Project Report on

NFL Safety and Helmet Collision Detection

Submitted by:

Taylor Thomas

Bhoj Raj Thapa

Samaneh Rabienia

Laziz Muradov

Mahsen Al-Ani

December 2021

1. Introduction

The National Football League (NFL), America's most popular sports league, has been invested in player safety. Recently, NFL has increased its commitment to player safety and collaborated with Amazon Web Services (AWS) to develop the best sports injury surveillance and mitigation program. The objective of this study was to identify all possible risks such as helmet-to-helmet, helmet-to-shoulder, and helmet-to-ground collisions to protect players and make the game safer. So, developing a digital athlete program will improve player safety, and make the game safer. Our contribution in this study was to design a model using machine learning algorithms and train a neural network to detect the player's helmet from live video. Helmet detection is one of the most critical steps of the implementation. One of the challenges of helmet detection is Helmet size. The helmet size is small compared with the image size, which causes the miss and false detection and subsequently will affect the detection accuracy. So, to solve this issue we used the YOLOV 5 object detector that operates well on small objects to detect helmet objects in each video frame.

1.1 Background

Injury prediction and prevention in contact sports has become an increasingly important aspect of large sports leagues across the world. Using technology to automate the retrieval and analysis of data is valuable for these sports leagues because it ultimately makes the games safer for the players, which increases support for the game. The National Football League has been increasing its commitment to promoting the health and safety of its players for several years now. With increased awareness of the dangers of impact sports like football, ensuring the proper diagnosis and prevention of injuries has become a high priority for the NFL. To improve upon its current methodologies of manually detecting exposures from a small subset of plays, the NFL wants to take advantage of modern-day computing to enhance the effectiveness and scalability of this particular safety protocol. One way the sports league wants to leverage the power of computing to accomplish this involves machine learning and image processing algorithms to automate the detection and classification of dangerous helmet impacts that occur during games. Developing a program that will automate the process of reviewing footage and providing critical information on players' exposures to harmful impacts will increase the efficiency of the process and ultimately allow this safety feature to be implemented at a larger scale. Additional uses of the program include strategic analysis of plays and implementation of automated game analysis based on the movement and location of different players during the game.

1.2 Statement of Problem

Currently, the NFL spends many hours reviewing a small subset of available game footage to assess and obtain data on critical hits that may put certain players at risk. This inefficient analysis method limits the scope of this safety protocol and adds a large overhead cost for the league. By creating a program that automatically detects all 22 player helmets on the field and tracks these helmets consistently throughout the duration of the play, a solid foundation is made for automated tracking and analysis of the players' injuries.

1.3 Objective

To effectively contribute to the endeavors of the NFL to improve the scalability and speed of player safety processes, a high priority must be placed on accuracy and validity of our results. Adequately solving this problem means correctly identifying the 22 helmets in each video frame and keeping a consistent assignment throughout the play. Accomplishing this task to a high degree of accuracy will provide a critical baseline framework for the NFL to use for important data extraction and analysis for injury prevention. Therefore, our objective would be:

- To detect the helmet from the NFL dataset
- To use YOLOv5 object detection algorithm to detect the helmets and compare the result with the baseline helmet boxes provided.

1.4 Related Work

Machine learning methods have vast application in sport medicine, one of them being the identification and prediction of risk factors. A couple of recurring results stood out between all the studies. Young elite players were most vulnerable within most models because of the high physical performance and motor coordination. These studies were all flawed in that they suffered from imprecision due to the lack of available data. Wider spread adoption of machine learning techniques in large sports leagues could therefore yield very beneficial data not only for the sports leagues, but also for all others involved in sport analysis. There are infinite different ways of implementing machine learning methods to enhance the safety of sports; currently, tree-based models are the most popular in sports medicine because of their simplicity in development and interpretation. One important finding with this review was that approaches currently lack proper quality in terms of the methodology applied. One major recommendation would be acknowledging the dependencies between testing and training datasets and attempting to alleviate the issues associated with this.

2. Datasets

The dataset for the competition has been provided by the organizer. A vital component of understanding how to approach this problem is being familiar with the datasets that are provided. Each play has two different perspectives with two respective videos: an endzone view of the play and a sideline view of the play which have shown in Figure 1 and Figure 2 respectively.

2.1 Images

The images directory contains 9947 supporting photos of both the endzone views of the play and sideline views of the play that can be used for making a helmet detector for the train/test video frames. The training set videos are in train with corresponding labels in train_lables.csv, while the videos for prediction are in the test folder.

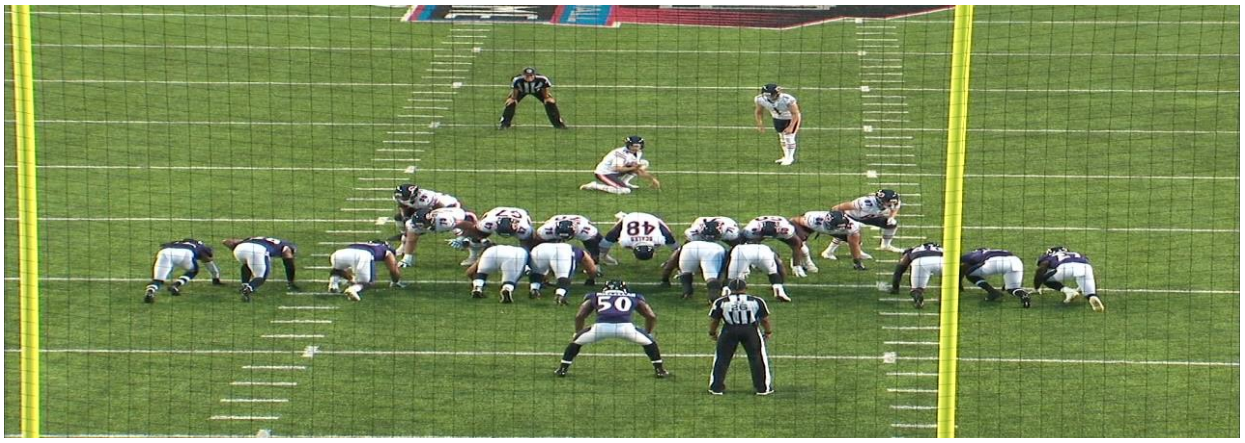


Figure 1: Endzone view of play

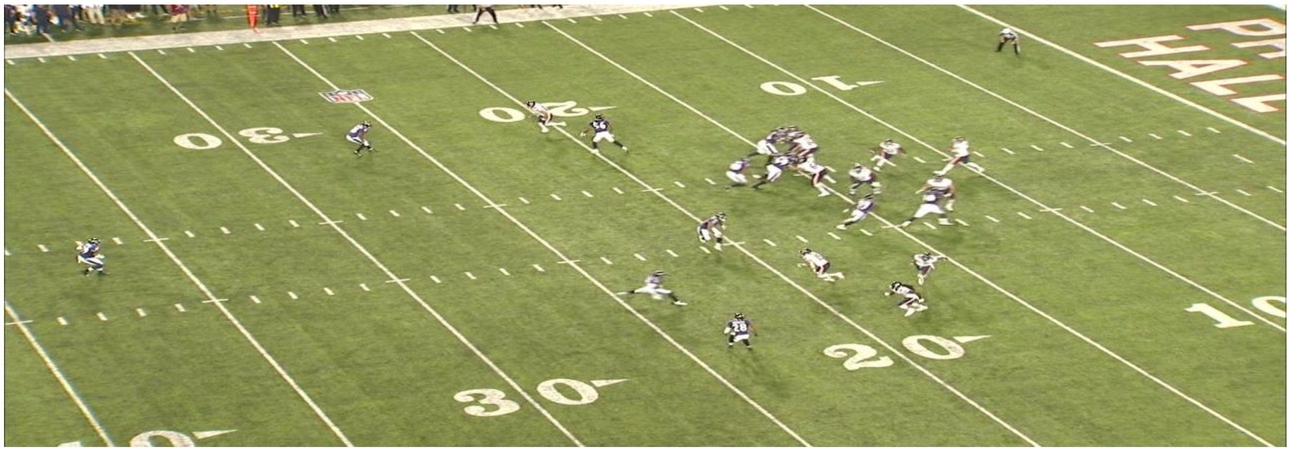


Figure 2: Sideline view of play

Figure 3 has been provided by Kaggle to provide a frame of reference for some of these player tracking variables.

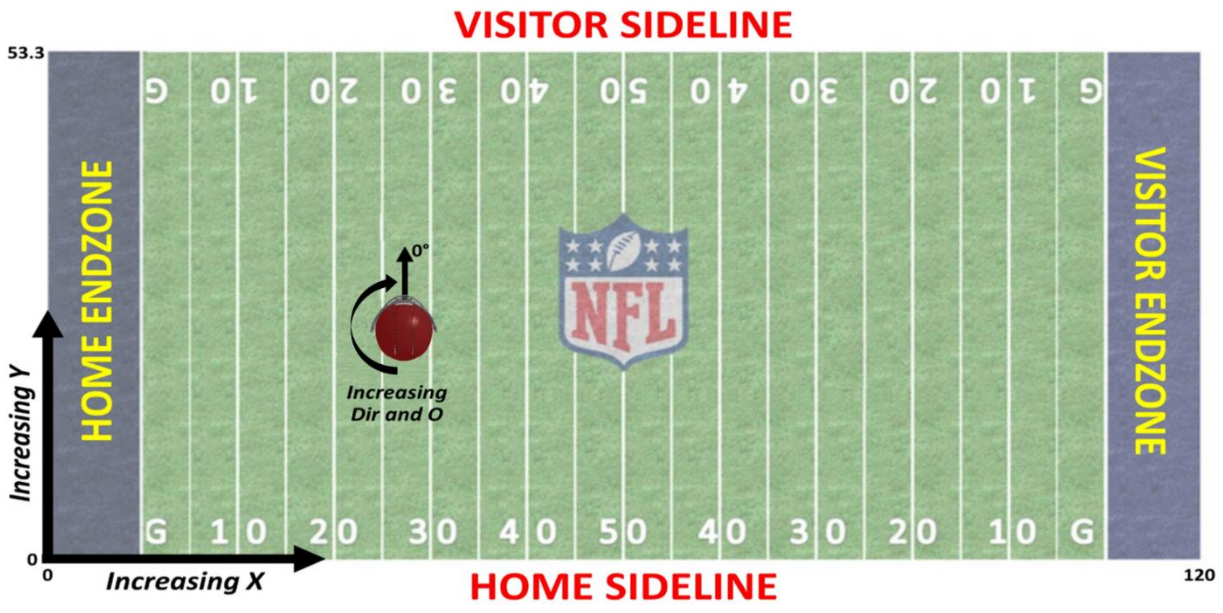
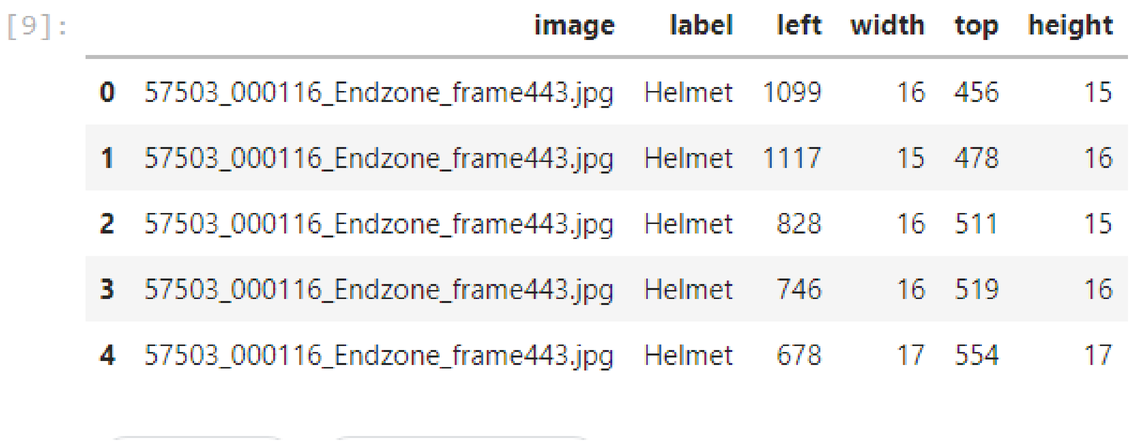


Figure 3: Depiction of football field with frame of reference

2.2 Image_labels.csv

The image_labels.csv file contains the attributes to correctly identify bounding boxes of the supporting images for a helmet detector.



