Machine vision in hay bale production

by

Benjamin Vail

B.S, University of Kansas, 2013

A THESIS

submitted in partial fulfillment of the requirements for the degree.

MASTER OF SCIENCE

Carl and Melinda Helwig Department of Biological and Agricultural Engineering
Carl R. Ice College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2024

Approved by:

Major Professor
Edwin Brokesh

Copyright

Abstract

The goal of this project is to develop a system capable of real-time detection, pass/fail classification, and location tracking of large square hay bales under field conditions.

A review of past and current methods of object detection was carried out. This led to the selection of the YOLO family of detectors, due to its relatively high precision, recall and mAP capabilities while still being able to maintain 10 to 15 fps on a Jetson Nano during detection.

A dataset with over 6000 images of both good and bad hay bales was collected. Unfortunately, the dataset developed a class imbalance, with more good bale images than bad bales. This caused the bad bale class to over train and the good bale class to under train, severely impacting precision, and recall. To correct this imbalance, three different methods of image generation were tested, PFGM++, DCGAN, and Least Squares GAN. The methods were tested against two baselines from the original imbalanced data, one from just the data without any pre-training and the other with pre-training on Microsoft's MSCOCO multiclass image dataset, commonly used a baseline by the computer vision community. The results from this showed a clear improvement in model quality for the PFGM++, a slight improvement was seen with the Least Squares GAN, but the results from the DCGAN were mixed. The results also showed a clear improvement when using pre-training on the MSCOCO dataset over the from scratch training.

For the prototype, the system would need to be small but still powerful enough to run real-time detection. The Nvidia Jetson Nano was selected to be the main computing platform for the system. Using a YOL0V8n model, the smallest parameter profile, the Jetson Nano can maintain 15fps even with multiple detected objects in camera view. The Luxonis Oak-D pro POE camera was chosen to carry out the detection and provide depth estimation of the bales. With IR vision, IR laser active stereo vision, and an IP-67 rating, it is a good choice for use on agricultural equipment that might be exposed to the elements and can also support night-time baling operations.

Results from testing showed that the system can detect good and bad bales. However, changes in bale material such as corn stover vs alfalfa and the reduction of the model from the v8x to the v8n, impacted the performance of the system. The system was able to achieve a

precision of 0.655 and recall of 0.799 during testing on corn stover bales. Using the v8x model would improve this moving forward.

Table of Contents

List of Figures

List of Tables

Acknowledgements

A huge thanks to Dr. Edwin Brokesh, if he hadn't mentioned having a research assistant position open during his Solid Modeling class and then hired me when I e-mailed him 5 minutes later, I wouldn't even be in the BAE graduate program. His support as my graduate advisor and research supervisor means a lot to me.

I would like to acknowledge the generous support of AGCO Corporation, whose funding, dataset contributions, testbed hay balers, and field locations made this project possible. I would also like to thank Kevin Hamilton from AGCO for all his work in the field while testing the prototype.

I would also like to thank my fellow students at Kansas State University, DJ Klamm, Benjamin Weinhold, and Zak Oster. DJ Klamm's research on camera positioning, baler motion and output rates, and test frame construction gave me a great starting point to my own research. Benjamin Weinhold's and Zak Oster's support was instrumental during the research on correcting the class imbalance of my dataset.

Dedication

I would like to dedicate this thesis to my Father Bill Vail, whose love of all things science and engineering inspired my own.

Chapter 1 - Introduction

1.1 Motivation:

The world's population is expected to reach 9.8 billion by 2050 and 10 billion by the end of the century (United Nations D. o., 2017). Even with the many advances in farming practices that have been made in recent history, we are still not currently able to produce enough food to feed that number of people. Meat, eggs, and dairy make up 14.5% of globally consumed calories on a yearly basis (United Nations F. a., 2021). In 2022, 18.5% of the $452.7 billion in annual farm production expenditure was due to feed costs associated with animal-based agriculture (NASS., 2023). In addition, roughly half of all livestock feed comes from harvested forages (Decision Innovation Solutions, 2020). This, coupled with the rising annual expenditure required of farmers since 2018 shown in Figure 1.1, means that efficient, cost effective, production and use of livestock feed resources is critical to feed the world in future generations.

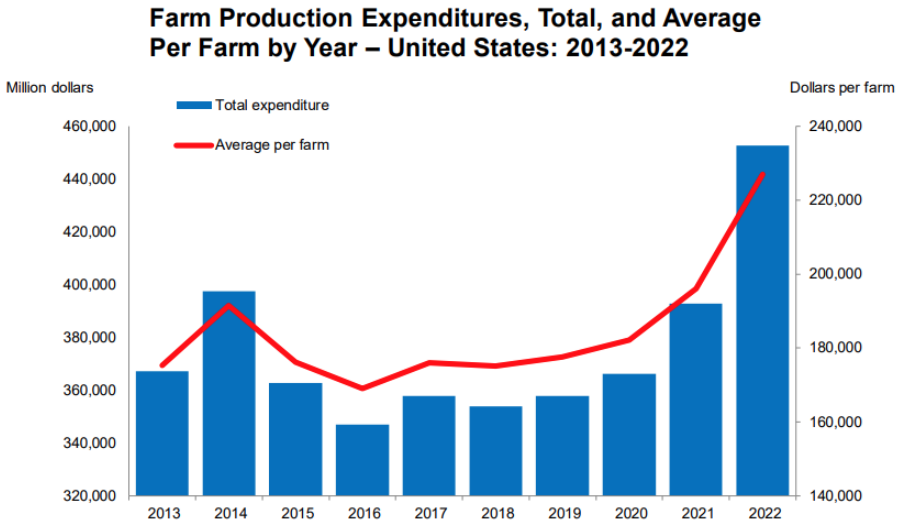

Figure 1.1: Farm Production Expenditures, Total, and Average Per Farm by Year – United States: 2013-2022 (NASS., 2023).

To meet increasing demands for food production while keeping the cost to farmers down, the efficiency of farm operations in general, and feed production in particular, will need to be greatly increased. To accomplish this, inputs such as labor, pesticides, fertilizer, and equipment

costs will need to be minimized while outputs are increased. One of the current focuses of the agricultural engineering industry is smart/precision agriculture, or the use of AI and machine learning in farming. Technologies such as targeted pesticide spraying, automated equipment operation, and computer-vision aided disease detection are just a few examples that have been showing up on the market recently. Smart/precision agriculture and automation are proving to be effective methods of increasing productivity while lowering input costs, such as pesticides, fertilizer, and labor (Shin, et al., 2023).

Because feed is the number one expenditure in farming, these new smart/precision agricultural techniques need to be translated to the feed production industry to meet the increasing demand. One common method of handling harvested forages for feed is to form it into either round or square bales. This increases the density of the forage into a size and volume that is easier to transport and distribute. During bale production, incorrect settings, machine failure, or operator error can lead to bale failure. Bad bales, due to their irregular shape and density are harder to transport and often spoil faster than correctly baled forage. For this project, the focus will be on detecting any failures that cause a significant change in the bale's shape, such as twine failure, or major compaction errors.

Due to a request from our corporate sponsor AGCO Corp. and the unique challenges that square balers face over round bales, we will be focusing on square hay bales for this research. Machine settings such as bale compaction, twine tension, and feed rate, are a particular concern for square hay bales. These failures necessitate either manually completing the process, or in the worst case, the breaking down of the bale and then attempting to re-bale the forage. Both cases increase production costs as well leading to increased loss of forage, but re-baling is particularly costly. And if the failure is a recurring issue and the operator doesn't detect the problem quickly, entire sections of bales can fail. The costs associated with these failures have created a need for research into ways to quickly identify bale failure during the production process and alert the baling operator.

1.2 Objectives:

While there are mechanical methods capable of detecting bale failure, this only works while the bale is in contact with the system. Many bale failures can occur after the bale has left the hay baler and so a system able to continue monitoring for an amount of time after dropping

the bale will be necessary. To quickly detect bale failures and alert the operator, a detection system would need to be able to recognize what parameters determine a good hay bale and be able to detect any deviation from those parameters. This system, to reduce distractions faced by the operator during baling, should also require as little interaction by the operator as possible. For this application, computer vision is uniquely suited, being able to continue monitoring the bale as long as the bale continues to be in the camera's field of view and within a reasonable distance.

Computer vision has been a growing area of research in agriculture for many years now, and technology is rapidly advancing. In the early stages, it was heavily impacted by lighting, movement, camera obstruction, high hardware requirements and changing viewing angles. These shortcomings of older computer-vision methods often prevented the use of computer-vision systems under field conditions, but advances in the past decade and even just the past few years have negated many of these problems. Newer technologies, such as deep learning architectures, improved computing power and miniaturization of integrated computing platforms, enable computer vision to be utilized under field conditions.

The goals for this project are as follows:

- Research common and emerging computer vision methods available to the market and select the best option for use in the detection of hay bales and performing pass/fail analysis of the finished bale. To perform this pass/fail analysis, the system should monitor the bale's shape for as long as possible.
- Collect an image dataset of both good and failed hay bales for the training of the previously selected computer vision model with a high degree of accuracy, precision, and recall metrics.
- Design and implement a prototype hardware system capable of deploying the trained model under field conditions. The prototype system must be capable of withstanding field environmental conditions, such as dust, water, and vibration. The system must also be able to determine the current GPS coordinates of the detected hay bale and record those coordinates along with the pass/fail determination of that bale for later retrieval.

1.3 Background:

1.3.1 Bale Failure Causes

Square hay baling operation begins by picking up the loose forage windrow with feeding tines, which is then carried into the bale chamber. In the bale chamber, a ram arm compresses the newly added forage material into the end of the bale. While the bale is being compressed, twine is fed through the chamber to the knotter and at an operator specified tension. Once the end of the current bale is reached, the twine is cut and knotted off to secure the bale. The completed bale is then ejected out of the back of the machine.

A bale can be considered a failure for several reasons; if the shape is not square and uniform, if the bale is larger or smaller than the desired dimensions, or if the bale falls apart upon ejection from the system. Each step of the baling process can experience failures caused by incorrect settings, system failure, or operator errors, such as:

1. During the forage pickup, moving at speeds greater than suggested rates can cause some of the forage to not be picked up properly. This type of failure results in crop loss rather than bale failure and will not be addressed in this project.
2. Inconsistent feeding rates into the chamber due to uneven windrow volume can cause changes in layer compression of the bale. This can lead to bales that are too tightly compressed and may break the twine, which is a contributing factor to failed bales.
3. Forage moisture levels necessitate changes to the feed rate into the bale chamber. If these are incorrectly adjusted, bale failure can occur, either during the baling process or afterwards when the forage moisture level changes.
4. Bale chamber tension can be improperly set by the operator, which can lead to incorrectly compressed bales. This can lead to mis-shaped bales, or broken twine failures due to over compression.
5. The twine tensioner being incorrectly set can lead to broken twine failures.
6. Wear or damage to the twine knotters can lead to a failure in securing of the twine at the end of the baling process which will cause the bale to fall apart once ejected.

