Enhancing YOLO Fire Detection System with Ensembling and Hard Negative Mining

April 30, 2025

John Bilbrey
Luddy School of Informatics
Indiana University)
Bloomington, Indiana
jbilbre@iu.edu

I. Introduction

There are practically unlimited applications for object recognition systems using Convolutional Neural Networks, though building a system that is accurate and reliable can be quite a challenge. In this paper, we will explore some techniques that can potentially increase the accuracy of these systems, using a baseline smoke and fire detection system to measure the effects. A system capable of detecting smoke and/or fire has valuable real-world applications, particularly integrating such a system into live video footage, like security cameras. Such a system could save companies, as well as individuals, from extensive monetary damage, and could potentially save lives. For this reason, it is absolutely crucial to maximize the accuracy and reliability of the system, and that makes a smoke and fire detection system a great candidate to test these potential methods of improvement on.

The methods for improvement that we will test are ensembling, and hard negative mining. To briefly summarize these methods, ensembling utilizes multiple different models to run predictions on an image, then combines the results from each model into one prediction, reducing individual biases of each model. Hard negative mining is the practice of recording false positives during the testing phase, then adding these images causing false positives back into the training set, and finally retraining on the updated training set. This allows the model or models to practice specifically on images that it had trouble accurately predicting during the initial test phase. Both of these techniques, in theory, should increase the accuracy and reliability of the system, though we will examine the results to come to a conclusion using hard evidence.

To measure the effects of these methods, we will first create a baseline model using a simple YOLOv8n convolutional neural network model, and test on our dataset in order to get some metrics to be able to measure against our improved models. The metrics we will use to compare models are a simple test accuracy, which only measures accuracy of classification, and also a mean average precision score, or mAP, to also measure the accuracy in terms of location and size for the prediction boxes that our models will draw. By using both of these metrics, we can get an idea of not only how well the models can detect smoke and fire, but also how accurately they can identify their location/s. Once we have our baseline model functioning, we can then implement the potential methods of improvement, and compare these metrics.

II. Background and Related Work

A. YOLO models

First, we must understand the type of convolutional neural network models that we will be working with. YOLO, standing for 'You Only Look Once', is a convolutional neural network model designed to find and label objects in one single pass, hence the name. While many other models break an image into smaller regions, and then evaluate these regions one at a time, YOLO models divide the whole image into a grid, then runs predictions for bounding boxes and their classifications all at once. This model is designed to work particularly quickly, making it a great fit for our end goal application of detecting fires in real-time video footage. YOLO also has many different versions, from version 1 to 11. These versions also include different sizes, including but not limited 'n' for nano, 'm' for medium, and 'l' for large. For our system, we will utilize these three sizes from the YOLOv8 model, trained at 10 epochs.

B. Ensembling

As briefly mentioned in the introduction, ensembling utilizes multiple different detection models to reduce biases that each individual model may have. By running predictions in the same image for each model, then combining these results into one prediction, we can allow the models to exhibit their strengths, while diminishing their weaknesses. In this application, we will utilize YOLOv8n, YOLOv8m, and YOLOv8l models to perform ensembling.

One limitation of this method is that it increases training time significantly, because each model must be individually trained. However, by implementing caching for these models, we can store the results of training to be accessed freely at any time

after training is complete. Essentially, what this means is that initial training will take much longer, but once this is complete, there will be no disadvantage in terms of running time.
A key aspect of this method is the technique in which you combine each models predictions into one. There are multiple methods to do this including non-maximum suppression and weighted box fusion, but for our application, we will use weighted box fusion. This algorithm works by collecting overlapping predictions of the same class, and then averaging their coordinates and confidence scores, into one bounding box. Furthermore, bounding boxes with higher confidence scores carry more weight as compared to lower confidence scores.

C. Hard Negative Mining

The hard negative mining technique allows our models to get extra training on images that are falsely registered as positives during training, allowing the models to fix inaccuracies encountered during the initial testing phase. After initial training, we perform the testing phase, collecting any images that the model incorrectly predicts positives where there actually is no smoke or fire. Then, we add those images back to the training set with their correct labels, and retrain them. This allows the models to more accurately distinguish between images that it previously had trouble with.

