Safety First

Improving Workplace Safety with Hard Hat Detection

Stanley Yang, Lennard Vanderspek

# Introduction

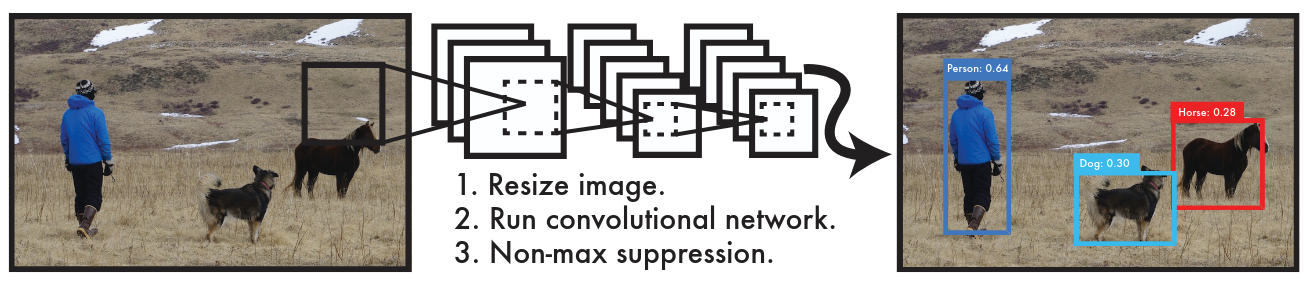
Object detection is a popular machine learning application with many use cases. In this paper we will explore the YOLO algorithm, which proposes an efficient, real time solution to object detection, and use it to detect whether an individual in an image is wearing a hard hat or not. This model may then be applied to camera feeds in construction zones to enforce the use of hardhats in dangerous locations. We used the Hard Hat Workers Dataset provided by Roboflow to train our model.

# Background

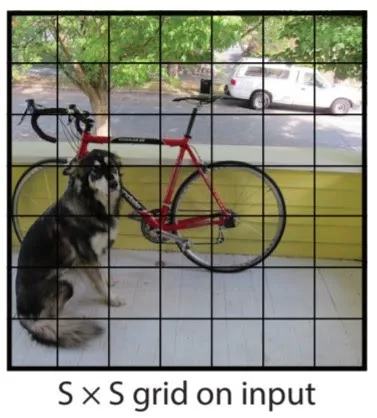
Object detection is the process of identifying a class object and its location within an image. The visualized output is shown as a number of colored boxes around objects within an image. Object detection is distinct from object classification, in which a model predicts the presence of a single object from a number of classes, but not its location.

YOLO is an object detection algorithm developed in 2016. It is popular for its real time performance and unifying object classification and bounding box prediction in a single model. Before YOLO, these two tasks were performed in separate models, but YOLO does both of these steps with a single neural net. As a result, YOLO is much faster than previous approaches, touting a 45 fps figure in its first implementation.