1.3.2 Bale Failure Costs

Estimating bale failure costs can be difficult due to the varied nature of the forage type, moisture content variability, skill in handling of the operator, and machine dry matter loss rates. In late 2023, Alfalfa prices for large square bales was between $200 and $325 per ton dry matter (USDA, 2023). If the operator has someone standing by that can unbind the failed bale, loosen up the forage matter, and place it onto the windrow, then it can be re-baled without needing to stop the baling process. This means that at minimum, the failure has resulted in doubling the baling twine and machine run time costs of that forage material while also requiring an assistant to process the broken bale. This doesn't consider dry matter loss during the re-baling process, where some loss is inevitable. Using an average of 2 tons or 4 large square bales per acre, 10 minute of work time by a farmhand to breakdown a bale, and 108ft of baling twine (3'x4'x6' square bale), the cost per fixing a bad bale is $11 (see Table 1.1)

	Cost	*Source*
Farm hand Pay (est. 10 min./bad bale)	$2.28	(Payscale, 2023)
Baling Twine (3x4x6 6 twine square)	$2.07	(CWC, 2023)
Machine (est. 3 min./bale, 2 bales/acre)	$6.65	(NCSU, 2013)
Total	$11.00	

Table 1.1: Bale Failure Cost Analysis.

1.4 Thesis Outline:

Chapter 1 provides an overview of the motivation, objectives, and results of this project. Chapter 2 covers the research carried out over different detections methods and the final method selection for this project. Chapter 3 explores the methods used to correct an imbalance in the training dataset classes and their effectiveness. Chapter 4 details the design process and implementation of the project prototype system. Chapter 5 summarizes the results of the project, testing of the prototype system, limitations of the system, and suggestions for a path forward in future research.

Chapter 2 - Object Detection Methods

2.1 Introduction:

The current state of the computer vision field can be broken down into two categories, shallow learning, and deep learning architectures. This classification of computer vision methods refers to whether the architecture utilizes artificial neural networks, in the case of deep learning, or doesn't, in the case of shallow learning.

Artificial neural networks are a way of attempting to simulate the highly complex and interconnected neural structure of the human brain. While modern computers can transmit signals from one part of a chip to another at nearly the speed of light compared to the human brain's comparatively sluggish 100 m/s, the human brain has a parallel processing capability far superior to computers (Cox & Dean, 2014). An average human brain has billions of neurons forming trillions of interconnections (Cox & Dean, 2014), this "neural network" of the brain enables us to carry out complex decision-making, pattern recognition, and self-learning that current artificial systems struggle with. Researchers in the field of deep learning work towards combining the extremely fast clock speeds, data processing, and data transmission capabilities of modern computing with the vast parallelism of the human neural structure. Current artificial neural networks attempt this by having a network of nodes, where each node looks for specific features in the image. The connections between these nodes have weights or importance values assigned to them (see Figure 2.1). The network takes the training images dataset from the input layer to the hidden layer, altering the values of the connection weights depending on the prevalence of the features that each node looks for. The output layer takes all the hidden layer values and decides if the object being searched for is there or not. This output is then compared to the labels provided by the trainer prior to training, if it matches then the weights found during training are accepted. If the output determination does not match the provided label, then that image that was wrong is run again using the combined weights from images that were guessed correctly. The training algorithm continues this process until it arrives at a set of feature weights that represents that class of object as best as possible with the training dataset provided.

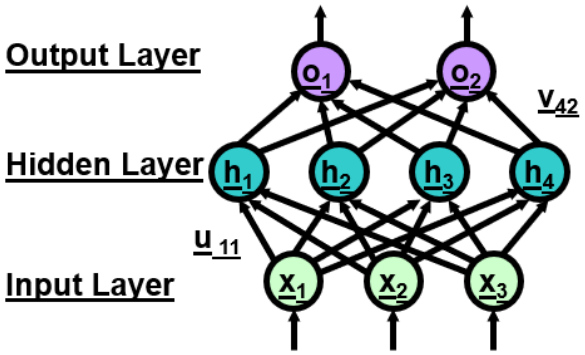

Figure 2.1: Neural network nodes and layers (Hsu, 2023).

Object detection has advanced far in the 20 years since the first facial recognition system was released in 2001 by Paul Viola and Michael Jones. This first architecture, the "Viola-Jones Detector" (see Figure 2.2), was a shallow or traditional machine learning method that searched the image for a set of specific features known as Haar-like features (Zou, Chen, Shi, Guo, & Ye, 2023). These shallow methods continued to develop, with algorithms such as HOG (histogram of gradients), LBP (local binary pattern), and DPM (deformable parts model), until 2012 when the first deep learning method, AlexNet was released (Zou, Chen, Shi, Guo, & Ye, 2023). Deep learning methods can be further broken down into one-stage and two-stage algorithms. One-stage algorithms perform classification and bounding regression in one step, while two-stage algorithms first generate region proposals using selective search or a region proposal network and then run classification on the proposed regions. In general, this will mean that one-stage algorithms are faster while two-stage algorithms are more accurate.

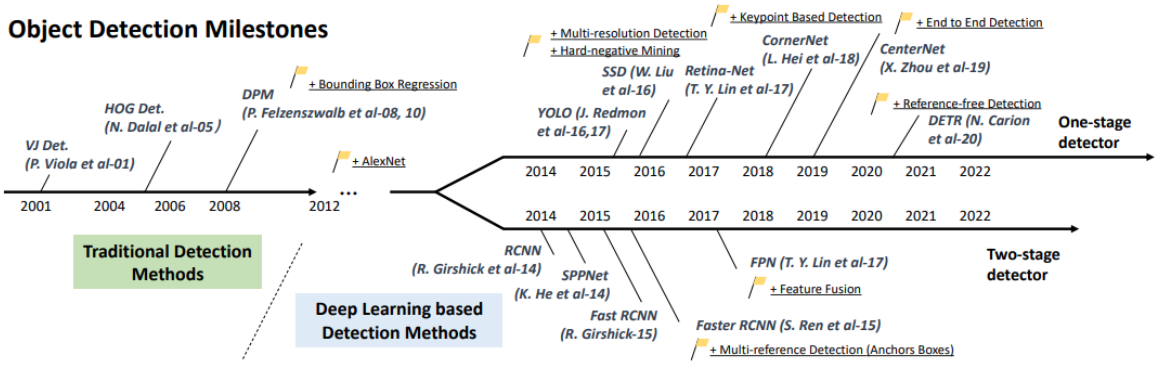

Figure 2.2: Object Detection Milestones (Zou, Chen, Shi, Guo, & Ye, 2023).

2.2 Model Evaluation Criteria:

When evaluating object detection models, three different performance metrics are primarily used: precision, recall, and the mean average precision at a 0.50 threshold (mAP50). While there are other statistics that can be useful for evaluating a computer vision model, these three metrics tend to be the standard used by many machine vision researchers and industry leaders.

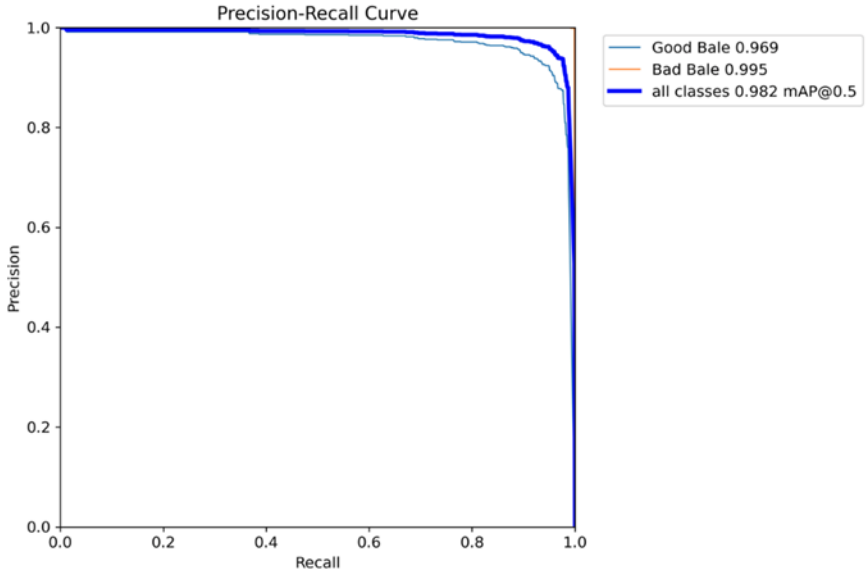

Figure 2.3: Precision-Recall curve from the YOLOV8 training of the corrected-200 epoch hay bale detection model.

2.2.1 Precision

Precision is the ratio of correctly identified positives to all predicted positives. Precision is calculated by first taking the intersection over union (IoU) which compares the predicted object bounding box and the ground truth object bounding box to a threshold value. If the IoU is greater than the threshold then it is a true positive (TP), otherwise it is a false positive (FP). For example, say you are using .5 as the threshold value, then if 50 percent or more of the pixels in the predicted bounding box match the pixels in the annotated bounding box, it will be a true positive. Once TP and FP have been found for each class, precision is then TP/(TP+FP) for that class.

2.2.2 Recall

Recall lets us know how good a model is at finding all positives. A model can have perfect precision because every positive it gives is correct but if it is only finding a small number of the positives in the dataset then its recall will be very low. Recall is calculated using the same TP values but also using false negatives (FN). FN is the measure of how many positive objects the model labeled as negative. Therefore, recall is TP/(TP+FN) for each class.

2.2.3 mAP50

Mean average precision (mAP) can be a confusing metric, as some authors use it interchangeably with average precision (AP). For instance, in YOLOv5 through v8, at the end of training you are given a mAP for both overall and per class. Since mAP stands for mean average precision which is the mean of all per class AP ratings, the per class mAP given by YOLO are AP ratings and the overall is the true mAP.

This can be further complicated by AP also being used to refer to the weighted average precision. In general AP is the area under the precision-recall curve which is a function of the threshold value used for IoU. mAP is the mean of AP for all classes but when a specific value is given, such as in the case of the commonly used mAP50 value, this means that instead of being the mean of the area under the precision-recall curve, it is the mean of the values for that curve function specifically at an IoU threshold of 0.50 (see Figure 2.3).

2.2 Viola-Jones & LBP:

One of the first applications of computer vision was facial recognition with the introduction of the Viola-Jones architecture in 2001 (Zou, Chen, Shi, Guo, & Ye, 2023). These first architectures revolved around the creation of feature sets during training and then searching through the image using differently size “windows” for these features. This approach worked well for facial recognition, due to the human face having common features such as eyes, nose, and a mouth that it could look for. These features can be rectangular features based on the brightness levels of the pixels (Figure 2.4), as is the case in Viola-Jones using Haar-Like features

(Viola & Jones, 2001). Or a varying sized local neighborhood of pixel values compared to a central pixel (Figure 2.5), as is the case with LBP features (Trefný & Matas, 2010).

