_id*].PNG. A separate file, *train.CSV*, contains a row for each bounding box. While there can be multiple rows indexed by a single image ID, which implies multiple wheat head within that image. An example image from the dataset with bounding boxes is shown in Figure 1 and the head of the CSV file is shown in Table 1.

Figure 1: Wheat field image (image ID: b6ab77fd7) from GWHD training data with bounding boxes drawn in red [2].

1.3 Evaluation

Conventionally, evaluation of predicted bounding boxes revolves around the intersection over union (IoU) metric. The IoU of a predicted bounding boxes A and the ground truth (target) boxes B is the area of overlapping divides the area of union, which is:

$$IoU(A, B) = \frac{A \cap B}{A \cup B}.$$

Generally, we threshold the IoU score with a constant threshold to keep those bounding boxes with the highest probability meaning a predicted bounding box is considered correct at a certain threshold if its IoU is greater than the threshold. But for each bounding boxes, the IoU threshold may vary, which is in the range [0.5, 0.75] in our task. Therefore, we sweep over a range of IoU thresholdsand for each of them, which is calculated

image_id	width	height	bbox	source
b6ab77fd7	1024	1024	[834.0, 222.0, 56.0, 36.0]	usask_1
b6ab77fd7	1024	1024	[226.0, 548.0, 130.0, 58.0]	usask_1
b6ab77fd7	1024	1024	[337.0, 504.0, 74.0, 160.0]	usask_1
b6ab77fd7	1024	1024	[834.0, 95.0, 109.0, 107.0]	usask_1
b6ab77fd7	1024	1024	[26.0, 144.0, 124.0, 117.0]	usask_1

Table 1: First 5 rows of training DataFrame before any data cleaning or augmentation. Bbox data is in the form [x, y, width, height] [2].

at threshold values in the range [0.5, 0.75] with a step size of 0.05, and compute their average. [2].

The set of predictions for a single image is scored by the average of the precision values for each threshold:

$$\frac{1}{|thresholds|}\sum_{t\in thresholds}\frac{TP(t)}{TP(t)+FP(t)+FN(t)}$$

where t is a threshold value and $TP(t), FP(t)$, and $FN(t)$ are the number of true positives, false positives, and false negatives at threshold t, respectively. A predicted bounding box around a wheat head is considered as a true positive at threshold t if its IoU with the target box at that head is greater than t. A false positive means that a predicted bounding box had no corresponding target box, and a false negative means a target bounding box had no corresponding predicted box.

To submit a model for evaluation, a submission CSV file must be generated with rows in the format [*image_id, PredictionString*], where *PredictionString* contains a space delimited list of predcited bounding boxes for the image in the form *confidence_score x y width height*. Once a Kaggle notebook is created to preform model evaluation on the given subset of testing data, the notebook must be submitted to Kaggle for private evaluation on the full set of testing data. Kaggle then outputs the "public score" and "private score" for the model, which are evaluation scores on 62% and 38% of the full testing data, respectively.

2 Literature Review

Our task is essentially to detect the wheat heads as many as possible within a single image. The object detection task is to first localize the object and then identify which class it belongs to. The most common output of an object detection system is the bounding box of possible object with corresponding probability.

In the history of object detection, tons of methods were proposed to solve this task. Before the popularity of deep neural network, Felzenszwalb et al.[3], proposed to use deformable Part Models which essentially represents the target object as handcrafted SIFT features and computes response of local filter. Then they slack in part locations by maximizing scores in area around expected location of part and combine these individual part scores to score the entire object. These process is essentially convolutional neural network, we use convolutional layers to automatically extract features and be the local filter. Then we put max pooling layer and fully connected layer or convolutional layer to score the entire object. This method is proposed by Girshick et al.[4].

While, after the wide-spread of deep neural network, two methods remains state-of-the-art: 1) Regional Proposal-based methods (including RCNN[5], Fast RCNN[6], and Faster RCNN[7]) originally proposed still by Girshick et al.[5, 6]; 2) YOLO-based methods originally proposed by Redmon et al.[8, 9, 10], Although the fundamental ideas of these two methods vary a lot, there is no leading leverage that which method outperforms the other in reality. However, Regional Proposal methods are inherently computationally expensive. While YOLO uses lighter architecture, only one end-to-end network is trained, which leads to faster training and inference. In our work, we literally try both of them in our task.

2.1 YOLO

The YOLO algorithm, proposed by Joseph Redmon et al.[8], essentially, reframes object detection as a single regression problem. It takes in the image pixels straightly and outputs the bounding box coordinates and class probabilities. The YOLO uses features from the entire image to predict each bounding box and predicts all bounding boxes across all classes for an image.