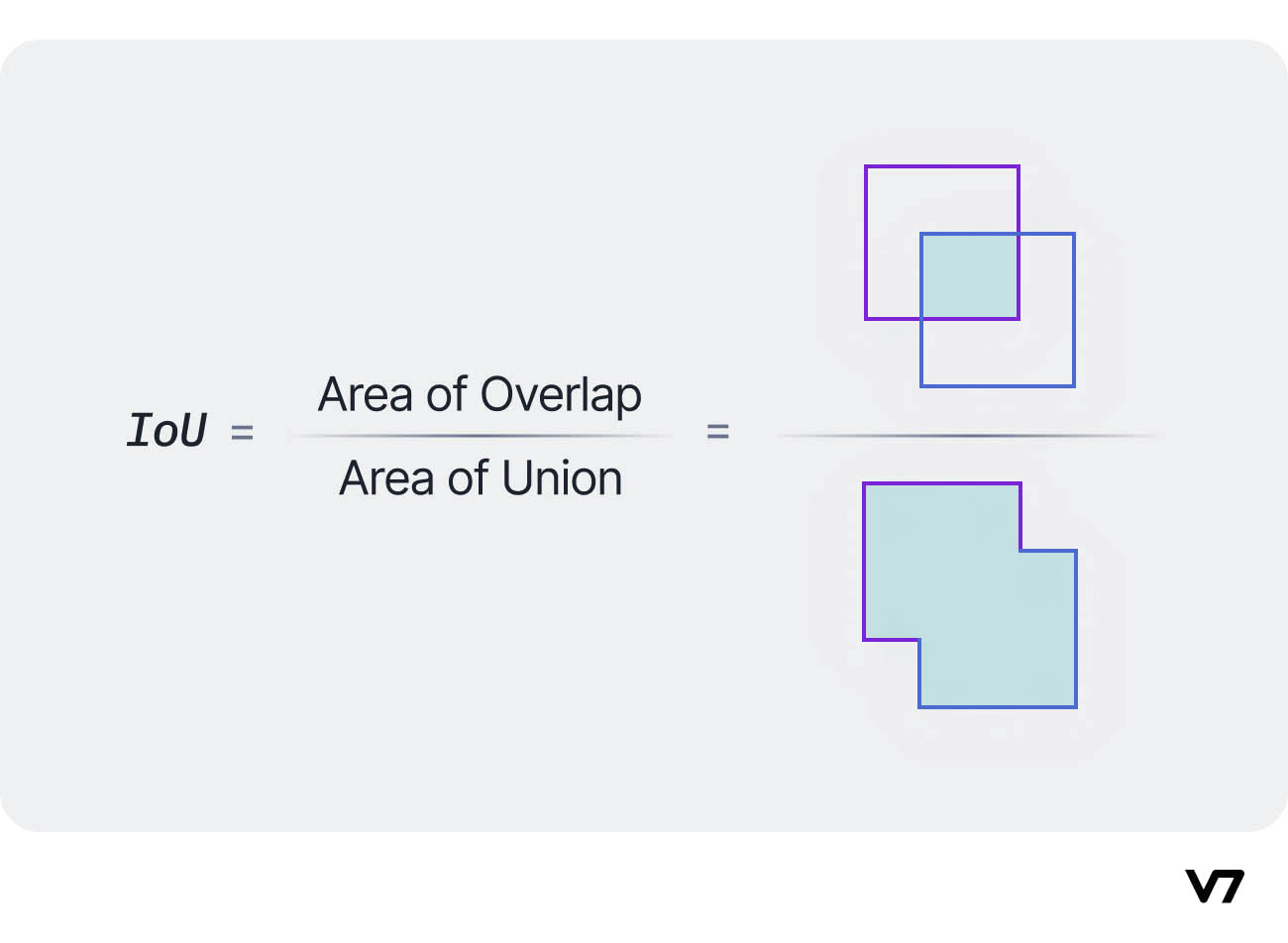
While image classification is the process of mapping an image to a number of classes, the goal with image detection is to locate objects from these classes within a given image. The output for an image detection model should be a set of classifications, each with the location, height, and width of the object, shown graphically below.

[4]

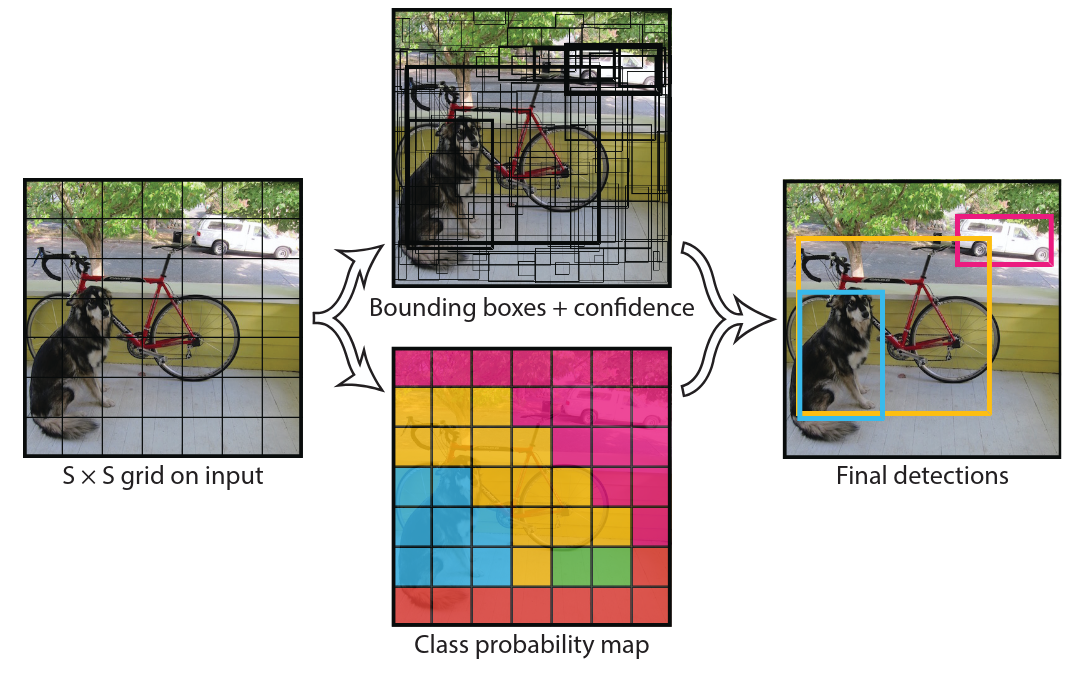
YOLO achieves this by dividing the image into a fixed number of cells, each of which is trained to predict 1) a number of bounding boxes and 2) the predicted class. The predicted class is represented as a one hot vector with a confidence score for each possible class. The size of each of these cells is C+B\*5, where C is the number of classes, B is the max number of predicted bounding boxes, and 5 represents the values x, y, h, w, and confidence of each bounding box. Note that the cell is only meant to predict a single class.