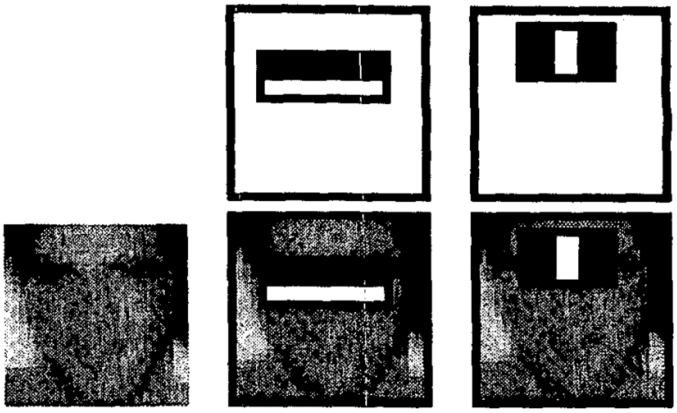

Figure 2.4: Haar-Like Features for Face Detection (Viola & Jones, 2001)

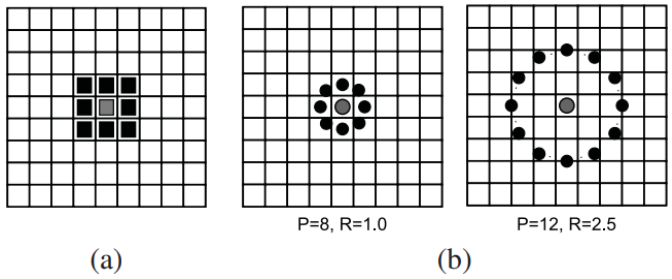

Figure 2.5: (a) LBP (b) rotation and multiscale $LBP_{P,R}$ (Trefný & Matas, 2010)

Both Haar-Like and LBP features have been used in facial recognition for smart phones and other portable devices, and the decision was made to test these methods for compatibility with the detection of hay bales before moving to more modern deep learning methods.

2.2.1 Viola-Jones & Haar-Like

The Viola Jones detection algorithm is the first true object detection method, as the prior methods were simple template matching algorithms. In template matching, the algorithm checks pixel values of an image against a provided template image. If the image pixel values are close enough to the original template image, then it shows a match. These older methods were not capable of looking for a class of objects but instead checking for a match to a single example object.

While the Viola-Jones detector was a common method used in facial recognition, it was quickly decided to move on from it for this project. Because of the rectangular features that it uses, the detector requires images from a multitude of angles and lighting conditions to be able to accurately detect objects. This dependence on views from the same angle, shape, and scale as the dataset it was trained on, leaves the Viola-Jones detection method often unable to properly bound the detected object as in the case of Figure 2.6, or even detect it at all.

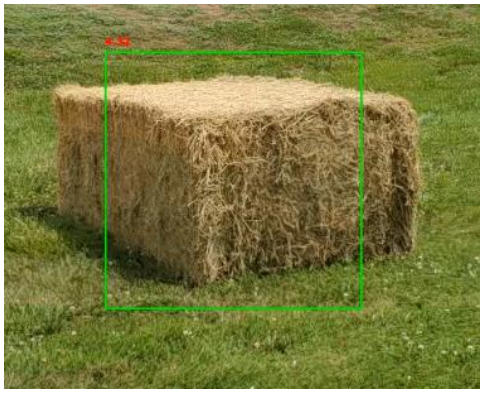

Figure 2.6: Viola-Jones detection test.

2.2.2 LBP

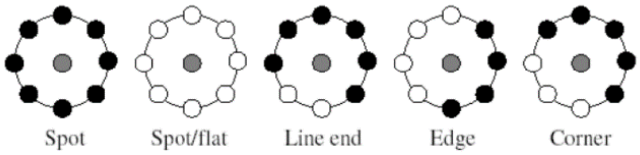

Figure 2.7: LBP feature examples (Chong-Yeon, 2008).

Object detection using LBP features, performs detection by breaking the image into a histogram, analyzing each bin by thresholding the surrounding pixels to the center pixel and assigning the resultant value to that bin. The method can then compare the values of each bin to its neighbors to calculate texture primitives for detection (see Figure 2.7). Once an image has been broken down into texture primitives, the results are then compared to an increasingly complex cascade of trained feature templates. If an image shows a match to a large enough number of cascades, then the program will show a positive detection and provide a resultant confidence rating. If the image does not match the initial lower detail cascades, then the program

moves on. This way, calculation time is not wasted on further cascade levels and the detection speed is maximized**.**

The LBP method, when properly trained, can achieve a detection speed up to 101ms per 2000 detections while still maintaining a hit rate (accuracy) of 89% (see Table 2.1). This high level of speed with a relatively good level of accuracy has made LBP a commonly used method for mobile and embedded detection systems, such as facial recognition in smartphone cameras (Chong-Yeon, 2008).

LBP Result

Algorithm	Dataset	Detected Faces	Hit Rate	Detection Speed (ms)
LBP	Color FERET	1004/1127	89%	95.44924
	MIT	1779/2000	89%	101.8864
	Taarlab	674/759	88.8%	104.9335
Overall		**3457/3886**	**89%**	**101**

Table 2.1: LBP detection statistics (Kadir & et.al., 2014).

To prepare an LBP feature model for object detection, it must first go through training on a dataset of images for the class of object being looked for, to extract the commonly shared features of the object. LBP feature extraction for use in object detection is carried out through first converting each image in the training set to a single channel gray scale image. Once the image is in grayscale, it is converted into a matrix of luminance values for each pixel and binarized by comparing each pixel's luminance to its nearest neighbors. If the central pixel's luminance is greater than the neighbor then its value is 1 otherwise it is 0, this process is repeated for all neighbors until a binary representation of the central pixel is created. This binary representation is an 8-bit number representing a pixel's 8 neighbors (see Figure 2.7). The training algorithm then creates a histogram of the binary pixel representations of the image area that is occupied by the object, which provides a feature representation of the object in that image. These representations of the object in each training image are then combined into a single cascade of features file representing the class of object being looked for.

The LBP detector for this project was trained using over 6000 images of good and bad hay bales with an 80/20 train/validate split, with a separate set of 670 images used for testing of the completed model. Since the method of LBP detection being used here cannot create a single

model for multiple classes, The dataset was split into separate sets for good bales and bad bales. These two separate sets were then used to create two LBP models, which a program can then use to check each test image for both good and bad bales at the same time. Using this program, the models were run on the test dataset and the resulting images verified by hand to provide precision and recall statistics, the results can be seen in Table 2.2.

	Precision	*Recall*
Good Bale	1	0.582
Bad Bale	0.6	0.164

Table 2.2: LBP model testing results.

From the results in Table 2.2, the LBP model for good and bad hay bales shows a moderate to high performance in precision but a low quality of performance in recall. The results show that when it does identify something as a good bale it was right every time but missed almost 42% of good bales. When it identified something as a bad bale, it was right 60% of the time but missed nearly 84% of the bad bales, which is a significantly poor performance. For a commercial application such as for this project, where failing to properly detect bad bales will cost the operator money, results like those seen in Table 2.2 are not acceptable. At this point, it was decided to move onto more modern methods.

While LBP cascades were a leading method for detection systems in the past, the method is showing its age. Newer methods utilizing deep learning algorithms, along with vastly increased computing power available even to mobile devices, have led to systems capable of 95% and higher accuracies while still maintaining detection rates fast enough to perform detection in real-time. Couple these advancements with LBP's relative weakness in being able to detect objects at different scales and lighting conditions, and an inability to use color information, and many other methods begin to outshine LBP cascades.

2.3 Deep Learning Methods:

Deep learning methods, since their introduction in 2012, have quickly taken over as the focus of research and implementation in computer vision. Their ability to automatically learn features from a dataset without being totally dependent on user input makes them far more versatile than most shallow learning methods. Deep learning architectures can take in multiple feature domains, and through multi-layer abstraction, create complex feature hierarchies (Picon, Alvarez-Gila, Irusta, & Echazarra, 2020). Their shallow learning counterparts were often limited to just one or two domains, and if more were needed, then often multiple models would have to be run simultaneously, and their inputs combined (Picon, Alvarez-Gila, Irusta, & Echazarra, 2020).

In the current market, there are many different deep learning architectures and even more custom variants of each. Out of this plethora of architectures, there are a few that have received more focus from the industry and research community than others, such as the two-stage RCNN and single-stage YOLO architecture families (see Figure 2.2). Because this is a project focused on the needs of our sponsors, certain considerations are necessary; the detection algorithm should be as open-source as possible, and it needs to be accurate while still running in real-time. The task of detecting hay bales also requires consideration for the environment that the equipment will be operating in. Such as needing whole-image context during training, since hay bales are very similar to the background of the field they are produced from. With these guidelines in mind, it was determined that the YOLO architecture family; with its simplified detection pipeline, short latency, and relatively high degree of accuracy, would be a good match. RCNN, being a two-stage architecture, has a higher computing overhead and greater detection latency, and thus is not as well suited for real-time detection as YOLO.

2.3.1 YOLO

The YOLO architecture was first introduced by Joseph Redmon et.al. in their 2016 paper "You Only Look Once: Unified, Real-Time Object Detection". Most other architectures of the time were using one network to propose regions that are most likely to contain the object and then a second to determine the actual location and bounding region. Redmon and his team, instead proposed treating detection as a regression problem and carrying out the process with a

single pipeline using one network. YOLO does this by splitting the image into an SxS grid and determining which cell of the grid contains the center of the object, and then moves out from the center, determining the bounding region of the object.

YOLO has undergone many improvements since its inception, with the most current version being YOLOv8 (Jocher, Chaurasia, & Qiu, 2023). At the start of this project YOLOv5 was the current version and was used to train a model on a dataset of images gathered from the K-State dairy farm, online image repositories, and images supplied by AGCO corp. In January 2023, Ultralytics released YOLOv8, which showed a significant improvement in detection speed over previous versions (Jocher, Chaurasia, & Qiu, 2023) (see Figure 2.8). Due to the limited size of the hardware being used, it was decided that the increased speed but similar size and hardware requirements of YOLOv8 was worth updating the model.

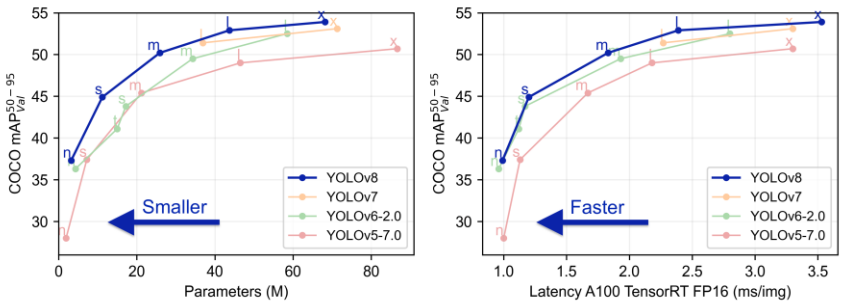

Figure 2.8: YOLOv8 latency performance vs. previous YOLO versions (Jocher, Chaurasia, & Qiu, 2023).