[9]:

	image	label	left	width	top	height
0	57503_000116_Endzone_frame443.jpg	Helmet	1099	16	456	15
1	57503_000116_Endzone_frame443.jpg	Helmet	1117	15	478	16
2	57503_000116_Endzone_frame443.jpg	Helmet	828	16	511	15
3	57503_000116_Endzone_frame443.jpg	Helmet	746	16	519	16
4	57503_000116_Endzone_frame443.jpg	Helmet	678	17	554	17

- Image describes file name of the image
- Label describes the label type
- Left/width/top/height describes the bounding box of the label

2.3 train/test_baseline_helmets.csv

The train/test_baseline_helmets.csv Contains the predictions that were obtained by training on the images found in the images folder and labels from image_lables.csv. It's important to note that these are just baseline predictions from a model and are not perfect. The important fields in this file:

- Video shows the filename of the associated video.
- Frame shows the frame number for this play.
- left/width/top/height describes the predicted bounding box

2.4 Dataset preparation

Dataset preparation is the most important part of YOLOV5 training. To train a YOLOv5 model the following steps are needed

- Create train-validation set
- Create the necessary /dataset folder structure and add the images to it.
- Create the data.yaml file to train the model
- Create bounding box coordinates in the YOLO format

3. State-of-the-art Results

A baseline implementation to detect and track helmets was provided by the NFL Competition Organizer. They also provided us with a dataset of images showing helmets with labeled bounding boxes. The baseline helmet detection boxes for the training and test set were provided as well. Two different frames of endzone side with bounding box have been shown in figure 4 and figure 5.

We set out to improve upon this program by changing the underlying architecture of the program using YOLOV 5 model that can be used for helmet detection with high performance. The implementation and result of the state-of-the-art implementation can be found in section 1 of the project's Kaggle notebook.

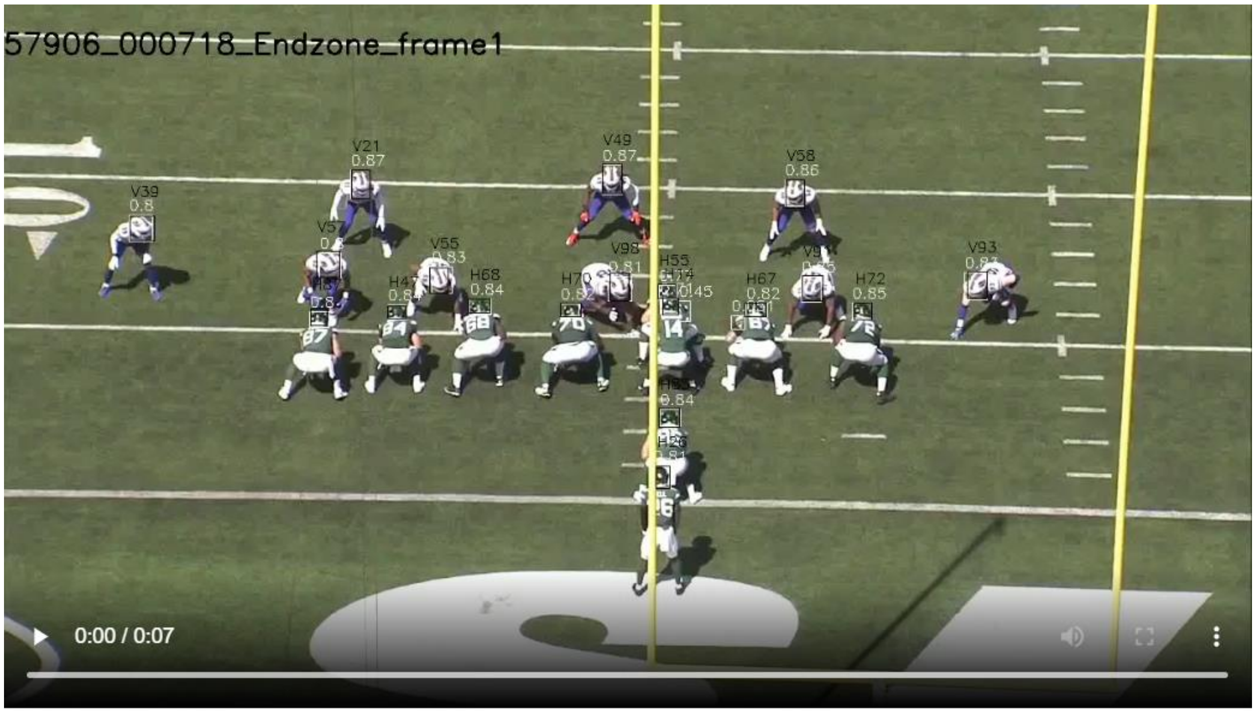


Figure 4: Frame number 1 of Endzone view

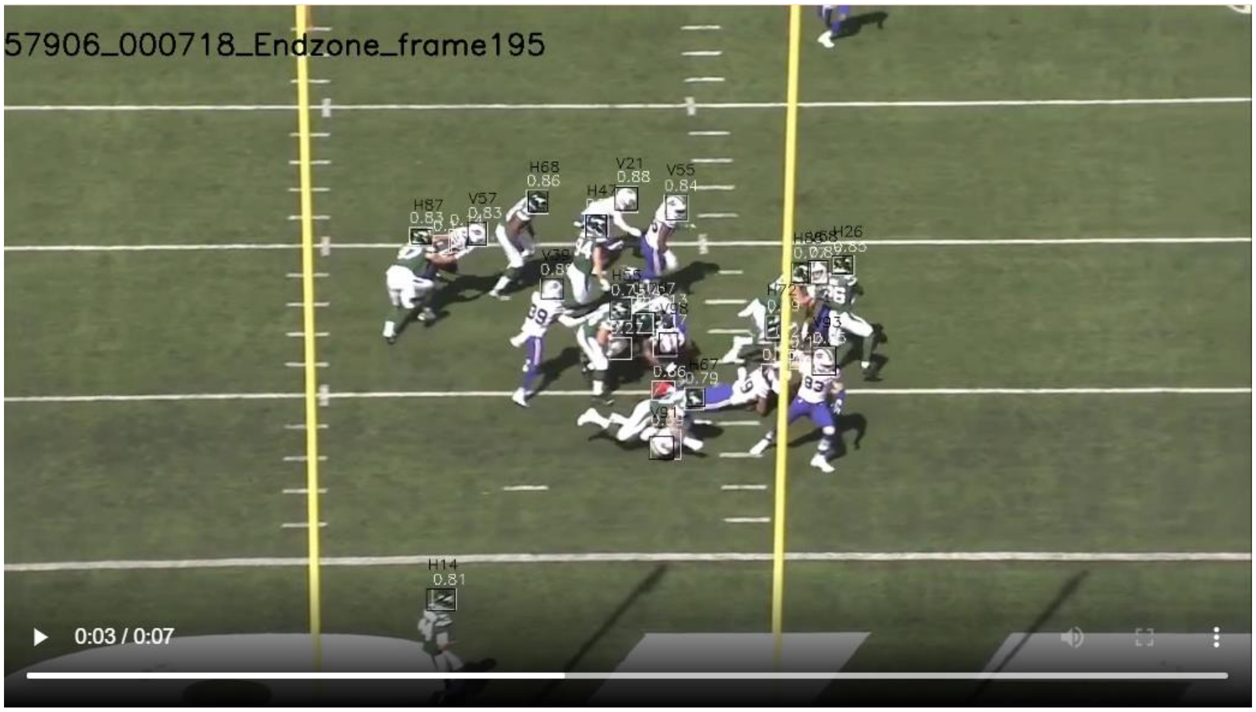
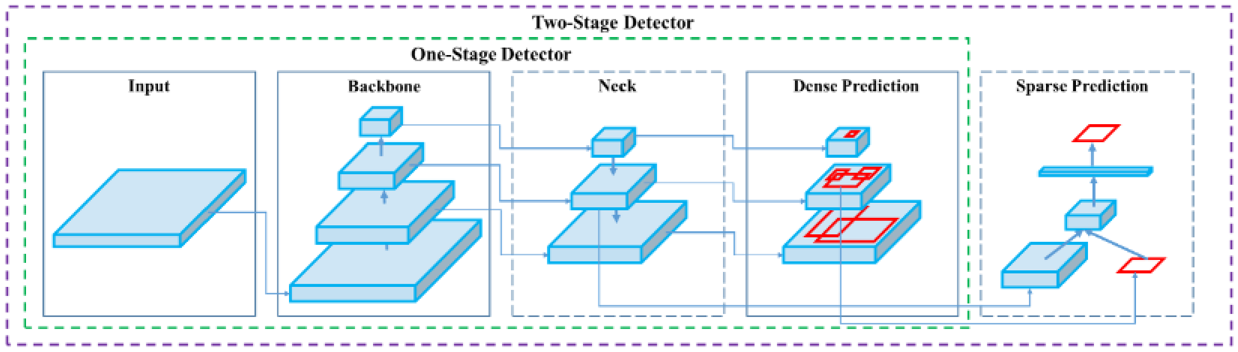


Figure 5: Frame number 196 of Endzone view

4. Proposed Neural Architecture

To improve upon the baseline implementation, an object detection algorithm using neural networks called YOLO (You only look once) is implemented. This framework only passes the training images through the CNN once and divides an input image into a grid of pixels, making each grid responsible for object detection within its boundaries. A convolutional neural network (CNN) is a deep learning algorithm that can take an image, assign weights and biases to objects in the image, and then, using these weights and biases, differentiate these objects within the image from one another. Specifically, YOLOv5, in which bounding boxes and confidence scores for those boxes are predicted, was implemented. The output of this algorithm involves center point values (vertical and horizontal), the width and height of a prediction box, and a confidence value to showcase the degree of confidence that the algorithm has for the box it predicts. The confidence for the prediction is equal to the intersection over union between the box predicted by the algorithm and the ground truth. Based on the PyTorch framework and developed for speed and lowcomputational cost, this architecture was ideal for our implementation to detect helmet objects in each video frame.