D. Dataset

The dataset we will utilize to create and test our models based on is titled ”Smoke-Fire-Detection-YOLO”, which is specifically designed for YOLO models. This dataset contains 14,210 training images, 3,099 validation images, and 4,306 testing images. While this dataset is not especially large, it is adequate for our intended purpose. Also, the goal is not necessarily to maximize accuracy, but rather to maximize the increase in accuracy between our baseline model, and the models utilizing ensembling and hard negative mining.

E. Related Work

The original paper that used this dataset to create a smoke and fire recognition system, titled ”An automatic fire detection system based on deep convolutional neural networks for low-power, resource-constrained devices” was focused on creating this system using a convolutional neural network, although this system was designed to work with live video feed. While this is the end goal of my system, it is not currently implemented to do so. Rather, it is designed to work with single images. However, this paper did not employ the techniques of ensembling and hard negative mining. My goal is to explore potential improvements to this system, by implementing these methods.

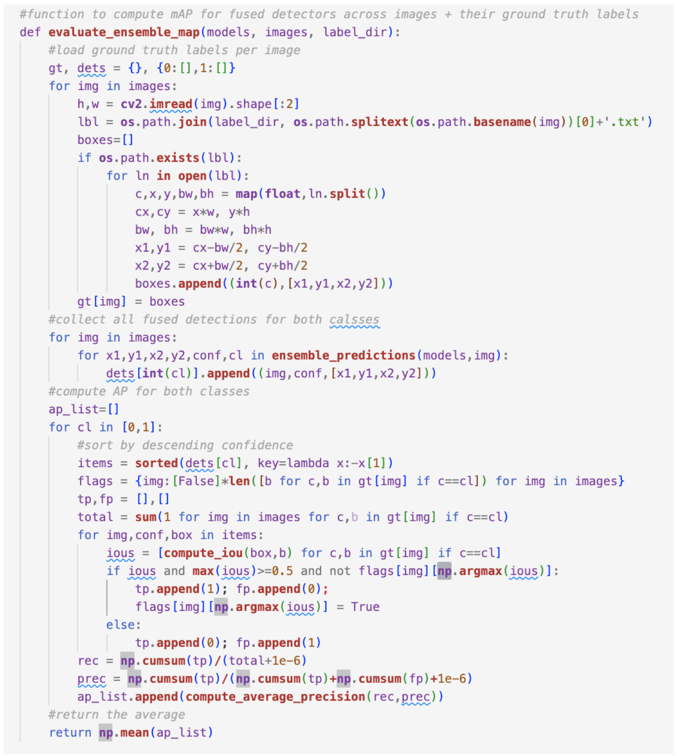
III. Methods and Implementations

A. Libraries and Frameworks

To create a functioning system, we will utilize three main libraries: openCV, numpy, and ultralytics. OpenCv will be used to draw bounding boxes on images, as well as load and display images. Numpy will be used to create necessary arrays and organize relevant data. Lastly, ultralytics is the library that contains all of the pre-trained YOLO models that we will utilize. This allows us to train and evaluate models, as well as retrieve metrics for each individual model.