2.3.2 YOLO Backbone

The backbone of the YOLOv8 detection architecture is composed of P1 through P5 sections with 168 total layers, 53 of which are convolutional layers (King, 2023). P1 and P2 are base convolutional layers with the neck and head layers P3 through P5 interlinked to form a combined header layer (King, 2023).

The diagram in Figure 2.9, by RangeKing of GitHub in cooperation with one of the YOLOv8 developers Glenn Jocher, shows the composition of the YOLOv8 structure (King, 2023). The first two layers extract object features from the image. The SPPF layer and its convolutional layers process those features at multiple scales. The up-sample layers process the feature maps at increasing resolutions. The C2F layers combine features with contextual information gathered from the image background to improve accuracy. Once all of this has been done, the detection layers then map the features to a bounding box and object class.

Training of a model involves running this structure over a large dataset of images that have been annotated to include a bounding box around the object and the object class name. The structure extracts object features from the bounding box area while also gathering contextual information from outside the bounding box and then condenses this into a detection model. Contextual information can be very important to the accuracy of a model, which is why many groups will provide pre-training on other image datasets that don't include the object being looked for but have similar environments or backgrounds. This pre-training gives the model better context for the environment in the background and what isn't the object being searched for compared to training from scratch. This model is then run over a separate validation dataset for accuracy checks. This process is run in segments known as epochs, where one epoch is when the algorithm has trained once on every image in the training dataset. YOLOv8 continues training until the number of epochs set by the user or until there hasn't been an increase in mAP50, precision, or recall for a defined number of epochs.

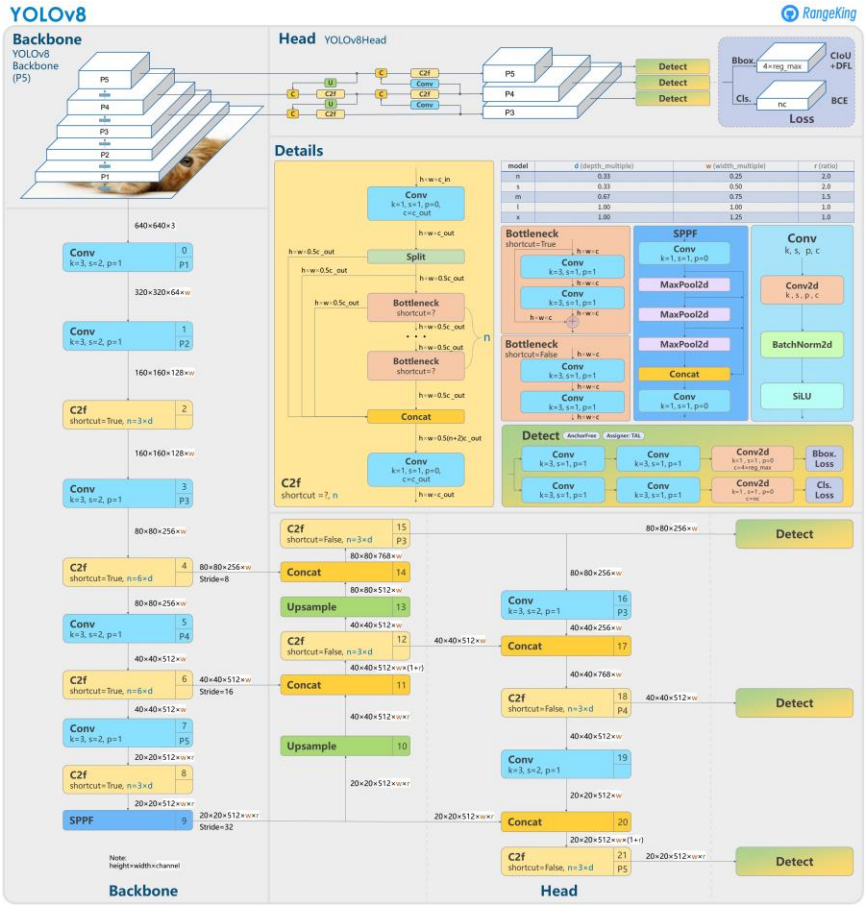

Figure 2.9: YOLOv8 Backbone Structure (King, 2023).

2.4 Training Results:

Thanks to the images supplied by AGCO Corp., images taken of hay bales supplied by the K-State research farms, and images collected from the internet, an image dataset of over 6000 hay bale images was collected. This first dataset, due to a lack of bad bale images, was heavily class imbalanced, with there being more images of good bales than bad bales. It was decided to validate the results of this model first without fixing the imbalance as a baseline. To provide a good baseline for further testing, two separate baselines were trained; one that was trained on the data without pretraining on a commonly used dataset such as MSCOCO, and one that did include pretraining with MSCOCO. Because the dataset that was collected was from such a limited number of sources, the overall training statistics showed an extremely high level of accuracy, precision, and mAP50, even on images from the set that had been set aside for testing. To run a proper test of the system, a set of test images was obtained from the internet. This test set was comprised of images that were from different fields and baler manufacturers than what was used in the training dataset. This new set was then used so that the test dataset would be as unrelated to the training dataset as possible. These tests showed a drop in mAP50 in comparison to the original test dataset statistics as expected, providing a more realistic test of the models (see Table 2.3).

	Scratch mAP50 original test	*MSCOCO mAP50 original test*	*Scratch mAP50 internet images*	*MSCOCO mAP50 internet images*
All	0.983	0.988	0.464	0.689
Good Bale	0.97	0.98	0.527	0.563
Bad Bale	0.995	0.995	0.401	0.815

Table 2.3: YOLOv8 class-imbalance uncorrected test statistics.

	Precision	*Recall*	*mAP50*
All	0.468	0.383	0.464
Good Bale	0.448	0.534	0.527

Bad Bale	0.489	0.232	0.401

Table 2.4: Expanded test statistics for YOLOv8 without pretraining model.

	Precision	*Recall*	*mAP50*
All	0.829	0.553	0.689
Good Bale	0.811	0.42	0.563
Bad Bale	0.848	0.686	0.815

Table 2.5: Expanded test statistics for YOLOv8 MSCOCO pretrained model.

2.5 Conclusion:

While deep learning architectures have many advantages over their shallow counterparts, one of the main difficulties with deep learning is their dependence on very large datasets. Dataset collection and annotation is the most time-consuming part of creating a deep learning model. And the quality of the dataset will heavily impact the accuracy of the model. This is one of the reasons that pre-training is a valuable tool in model training. By training a model starting from weights that were trained on a large and varied multi-class dataset, bias in detection can be removed. Pre-training teaches the model a wide variety of different objects, allowing the model to better discern what isn't the new classes it is being trained in. As can be seen from Table 2.4 and Table 2.5, pretraining leads to a large increase in overall performance in all but the good bale recall statistics.

Another common issue that deep learning models can encounter, and one that this project initially suffered from, is class imbalance. Class imbalance occurs in multi-class models when one class has a much higher or lower number of images than the other classes. Most training algorithms will attempt to stop training if the validation phase encounters a dip in performance over multiple epochs to prevent over-training. If one class is much smaller than the other classes, then that class will start to over-train before the others have reached their full potential accuracy. From Table 2.4 and Table 2.5, a large difference between the good bale and bad bale classes can be noticed. This difference in statistics is due to the much smaller bad bale class causing the program to end training before the good bale class was fully trained. In chapter 3, I

will discuss how I went about correcting the class imbalance between the good and bad bale classes, and how this impacted the final model statistics.

Despite these challenges, the YOLO architecture shows that it can detect good and bad hay bales and was chosen to be the detection architecture for this project moving forward. It was also decided that the advantages of pre-training on a large multiclass dataset such as the MSCOCO dataset are worth the effort and will be included in all models for this project in the future. With the architecture and pre-training method decided, all that remains to complete this stage of the detection model development is to further expand the dataset and correct the class imbalance issues.

Chapter 3 - Dataset Augmentation for Class Imbalance Correction

3.1 Introduction:

The original dataset was comprised of 6000+ hay bale images with 1600+ of those being bad hay bales. After initial training of a YOLO v8 model from this dataset, it became apparent that the smaller sample size for the bad hay bale class compared to the good hay bales was causing a reduction in performance. The original training showed poor mAP performance, indicating that with an intersection over union of 50%, the area under the precision-recall curve was low. The pattern of performance effects; with the bad bale class being higher, and the good bale class being lower than expected, suggests a large under/over training due to class imbalance.

During initial research, some common methods such as over/under sampling and synthetic minority oversampling (SMOTE) were considered as possible fixes, but it was determined that these methods would not combat class imbalance effectively enough for this dataset. The first case, over/under sampling is more of a temporary fix and tries to improve the model by cutting images from the larger class. The second case, SMOTE, takes images from the smaller dataset class and copies them but with different backgrounds, rotations, and scales to artificially expand the class. This method is already employed by YOLO training to improve the trained model's quality and would have minimal effect in combatting class imbalance. It was decided that three different generative methods to fix the class imbalance would be used, and the best out of those three would be chosen to create the final dataset. The three methods chosen were, the Poisson Flow Generative Model (PFGM++), the Least Squares Generative Adversarial Network (LSGAN), and the Deep Convolutional Generative Adversarial Network (DCGAN). Creating new images through artificial generative methods would expand the smaller dataset class without needing to reduce the larger class as in over/under sampling, but the images would be sufficiently unique from the original images to improve the class imbalance, unlike the SMOTE technique.

PFGM++ is an update to Poisson Flow Generative Modelling (PFGM) based on the paper *PFGM++: Unlocking the Potential of Physics-Inspired Generative Models* (Xu & et.al., 2023). This paper and its associated model were released in February 2023 and unifies PFGM with Diffusion modeling and makes many improvements and optimizations to the original model. PFGM generates images through treating each pixel in an image as a charged particle using the

Poisson equation, generalizing those equations for the entire dataset and then running the equations in reverse for a randomly generated seed image. This results in a new randomly generated image of the trained object class per static seed image.

The LSGAN comes from a 2017 paper in the International Conference on Computer Vision called *Least Squares Generative Adversarial Networks* (Mao & et.al., 2017). The authors claim the LSGAN produces superior quality images with greater stability than other models because it uses a mean squared error loss function to pull the generated images closer to the true data distribution even if the generated images are on the true side of the decision boundary.

The DCGAN originates from a 2016 paper titled *Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks* (Radford & et.al., 2016). This GAN uses a pair of convolutional neural networks (CNNs) with several architectural restrictions to learn a hierarchy of object and scene features, one network then works to discriminate real from fake images and the other works to generate fake images the can fool the first network. The authors claim that the model can learn both the object of the scene and the background.