[4]

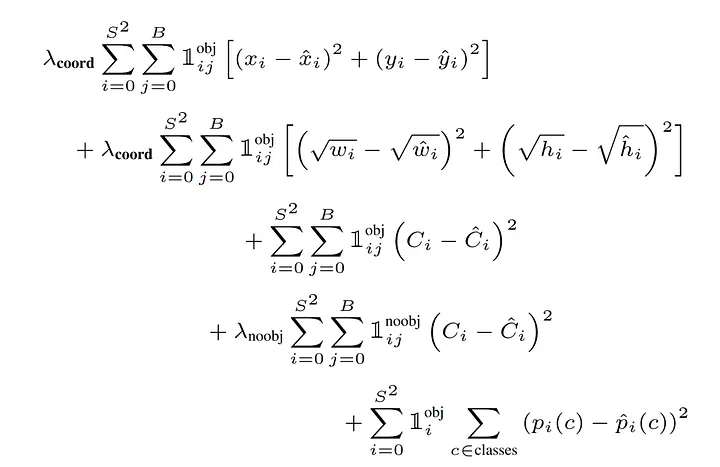
In training, the confidence is calculated as the Intersection of Union (IoU), which is the area of intersection divided by the area of union of the test and predicted bounding boxes.

[8]

The complete approach can be seen below. The image is divided into cells, each of which predicts a number of bounding boxes and a singular class. These values are then aggregated to estimate the true bounding box and corresponding class.

[4]

The loss function is a function of the object center location (x,y), size (w,h) class (C), and confidence (IoU). The loss is the sum of the mean squared difference between the test and predicted values for each.

[4]

The loss function also incorporate the following four lambdas/weight constant, which are all crucial in the outcome of the training and the target focus of the model:

* **Lambda\_coord** - controls the bounding box precision
* **Lambda\_obj** - controls the confidence score on when an object is present in the cell
* **Lambda\_noobj** - controls the confidence score on when an object is NOT present in the cell
* **Lambda\_class** - controls the classification accuracy

In the YOLO model, there is also the concept of backbone/neck/head training paradigm, which breaks the model into three parts with specific jobs.

* **Backbone** is the feature extractor of the model, which processes the input images and learns patterns such as edges and shapes.
* **Neck** aggregates the features from different stages of the backbone, which helps the model detect objects by combining low and high resolution features.
* **Head** is the prediction part of the model, which takes the aggregated feature from the neck and outputs the final object prediction including bounding box coordinates, objectness scores and class probabilities.

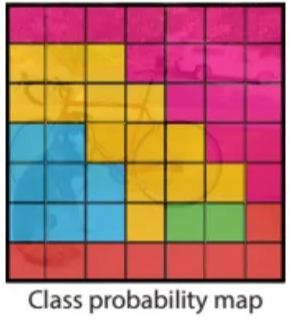
# Implementation

## Differences from YOLO

Our solution generally follows the YOLO algorithm with some minor differences.