B. Evaluation metrics

To evaluate the performance of our models, we will utilize a simple test accuracy, as well as a mean average precision score, or mAP. The former can be used to measure purely how well the model can detect smoke and/or fire, without factoring in the locational accuracy of the bounding boxes it draws. This simply considers a prediciton accurate if classes of the bounding boxes drawn match the ground-truth labels. The latter will factor in the locational accuracy, giving us an idea of how precisely the models can locate the smoke and/or fires. To obtain the mAP values for our baseline YOLOv8n model, we can simply retrieve it after evaluating on the test set using the ultralytics library, as this is one of the metrics that is computed and stored during the testing phase. However, to obtain the mAP for our system utilizing ensembling and hard negative mining, we will need to calculate these values ourselves. Furthermore, the YOLO models within the ultralytics library do not compute a simple test accuracy, so we must also compute these values manually. To compute mAP after ensembling, a custom function is needed, which is shown in figure 1.



```python
#function to compute mAP for fused detectors across images + their ground truth labels
def evaluate_ensemble_map(models, images, label_dir):
    #load ground truth labels per image
    gt, dets = {}, {0:[],1:[]}
    for img in images:
        h,w = cv2.imread(img).shape[:2]
        lbl = os.path.join(label_dir, os.path.splitext(os.path.basename(img))[0]+'.txt')
        boxes=[]
        if os.path.exists(lbl):
            for ln in open(lbl):
                c,x,y,bw,bh = map(float,ln.split())
                cx,cy = x*w, y*h
                bw, bh = bw*w, bh*h
                x1,y1 = cx-bw/2, cy-bh/2
                x2,y2 = cx+bw/2, cy+bh/2
                boxes.append((int(c),[x1,y1,x2,y2]))
        gt[img] = boxes
    #collect all fused detections for both calsses
    for img in images:
        for x1,y1,x2,y2,conf,cl in ensemble_predictions(models,img):
            dets[int(cl)].append((img,conf,[x1,y1,x2,y2]))
    #compute AP for both classes
    ap_list=[]
    for cl in [0,1]:
        #sort by descending confidence
        items = sorted(dets[cl], key=lambda x:-x[1])
        flags = {img:[False]*len([b for c,b in gt[img] if c==cl]) for img in images}
        tp,fp = [],[]
        total = sum(1 for img in images for c,b in gt[img] if c==cl)
        for img,conf,box in items:
            ious = [compute_iou(box,b) for c,b in gt[img] if c==cl]
            if ious and max(ious)>=0.5 and not flags[img][np.argmax(ious)]:
                tp.append(1); fp.append(0);
                flags[img][np.argmax(ious)] = True
            else:
                tp.append(0); fp.append(1)
        rec = np.cumsum(tp)/(total+1e-6)
        prec = np.cumsum(tp)/(np.cumsum(tp)+np.cumsum(fp)+1e-6)
        ap_list.append(compute_average_precision(rec,prec))
    #return the average
    return np.mean(ap_list)
```

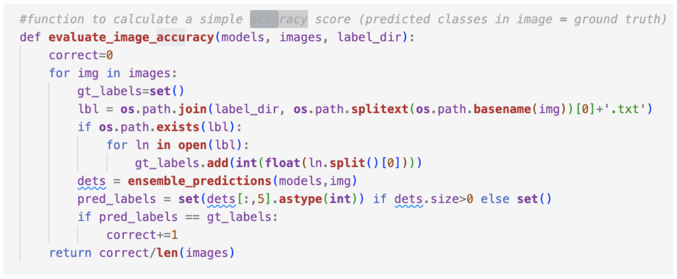
Fig. 1. Function to compute mAP for model after ensembling

The general algorithm for this function is as follows:

1) load ground-truth labels for each image in the test set

2) evaluate the ensembled model on the test set, collecting the coordinates and confidence levels for each bounding box in each image
3) Match detections to ground-truth labels
4) Calculate average precision
5) average the two classes' average precisions to find mAP

To calculate the simple test accuracy, we need only to compare the predicted classes on an each image to the actual classes in the image listed in the ground-truth labels. The function that does this is shown in figure 2. The general



```python
#function to calculate a simple accuracy score (predicted classes in image = ground truth)
def evaluate_image_accuracy(models, images, label_dir):
    correct=0
    for img in images:
        gt_labels=set()
        lbl = os.path.join(label_dir, os.path.splitext(os.path.basename(img))[0]+'.txt')
        if os.path.exists(lbl):
            for ln in open(lbl):
                gt_labels.add(int(float(ln.split()[0])))
        dets = ensemble_predictions(models,img)
        pred_labels = set(dets[:,5].astype(int)) if dets.size>0 else set()
        if pred_labels == gt_labels:
            correct+=1
    return correct/len(images)
```

Fig. 2. Function to compute simple test accuracy for any model

algorithm for this function is as follows:

1) load ground-truth labels for each image in the test set
2) evaluate the model on the test set and record predicted classes for each image
3) compare predicted classes to ground-truth classes
4) divide the number of correctly classified images by the total number of images in test set