3.2 Methodology:

It was suggested by Dr. Hsu of KSU's Department of Computer Science that pre-training a model on a multi-class dataset can in most cases improve performance. So, to verify this, while at the same time comparing the different GANs chosen, an 80-10-10 percent (train/validate/test) random split of the data was created and from that dataset, two baselines were trained for 200 epochs using YOLO v8. The first was trained from scratch and the second was fine-tuned from a model trained on the MSCOCO dataset. These baselines will serve as the point of comparison for the models trained on class balance corrected datasets created by each GAN.

Each GAN architecture was trained on a dataset made up of only the bad haybale images, and then used to generate artificial bad bale images. These images were added to a copy of the existing baseline dataset's bad bale class to balance that class in comparison to the good bale class. The three new balanced datasets were then used to train from-scratch YOLOv8 models, as well as finetune three new YOLOv8 models pretrained on MSCOCO.

Because most of the haybales in the dataset images were produced by a rather small number of balers and fields, it was decided to use images collected from outside sources for the

testing phase. This independent test set would help to prevent natural similarities that occur when most of the bales are produced by the same machine, from skewing the results. Finally, all the models were tested on this new dataset gathered from internet images and the results compared to the baseline results. Precision, recall, and mAP-50 were used when comparing the different models, since they best portray the quality of the compared models.

During training, utilizing cloud computing platforms such as Google Collab and Kaggle were attempted, but their runtime and memory constraints were insufficient for running the hardware intensive GANs. As a result, it was decided to use Kansas State University's BeoCat super-computer to run the models. This required tweaking the training code for each GAN and learning the job submission systems for BeoCat, but this proved to be a good move. This was particularly useful in the case of PFGM++, as even with multiple GPU, training required multiple days.

3.3 Baseline Statistics:

When first beginning the imbalance correction project, it was decided to train two different baselines for comparison: one trained from scratch on our dataset, and the other given pretraining on the MSCOCO image dataset before training on our dataset. The MSCOCO dataset is a commonly used, freely available dataset with many classes and many images per class. It was hypothesized that pretraining on MSCOCO could possibly improve our model's ability to avoid false positives on other objects, add context for image background environment and improve performance in general. This hypothesis was later shown to hold true, the MSCOCO pretrained model when tested on an outside testing image set that the model was never trained on, showed significantly better performance over the trained from scratch baseline model (see Table 3.1). Due to this, the MSCOCO pretrained baseline was chosen as the main source of comparison.

Scratch Baseline

Class	**images**	**instances**	**Precision**	**Recall**	**mAP50**	**mAP50-95**
All	24	233	0.468	0.383	0.464	0.228
Good Bale	24	163	0.448	0.534	0.527	0.241
Bad Bale	24	70	0.489	0.232	0.401	0.323

MSCOCO Baseline

Class	**images**	**instances**	**Precision**	**Recall**	**mAP50**	**mAP50-95**
All	24	233	0.829	0.553	0.689	0.446
Good Bale	24	163	0.811	0.42	0.563	0.268
Bad Bale	24	70	0.848	0.686	0.815	0.623

Table 3.1: Baseline YOLO statistics.

3.4 Results:

Each GAN was tackled by a different member of the team and YOLO models were trained separately. No model was trained with a combination of the GANs. This section details the work done for each model.

3.4.1 Least Squares GAN

Least Square GAN (LSGAN) is a generative adversarial network that uses the least squares loss function for its discriminator. LSGAN was initially trained for 3000 epochs on the whole bad bale set. The generated images were not of sufficient detail to train with, and as training continued, the model suffered from mode collapse, only producing images of a baling machine featured prominently in the dataset. Based on this, an altered bad bale dataset which was cropped to just the bales themselves was produced. LSGAN was then trained for 1500 epochs and produced usable results. A YOLOv8 model was then fine-tuned using this new dataset. The results from training and testing on the internet test set can be seen in Table 3.2 below:

	End of Training	*Test (Internet Set)*
Precision	0.969	0.852
Recall	0.963	0.558
mAP 50	0.977	0.749

Table 3.2: LSGAN training and testing comparison.

After this, LSGAN's performance was compared to our baselines. The results can be seen in Table 3.3 below. LSGAN performed significantly better than the scratch baseline in all metrics and slightly better than the MSCOCO baseline on mAP 50. There was a negligible difference in precision and recall when compared to the MSCOCO baseline.

	d-mAP-50 (Scratch)	d-mAP-50 (MSCOCO)	d-precision (MSCOCO)	d-recall (MSCOCO)
All	0.285	0.06	0.023	0.005
Good Bale	0.127	0.091	-0.083	0.126
Bad Bale	0.443	0.029	0.128	-0.115

Table 3.3: LSGAN baseline comparison

During the setup of LSGAN, a clerical error caused LSGAN to be trained at an image size of 512 instead of 640 like the baselines. When retrained at 640, there was a larger performance improvement of the YOLO model compared to baseline than was seen by the 512 set. This means image resolution makes a big difference in the quality of the generated images. Moving forward, all generative methods will utilize the largest resolution permitted by the hardware available and the model architecture being used.

The takeaways from this are that pretraining significantly improves performance on novel images and that the LSGAN generated images provide a slight benefit to detection when fine tuning compared to the MSCOCO baseline. LSGAN is not nearly as hardware intensive as PFGM++ which is covered in the next section and can be a good option for class imbalance correction when on a constrained hardware budget. Otherwise, as discussed in the next section, the larger improvement compared to the MSCOCO baseline model given by PFGM++, means that it would be the preferred method of the three methods compared here.

3.4.2 PFGM ++

Poisson Flow Generative Modelling (PFGM++) was released in February of 2023 on paperswithcode.com. At the time of its release, it was the SOTA in image generation on the CIFAR-10 dataset and at the time of writing this report was SOTA for the FFHQ 64x64 – 4x upscaling dataset. The authors sought to unify the original PFGM model with Diffusion modelling. By altering the parameter D, the model can be switched from PFGM at D=1 to a

Diffusion model as D goes to infinity. This allows for setting the model as a mix between the two models as best fits the data being used. PFGM attempts to generate images by taking individual images from the dataset and treating them like charged particles on the z=0 hyperplane. This creates a modeled representation of an electric field which is the gradient of the solution to the Poisson equation originating from each pixel of the image (see Figure 3.1). Moving away from the image along the field lines leads towards static diffusion. Once PFGM++ has been trained on the entire dataset, it will have generated a set of equations capable of moving from a generalized object of that class to a static image. After this, a random static seed image can be run through the equations in reverse, turning the random static into a randomly generated image of the class object.

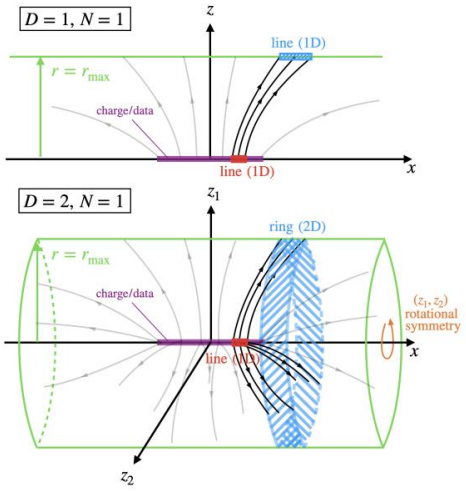

Figure 3.1: Graphical representation of PFGM++ treatment of data as a charged particle (Xu & et.al., 2023).

Due to the high hardware demands of the PFGM++ model and the tendency of the full images with background to cause mode collapse, it was decided to use a cropped set of bad bale images during the training like what was used for the other GANs employed in this project. This dataset was in 256x256 pixels resolution, cropped to just the hay bale with D=128 and trained for a period of 50 epochs at 50 k-imgs per epoch. It was decided at the beginning of this portion of the project that all images generated would be used even if misshapen so that the quality of the method can be determined versus the others. And while some of the images from PFGM++ were misshapen, most came out with an impressive level of texture, color, and shape realism in comparison to the training image set (see Figure 3.2).

Figure 3.2: Artificial images generated by PFGM++.

Once enough images of sufficient realism were generated with the trained model to bring the bad hay bale class into parity with the size of the good hay bale class, a corrected YOLOV8 model was trained for 200 epochs with pre-training from the MSCOCO dataset with parameters matching those used in training the MSCOCO baseline model. The results from validating this new corrected model on outside images that were not used during the training of either the model or the baseline, showed an appreciable improvement in multiple metrics (see Table 3.4). The largest increase (mAP50 =+10.8%, Recall =+18.1%, though precision was +-7.4%) was seen in the good hay bale class which was the class that was overrepresented in the original dataset. This is to be expected, as the class imbalance was originally causing the bad hay bales to over train which caused YOLOV8 to halt training before the good hay bale class could reach full training. This increase in model quality is even more drastic when compared to the from scratch baseline model. Overall, PFGM++ provided a net increase in model performance when used to correct class imbalance, in both classes and baselines.

Class	*Precision*	*Recall*	*mAP50*	*d mAP50 (scratch)*	*d mAP50 (COCO)*	*d Recall (COCO)*	*d Precision (COCO)*
All	**0.858**	**0.629**	**0.762**	**0.298**	**0.073**	**0.076**	**0.029**
Good Bale	**0.737**	**0.601**	**0.671**	**0.144**	**0.108**	**0.181**	**-0.074**
Bad Bale	**0.979**	**0.657**	**0.853**	**0.452**	**0.038**	**-0.029**	**0.131**

Table 3.4: PFGM++ corrected YOLOV8 model metrics compared to baseline.

3.4.3 Deep Convolutional GAN

The DCGAN was initially trained using the full bad bale dataset at resolutions of both 256x256 and 512x512 pixels. Initial images prominently featured the bale, but with very low detail (see Figure 3.3). However, after 1000 epochs of training, the model suffered from mode collapse and began generating the same image every time. After 2000 epochs, every generation would output a green image.

Figure 3.3: Initial Images Generated using DCGAN.

After the creation of the cropped bale dataset, new training was run at resolutions of 256 and 512. These images portrayed a more lifelike bale, better capturing the details and the shadows (see Figure 3.4). The best-looking images came around the 1000 epoch mark with a training size of 512. Due to limitations in the training computer's RAM, attempting resolutions above 512 was not possible, though with increased resources, the results could be improved upon.

Figure 3.4: Images Generated with Bale Only Dataset

After generating enough images to counter the class imbalance, a the pretrained MSCOCO model was trained using the new hay bale dataset. The training statistics were very similar to the between the initial MSCOCO baseline and the DCGAN MSCOCO model. When tested using a new internet set, the differences became more apparent (see Table 3.5). The precision of the new model severely underperformed the baseline, suggesting an increase in false positives. The recall, however, saw a modest increase while the mAP saw a small drop.

Model	*Baseline w/ MSCOCO*	*DCGAN*
Precision	0.829	0.559
Recall	0.553	0.672
mAP 50	0.646	0.644

Table 3.5: DCGAN results vs baseline comparison.