### Cell Class Prediction

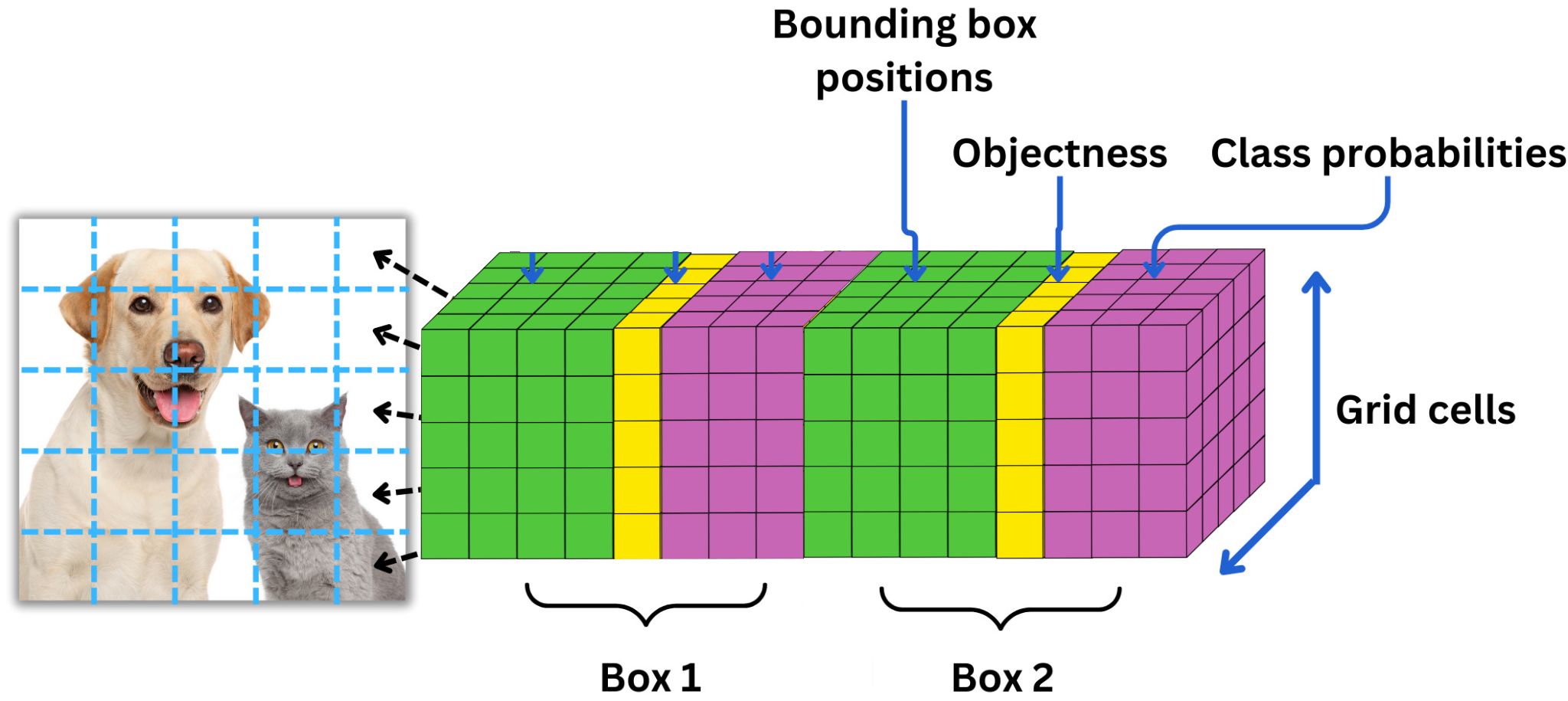
Recall that each cell in the original YOLO predicted the following values: one class per cell and x, y, h, w, c for each bounding box. The size for the cell can be computed as C+(B\*5), where C is the number of classes and B is the number of bounding boxes. With this approach, a class probability map can be constructed:

[4]

Our approach attempts to identify the class for each bounding box, allowing for separate predictions of overlapping objects. This means each cell predicts the following values: class, x, y, h, w, c for each bounding box. The size for the cell can be computed as B\*(C+5). In this case, there is more than one predicted class per cell, so the class probability map is not applicable.

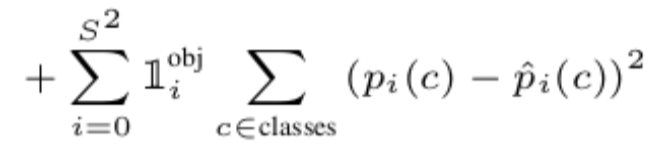
[10]