The key idea of YOLO algorithm is to train a single end-to-end network to do inference as suggested by its name. To do so, the YOLO first divides the whole image into $S \times S$ grids and each grid cell contains B bounding boxes (the later

version uses anchor boxes). Then for each box, we further define a confidence score that reflects how certain this bounding box contains an object.

$$\textbf{confidence score} = Pr(Object) * IOU_{pred}^{truth}$$

$$IOU_{pred}^{truth} = \frac{\text{Area of Overlap}}{\text{Area of Union}}$$

And for each grid cell, we compute the conditional probability $Pr(Class_i|Object)$ that which class the contained object belongs to. Putting all together, during inference, the confidence score for each box with specific class equals to

$$Pr(Class_i) * IOU_{pred}^{truth} = Pr(Class_i|Object) \\ *Pr(Object) * IOU_{pred}^{truth},$$

which measures how certain a specific object within a box and how well the predicted box fits the object.

For a single bounding box, the predict result contains the location of the this bounding box and the confidence score of whether there contains an object, as well as the probability of which class the contained object belongs to. Therefore, for a single input image, the predictions are encoded as an $S \times S \times B * (5 + C)$ dimensional tensor, where B denotes the number of bounding boxes or anchor boxes, C denotes the number of object classes. Thus, we design our network to output tensor with the above dimensionality and train it with an unified loss.

During the one pass of forward propagation, the YOLO determines the probability that the cell contains a certain class with the following equation:

$$score_{c,i} = p_c \times c_i$$

After predicting the class probabilities, the unnecessary anchor boxes are get rid of by the Non-max suppression. The Non-max suppression eliminates the bounding boxes that are very close by performing the IoU (Intersection over Union) with the one having the highest class probability among them. It calculates the value of IoU for all the bounding boxes respective to the one having the highest class probability and rejects the bounding boxes whose value of the IoU is greater than a threshold. This process would be repeated untill all the bounding boxes are different.

Since the first time being proposed, the first three YOLO versions have been released in 2016, 2017 and 2018 respectively. The 4th generation of YOLO[11] has been released in April 2020 and it takes the influence of state of art BoF (bag of freebies) and several BoS (bag of specials).

2.2 Faster R-CNN

Faster R-CNN (F-RCNN) is the combination of two separate convolutional neural networks: Fast R-CNN and a Region Proposal Network (RPN). The Fast R-CNN network acts as the detection network and has access to full-image convolutional features provided by the RPN, which has been trained to generate high-quality region proposals for detection. These two networks are merged into a single Faster R-CNN network by sharing their convolutional features [7].

An RPN takes an image as input and outputs bounding boxes (region proposals) and corresponding "objectness" scores. These proposals are generated by sliding a so-called "mini-network" over a square sliding window of the last shared convolutional layer. Each sliding window is mapped to a lower-dimensional feature and fed into both a box-regression layer (reg) and a box-classification layer(cls). Multiple region proposals are generated at each sliding window location, and these proposals are parameterized relative to a corresponding number of translation-invariant reference boxes known as anchors, with each anchor centered at the current sliding window [7].

The loss function for an image is defined as

$$L(\{p_i\}, \{t_i\}) = \frac{1}{N_{cls}} \sum_i L_{cls}(p_i, p_i^*) + \lambda \frac{1}{N_{reg}} \sum_i p_i^* L_{reg}(t_i, t_i^*)$$

[7] Where i is the index of an anchor in a mini-batch, p_i is the predicted probability of anchor i being an object, p_i^* is an indicator regarding the sign of the anchor, t_i is a vector representing the 4 parameterized coordinates of the predicted bounding box, and t_i^* is the indicator for the ground-truth box corresponding to a positive anchor. $N_{cls} = 256, N_{reg} = 2400$, and $\lambda = 10$. L_{cls} is the log loss over the classes *object* and *background*, and $L_{reg} = R(t_i - t_i^*)$ where R is the smooth L_1 loss function [7]. The model is trained end-to-end using stochastic gradient descent parameterized by a momentum of 0.9, a weight decay of 0.0005, and a learning rates of 0.001 and 0.0001 for the first 60k and last 20k mini-batches on the PASCAL VOC

dataset, respectively [7].

3 Implementation

3.1 YOLO

In our task, we make use of the YOLO version 3, the most classic version, while the basic idea of implementing YOLO remains the same. The YOLO is implemented as a convolutional neural network. Just as usual, the convolutional neural network is used to automatically extract the features, and finally, another convolutional layer is added at the end of the network to map the feature map to our output.

Then we optimize the unified loss, a specially constructed squared error including 1) the confidence score loss; 2) the bounding box coordinate predictions loss; 3) classification loss:

$$
\begin{aligned}
loss = \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{i,j}^{obj} [(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2] \\
+\lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{i,j}^{obj} [(\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2] \\
+ \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{i,j}^{obj} (C_i - \hat{C}_i)^2 \\
+\lambda_{noobj} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{i,j}^{noobj} (C_i - \hat{C}_i)^2 \\
+ \sum_{i=0}^{S^2} \mathbb{1}_{i,j}^{obj} \sum_{c \in classes} (p_i(c) - \hat{p}_i(c))^2,
\end{aligned}
$$

where $\mathbb{1}_i^{obj}$ denotes if object appears in cell i and $\mathbb{1}_{i,j}^{obj}$ denotes that the jth bounding box predictor in cell i is corresponded to that prediction. There are two parameters λ_{coord} and λ_{noobj} in the loss function. They are setting to increase the loss from bounding box coordinate loss while decreasing the confidence loss, since the majority of the confidence score is zero which causes the explosion of gradient.