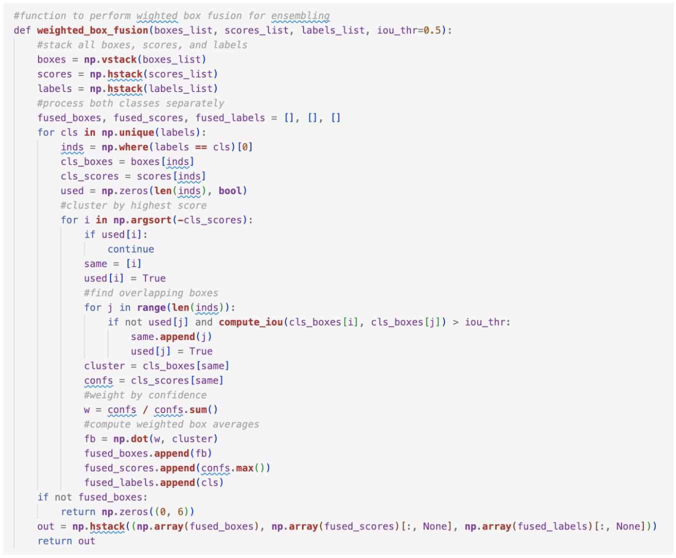
C. Ensembling

To perform ensembling, we must create our three chosen models (YOLOv8n, YOLOv8m, YOLOv8l) and train them on the training set. Then, for each image in the test set, we run predictions using each model, and use weighted box fusion to combine these results.
Firstly, weighted box fusion is performed by collecting all the predictions in a given image, and combining overlapping predictions of the same class into one by averaging the coordinates and confidence scores of each prediction. The function utilized to perform this is shown in Figure 3, and the general steps of this algorithm are as follows:

1) gather all predictions from all models in an image
2) for each possible class (smoke and fire), cluster overlapping boxes
3) fuse each cluster by computing the weighted average of the coordinates and confidence scores, for each prediction in the cluster
4) return fused detection

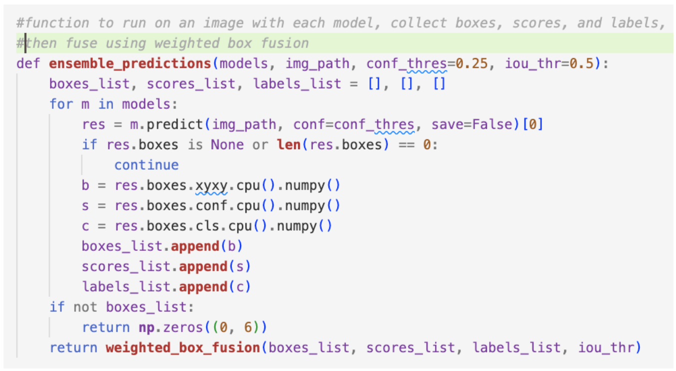
Now, in order to run a prediction on an image using ensembling, we will utilize another function, shown in figure 4, to run predictions on the image for each model, and weighted box fusion to combine the results.

This function can be utilized to run ensembled predictions on each image in the test set. Then, using the previously



```python
#function to perform wighted box fusion for ensembling
def weighted_box_fusion(boxes_list, scores_list, labels_list, iou_thr=0.5):
    #stack all boxes, scores, and labels
    boxes = np.vstack(boxes_list)
    scores = np.hstack(scores_list)
    labels = np.hstack(labels_list)
    #process both classes separately
    fused_boxes, fused_scores, fused_labels = [], [], []
    for cls in np.unique(labels):
        inds = np.where(labels == cls)[0]
        cls_boxes = boxes[inds]
        cls_scores = scores[inds]
        used = np.zeros(len(inds), bool)
        #cluster by highest score
        for i in np.argsort(-cls_scores):
            if used[i]:
                continue
            same = [i]
            used[i] = True
            #find overlapping boxes
            for j in range(len(inds)):
                if not used[j] and compute_iou(cls_boxes[i], cls_boxes[j]) > iou_thr:
                    same.append(j)
                    used[j] = True
            cluster = cls_boxes[same]
            confs = cls_scores[same]
            #weight by confidence
            w = confs / confs.sum()
            #compute weighted box averages
            fb = np.dot(w, cluster)
            fused_boxes.append(fb)
            fused_scores.append(confs.max())
            fused_labels.append(cls)
    if not fused_boxes:
        return np.zeros((0, 6))
    out = np.hstack((np.array(fused_boxes), np.array(fused_scores)[:, None], np.array(fused_labels)[:, None]))
    return out
```

Fig. 3. function to perform weighted box fusion



```python
#function to run on an image with each model, collect boxes, scores, and labels,
#then fuse using weighted box fusion
def ensemble_predictions(models, img_path, conf_thres=0.25, iou_thr=0.5):
    boxes_list, scores_list, labels_list = [], [], []
    for m in models:
        res = m.predict(img_path, conf=conf_thres, save=False)[0]
        if res.boxes is None or len(res.boxes) == 0:
            continue
        b = res.boxes.xyxy.cpu().numpy()
        s = res.boxes.conf.cpu().numpy()
        c = res.boxes.cls.cpu().numpy()
        boxes_list.append(b)
        scores_list.append(s)
        labels_list.append(c)
    if not boxes_list:
        return np.zeros((0, 6))
    return weighted_box_fusion(boxes_list, scores_list, labels_list, iou_thr)
```