Input: { Image, Patches, Image Pyramid, … }

Backbone: { VGG16 [68], ResNet-50 [26], ResNeXt-101 [86], Darknet53 [63], … }

Neck: { FPN [44], PANet [49], Bi-FPN [77], … }

Head:

Dense Prediction: { RPN [64], YOLO [61, 62, 63], SSD [50], RetinaNet [45], FCOS [78], … }

Sparse Prediction: { Faster R-CNN [64], R-FCN [9], … }

General Object Detector will have a backbone for pre-training it and a head to predict classes and bounding boxes. The backbones can be running on GPU or CPU platforms. The head can be either one-stage (e.g., YOLO, SSD, RetinaNET) for dense prediction or two stages (e.g., Faster R-CNN) for the sparse prediction object detector. Recent object detectors have some layers (Nect) to collect feature maps, and it is between the backbone and the head.

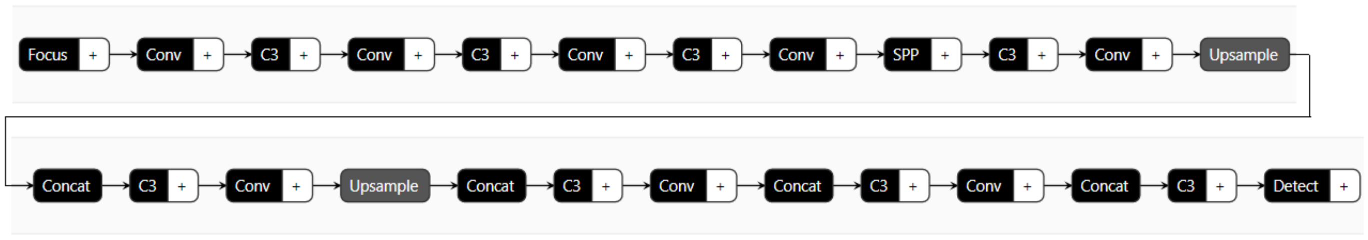


Figure 4: Model of Yolov5

5. Overall approach to the problem

In order to solve the problem of detection of the helmet, we need a lot of image data with the labels in it. The dataset is provided by the organizer in the Kaggle. We thought of using those images to get better detection. Our approach to find the better detection is to use the Yolov5 algorithm (described in section 4) with different parameters and trained with different dataset. We already have a large amount of data which is described in the dataset images. Then, we will split the dataset into training and validation dataset. After that, we will be passing those datasets through Yolov5 algorithm to train and validate.

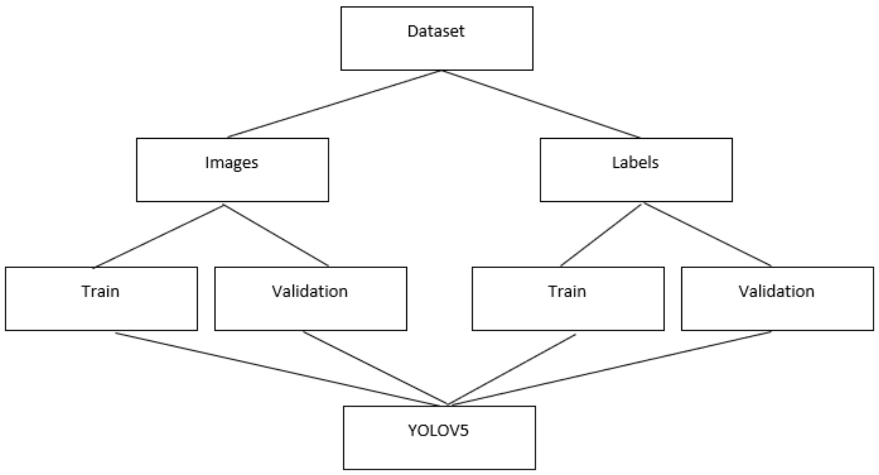


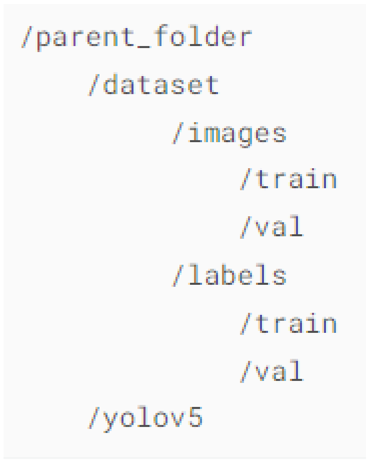
Figure 6: Basic Block Diagram of the approach

Our first experiment will be to train the data with the default parameters of the yolov5, and later we will compare the results by changing the hyperparameters of yolov5 algorithm. Our ideal output after changing the hyperparameters would be to get better confidence than the detected baseline data provided by the organizer. We will test it on a test video. We will try to look at the first and last frame to check if we got better results than the provided baseline.

6. Implementation Details

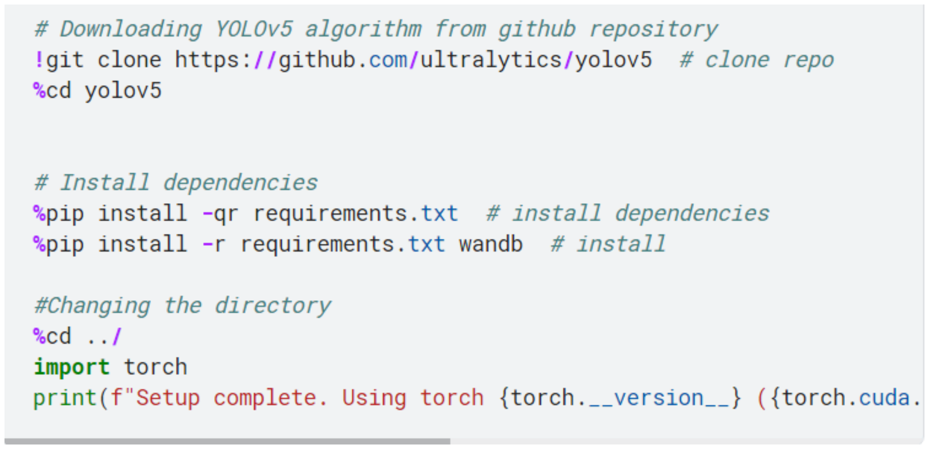
We have written the code in Python 3.8. Our program would run in the Kaggle notebook environment itself. We thought of choosing Kaggle notebook so that we don't have to extract the huge data to our local computer, and also it offers the GPU, TPU and CPU of its own to run the notebook there. It is more convenient to do that way. Since none of our group members had used Kaggle before, It took some time to figure out how the Kaggle notebook works. Eventually we were able to use the GPU provided by the Kaggle. Our selection of GPU is because the GPU is specifically made to do the neural network calculations. We had tried to use CPU in the initial phase but with the number of epochs increasing, the time taken to train was quite high. The following are the steps to implement our whole project:

1. Baseline Detection of Helmets
   We were given the box locations and the actual labels of the helmets. And also, the images in a video. Then we will put those boxes in a test video. We arbitrarily selected a test video (test/57906_000718_Endzone.mp4) to compare the result. We will compare the 1st and last frame of the video. Each video is around 7 seconds. After we got the confidence level for the test video, we will use Yolov5 to train the model with the given images to get better results. We have used CV2 (OpenCV) library to play the video from frames. And only the 65% of the screen will be displayed because that will be enough to show every player/helmet from the video.
2. YOLOv5 Implementation
   i) Setup – first the appropriate folder structure was needed for yolo5 displayed below:



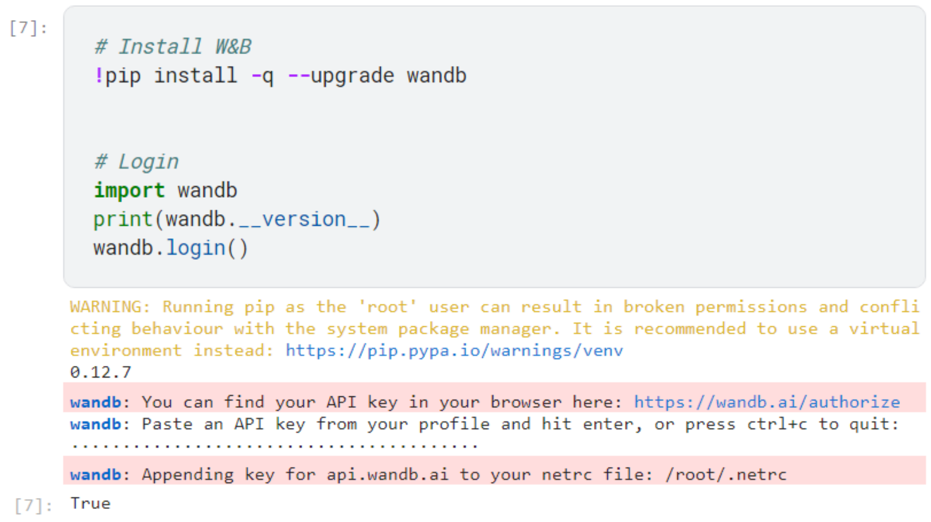
```
/parent_folder
    /dataset
         /images
             /train
             /val
         /labels
             /train
             /val
    /yolov5
```

   ii) Required Yolo and required dependencies were installed.



```
# Downloading YOLOv5 algorithm from github repository
!git clone https://github.com/ultralytics/yolov5  # clone repo
%cd yolov5


# Install dependencies
%pip install -qr requirements.txt  # install dependencies
%pip install -r requirements.txt wandb  # install

#Changing the directory
%cd ../
import torch
print(f"Setup complete. Using torch {torch.__version__} ({torch.cuda.
```

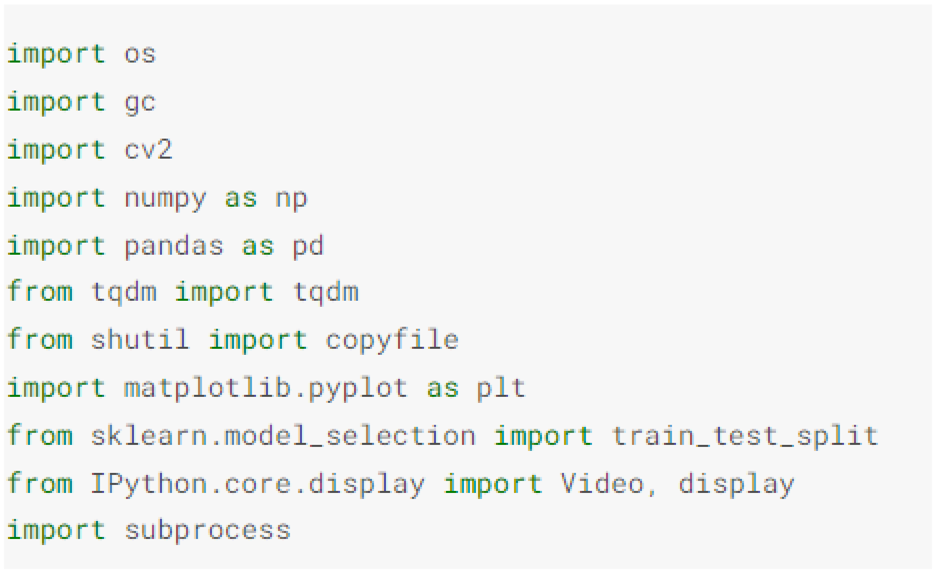
iii) Weights & Biases Setup: We installed wandb module which will be used to show the weights and biases and their training results. Each time you the command, you will get an API asking for the code, you need to go the link provided, copy/paste the code in the notebook, and press enter. It should display True to get the updates. Weights and biases are basically a tool to look at the metrices of that run. It is a separate feature for YOLOv5 algorithm. It will provide us with the performance metrics.



```
[7]:
# Install W&B
!pip install -q --upgrade wandb


# Login
import wandb
print(wandb.__version__)
wandb.login()
```

```
WARNING: Running pip as the 'root' user can result in broken permissions and confli
cting behaviour with the system package manager. It is recommended to use a virtual
environment instead: https://pip.pypa.io/warnings/venv
0.12.7
wandb: You can find your API key in your browser here: https://wandb.ai/authorize
wandb: Paste an API key from your profile and hit enter, or press ctrl+c to quit:
··································
wandb: Appending key for api.wandb.ai to your netrc file: /root/.netrc
[7]: True
```