The final step is to optimize this squared error loss. The gradient method works in optimizing this squared error loss. Conventionally, SGD with momentum and Adam [12] are used in different YOLO versions. Still there is no evidence suggesting which method outperforms the other, we try both in our implementation.

$$
\begin{aligned}
\Delta\theta_t^{SGD} &= -\alpha m_t \\
\Delta\theta_t^{Adam} &= -\alpha m_t / \sqrt{v_t}
\end{aligned}
$$

$$
v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2,
$$

where g_t is the gradient and m_t is the momentum term.

3.1.1 Data Preprocessing

As a convention, we first split the entire training data 80/20 into a training set and validation set, respectively. Since we are dealing with training neural network, it is generally better to perform data augmentation. First, we adjust the images in their HSV domain, which is informed by the YOLO paper, by partially adjusting their Hue, Saturation, and Value. Secondly, we perform image translation for 10 percent of the training set and horizontally flip the images with a probability of 0.5. Finally, image mosaic is performed.

We resize the original image with 1024 by 1024 pixels into 640 by 640 pixels due to the computational resource we have access to. Otherwise, the RAM will be out of memory after loading the image cache. Then we have to normalize the coordinates of the bounding boxes between 0 and 1 by dividing its corresponding height or width of each image. Also note that, during inference, we can also downsample the images into 640 by 640 pixels or we can denormalize the coordinates of the bounding box back to its original resolution.

3.1.2 Model & Training

The model we used is built on top of the Darknet which is used in the YOLO v3 paper. The model has 333 layers with residual blocks to stabilize the training process. The total number of parameters is 61523734, which turns out that this network is super heavy and very hard to train from scratch. Therefore, as both informed by the original paper and the limitation of computational resource, we use the weights pretrained on COCO 2017 dataset to initialize our training. we train our network 5 epochs with different hyperparameters to select some of the hyperparameters. While there are

literally too many hyperparameters to tune in this sophisticated system, there is no guarantee that the hyperparameters we used is the most optimal. Finally, we train our network 50 epochs using SGD with momentum and Adam, respectively.

3.2 Faster R-CNN

In order to gain more context regarding the efficacy of YOLO for this particular application, we also implemented an entirely separate image classification model using a Faster R-CNN model. We then used Bayesian optimization to tune the hyperparameters of the optimizer before running a full training loop on the F-RCNN model with the resultant hyperparameters.

3.2.1 Data Preprocessing

Data cleaning and preparation was fairly straightforward for this model. Once the bounding box parameters for each datapoint were separated into individual columns in the training dataframe, we split the training data 80/20 into a training set and validation set, respectively. We then created a custom WheatDataset class which inherits from the PyTorch Dataset class, and implemented the required __getitem__ and __len__ functions. Given an image ID, the __getitem__ function reads in the image and its bounding box data, re-formats the image to be in [C,W,H] form with values between 0 and 1, and re-formats the bounding box data to the pascal_voc format (both of these augmentations are required in order to use F-RCNN on the data). It then sets up the labels (all 1s as our only class is the wheat head), preforms transforms, and returns the image, image ID, and target.

We then created PyTorch DataLoaders for our training and validation WheatDataset objects, with the training set having a flip transform with probability of .5. The F-RCNN model automatically resizes the images, so a resize transform was not necessary.

3.2.2 Model

We set up our model for transfer learning by taking an instance of TorchVision's Faster R-CNN model, which has a ResNet-50-FPN backbone and is pretrained on the COCO dataset, and replacing the model's RPN box predictor with a new FastRCNNPredictor instance.

3.2.3 Hyperparemeter Optimization

Before conducting a full training loop on the model, we preformed Bayesian optimization on the hyperparameters (learning rate, weight decay, momentum) of the optimizer (SGD with momentum). The Bayesian optimization loop was implemented using the Adaptive Experimentation (Ax) library. After setting the parameters to be optimized and search space over which the Bayesian optimizer can select values from, we optimized the parameters in an Ax Managed Loop by maximizing the validation accuracy as a function of the parameters for 20 trials. For the arm generation strategy, we set the first 5 trials to use a SOBOL random sampler on the search space (which ensures the randomly selected parameters are sufficiently far apart in each trial), while the subsequent 15 trials used a Gaussian Process model with the Expected Improvement acquisition function. A single trial of the Bayesian optimization loop is as follows:

1. Generate arm using generation strategy
2. Train instance of our F-RCNN model on training set for 3 epochs using SGD with hyperparameters from the current arm
3. Evaluate trained model on validation set by calculating total IoU precision for each validation image and averaging
4. Add the arm and resulting precision to the Bayesian optimization dataset and condition the prior on the available data

After repeating this process for 20 trials, we select the best arm from the experiment to obtain our optimized hyperparameters:

- Learning Rate: 0.001
- Weight Decay: $2.5429 * 10^{-6}$
- Momentum: 0.9

4 Experimental Results

4.1 YOLO v3

4.1.1 Training

We train our YOLO network on a Nvidia Tesla T4 (15079MB). Training 50 epochs with SGD or Adam takes roughly the same amount of time, approximately 2 hours. Due to the limitation of our GPU memory, we use the batch size of 16, otherwise, the GPU will be out of memory. The training results is as shown in the Appendix in figure 6,7,8. As is shown from the result, the classification converges to 0 in all the scenarios. The SGD with momentum works slightly better than the Adam optimizer with selected hyperparameters in terms of training. While the Adam works even slightly better during validation in terms of objectness loss, and converge to roughly the same point as SGD with momentum in terms of box loss. Another fact is that SGD with momentum converges slightly faster than Adam.