YOLO Model Size



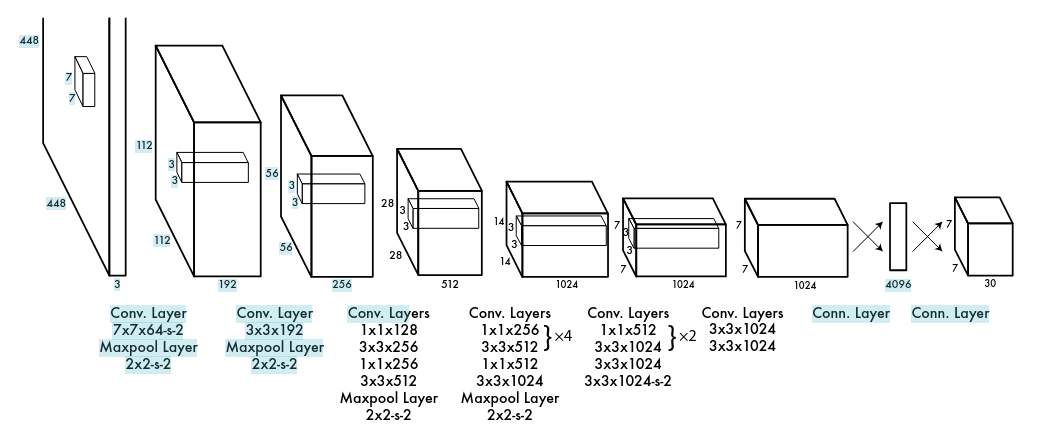
Our Model Size

This changes the loss function slightly by simply increasing the number of class probability losses to compute. In the last summation in YOLO’s loss function, we would compute the mean squared difference for each class, for each cell, *for each bounding box.*

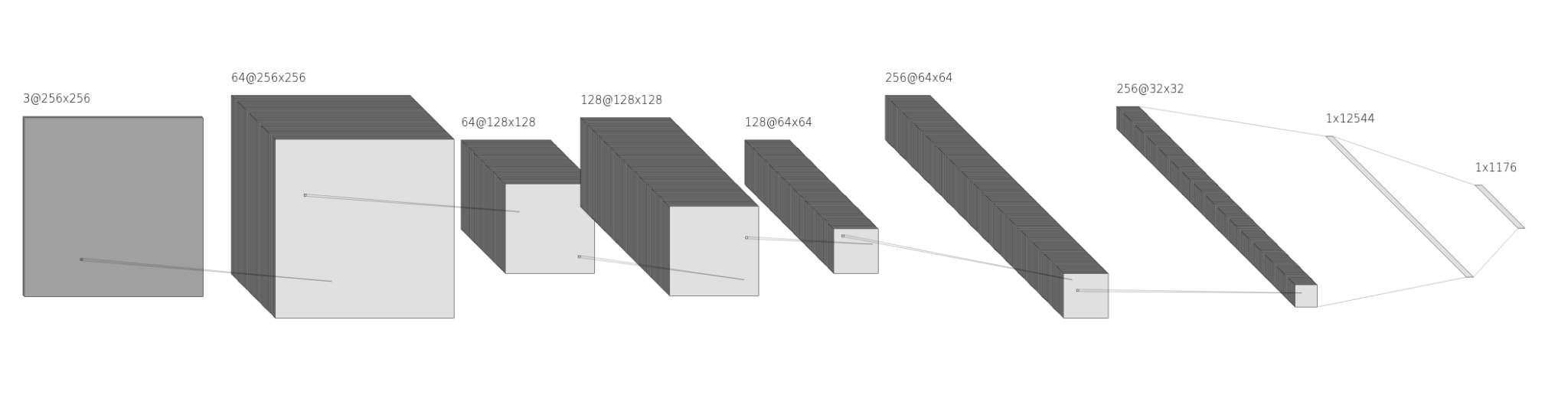
[4]

### Network size

Due to hardware constraints, it was not feasible for us to train a network as large as the one used in the paper. The network described in the YOLO paper has 24 convolutional layers and 4 max pooling layers.

[4]

Our network is much smaller, limited to 3 much smaller convolutional/pooling pairs. It is also smaller than the Fast YOLO model mentioned in the paper, which still uses 9 convolutional layers.



### Image Resolution

In our approach, we used images of size 256x256. While YOLO used 224x224 images for object classification training, the images were sized to 448x448 due to object detection needing higher resolution images.

### Object Classification Pretraining

The first 20 convolutional layers in YOLO were pretrained (for a week) for object classification on a separate training set followed by an average pooling layer. This trained model was then converted to perform detection. In our approach, we attempted to do all of the learning in a single training phase.

## Training

The training phase consisted of anywhere between 10 to 50 epochs. For the training data, we used a smaller subset of the Hard Hat Dataset, ranging from 2,000 to 5,259 images. Due to hardware limitations, each training took anywhere from 1 to 10 hours. We understand that small datasets and low epoch counts can significantly impact the accuracy of the YOLO model; however, this was the only way we could allocate enough time to experiment with various parameters and training techniques.