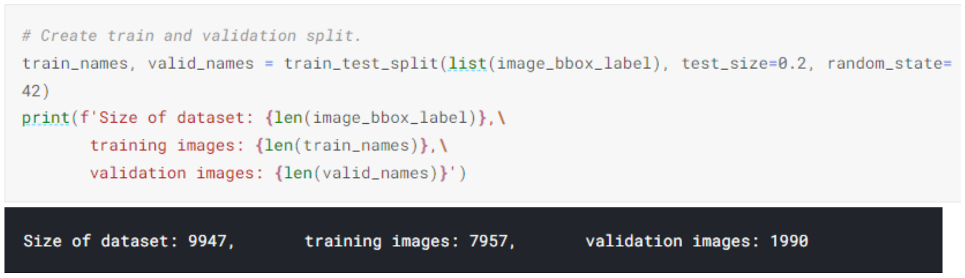
iv) Importing the required libraries: We have imported OS for the operating system operations, GC for the garbage collector, cv2 for the computer vision application such as playing video, changing video to frame, numpy for the numeraical operations, pandas for the data exploration and analysis, tqdm for the smart progress bar output, matplotlib for plotting, sklearn for the train_test_split function.



```
import os
import gc
import cv2
import numpy as np
import pandas as pd
from tqdm import tqdm
from shutil import copyfile
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from IPython.core.display import Video, display
import subprocess
```

v) Setting up the hyper-parameters: we have set up the batch size and epochs as the main hyperparameters. We are planning to use different epochs to see the differences between the epochs. As a default, we will make batch size equal to 16 and epochs as 1 so that it does not take a lot of time to run. But we will increase the epochs to show in the report. But in the code that we submit, the value will be 1 so that reader can run the file within limited time.

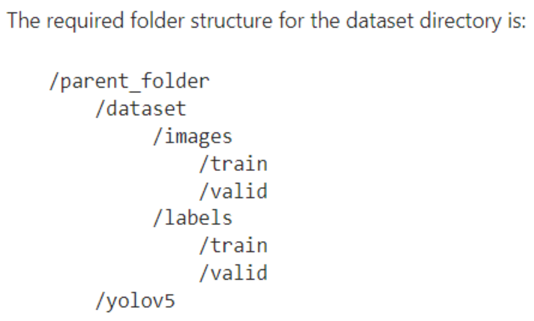
vii) Training and validation split -



```
# Create train and validation split.
train_names, valid_names = train_test_split(list(image_bbox_label), test_size=0.2, random_state=
42)
print(f'Size of dataset: {len(image_bbox_label)},\
       training images: {len(train_names)},\
       validation images: {len(valid_names)}')
```

```
Size of dataset: 9947,       training images: 7957,       validation images: 1990
```

viii) Making directories and setting up the folder as required for the YOLOv5 to run.

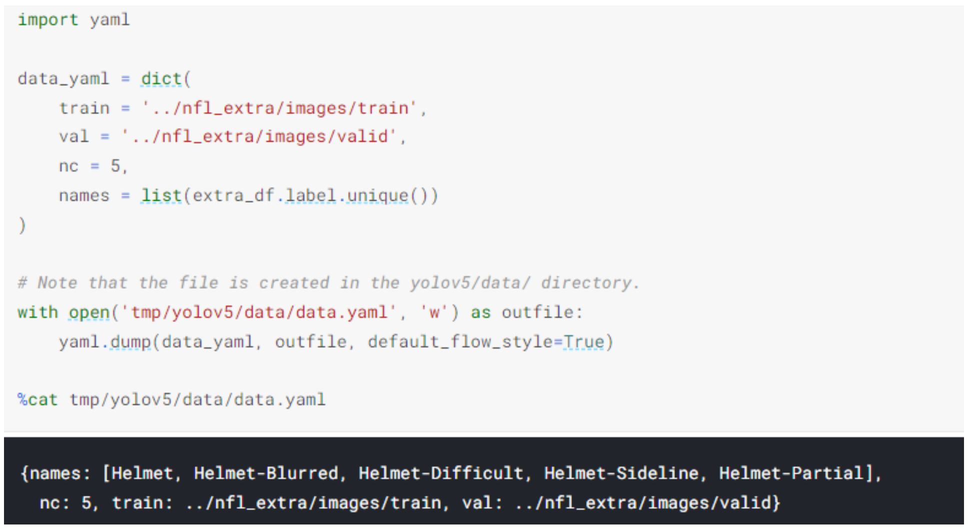


The required folder structure for the dataset directory is:

```
/parent_folder
    /dataset
         /images
             /train
             /valid
         /labels
             /train
             /valid
    /yolov5
```

We will make this kind of folder map.

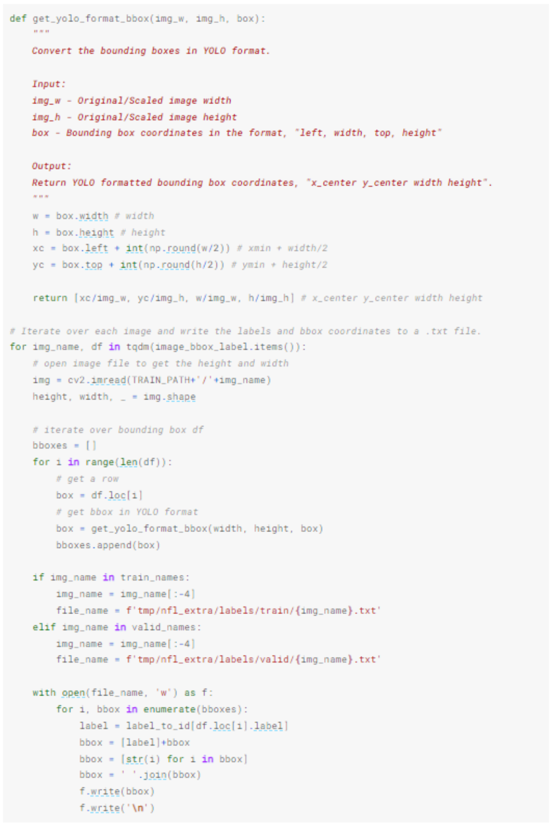
viii) Setup the YAML files for training: To train a YOLO-v5 model, we need to have two YAML files. One is to specify where our training and validation data is, the number of classes that we want to detect, and the names corresponding to those classes. Second one is the configuration. But in this case, we are just using data.yaml file of the default.



```
import yaml

data_yaml = dict(
    train = '../nfl_extra/images/train',
    val = '../nfl_extra/images/valid',
    nc = 5,
    names = list(extra_df.label.unique())
)

# Note that the file is created in the yolov5/data/ directory.
with open('tmp/yolov5/data/data.yaml', 'w') as outfile:
    yaml.dump(data_yaml, outfile, default_flow_style=True)

%cat tmp/yolov5/data/data.yaml
```

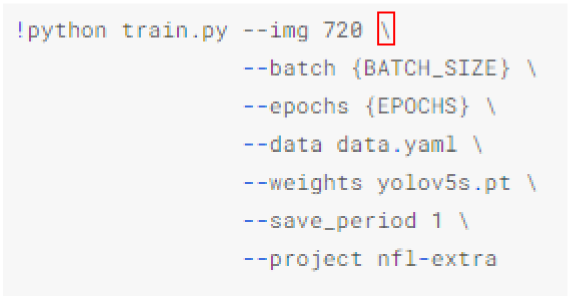
```
{names: [Helmet, Helmet-Blurred, Helmet-Difficult, Helmet-Sideline, Helmet-Partial],
  nc: 5, train: ../nfl_extra/images/train, val: ../nfl_extra/images/valid}
```

ix) Convert bounding boxes to YOLO format – done iteratively over every image with labels and image coordinates written in a new file.

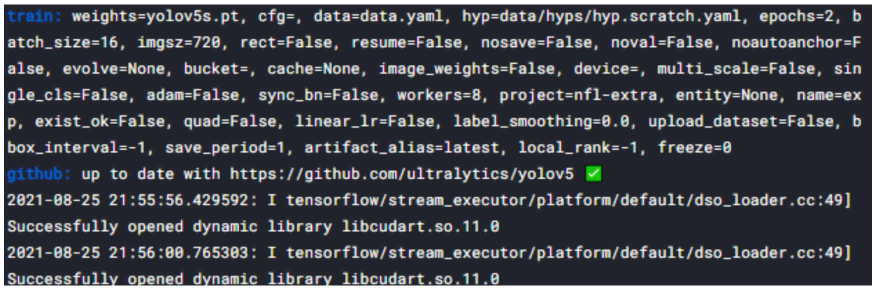


```python
def get_yolo_format_bbox(img_w, img_h, box):
    """
    Convert the bounding boxes in YOLO format.

    Input:
    img_w - Original/Scaled image width
    img_h - Original/Scaled image height
    box - Bounding box coordinates in the format, "left, width, top, height"

    Output:
    Return YOLO formatted bounding box coordinates, "x_center y_center width height".
    """
    w = box.width # width
    h = box.height # height
    xc = box.left + int(np.round(w/2)) # xmin + width/2
    yc = box.top + int(np.round(h/2)) # ymin + height/2

    return [xc/img_w, yc/img_h, w/img_w, h/img_h] # x_center y_center width height

# Iterate over each image and write the labels and bbox coordinates to a .txt file.
for img_name, df in tqdm(image_bbox_label.items()):
    # open image file to get the height and width
    img = cv2.imread(TRAIN_PATH+'/'+img_name)
    height, width, _ = img.shape

    # iterate over bounding box df
    bboxes = []
    for i in range(len(df)):
        # get a row
        box = df.loc[i]
        # get bbox in YOLO format
        box = get_yolo_format_bbox(width, height, box)
        bboxes.append(box)

    if img_name in train_names:
        img_name = img_name[:-4]
        file_name = f'tmp/nfl_extra/labels/train/{img_name}.txt'
    elif img_name in valid_names:
        img_name = img_name[:-4]
        file_name = f'tmp/nfl_extra/labels/valid/{img_name}.txt'

    with open(file_name, 'w') as f:
        for i, bbox in enumerate(bboxes):
            label = label_to_id[df.loc[i].label]
            bbox = [label]+bbox
            bbox = [str(i) for i in bbox]
            bbox = ' '.join(bbox)
            f.write(bbox)
            f.write('\n')
```