4.1.2 Inference

During inference, we first threshold the bounding boxes with lower probability. Then the Non-maximum Suppression is applied to further filter out other candidates, except for the most possible bounding boxes. Also note that, we need to denormalize the coordinates of the bounding boxes back to its original resolution.

The provided test dataset contains 10 images, we do inference on these 10 images and submit the score to kaggle. The obtained results used the most optimal model is 0.6575 for private score and 0.7208 for public. The inference process is super fast taking 0.032s for a single image. Some sample detection can be viewed in the Appendix in figure 2,3,4,5.

4.2 Faster R-CNN

4.2.1 Training

After obtaining the optimized hyperparameters, we used them to train a new instance of our F-RCNN model on the combined training and validation sets for 30 epochs. Training was conducted using the standard PyTorch training loop, and took about 3 hours on a Paperspace Gradient cloud instance with a Nvidia Quadro P5000 GPU.

4.2.2 Inference

We performed inference on the test set by first defining a TestDataset class, which works similarly to the WheatDatset class but only shapes the image into the required format and returns the image and image ID. We created an instance of this class on the test data, passed it into a DataLoader, and iterated over the contents of the DataLoader. We called the trained model (in evaluation mode) on each image, which returned a dictionary containing the keys "boxes," "scores," and "labels," and discarded all predicted bounding boxes that had a corresponding score less than a threshold of 0.5. These bounding boxes were then reformatted into their original x, y, w, h format. We then loaded the predicted bounding boxes and their scores for each image into a DataFrame in the format specified by the Kaggle competition, saved the DataFrame as a CSV, and submitted it to Kaggle for evaluation on the private test set.

We received private and public scores of 0.5721 and 0.6687, respectively, which put us at about halfway up both leaderboards. The winning scores were 0.6897 for private and 0.7746 for public, which shows just how difficult of a detection task this was. Our score for this model could have been improved by performing much more in-depth data augmentation such as generating synthetic training data through mosaic tiling and implementing other techniques such as pseudo-labeling.

5 Conclusion

In this project, we perform object detection to detect wheat head, which is of vital importance to estimate the pertinent wheat traits. Two state-of-the-art techniques are performed to achieve our goals with different optimization methods. The experimental result turns out that, as promised, the training as well as the inference of YOLO is way faster than those of the Faster R-CNN. This enables more potential applications of wheat head detection in real time, for example using robotics to detect wheat head. Because YOLO simply reframes object detection as a single end-to-end network without regional proposal and separate classifiers

that the R-CNN-based methods have; while during inference, the cost of computing non-maximum suppression is negligible. And the training process of YOLO still saves a lot of computation making it easier to be deployed.

We tested out two very powerful optimizer in our project. As is described in the experimental results section, the SGD with momentum with selected hyperparameters works slightly better than the Adam in our project in terms of the speed of convergence and the convergence rate of part of the loss function. While the Adam also shows good results in the validation process. Again, there are a lot of area that this project cannot be covered due to the limitation of time and computational resources we can have access to, for example tuning the batch size, trying more sophisticated hyperparameters selection methods.

In terms of the final inference precision, while YOLO outperforms R-CNN in our training, the original YOLO paper still shows that the Faster R-CNN outperforms YOLO in many dataset regarding precision. The reason might be the hyperparameters we choose in training Faster R-CNN and the number of epochs we trained might not converge.

References

[1] F. Fourati, W. Souidene, and R. Attia, "An original framework for wheat head detection using deep, semi-supervised and ensemble learning within global wheat head detection (gwhd) dataset," *arXiv preprint arXiv:2009.11977*, 2020.

[2] E. David, S. Madec, P. Sadeghi-Tehran, H. Aasen, B. Zheng, S. Liu, N. Kirchgessner, G. Ishikawa, K. Nagasawa, M. A. Badhon, C. Pozniak, B. de Solan, A. Hund, S. C. Chapman, F. Baret, I. Stavness, and W. Guo, "Global wheat head detection (gwhd) dataset: a large and diverse dataset of high resolution rgb labelled images to develop and benchmark wheat head detection methods," 2020.

[3] P. F. Felzenszwalb, R. B. Girshick, D. McAllester, and D. Ramanan, "Object detection with discriminatively trained part-based models," *IEEE transactions on pattern analysis and machine intelligence*, vol. 32, no. 9, pp. 1627–1645, 2009.