Given the results from Table 3.5, while DCGAN images did show improvement over the from-scratch baseline, the DCGAN is a downgrade in performance over the initial MSCOCO baseline. Due to this, DCGAN is not recommended for correcting the class imbalance in the haybale dataset.

3.5 Conclusion:

While all three methods did not provide direct classification improvements, the LSGAN method showed slight increases over the MSCOCO baseline model, and the PFGM++ method showed notable improvements across the board. In mAP50, the images generated with PFGM++ provided a 29.8% increase over the original scratch model and a 7.3% increase over the pretrained original model. In precision, PFGM++ provided a 2.9% increase over the pretrained original model. In recall, PFGM++ showed a 7.6% increase of the pretrained original model. These results show that correcting a class imbalance in image datasets through artificial image generation is a viable technique and can provide a notable improvement in model performance. Image generation using machine learning is a major topic of current research, and there are new models being published often, but not all image generation techniques are equal and out of the three methods shown here, PFGM++ is the clear winner. For a comparison of the improvements seen between the balanced vs unbalanced models for all three methods (see Table 3.6). Moving

forward with the project, PFGM++ will be the method of choice for correcting dataset class imbalance.

	d-mAP-50 (Scratch)	*d-mAP-50 (MSCOCO)*	*d-precision (MSCOCO)*	*d-recall (MSCOCO)*
PFGM++	0.298	0.073	0.029	0.076
LSGAN	0.285	0.06	0.023	0.005
DCGAN	0.180	-0.045	-0.270	0.119

Table 3.6: GAN margin of improvement comparison.

Acknowledgment: This research was carried out in conjunction with Benjamin Weinhold (M.S. student with the KSU BAE Department) and Zak Oster (M.S. student with the KSU MNE Department)

Chapter 4 - Detector Validation Platform, Design, and Initial Testing

4.1 Introduction:

The hay baling process is carried out in an environment that is extremely dusty, with limited power options and space; these factors played a large part in the selection of materials for the design of the prototype. Since not all tractors have a cab cover, all parts including the portion contained in the cab will need to be IP67 rated or higher. Baling operations are sometimes carried out in the very early or late parts of the day where lighting is limited; and while the current trained model does not include IR images, for the sake of future expandability, it was decided to design the system with low light capability.

4.2 Objectives:

The purpose of this prototype is for the validation of the detector model developed over the course of this project, as well as any future model versions. This requires the prototype to be modular, so that changes needed for future expansions can be easily implemented. The prototype will also need to have components with capabilities that are anticipated for future work, such as IR capabilities for nighttime operation. The system will also need to be able to collect necessary data during field operations, such as images of detected objects, their GPS coordinates, and the class of the object. Below are the main design objectives chosen for the prototype.

The goals of the prototype validation platform are as follows:

- Easy to install and operate.
- Protects the components from dust and rain.
- Allows for easy implementation of future upgrades.
- Provides a mounting point for the camera and GPS that can be adjusted in position and angle.
- Saves an image for each detected object as well as a text file with the GPS coordinates, and object class.
- Provides an in-cab monitor so that the operator can monitor system functions.

4.3 Hardware:

4.3.1 Camera

During the initial stages of this project, a FLIR BlackFly S camera was used. It was decided to move to a different camera due to BlackFly's dependency on proprietary software and general difficulty in setup for depth imaging. The market for stereo cameras is already limited, adding in the requirement for nonproprietary software, IP67 rating, and capability of withstanding field conditions, and the options are reduced to either the Luxonis Oak D pro POE camera or custom ordering a camera package. The Oak D pro camera met all the anticipated needs for this project, IP67, strong design, stereo depth imaging, IR capabilities, and even has on-board processing that will reduce computing load on the main system. And since a custom camera design would be prohibitively time consuming and expensive, the Luxonis camera was chosen.

4.3.2 Computing

For embedded and mobile computing, there are several manufacturers to choose from depending on the needs of the system, such as Arduino, Raspberry, AdaFruit, and Nvidia. Since this system will need to be capable of carrying out machine vison algorithms, AdaFruit and Arduino, brands who specialize in extremely small and thus limited systems, are automatically eliminated. While YOLOv8 can be run on a Raspberry Pi, it is at a very low frame rate and would be far too limited in capability for this model and any future expansions. Due to this, the Nvidia Jetson Nano was chosen. The Jetson Nano has USB, WIFI, Bluetooth, and Ethernet capabilities, as well as having GPIO pins for electrical components. Importantly for this application, it also has a built-in GPU and ARM processor, allowing it to run a YOLOv8n model at 15FPS. Since the system runs a modified ARM64 version of Linux, it is easy to integrate and control all the components through a Python program.

4.3.3 GPS

No matter how good the stereo depth camera is at finding the distance between the camera and the haybale, without highly accurate GPS coordinates, that capability will be useless.

When selecting a GPS module for the system, there are several factors that need to be accounted for: accuracy, speed of signal acquisition, and data connection method. For accuracy, higher is usually better, but hay bales are large and the distance between them can be large as well. Accuracy within one to two meters is acceptable for this application. Signal acquisition speed is very important in this situation, where the tractor might not be turned on until right before testing. If a GPS module doesn't have the capability of hot-starting, where the backup battery allows the module to keep signal acquisition even when the system is off, then re-acquiring the signal can take up to 20 minutes in some cases. This could lead to the system being unable to calculate the location for detected bales. For data connection, a module capable of transmitting over its USB cord rather than needing a direct serial pin connection is much simpler to use and troubleshoot. There are many GPS module manufacturers, thought the necessity for non-proprietary software does limit the field a little. It was ultimately decided to go with the SparkFun NEO-M9N SMA GPS board because it had all the required features while being relatively inexpensive. It has a USB-C port, a backup battery for hot-starting, and a horizontal accuracy of 1.5m. It is also capable of receiving GPS, GLONASS, Galileo, and BeiDou signals, allowing it to find a connection in a wider range of locals and with faster acquisition. It has a built-in SMA antenna port that makes connecting to the antenna on the camera mount very easy. This way, the GPS module can be stored in the protective case with the Jetson nano but provide coordinates for the camera since the antenna is located next to it.

4.3.4 Camera Mount

Since there are no commercially available camera mounts for the Oak D Pro POE camera that also can hold a GPS antenna, a mount was custom designed and manufactured in the KSU BAE machine shop. The mount was designed in SolidWorks and machined out of aluminum (see Figure 4.1). A magnetic base, bolts onto the lower chassis and allows adjustment of the angle and position of the camera (see Figure 4.3).

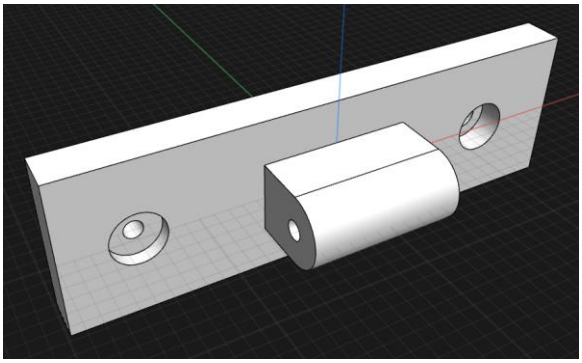

Figure 4.1: Camera mount part model.

Part #	*Part Name*	*Manufacturer*	*Part Amount*
1	Oak-D Pro POE Camera	Luxonis	1
2	Jetson Nano B01	Nvidia	1
3	NF-A4x20 5V PWM Fan	Noctua	1
4	IP68 RJ45 Couplers	Iwillink	2
5	Coaxial SMA Connector Cable 33ft	Superbat	2
6	Cat6 Outdoor Ethernet Cable 75ft	Ecjtu	1
7	1” Adjustable Cable Gland	Ampele	1
8	NEO-M9N SMA GPS Board	SparkFun	1
9	SMA GPS Antenna	Proxicast	1
10	300W DC 12V to 110V AC Inverter	Bestek	1
11	IP67 Clear Cover Junction Box	LeMotech	1
12	DM AC8265 Jetson Nano Wi-Fi Module	Waveshare	1
13	10.1” Capacitive Touch Screen	Roadom	1
14	Camera & GPS Mount	Custom	1

Table 4.1: Prototype Bill of Materials.

Figure 4.2: Prototype In-Cab Monitor (screen on the right).

Figure 4.3: Prototype Magnetic Camera/GPS Mount.

4.4 Construction & Lab Testing:

The process of construction began with drilling holes into the protective case for the WIFI antenna, power button, and cable gland. The gland can be cinched down over all the cables running into the protective case to prevent water from entering. The Jetson Nano and SparkFun GPS module were secured onto the case's isolation floor, and then the GPS module

connected to the Jetson Nano via USB. The ethernet and GPS antenna cables were all wrapped in cable protectors, waterproof connectors, and ran from the case down the length of the tractor and baler to the camera mount positioned over the baler output chute. The camera mount body was bolted to the magnetic base, and the camera and GPS antenna secured to the mount. The invertor connects to the DC power port in the tractor cab and powers the Jetson Nano, its connected systems and screen, and the camera POE injector.

To test the system before sending it out for field-tests, a printed-out picture of a haybale was used to trigger the completed system's detection program. From these tests, it was noticed that the system was trying to save an image and location coordinates every loop, which was due to a failure with the timing method in the program. This was corrected prior to sending the system out for field testing.

4.5 Field-Test Results:

Once the design and construction of the prototype was complete, testing was performed with support from AGCO Corp. The prototype was mounted to one of their large square balers and run through multiple baling operations in the Great Bend Kansas area. The testing was delayed multiple times due to rainy conditions, but the fields had completely dried out by the time baling commenced. Overall field conditions were sunny and dry during the baling, with the soil being dry enough to see minor dust kick-up during baling. The forage being baled during this process was corn stover, while the model had mostly been trained on alfalfa. This allowed testing of the model's ability to generalize hay bales and how different forage species will impact detection. The system saves an image and the associated bale's GPS coordinates every 4 seconds while an object is detected in the ROI area of the screen. Over the baling process, 642 images were collected, although the last 13 images had to be discarded due to the camera being displaced during over-night storage. Over the trial, the detection of good bales showed impressive precision but a lowered recall due to a larger than normal number of false negatives. These false negatives were due to the system over-classifying the corn stover bales as bad, leading to increased bad bale false positives and good bale false negatives.

Figure 4.4: Example detection image from the AGCO test of the prototype system.

	Precision	*Recall*
All	0.655	0.799
Good Bale	0.899	0.683
Bad Bale	0.410	0.914

Table 4.2: Prototype testing statistics.

4.6 Conclusion:

While the resulting precision and recall metrics were lower than the original training validation statistics, that was to be expected with the bales being corn stover rather than alfalfa and on the smaller YOLOv8n model rather than the YOLOv8x. A better comparison is to the test statistics run on images collected from random internet hay bale images (see Table 3.4). The results from that test showed (over-all precision = 0.858, recall = 0.629). A lower precision and higher recall compared to the values in Table 3.4, shows that it was having a larger number of false positives but a lower number of false negatives. Meaning that it was misclassifying the corn stover bales more often but failing to detect bales less often.