Fig. 4. Function to perform ensembled predictions on an image

discussed functions to evaluate the metrics, we can calculate the overall test accuracy and mAP for the ensembled model on the entire test set. The general steps of this function are as follows:

1) initialize empty lists to store boxes, scores, and labels, separately.
2) run predictions on given image for each model
3) call on weighted box fusion function to combine results

D. Hard Negative Mining

Performing hard negative mining entails running each model on the test set, recording predictions with false positives, adding those images back to the training set, and retraining each model.

First, we must obtain the hard negatives, which can be done using a function, as found in figure 5. Hard negatives can be classified, in our case, as images that do not actually contain any smoke or fire (more specifically, images whose label files are empty), but are predicted to by one or more of the models. As such, we need to find all test images whose labels are empty, and for these images, if the models detect a fire or smoke, we will consider them as hard negatives, and add them to a directory in our working repository. Then, after hard negative mining is finished, we can add these images into the training set, and retrain the models on the updated training set. When adding to these images to the training set, it is a good idea to mark them as hard negatives so that it is easy to identify them in the case that we would like to utilize the original training set for any reason. To do so, I added a 'HNM' prefix to each image and corresponding label file. The general steps of this algorithm are as follows:

1) create a list that contains all empty label files in the test set
2) run ensembled predictions on each corresponding image
3) record predictions containing smoke and/or fire
4) add these images to their own separate directory with an 'HNM' prefix in the filename

Once this function is ran, we can then simply add all of the images in the created directory to the training set, along with their corresponding ground-truth labels, which is simply an empty text file. Then, we can retrain all models on the updated training set.

E. Main Workflow

In order to combine these methods into a fully functioning system, we can abide by the following general workflow:

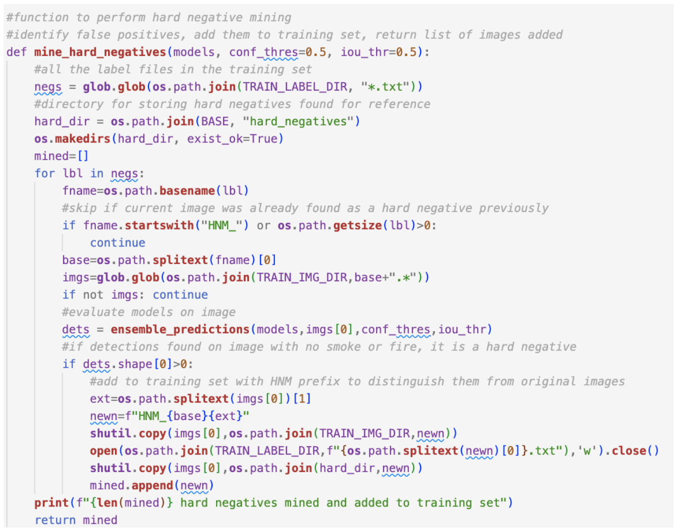
1) create and train all 3 models on the original training set
2) test baseline model on test set and record accuracy and mAP
3) run ensembled predictions on test set and record accuracy and mAP (for ensembling only metrics)
4) perform hard negative mining
5) retrain all models on updated training set
6) test YOLOv8n model on test set after hard negative mining and record accuracy and mAP (for hard negative mining only metrics)
7) perform ensembled predictions on test set after hard negative mining and record accuracy and mAP (for ensembled + hard negative mining metrics)

IV. Results

	Baseline	Ensembling	HNM	Ensembling + HNM
Test accuracy	82.72%	84.42%	82.86%	85.28%
mAP	0.6560	0.5651	0.6476	0.5978