[4] R. Girshick, F. Iandola, T. Darrell, and J. Malik, "Deformable part models are convolutional neural networks," in *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pp. 437–446, 2015.

[5] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," 2014.

[6] R. Girshick, "Fast r-cnn," 2015.

[7] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," 2016.

[8] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 779–788, 2016.

[9] J. Redmon and A. Farhadi, "Yolo9000: Better, faster, stronger," 2016.

[10] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," 2018.

[11] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao, "Yolov4: Optimal speed and accuracy of object detection," 2020.

[12] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2017.

6 Appendix

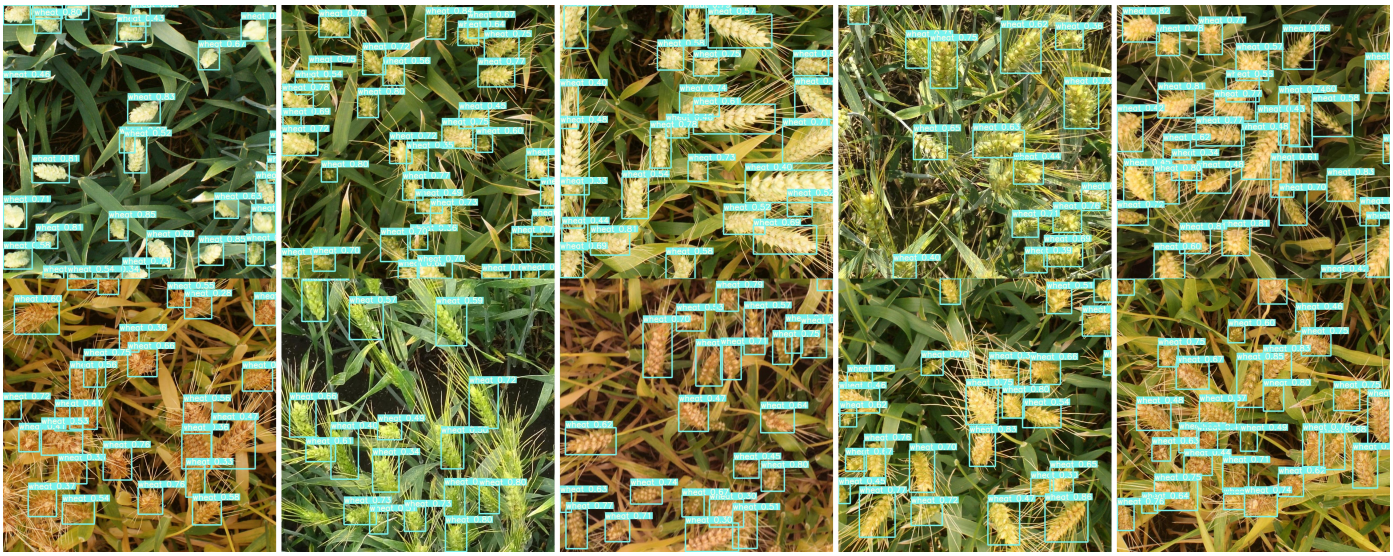

Figure 2: The inference results trained using Adam optimizer with selected hyperparameters with confidence score greater than 0.25