The driving force behind these results can be broken down to two main reasons; the material of the bales (corn stover) being different from the material of the training bales (alfalfa), and the much smaller number of parameters of the YOLOv8n model compared to the YOLOv8x model that was used the training and augmentation phases (see sections 2.3 and 3.4.2). The YOLOv8n model has 3.2 million parameters while the YOLOv8x model has 68.2 million parameters. This smaller model was chosen so that it would be able to run on the Luxonis Oak-D pro camera that was used in the prototype, but this will need to be re-evaluated moving forward.

During the beginning phases of this project, dust in the field obscuring the camera's view was a concern that we wanted to investigate but didn't have a way of testing. During testing, several images were collected with dust being blown between the bale and camera. This dust was not thick enough to obscure the bale from the camera.

Chapter 5 - Conclusion and Future Work

5.1 Summary:

The goals of this project were to find a good architecture for detection and pass/fail inspection of hay bales, collect good and bad hay bale images and train a detection model, and implement a prototype system for in-field testing of that model.

Chapter 2 outlines the difference between shallow and deep learning methodologies and some of the most popular architectures for each. The most promising methods were tested, showing a clear winner in the deep learning methods and the YOLO architecture specifically for this application.

In Chapter 3, with the aid of fellow students Zak Oster and Benjamin Weinhold, a class imbalance that had developed in the dataset was corrected. This imbalance was due to the difficulty in getting bad bale images compared to the good bales. Dataset augmentation using the PFGM++ architecture to artificially generate bad bales images proved to show the greatest increase in testing over DCGAN and Least Squares GAN.

In Chapter 4, the design and implementation of a prototype system capable of carrying out field-testing of the detection model was discussed. The goals of the prototype were to carry out pass/fail detection of the hay bales, calculate the distance of the bale from the camera, and then use that to record the GPS coordinates of each detected bale. Using an Nvidia Jetson Nano and a Luxonis Oak-D Pro camera as the main components, a prototype was built and tested.

5.2 Results:

After selecting the YOLO architecture as the method of object detection and collecting a hay bale dataset, initial training of the model was carried out using both from scratch and pre-training. This revealed that including pretraining on a large multi-class dataset greatly improved the capability of the model. It also revealed a class imbalance in the original dataset, which artificial image generation using PFGM++ proved to be effective at balancing. After balancing the dataset, the design and implementation of a prototype system was carried out.

The results from the prototype and the YOLOv8n model showed that on corn stover bales it was able to reach an over-all precision of 0.655 and a recall of 0.799. This is sufficient for an

initial proof-of-concept but could be greatly improved through continued development. The suggested next steps for this project will be covered in the next section.

5.3 Future Work:

The prototype system had an emphasis on saving space as much as possible, and looking back, it perhaps placed too much emphasis on that. To make the model capable of running on the Luxonis Oak-D Pro camera's internal processor, the model was reduced from the v8x with 68.3m parameters to the v8n with only 3.2m. This led to a noticeable drop in performance. Moving forward, the next prototype would instead run the detection on a more powerful computer such as the Nvidia Orin NX which can run the v8x model in real-time detection at very high frame rates with capacity left over for running the rest of the system. This would also free up the camera's processor for additional image processing techniques that might be added in the future.

The dataset for this project was collected over several years and the focus was on getting as many images as possible, particularly for the bad bale class. Now that the dataset has over 6000 images and has baseline statistics, a pruning of the dataset down to the best images would increase the accuracy of the model. This pruning is particularly needed for the artificially generated bad bale images, since it included all images that were generated so that the three different GANs could be properly compared. While this showed an over-all increase in the case of the dataset including the PFGM++ generated images, review of the artificial images could maximize this benefit.

It would be beneficial to explore ways of detecting even smaller discrepancies in hay bales than the exaggerated curving failures that the system was looking for. A similar system looking from a top-down view as the bale travels down the chute, utilizing a segmentation model trained on the dataset could hypothetically be used to measure length discrepancies down to only a few inches. This would allow the detection of bad bales that do not present a major change in curvature and would be missed by this current system.

The system's camera was chosen with an eye on being able to run detection during nighttime baling operations. Further exploration of that and whether a separate model for night vision would be needed, could hold promise for commercialization of the system, since many

farmers like to get out in the early or late hours of the day to get specific field conditions for baling.

Chapter 6 - Bibliography

Chauhan, S. (n.d.). *mAP (Mean Average Precision) for Object Detection - with python code.* Retrieved October 24, 2021, from Datanics: https://www.datanics.tech/2020/11/understanding-mean-average-precision.html

Chong-Yeon, J. (2008). *Face Detection using LBP features.* Stanford University. Retrieved from https://cs229.stanford.edu/proj2008/Jo-FaceDetectionUsingLBPfeatures.pdf

Cox, D. D., & Dean, T. (2014). Neural Networks and Neuroscience-Inspired Computer Vision. *Current Biology, 24*(18), R921-R929. doi:https://doi.org/10.1016/j.cub.2014.08.026.

CWC. (2023). *Big Baler Twine*. Retrieved from CWCGlobal: https://www.cwcglobal.com/twine/baling/big-baler

Decision Innovation Solutions. (2020). *Animal Feed/Food Consumption and COVIS-19 Impact Analysis.* Retrieved April 4, 2023, from Decision-Innovation: http://www.decision-innovation.com/webres/File/docs/210525%20IFEEDER%20Animal%20Feed-Food%20Consumption%20COVID-19.pdf

Dougherty, A. (2020, November 2). *The Power of (Local Binary) Patterns*. Retrieved from Towards Data Science: https://towardsdatascience.com/the-power-of-local-binary-patterns-3134178af1c7

Hsu, W. (2023). Multilayer Perceptrons (MLP) & Error Backpropagation (aka Backprop, EBP). Manhattan, Kansas, USA.

Jocher, G., Chaurasia, A., & Qiu, J. (2023). YOLO. (8.0.0). Retrieved from https://github.com/ultralytics/ultralytics

Kadir, K., & et.al. (2014). A comparative study between LBP and Haar-like features for Face Detection using OpenCV. *4th International Conference on Engineering Technology and Technopreneuship (ICE2T)*. doi:10.1109/ICE2T.2014.7006273

King, R. (2023). *Brief summary of YOLOv8 model structure*. Retrieved July 2023, from https://github.com/ultralytics/ultralytics/issues/189

Liu, J., & Wang, X. (2020). Early recognition of tomato gray leaf spot disease based on MobileNetv2-YOLOv3 model. *Plant Methods, 16*, 83. doi:https://doi.org/10.1186/s13007-020-00624-2

Luxonis. (2023, July 2). DepthAI v1.12.1. Retrieved from https://github.com/luxonis/depthai

Mao, X., & et.al. (2017). Least Squares Generative Adversarial Networks. *IEEE International Conference on Computer Vision (ICCV).*

Meel, V. (n.d.). *YOLOv3: Real-Time Object Detection Algorithm (What's New?).* Retrieved October 4, 2021, from https://viso.ai/deep-learning/yolov3-overview/

NASS. (2023). *Farm Production Expenditures 2022 Summary.* USDA., NASS. Retrieved from https://usda.library.cornell.edu/concern/publications/qz20ss48r

NCSU. (2013). *Hay Harvest Costs.* NCSU, Extension. Retrieved from https://cals.ncsu.edu/are-extension/wp-content/uploads/sites/27/2018/03/ForageBudHayLRB84-2_Print_2013_0.pdf

Payscale. (2023, October 25). *Average Farm Hand Hourly Pay*. Retrieved from Payscale: https://www.payscale.com/research/US/Job=Farm_Hand/Hourly_Rate

Picon, A., Alvarez-Gila, A., Irusta, U., & Echazarra, J. (2020). Why deep learning performs better than classical machine learning? *Dyna Ingenieria E Industria*. doi:http://dx.doi.org/10.6036/9574

Radford, A., & et.al. (2016, January 7). *Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks.* Retrieved from arXiv: https://arxiv.org/abs/1511.06434

Redmon, J., & et.al. (2016). *You Only Look Once: Unified, Real-Time Object Detection.* Retrieved from ArXiv: https://arxiv.org/abs/1506.02640

Shin, J., Mahmud, M., Rehman, T., Ravichandran, P., Heung, B., & Chang, Y. (2023). Trends and Prospect of Machine Vision Technology for Stresses and Diseases Detection in Precision Agriculture. *AgriEngineering*(5), 20-39. doi:https://doi.org/10.3390/agriengineering5010003

Solutions, D. I. (2020). *Animal Feed/Food Consumption and COVID-19 Impact Analysis.*

Syed, T. N., Lakhiar, I. A., & Chandio, F. A. (2019). Machine vision technology in agriculture: A review on the automatic seedling transplanters. *International Journal of Multidisciplinary Research and Development, 6*(12), 79-88.

Trefný, J., & Matas, J. (2010). Extended set of local binary patterns for rapid object detection. *Computer vision winter workshop,* 1-7. Retrieved from https://cmp.felk.cvut.cz/~matas/papers/trefny-lbp-cvww10.pdf

Ultralytics. (2023). Ultralytics. Retrieved October 2023, from https://github.com/ultralytics/ultralytics

United Nations, D. o. (2017). World Population Prospects: The 2017 Revision, Key Fidnings and Advance Tables. *United Nations*.

United Nations, F. a. (2021). Statistical Yearbook: World Food and Agriculture 2021. *FAO*.