TABLE I
METRICS COMPARISON OF BASELINE, ENSEMBLING, HNM, AND COMBINED METHODS

As shown in the table, our baseline model has a test accuracy of 82.72% and an mAP of 0.6560. While this is relatively good for a baseline model, it still leaves room for improvement, and is still prone to producing inaccurate predictions from time to time. By implementing ensembling, test accuracy was improved by about 2%, at 84.43%. While this is an improvement, it is not a drastic one, and the model is still very much capable of producing inaccuracies. Perhaps more notably, the mAP for ensembling only significantly decreased, down to 0.5651. This is potentially worrying, although in this particular system, test accuracy should be prioritized over mAP, because being able to recognize if there is a fire or not is more important than being able to accurately locate it.



```
#function to perform hard negative mining
#identify false positives, add them to training set, return list of images added
def mine_hard_negatives(models, conf_thres=0.5, iou_thr=0.5):
    #all the label files in the training set
    negs = glob.glob(os.path.join(TRAIN_LABEL_DIR, "*.txt"))
    #directory for storing hard negatives found for reference
    hard_dir = os.path.join(BASE, "hard_negatives")
    os.makedirs(hard_dir, exist_ok=True)
    mined=[]
    for lbl in negs:
        fname=os.path.basename(lbl)
        #skip if current image was already found as a hard negative previously
        if fname.startswith("HNM_") or os.path.getsize(lbl)>0:
            continue
        base=os.path.splitext(fname)[0]
        imgs=glob.glob(os.path.join(TRAIN_IMG_DIR,base+".*"))
        if not imgs: continue
        #evaluate models on image
        dets = ensemble_predictions(models,imgs[0],conf_thres,iou_thr)
        #if detections found on image with no smoke or fire, it is a hard negative
        if dets.shape[0]>0:
            #add to training set with HNM prefix to distinguish them from original images
            ext=os.path.splitext(imgs[0])[1]
            newn=f"HNM_{base}{ext}"
            shutil.copy(imgs[0],os.path.join(TRAIN_IMG_DIR,newn))
            open(os.path.join(TRAIN_LABEL_DIR,f"{os.path.splitext(newn)[0]}.txt"),'w').close()
            shutil.copy(imgs[0],os.path.join(hard_dir,newn))
            mined.append(newn)
    print(f"{len(mined)} hard negatives mined and added to training set")
    return mined
```

Fig. 5. function to mine hard negatives

When only hard negative mining is performed, there is a negligible difference from the baseline model. Although test accuracy improves slightly from 82.72% to 82.86%, the mAP decreases slightly from 0.6560 to 0.6476. However, these differences are so small that they will likely not produce meaningfully different results.

By combining ensembling and hard negative mining, we see a noteworthy, though not substantial, increase in test accuracy compared to the baseline, from 82.72& to 85.28%. However, mAP is still considerably lower than the baseline at 0.5978.