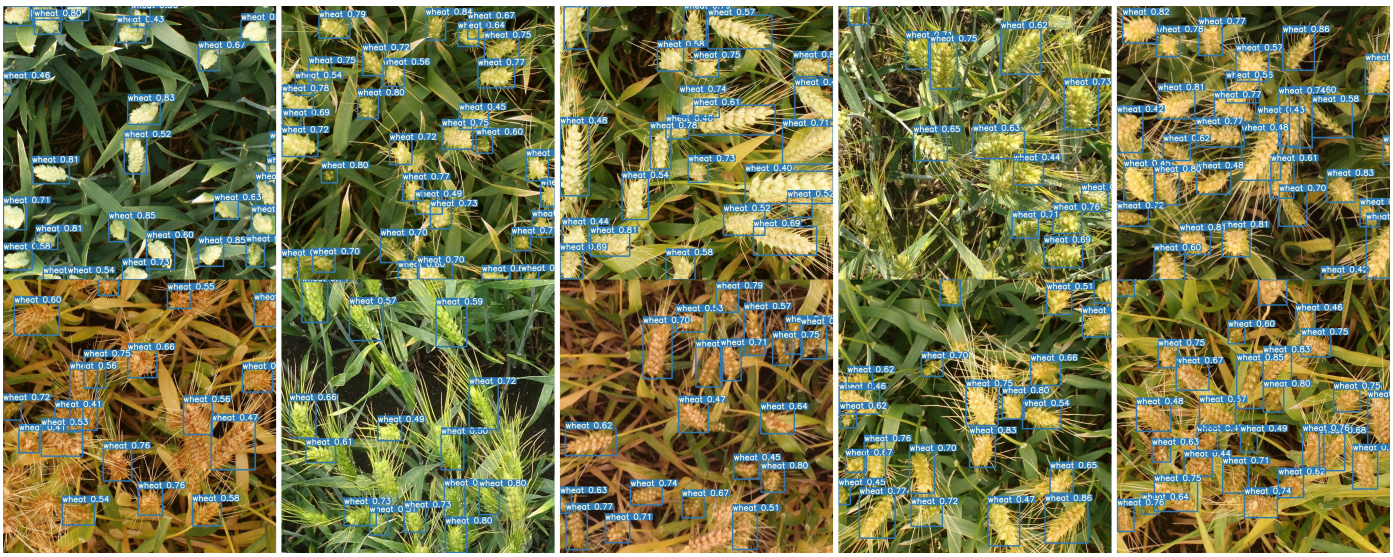

Figure 3: The inference results trained using Adam optimizer with selected hyperparameters with confidence score greater than 0.4

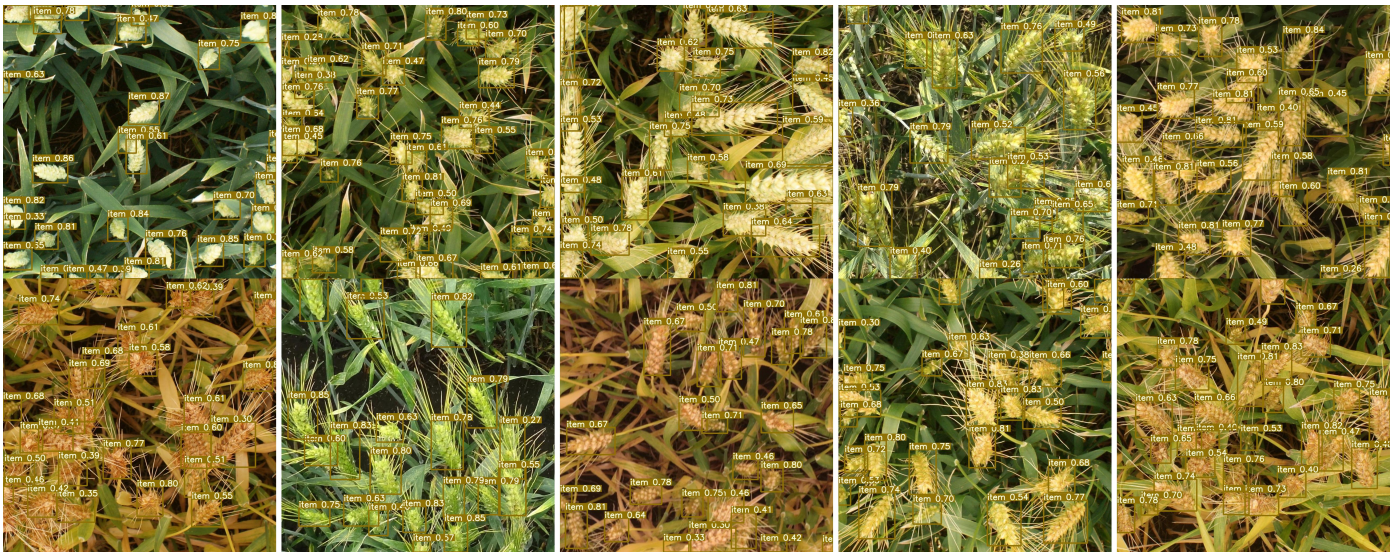

Figure 4: The inference results trained using SGD with momentum with selected hyperparameters with confidence score greater than 0.25