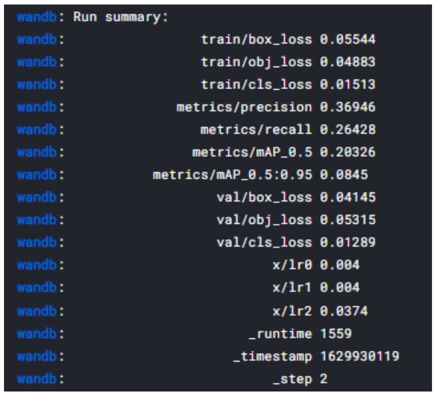
x) Train the model: There are multiple hyperparameters that we can specify, which are: img (input image size), batch (batch size), epochs (number of training epochs), data (YAML file), cfg (model configuration), weights (specify a custom path to weights), name (result names), nosave (only save the final checkpoint), cache (cache images for faster training). But in our case, we have just used the img, batch, epochs, data, save-period, and project.



```
!python train.py --img 720 \
                 --batch {BATCH_SIZE} \
                 --epochs {EPOCHS} \
                 --data data.yaml \
                 --weights yolov5s.pt \
                 --save_period 1 \
                 --project nfl-extra
```



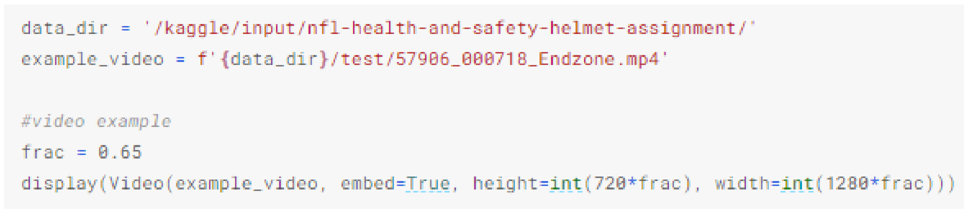
```
train: weights=yolov5s.pt, cfg=, data=data.yaml, hyp=data/hyps/hyp.scratch.yaml, epochs=2, b
atch_size=16, imgsz=720, rect=False, resume=False, nosave=False, noval=False, noautoanchor=F
alse, evolve=None, bucket=, cache=None, image_weights=False, device=, multi_scale=False, sin
gle_cls=False, adam=False, sync_bn=False, workers=8, project=nfl-extra, entity=None, name=ex
p, exist_ok=False, quad=False, linear_lr=False, label_smoothing=0.0, upload_dataset=False, b
box_interval=-1, save_period=1, artifact_alias=latest, local_rank=-1, freeze=0
github: up to date with https://github.com/ultralytics/yolov5 ✅
2021-08-25 21:55:56.429592: I tensorflow/stream_executor/platform/default/dso_loader.cc:49]
Successfully opened dynamic library libcudart.so.11.0
2021-08-25 21:56:00.765303: I tensorflow/stream_executor/platform/default/dso_loader.cc:49]
Successfully opened dynamic library libcudart.so.11.0
```



```
wandb: Run summary:
wandb:                    train/box_loss 0.05544
wandb:                    train/obj_loss 0.04883
wandb:                    train/cls_loss 0.01513
wandb:                 metrics/precision 0.36946
wandb:                    metrics/recall 0.26428
wandb:                   metrics/mAP_0.5 0.20326
wandb:              metrics/mAP_0.5:0.95 0.0845
wandb:                      val/box_loss 0.04145
wandb:                      val/obj_loss 0.05315
wandb:                      val/cls_loss 0.01289
wandb:                             x/lr0 0.004
wandb:                             x/lr1 0.004
wandb:                             x/lr2 0.0374
wandb:                          _runtime 1559
wandb:                        _timestamp 1629930119
wandb:                             _step 2
```

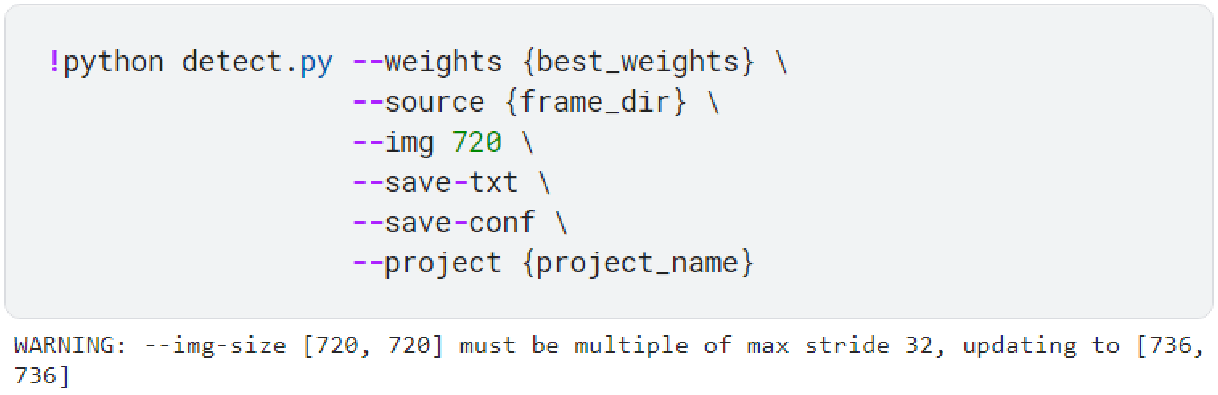
3) Testing it on the video.

i) We got the best weights from the training. Now we will use those weights to detect the helmets in the test video.



```
data_dir = '/kaggle/input/nfl-health-and-safety-helmet-assignment/'
example_video = f'{data_dir}/test/57906_000718_Endzone.mp4'

#video example
frac = 0.65
display(Video(example_video, embed=True, height=int(720*frac), width=int(1280*frac)))
```

ii) Helmet Detection on test video:



```
!python detect.py --weights {best_weights} \
                  --source {frame_dir} \
                  --img 720 \
                  --save-txt \
                  --save-conf \
                  --project {project_name}
```

```
WARNING: --img-size [720, 720] must be multiple of max stride 32, updating to [736, 736]
```

Then, it will provide us the detected values for each frame. Then, we will try to put that in each frame of the video and play that video using CV2 library. We will display only the 65% percent of the original video resolution.

7. Experiment and Results

7.1 Hyper-parameterizing YOLOv5 Model

The following section elaborates on results found from testing our program across a different number of epochs. Since a lot of time is required to run this on larger epochs, we are only able to test up to 10 epochs. Before diving into this section, it's important to understand the different classes (Helmet, Helmet-Blurred, Helmet-Difficult, Helmet-Sideline, Helmet-Partial). Each of these represent a classification of any given helmet at a certain frame in time. Inherently, the classifications that fall under "partial", "blurred", etc. are less detectable by the algorithm and will perform worse. For this reason, and because most helmets fall under the Helmet class, the Helmet class is the class that will be prioritized when analyzing the F1 score and other metrics. Also, the numbers next to each detected helmet show the algorithm's confidence of that detection. Thus, a higher confidence means a better result. We are looking to tune our hyperparameters in a way that maximizes both the confidence and the F1 score as to not overfit to our training data. A summary of important data points is shown in this report and full access to graphs are available at a link at the bottom of each following section.

7.1.1 Epoch = 1

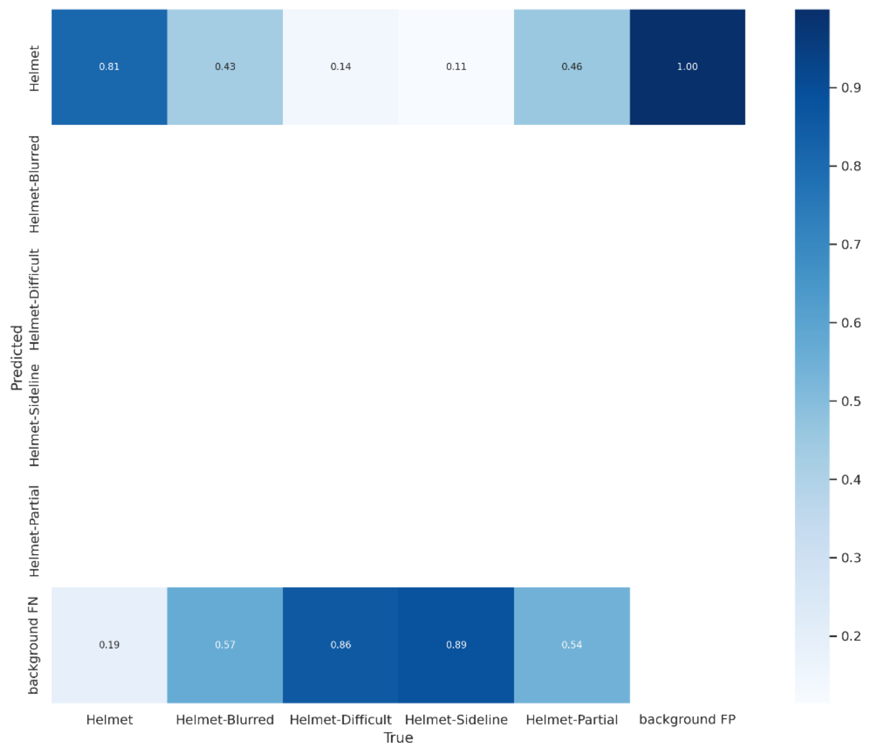


Figure 7: Confusion Matrix for Epoch=1

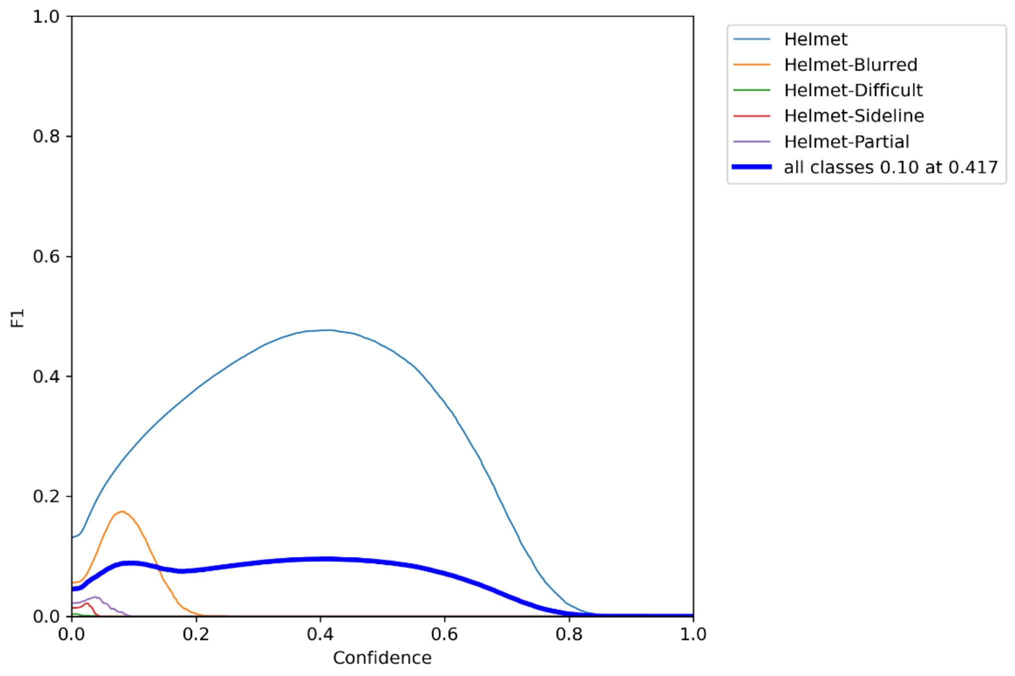


Figure 8: F1 score for Epoch=1

When epoch is set to 1, we see that a confidence score of .417 maximizes our F1 score for all helmet types in the frames. We also see that for only the helmet class an F1 score of about .5 is obtained with a confidence level of .45. Since epoch is only 1, we expect a low maximized F1 score with this run.