Since we were training the model from scratch with random weights, we implemented several techniques to assist with the early phase of training. We started with a learning rate warm-up, which gradually increased the learning rate after each batch until it reached the base learning rate. This helps prevent the model from overcorrecting and avoids the exploding gradient problem. We also applied warm-up phases to the confidence and IoU thresholds. Doing so allows the model to learn from as many images as possible, regardless of initial prediction accuracy, helping to prevent issues with the model failing to produce matches early on.



As training progressed into later epochs, we aimed to stabilize the model and make smaller adjustments to avoid overshooting. To achieve this, we implemented a learning rate scheduler that decreased the learning rate by 5% at the end of each epoch.



# 

# Results

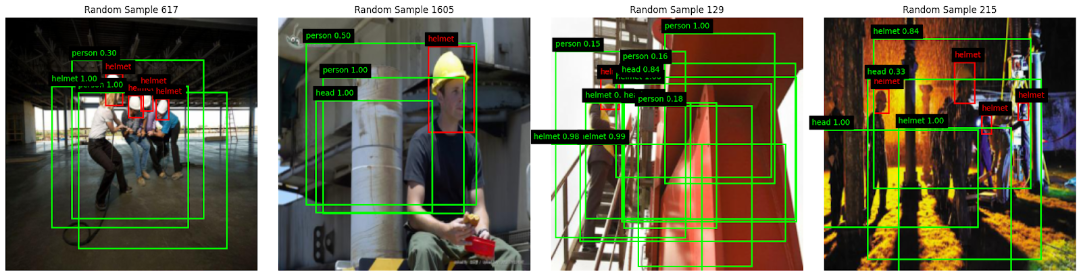
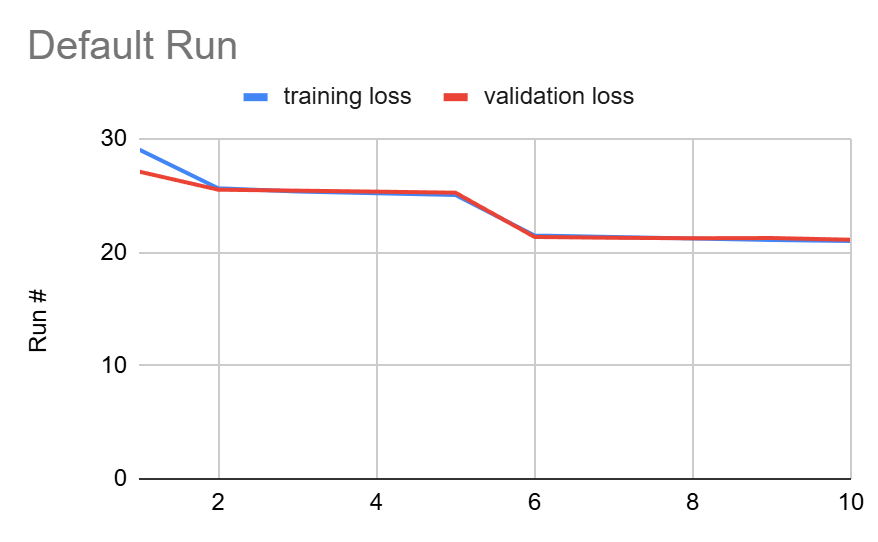
To understand the output visualization:

Red box - the label and bounding box provided by the training data set

Green box - the label and bounding box predicted by our trained model at each of the runs