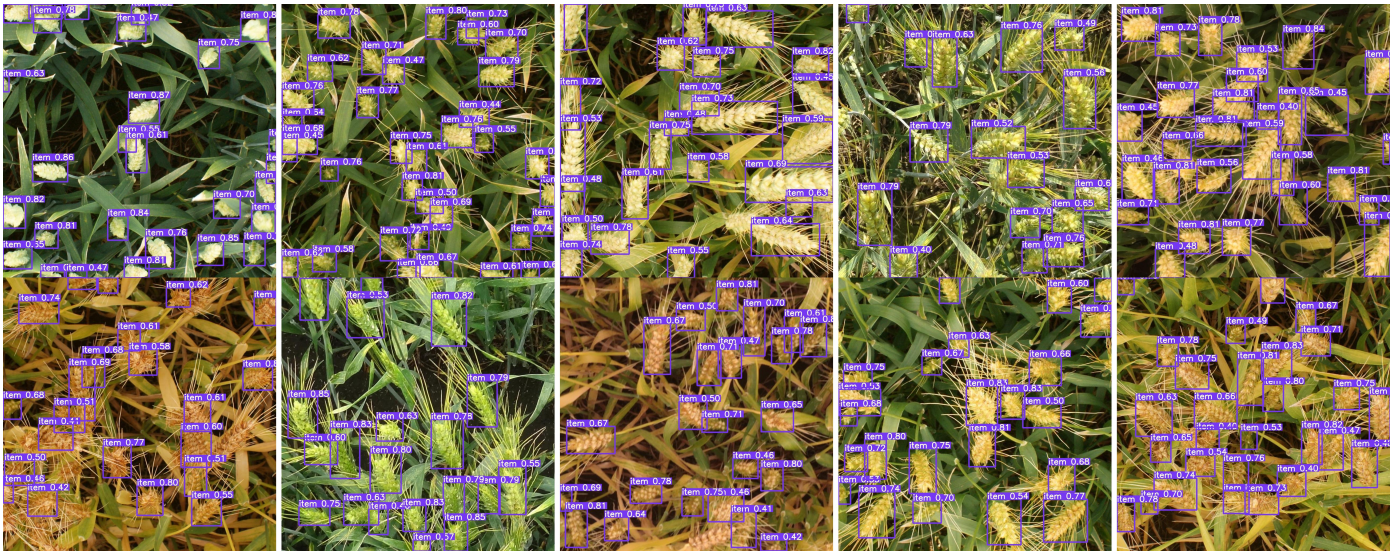

Figure 5: The inference results trained using SGD with momentum with selected hyperparameters with confidence score greater than 0.4

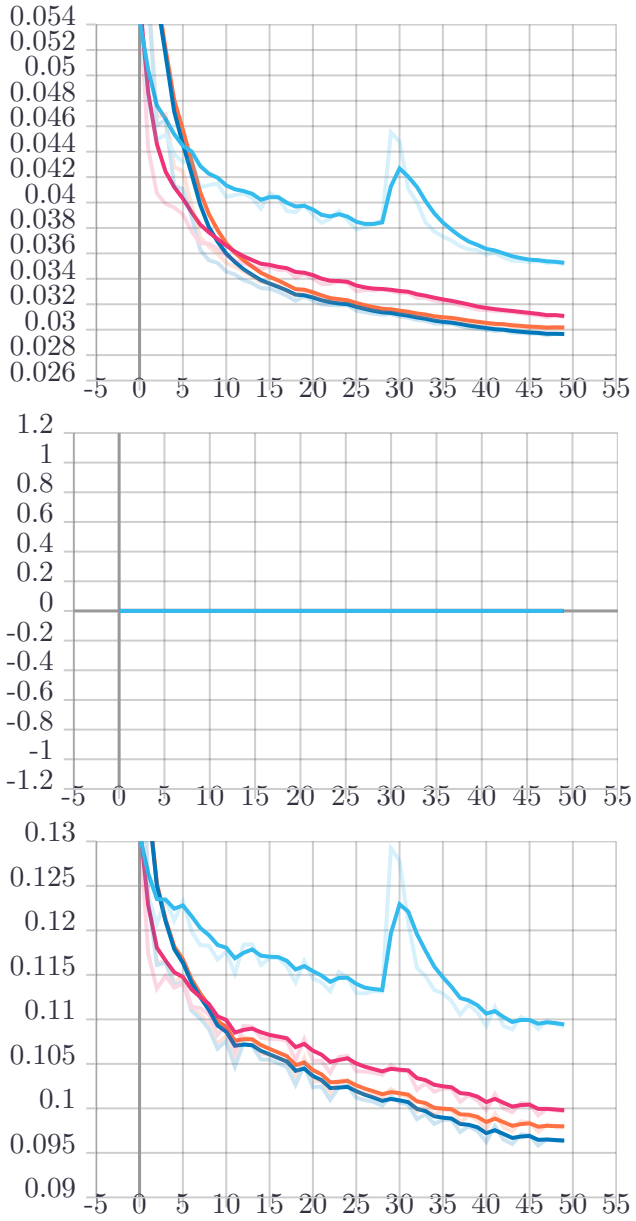

Figure 6: Some Train results using SGD with momentum and Adam optimizer using different parameters. The BLUE line denotes Adam with momentum equals to 0.935 and learning rate 0.0001; the PINK line denotes Adam optimizer with momentum 0.9 and learning rate 0.0001. the Orange line denotes SGD with momentum 0.9 and learning rate 0.001; The dark BLUE line denotes SGD with momentum 0.935 and learning rate 0.001. From top to bottom: Box Loss, Classification Loss, and Objectness Loss