A full description of the data for this run can be found here: https://wandb.ai/staylort/nfl-extra/runs/17y2nlqm?workspace=user-staylort.

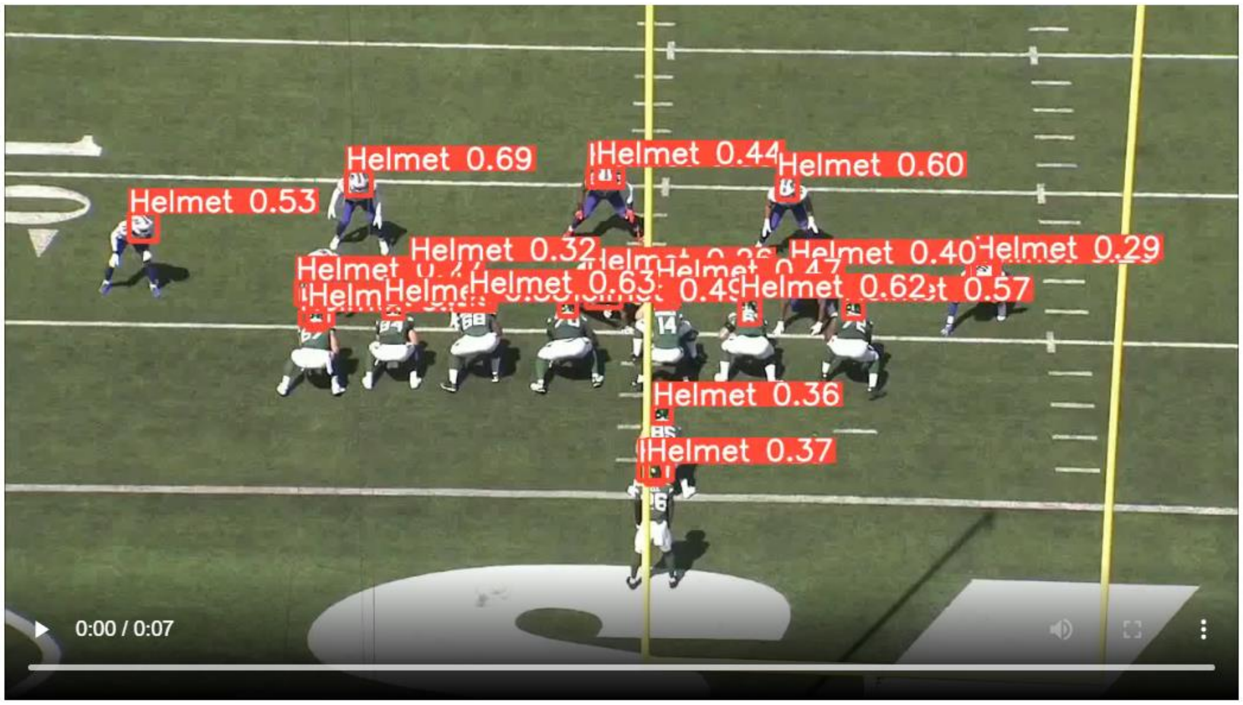


Figure 9: First frame with epoch = 1

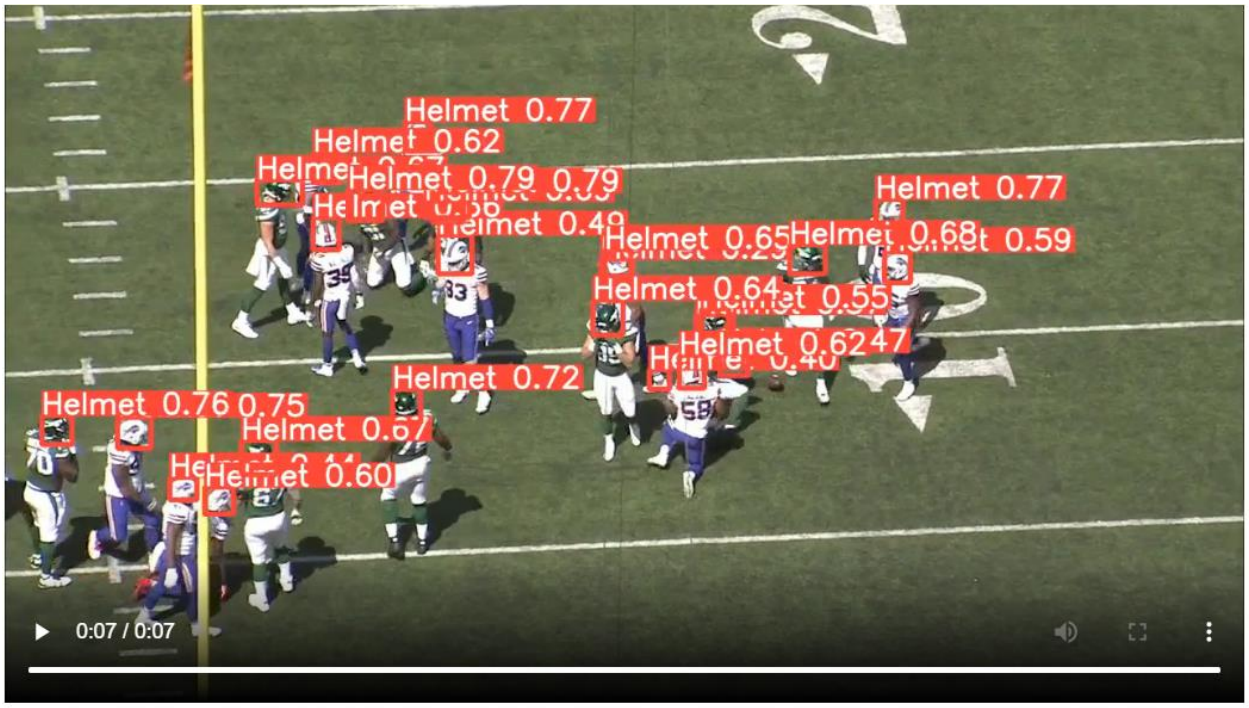


Figure 1: Last frame with epoch 1

From the first epoch, we can see that the confidence values are quite low. But our hypothesis is that it will increase with the increase in number of epochs.

7.1.2 Epoch = 5

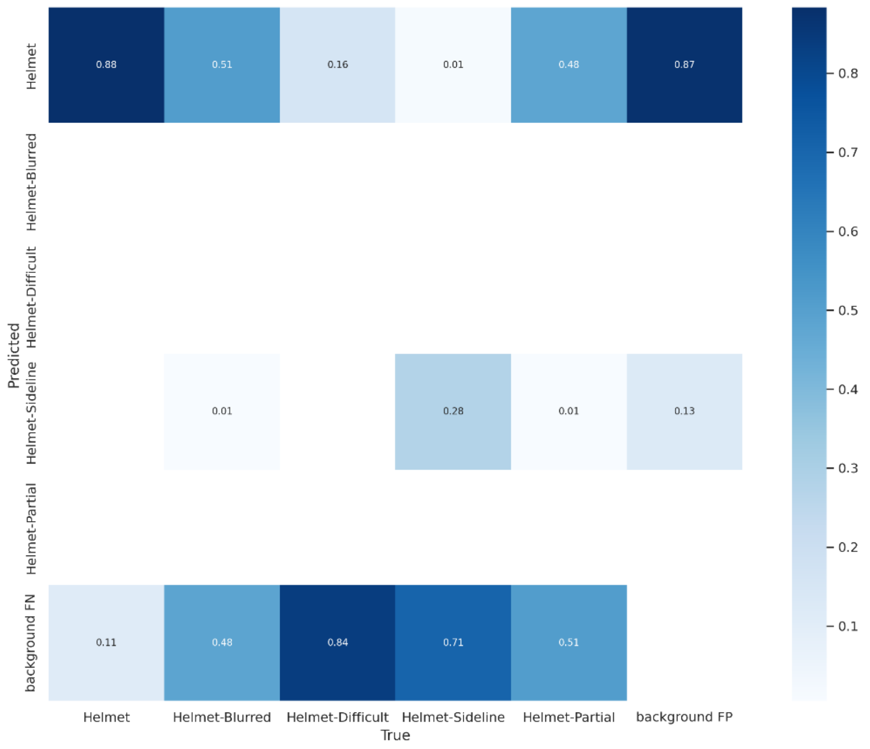


Figure 9: Confusion Matrix for Epoch=5

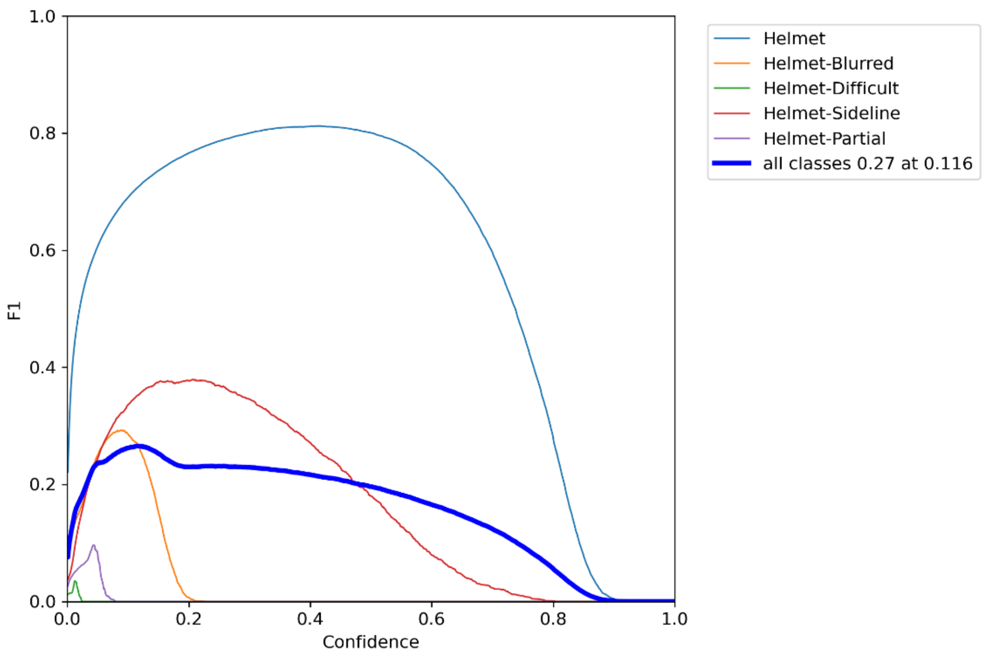


Figure 10: F1 Curve for Epoch=5

When epoch is set to 5, we see a dramatic increase in the F1 score and confidence levels present in the classes. The data shows that, including all classes, an F1 score of .27 is maximized at a confidence level of .116. Moreover, a confidence level of about .57 maximizes an F1 score of .8. This is a dramatic improvement from the previous epoch=1 result.