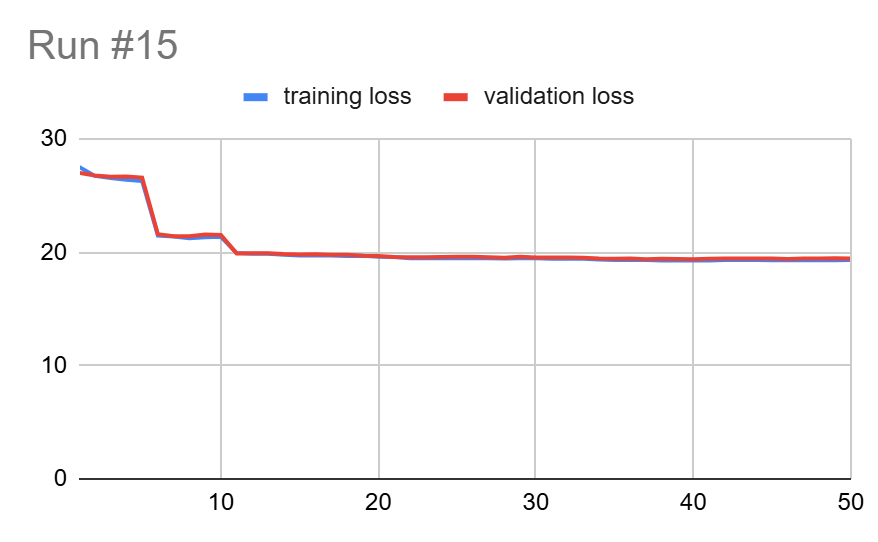
## Default Run

We start with the following run as the base line. In this run we only see significant improvement on the first and fifth epoch. This is most likely caused by the early warmups we have implemented, which all ends on the fifth epoch.



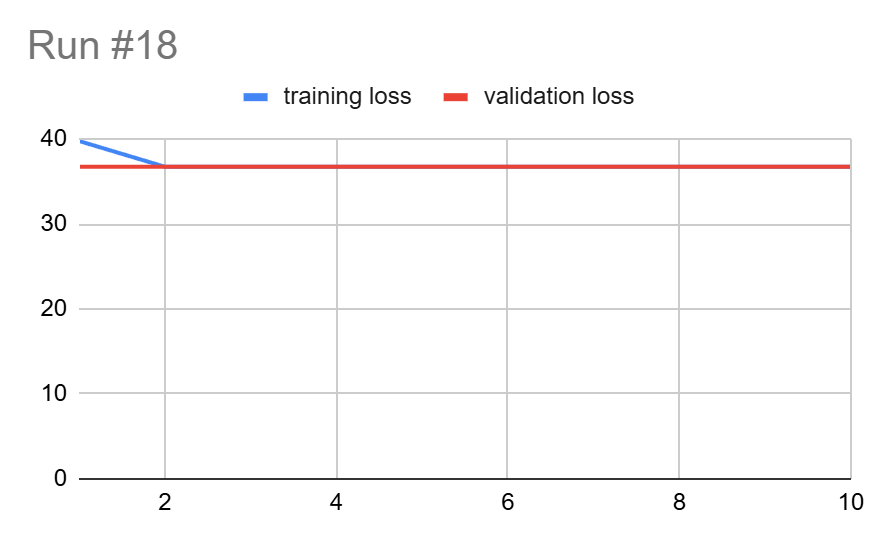
## Run #15

In the first 15 runs, we have been testing with different adjustments in the learning rate, loss function lambda constants, and other various training variables. We decide to perform a run #15 with significantly more data sets and epochs hoping that can give the model more time to learn and refine. However, this resulted in very little improvement.



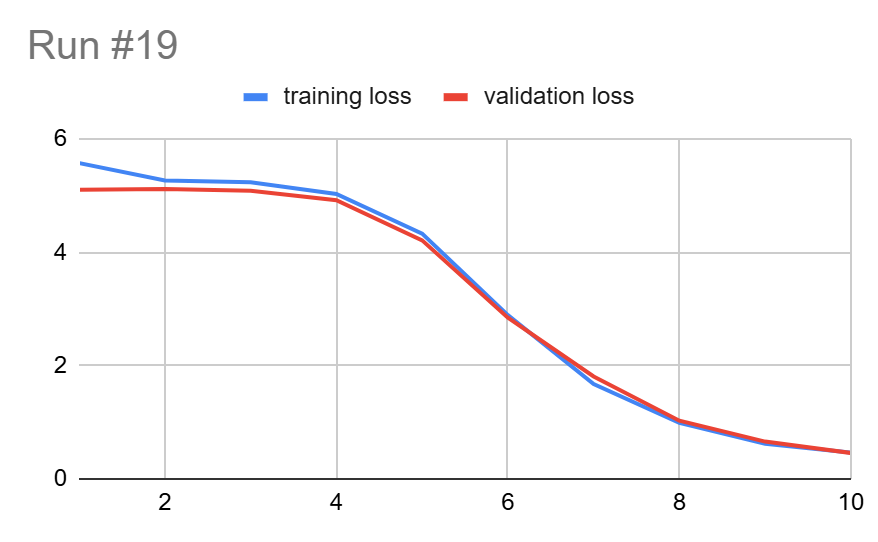
## Run #18

After seeing an extremely slow learning rate, we decided to perform some more tests without the warm ups. This change eliminates the sudden drop in training and validation loss but we still see a stale learning trend.