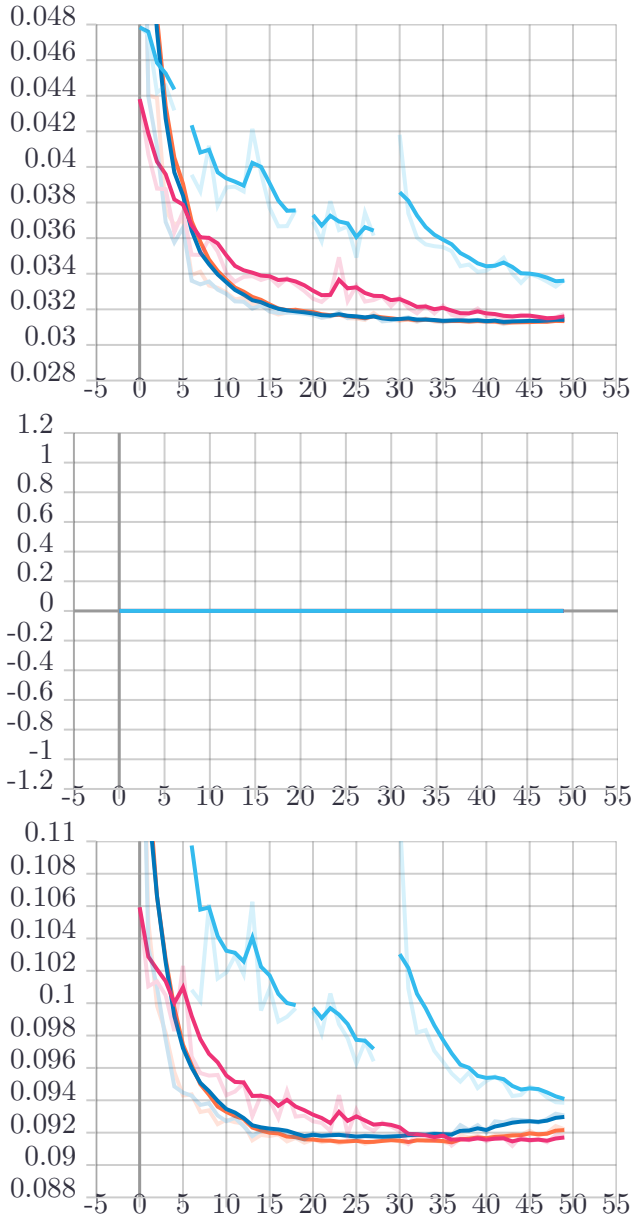

Figure 7: Corresponding Validation results using SGD with momentum and Adam optimizer using different parameters. The BLUE line denotes Adam with momentum equals to 0.935 and learning rate 0.0001; the PINK line denotes Adam optimizer with momentum 0.9 and learning rate 0.0001. the Orange line denotes SGD with momentum 0.9 and learning rate 0.001; The dark BLUE line denotes SGD with momentum 0.935 and learning rate 0.001. From top to bottom: Box Loss, Classification Loss, and Objectness Loss

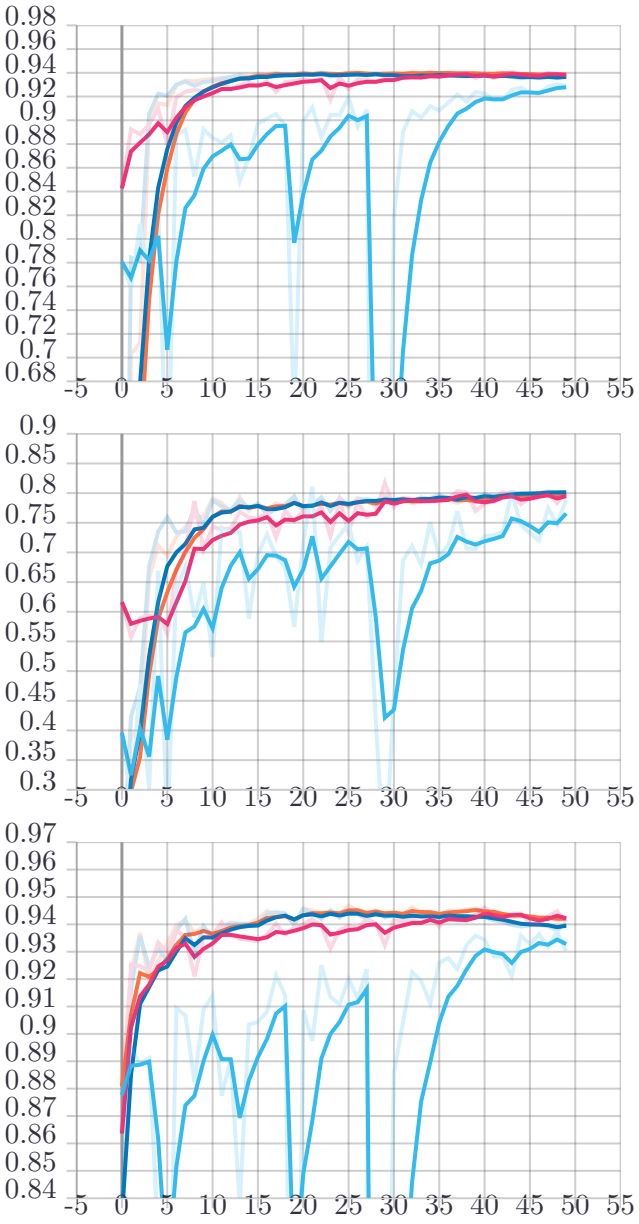

Figure 8: Corresponding metrics results using SGD with momentum and Adam optimizer using different parameters. The BLUE line denotes Adam with momentum equals to 0.935 and learning rate 0.0001; the PINK line denotes Adam optimizer with momentum 0.9 and learning rate 0.0001. the Orange line denotes SGD with momentum 0.9 and learning rate 0.001; The dark BLUE line denotes SGD with momentum 0.935 and learning rate 0.001. From top to bottom: mAP greater than 0.5, Precision, and Recall