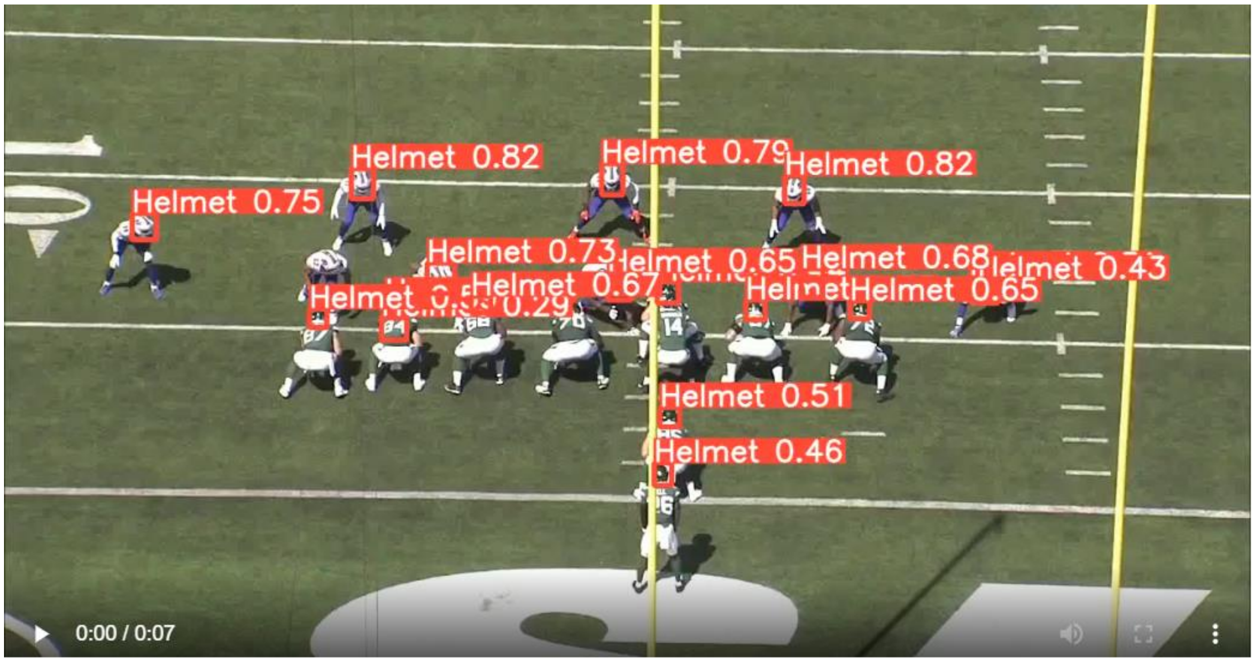


Figure 11: Start Frame with Epoch=5

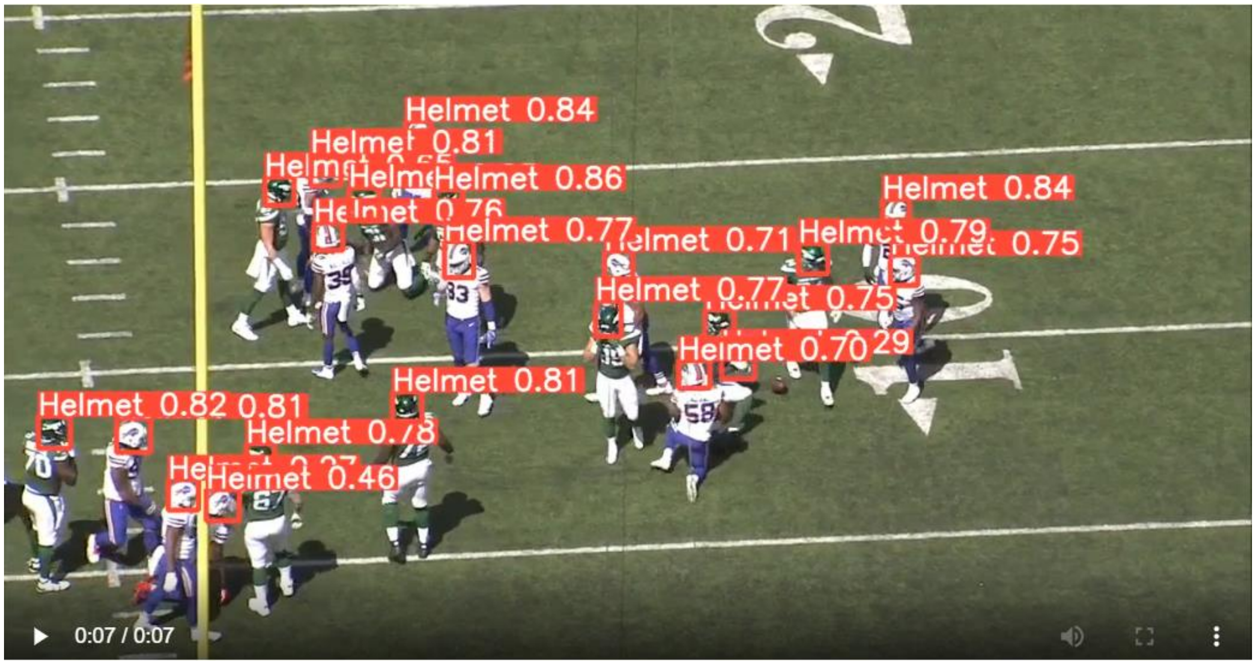
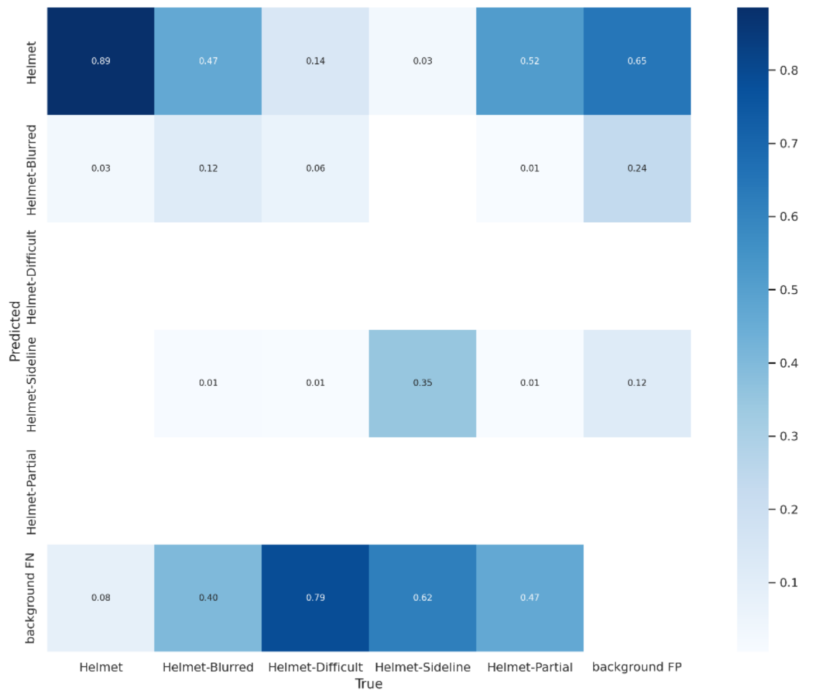


Figure 12: End Frame with Epoch=5

We do in fact see a substantial increase in the confidence levels of this algorithm after changing the epoch to 5.

A full description of the results for this run can be found here: https://wandb.ai/staylort/nfl-extra/runs/4nn6asvk?workspace=user-staylort.

7.1.3 Epoch = 10



2

Figure 13: Confusion Matrix for Epoch=10

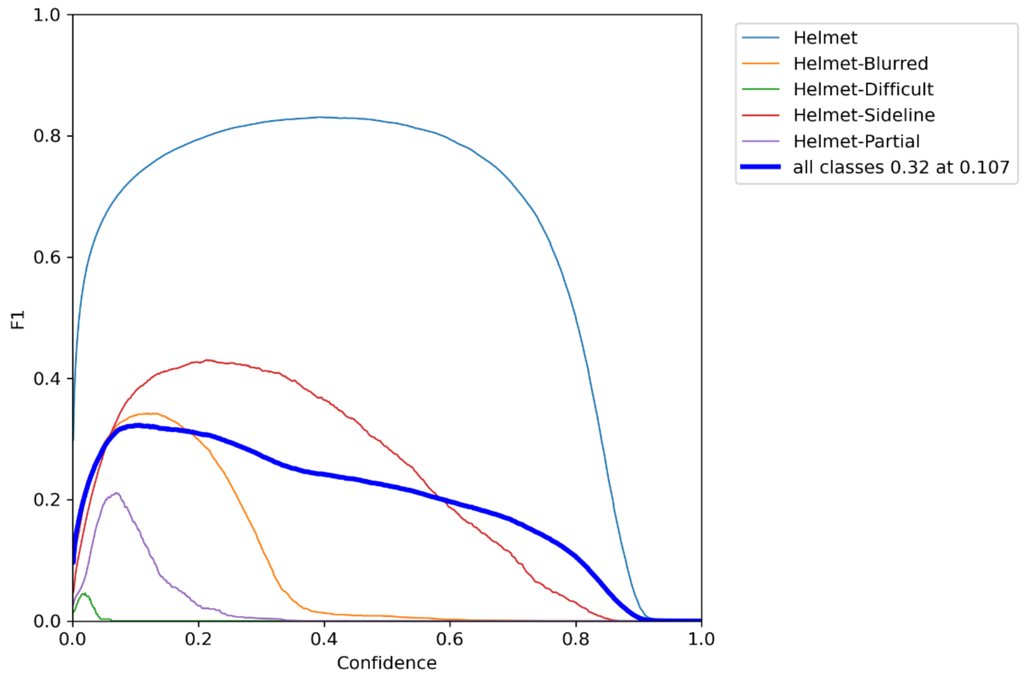


Figure 14: F1 Score for Epoch=10

With the epoch set at 10, we see a moderate increase in the F1 scores for the classes, especially those that are not clearly detectable. It is also obvious that there is a better result for the Helmet class, but the improvement is much less than what we saw moving from 1 to 5 epochs.