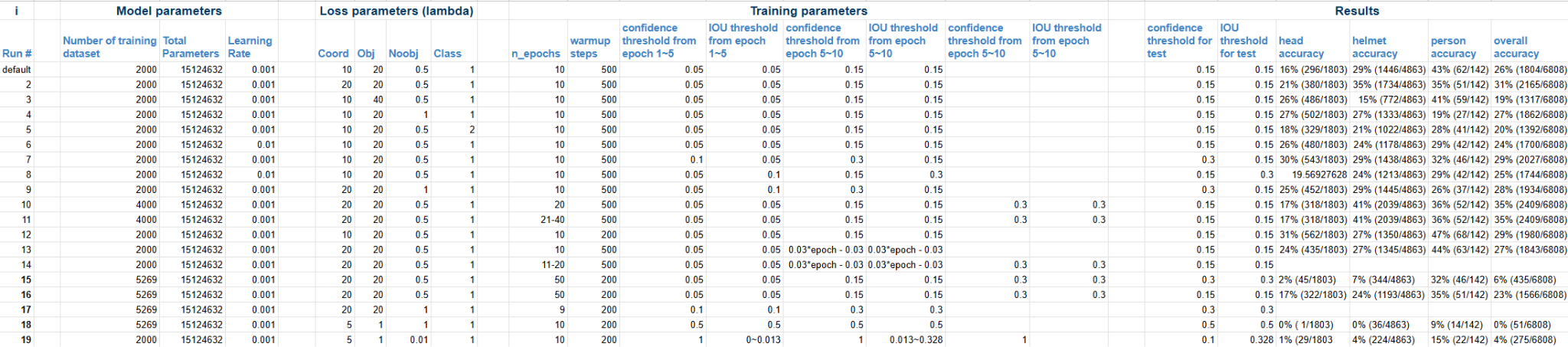
## Run #19

After further testing, we believe we have identified the reason why the training and validation losses appear stagnant. The issue came from the lambda noobj value being too high, causing the total loss to be dominated by the no-object loss. Reducing the lambda noobj significantly results in a much healthier training and validation loss trend; however, the final prediction seems to be even worse.



## All Runs

This shows the result of all the test runs we have done. Click [here](https://docs.google.com/spreadsheets/d/1tLdOCVmk66PDNjnhk-Kcpp3BZP11XCFAz4xebq8Ma9k/edit?usp=sharing) for the full test result.



# Conclusion

Although the final model was not able to confidently predict objects, we were able to identify the cause of the stagnant training and validation losses. This highlights the critical role of the loss function in model training, as well as the significant impact a single lambda constant can have on the model’s performance.

## Future Work and Improvements

Since we finally see an improvement in the training and validation loss trend, we can continue to tune other lambda constants to see if that improves the prediction.

One major area for improvement would have been our training strategy. In the YOLO paper, the researchers describe training 20 of the 24 convolutional layers for object classification on a separate dataset. This portion of the model was then used during the object detection training phase. By pretraining a portion of our model, or even leveraging an existing model, we may have had better luck in training.

A main constraint we faced was hardware limitations. This limited the size of our network, the resolution of our images, and the number tests we were able to run. By improving in any of these areas we may have been able to produce a more accurate model.

# Resources

1. Repo [https://github.com/yyang31/CSCI611\_Summer25\_Final\_Project](https://github.com/yyang31/CSCI611_Summer25_Final_Project#)
2. All test run results: [Hard Hat Detection Yolo Results](https://docs.google.com/spreadsheets/d/1tLdOCVmk66PDNjnhk-Kcpp3BZP11XCFAz4xebq8Ma9k/edit?usp=sharing)
3. Final Presentation: [Final Presentation](https://docs.google.com/presentation/d/1trLkGQRQCM7wBdbyA56u3VSkrFL8DZZOTOGhlcePbdU/edit?usp=sharing)
4. YOLO paper: <https://arxiv.org/abs/1506.02640>
5. Roboflow Hard Hat Dataset <https://public.roboflow.com/object-detection/hard-hat-workers>
6. <https://medium.com/@whyamit404/how-to-implement-a-yolo-object-detector-from-scratch-in-pytorch-e310829d92e6>
7. <https://medium.com/analytics-vidhya/yolo-explained-5b6f4564f31>
8. <https://www.v7labs.com/blog/yolo-object-detection>
9. <https://www.geeksforgeeks.org/computer-vision/yolov3-from-scratch-using-pytorch/>
10. <https://newsletter.theaiedge.io/p/deep-dive-how-yolo-works-part-1-from>