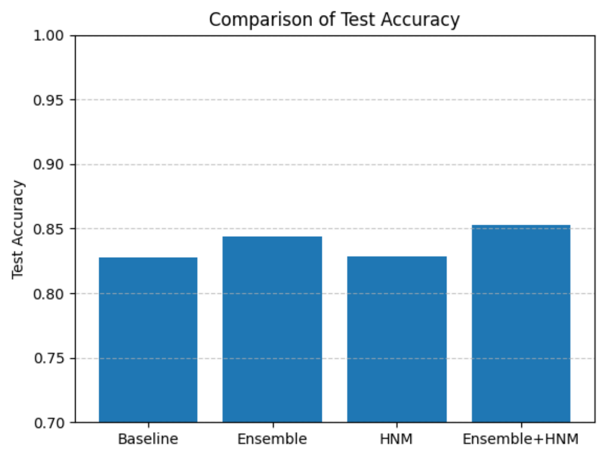


Fig. 6. Bar graph comparing test accuracy between methods

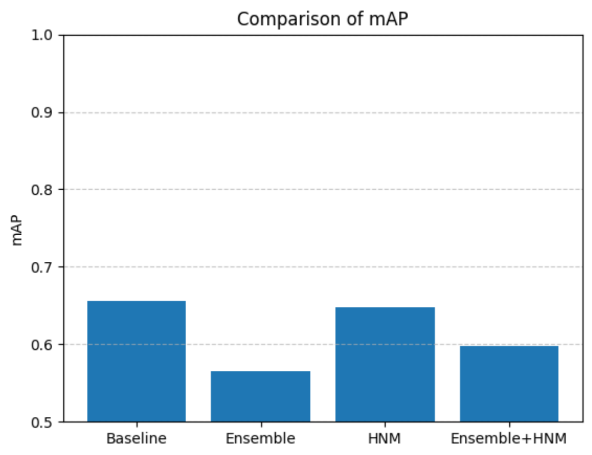


Fig. 7. Bar graph comparing mAP between methods

V. Discussion

While these methods arguably do not provide as significant of an increase in accuracy and reliability as one may expect, it still provides enough of an increase in test accuracy to justify using these methods over a simple baseline model. Especially in this application, where the ability to detect a fire is substantially more important than being able to accurately locate it, this could potentially be the difference between the system failing to recognize a fire, and succesfully recognizing it. However, in other applications where the precision of bounding boxes is more important, one could easily make the case that the drop off in mAP is enough to forego implementing these methods. It is also important to note that experimenting with different models in the ensembling process, as well as more models, could lead to further improvements. Additionally, performing multiple iterations of hard negative mining could also increase these metrics. Finally, in the end goal application of applying this system to live video feed, where each second of footage is composed of many frames (images), this test accuracy value means that about 85% of frames would be able to detect a fire. This means that it would be extremely unlikely for the system not to detect a fire for even just one second of video footage.

One limitation of this study was time, in various aspects. I did not have time to experiment with different models in ensembling, or to implement iterative hard negative mining before the due date. Also, training the models, especially when implementing these two methods, was very time consuming, even at just 10 epochs. Another limitation, which is directly correlated with the first, was the computing power of the machine used to carry out these tests. Performing training multiple times on a relatively old MacBook caused each epoch to take a long time to complete.

VI. Conclusion

By leveraging ensembling and hard negative mining, I was able to slightly improve the test accuracy of a baseline YOLOv8n model for detecting smoke and fire in images. Combining the results of multiple YOLO models into one allowed us to reduce the biases of each individual model and avoid errors that one model may make on its own. Hard negative mining allowed these models to confront the images that gave them trouble in the test set, and retrain on these images with ground-truth labels.

These findings are important because applications like smoke and fire detection require near absolute accuracy and reliability, as failing to detect a fire can result in potentially drastic consequences, ranging from significant monetary loss to the loss of life. By exploring ways to improve convolutional neural network models, these consequences can potentially be avoided. Furthermore, object detection systems have many uses beyond smoke and fire detection, and these methods can be applied to any object detection system, including those utilizing systems other than convolutional neural networks as well.

To continue this work, it would be valuable to experiment further with both ensembling and hard negative mining. By utilizing more models for ensembling and performing more iterations for hard negative mining, it is very plausible that accuracy could improve further. Also, exploring other methods entirely could help improve these models, and combining multiple methods could potentially allow us to reach near-perfect accuracy. Though basic models alone can create relatively accurate systems, this is not satisfactory for many applications of object detection, such as this one. However, this study has shown that employing various methods, even just two, can improve these models, and make them more viable for applications requiring high accuracy and consistency.

References

[1] Pedro Vinícius Almeida Borges de Venâncio, Adriano Chaves Lisboa, Adriano Vilela Barbosa. "An automatic fire detection system based on deep convolutional neural networks for low-power, resource-constrained devices." Neural Computing and Applications, vol. 34, no. 18, 2022, pp. 15349–15368. DOI: 10.1007/s00521-022-07467-z.

[2] Gamal, Sayed. Smoke-Fire Detection YOLO. Kaggle, 2022, www.kaggle.com/datasets/sayedgamal99/smoke-fire-detection-yolo.

[3] "Yolo Algorithm for Object Detection Explained [+examples]." YOLO Algorithm for Object Detection Explained [+Examples], v7, www.v7labs.com/blog/yolo-object-detection. Accessed 30 Apr. 2025.

[4] Ultralytics. "Home." Ultralytics YOLO Docs, 14 Apr. 2025, docs.ultralytics.com/.

[5] Sundardell. "Hard Negative Mining." Medium, Medium, 27 Apr. 2023, medium.com/@sundardell955/hard-negative-mining-91b5792259c5.

[6] 1308031437975826437. "Mean Average Precision (MAP): A Complete Guide." Kili, kili-technology.com/data-labeling/machine-learning/mean-average-precision-map-a-complete-guide. Accessed 30 Apr. 2025.

[7] Ibm. "What Is Ensemble Learning?" IBM, 16 Apr. 2025, www.ibm.com/think/topics/ensemble-learning.

[8] Mwiti, Derrick. "A Comprehensive Guide to Ensemble Learning: What Exactly Do You Need to Know." Neptune.Ai, 25 Apr. 2025, neptune.ai/blog/ensemble-learning-guide.