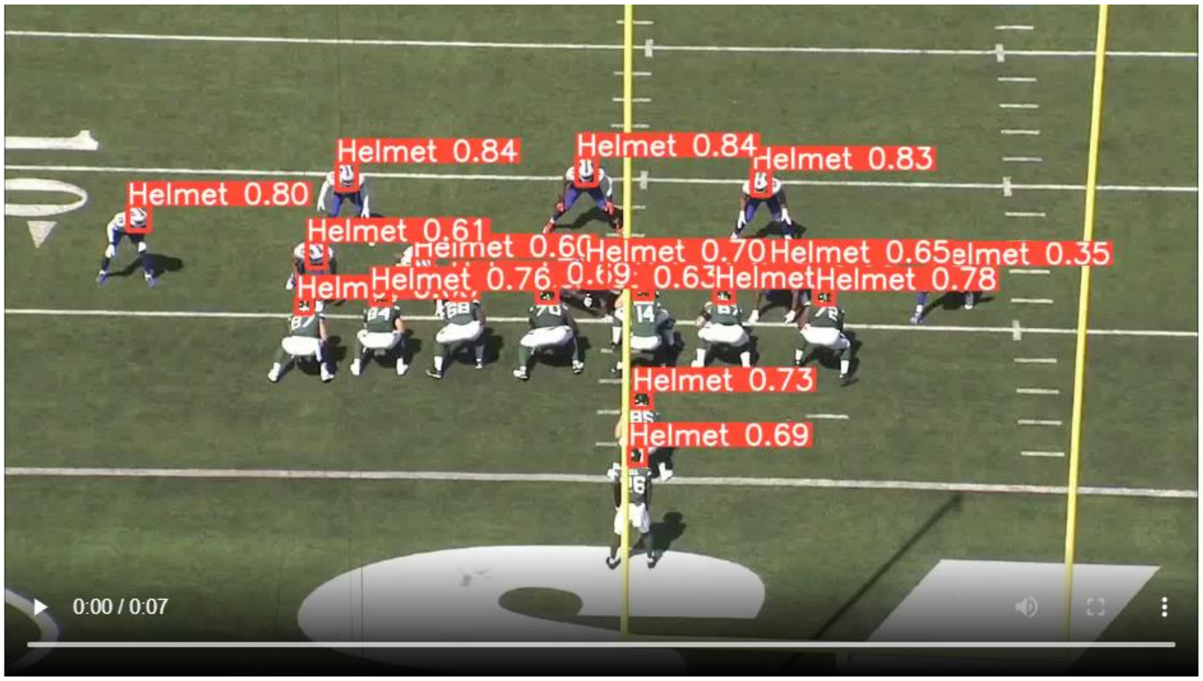


Figure 15: First frame for Epoch=10

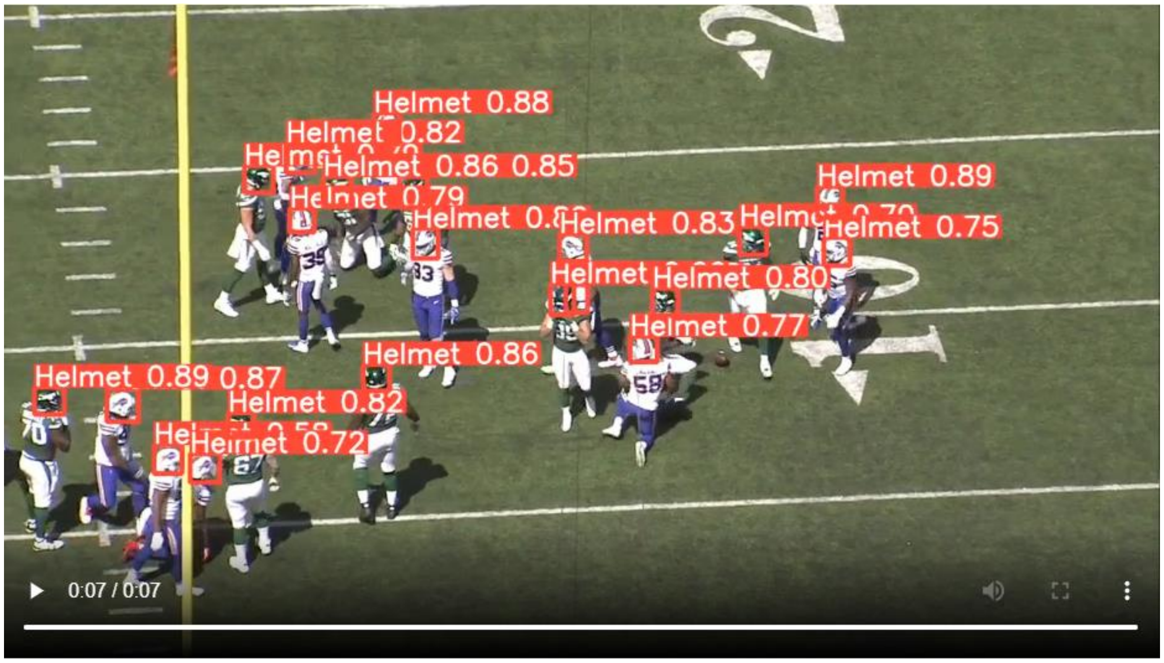


Figure 16: Final frame for Epoch=10

The confidence values for each of the detected helmets have increased, signaling the algorithm's improvement on this data with the increasing number of epochs.

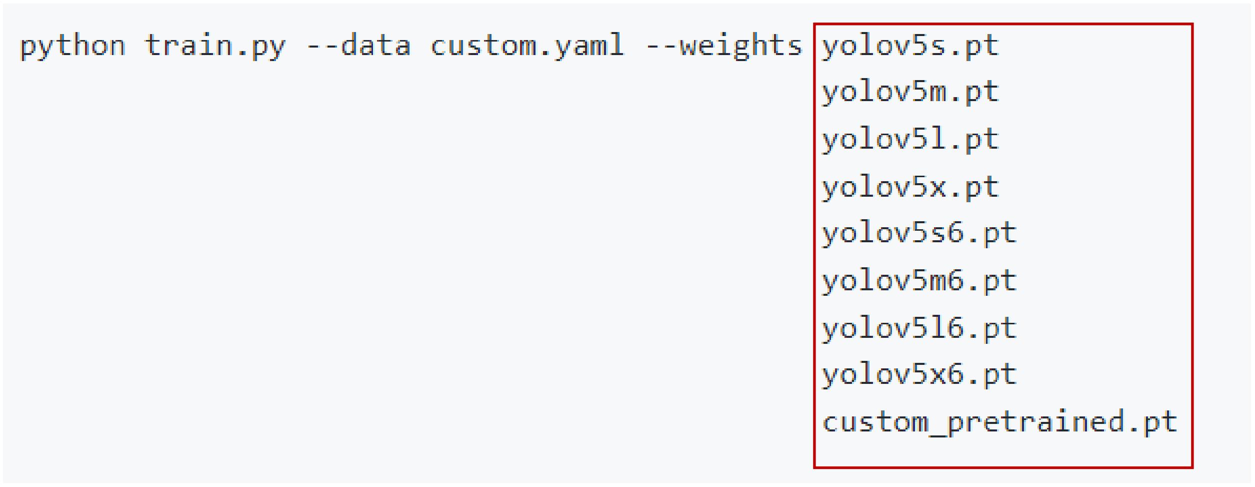
A full description of the results for this run can be found here: https://wandb.ai/staylort/nfl-extra/runs/3agght4y?workspace=user-staylort.

7.2 YOLOv5 versus State-of-the-Art

Comparing our results from the YOLOv5 algorithm with an epoch of just 10, we see comparable confidence values compared to the initial baseline results. With more time and/or more computer power, using the YOLOv5 method with more epochs will improve the algorithm significantly since we already see such a vast improvement between 1 epoch up to 10 epochs.

7.3 Future Prospects

While training, the pretrained weights were from YOLOv5s. Due to the time constraints for the training of the model, we didn't try all of the weights. But it can be tested and can be expected to get better performance. For example, here are other initial weight options:



```
python train.py --data custom.yaml --weights yolov5s.pt
                                             yolov5m.pt
                                             yolov5l.pt
                                             yolov5x.pt
                                             yolov5s6.pt
                                             yolov5m6.pt
                                             yolov5l6.pt
                                             yolov5x6.pt
                                             custom_pretrained.pt
```

If we have a large dataset, it is preferable to use the different configurations to see if the accuracy changes or not. In our case, we just used data.yaml file. But we can change it to a different configuration as shown in the figure below.

```
python train.py --data custom.yaml --weights '' --cfg yolov5s.yaml
                                                      yolov5m.yaml
                                                      yolov5l.yaml
                                                      yolov5x.yaml
                                                      yolov5s6.yaml
                                                      yolov5m6.yaml
                                                      yolov5l6.yaml
                                                      yolov5x6.yaml
```

8. Conclusion

To facilitate the injury prevention efforts of the NFL, we sought to develop an improved algorithm for detecting helmets in football play footage. To accomplish this, we researched different object detection models of machine learning and decided to use YOLOv5 because it offers a light and fast implementation without lowering the caliber of the results. We had trouble testing the hyperparameters of this model to its full potential due to limitations in computing power and time; however, we found that increasing the epoch size from 1 to 10 significantly increased the confidence and F1 score of the model. In other words, by increasing the number of epochs ran, this algorithm tries to find the optimal cost. To prevent from overfitting, we look at the F1 score on a plot with the confidence and try to maximize both of these variables to find the best result. When analyzing this F1-confidence plot, it is important to note that there are several classes accounted for, most importantly of which is the Helmet class because it makes up the overwhelming majority of the helmets at any given frame.

It is important to mention that the training data for models have a significant role in the limitations of the trained model. One of the limitations is in the application towards data utilization. Our algorithm is designed to detect individual helmets without accounting for their interaction during the actual game. The data provided was purposefully pulled from still images and video segments that are best able to train models to give a learned definition of a helmet, while during the game, different angles and various positions of people could result in either missed positives, thus providing incomplete medical data, or provide false positives, thus increasing the variation within theories or medical conclusions.

References

[1]. Van Eetvelde H, Mendonça LD, Ley C, Seil R, Tischer T. Machine learning methods in sport injury prediction and prevention: a systematic review. Journal of experimental orthopaedics. 2021 Dec;8(1):1-5.

[2]. https://github.com/ultralytics/yolov5

[3]. https://medium.com/analytics-vidhya/object-detection-algorithm-yolo-v5-architecture-89e0a35472ef

[4]. https://www.kaggle.com/profahmadhussein2010/nfl-baseline-simple-helmet-mapping

[5]. https://cocodataset.org/#home

[6]. https://github.com/ultralytics/yolov5/wiki/Tips-for-Best-Training-Results

[7] https://medium.com/axinc-ai/yolov5-the-latest-model-for-object-detection-b13320ec516b

[8] https://towardsdatascience.com/yolo-you-only-look-once-real-time-object-detection-explained-492dc9230006

[9] https://towardsdatascience.com/a-single-number-metric-for-evaluating-object-detection-models-c97f4a98616d

[10] https://medium.com/augmented-startups/yolov5-controversy-is-yolov5-real-20e048bebb08Th

[11] https://analyticsindiamag.com/top-8-algorithms-for-object-detection/