USDA. (2023). *Kansas Direct Hay Report.* USDA, Ks Dept of Ag Market News. Manhattan: USDA. Retrieved November 2023, from https://www.ams.usda.gov/mnreports/ams_2885.pdf

Viola, P., & Jones, M. (2001). Rapid object detection using a boosted cascade of simple features. *IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 1*. doi:10.1109/CVPR.2001.990517

Xu, Y., & et.al. (2023, February). PFGM++: Unlocking the Potential of Physics-Inspired Generative Models. *arXiv*. doi:https://doi.org/10.48550/arXiv.2302.04265

Zou, Z., Chen, K., Shi, Z., Guo, Y., & Ye, J. (2023, March). Object Detection in 20 Years: A Survey. *IEEE, 111*(3), 257-276. doi:10.1109/JPROC.2023.3238524

Appendix A - Code

A1 Bale_tracker.py

```
#!/usr/bin/env python3

from pathlib import Path
import sys
import cv2
import depthai as dai
import numpy as np
import time
from utils.Location_Save import Location_Save
import utils.GPS_Manager as GPS_Manager

# system variables
geoTagging_timer = 5
geotag_deadzone = 100
screen_preview_size = (416,416)

screen_centerx = screen_preview_size[0]/2
screen_centery = screen_preview_size[1]/2

cx1 = int(screen_centerx - geotag_deadzone)
cx2 = int(screen_centerx + geotag_deadzone)
cy1 = int(screen_centery + geotag_deadzone)
cy2 = int(screen_centery - geotag_deadzone)

# folder setup and model path
locationFolder = "/home/ksu-bale/Bale-Tracker/Bale Location Files/"
funcFile = Location_Save(locationFolder).saveLocation()
```

```
        print(funcFile)
        LocationTXTfile = str((Path(__file__).parent / Path(funcFile)).resolve().absolute())
        nnBlobPath = str((Path(__file__).parent / Path('/home/ksu-bale/Bale-
Tracker/Code/BaleNano_openvino_2022.1_6shave.blob')).resolve().absolute())

        Picture_Folder = "/home/ksu-bale/Bale-Tracker/Bale Detection Pictures/"
        picFolder = Location_Save(Picture_Folder).createPictureFolder()

        txtFile = open(LocationTXTfile, 'w')

        # GPS Poller
        global GPS_current_coords, GPS_last_coords
        GPS_current_coords = [0,0]
        GPS_last_coords = [0,0]

        gpsm = GPS_Manager.GPS_Updater()
        gpsm.run()

        if not Path(nnBlobPath).exists():
            import sys
            raise FileNotFoundError(f'Required file/s not found, please run "{sys.executable}
install_requirements.py"')

        labelMap = ["Good Bale", "Bad Bale"]
        syncNN = True

        # Create pipeline

        pipeline = dai.Pipeline()

        camRgb = pipeline.create(dai.node.ColorCamera)
```

```
spatialDetectionNetwork = pipeline.create(dai.node.YoloSpatialDetectionNetwork)
monoLeft = pipeline.create(dai.node.MonoCamera)
monoRight = pipeline.create(dai.node.MonoCamera)
stereo = pipeline.create(dai.node.StereoDepth)
nnNetworkOut = pipeline.create(dai.node.XLinkOut)

xoutRgb = pipeline.create(dai.node.XLinkOut)
xoutNN = pipeline.create(dai.node.XLinkOut)
xoutDepth = pipeline.create(dai.node.XLinkOut)

xoutRgb.setStreamName("rgb")
xoutNN.setStreamName("detections")
xoutDepth.setStreamName("depth")
nnNetworkOut.setStreamName("nnNetwork")

# Properties
camRgb.setPreviewSize(screen_preview_size)
camRgb.setResolution(dai.ColorCameraProperties.SensorResolution.THE_1080_P)
camRgb.setInterleaved(False)
camRgb.setColorOrder(dai.ColorCameraProperties.ColorOrder.BGR)

monoLeft.setResolution(dai.MonoCameraProperties.SensorResolution.THE_400_P)
monoLeft.setBoardSocket(dai.CameraBoardSocket.CAM_B)
monoRight.setResolution(dai.MonoCameraProperties.SensorResolution.THE_400_P)
monoRight.setBoardSocket(dai.CameraBoardSocket.CAM_C)

# setting node configs
stereo.setDefaultProfilePreset(dai.node.StereoDepth.PresetMode.HIGH_DENSITY)
# Align depth map to the perspective of RGB camera, on which inference is done
stereo.setDepthAlign(dai.CameraBoardSocket.CAM_A)
stereo.setOutputSize(monoLeft.getResolutionWidth(), monoLeft.getResolutionHeight())
```

```
stereo.setSubpixel(True)

spatialDetectionNetwork.setBlobPath(nnBlobPath)
spatialDetectionNetwork.setConfidenceThreshold(0.6)
spatialDetectionNetwork.input.setBlocking(False)
spatialDetectionNetwork.setBoundingBoxScaleFactor(0.5)
spatialDetectionNetwork.setDepthLowerThreshold(500)
spatialDetectionNetwork.setDepthUpperThreshold(15000)

# Yolo specific parameters
spatialDetectionNetwork.setNumClasses(2)
spatialDetectionNetwork.setCoordinateSize(3)
spatialDetectionNetwork.setAnchors([10,14, 23,27, 37,58, 81,82, 135,169, 344,319])
spatialDetectionNetwork.setAnchorMasks({ "side26": [1,2,3], "side13": [3,4,5] })
spatialDetectionNetwork.setIouThreshold(0.95)

# Linking
monoLeft.out.link(stereo.left)
monoRight.out.link(stereo.right)

camRgb.preview.link(spatialDetectionNetwork.input)
if syncNN:
    spatialDetectionNetwork.passthrough.link(xoutRgb.input)
else:
    camRgb.preview.link(xoutRgb.input)

spatialDetectionNetwork.out.link(xoutNN.input)

stereo.depth.link(spatialDetectionNetwork.inputDepth)
spatialDetectionNetwork.passthroughDepth.link(xoutDepth.input)
spatialDetectionNetwork.outNetwork.link(nnNetworkOut.input)
```

```
        # Connect to device and start pipeline
        with dai.Device(pipeline) as device:

            # Output queues will be used to get the rgb frames and nn data from the outputs defined above

            previewQueue = device.getOutputQueue(name="rgb", maxSize=4, blocking=False)
            detectionNNQueue = device.getOutputQueue(name="detections", maxSize=4, blocking=False)
            depthQueue = device.getOutputQueue(name="depth", maxSize=4, blocking=False)
            networkQueue = device.getOutputQueue(name="nnNetwork", maxSize=4, blocking=False);

            startTime = time.monotonic()
            counter = 0
            fps = 0
            color = (40, 255, 0)
            printOutputLayersOnce = True
            start_time = time.monotonic()
            irlaser = device.setIrLaserDotProjectorBrightness(765)
            shouldSave = True

            while True:
                inPreview = previewQueue.get()
                inDet = detectionNNQueue.get()
                depth = depthQueue.get()
                inNN = networkQueue.get()

                if printOutputLayersOnce:
                    toPrint = 'Output layer names:'
```

```
            for ten in inNN.getAllLayerNames():
                toPrint = f'{toPrint} {ten},'
            print(toPrint)
            printOutputLayersOnce = False;

        frame = inPreview.getCvFrame()
        depthFrame = depth.getFrame() # depthFrame values are in millimeters

        depth_downscaled = depthFrame[::4]
        min_depth = np.percentile(depth_downscaled[depth_downscaled != 0], 1)
        max_depth = np.percentile(depth_downscaled, 99)
        depthFrameColor = np.interp(depthFrame, (min_depth, max_depth), (0,
255)).astype(np.uint8)
        depthFrameColor = cv2.applyColorMap(depthFrameColor, cv2.COLORMAP_JET)

        counter+=1
        current_time = time.monotonic()
        if (current_time - startTime) > 1 :
            fps = counter / (current_time - startTime)
            counter = 0
            startTime = current_time

        detections = inDet.detections

        # If the frame is available, draw bounding boxes on it and show the frame
        height = frame.shape[0]
        width  = frame.shape[1]
        currenttime = time.monotonic()

        # if else to limit picture and location file saves
        if(not shouldSave):
```

```
                if (currenttime - start_time) >= geoTagging_timer:
                    shouldSave = True
                else:
                    for detection in detections:
                        roiData = detection.boundingBoxMapping
                        roi = roiData.roi
                        roi = roi.denormalize(depthFrameColor.shape[1],
depthFrameColor.shape[0])
                        topLeft = roi.topLeft()
                        bottomRight = roi.bottomRight()
                        xmin = int(topLeft.x)
                        ymin = int(topLeft.y)
                        xmax = int(bottomRight.x)
                        ymax = int(bottomRight.y)
                        cv2.rectangle(depthFrameColor, (xmin, ymin), (xmax, ymax), color, 1)

                        # Denormalize bounding box
                        x1 = int(detection.xmin * width)
                        x2 = int(detection.xmax * width)
                        y1 = int(detection.ymin * height)
                        y2 = int(detection.ymax * height)
                        try:
                            label = labelMap[detection.label]
                        except:
                            label = detection.label
                        cv2.putText(frame, str(label), (x1 - 75, y1 + 20),
cv2.FONT_HERSHEY_TRIPLEX, 0.4, (255,255,255), 3)
                        cv2.putText(frame, str(label), (x1 - 75, y1 + 20),
cv2.FONT_HERSHEY_TRIPLEX, 0.4, 255, 1)
                        cv2.putText(frame, "{:.2f}".format(detection.confidence*100), (x1 - 78, y1 +
35), cv2.FONT_HERSHEY_TRIPLEX, 0.3, (255,255,255), 3)
```

```
                    cv2.putText(frame, "{:.2f}".format(detection.confidence*100), (x1 - 78, y1 +
35), cv2.FONT_HERSHEY_TRIPLEX, 0.3, 255, 1)
                    cv2.putText(frame, f"X: {int(detection.spatialCoordinates.x)} mm", (x1 - 78,
y1 + 50), cv2.FONT_HERSHEY_TRIPLEX, 0.3, (255,255,255), 3)
                    cv2.putText(frame, f"X: {int(detection.spatialCoordinates.x)} mm", (x1 - 78,
y1 + 50), cv2.FONT_HERSHEY_TRIPLEX, 0.3, 255, 1)
                    cv2.putText(frame, f"Y: {int(detection.spatialCoordinates.y)} mm", (x1 - 78,
y1 + 65), cv2.FONT_HERSHEY_TRIPLEX, 0.3, (255,255,255), 3)
                    cv2.putText(frame, f"Y: {int(detection.spatialCoordinates.y)} mm", (x1 - 78,
y1 + 65), cv2.FONT_HERSHEY_TRIPLEX, 0.3, 255, 1)
                    cv2.putText(frame, f"Z: {int(detection.spatialCoordinates.z)} mm", (x1 - 78,
y1 + 80), cv2.FONT_HERSHEY_TRIPLEX, 0.3, (255,255,255), 3)
                    cv2.putText(frame, f"Z: {int(detection.spatialCoordinates.z)} mm", (x1 - 78,
y1 + 80), cv2.FONT_HERSHEY_TRIPLEX, 0.3, 255, 1)

                    cv2.rectangle(frame, (x1, y1), (x2, y2), color, thickness=2)#,
cv2.FONT_HERSHEY_SIMPLEX)

                    centroidx = (x1+x2)/2
                    centroidy = (y1+y2)/2

            else:
                for detection in detections:
                    roiData = detection.boundingBoxMapping
                    roi = roiData.roi
                    roi = roi.denormalize(depthFrameColor.shape[1], depthFrameColor.shape[0])
                    topLeft = roi.topLeft()
                    bottomRight = roi.bottomRight()
                    xmin = int(topLeft.x)
                    ymin = int(topLeft.y)
                    xmax = int(bottomRight.x)
```

```
                ymax = int(bottomRight.y)
                cv2.rectangle(depthFrameColor, (xmin, ymin), (xmax, ymax), color, 1)

                # Denormalize bounding box
                x1 = int(detection.xmin * width)
                x2 = int(detection.xmax * width)
                y1 = int(detection.ymin * height)
                y2 = int(detection.ymax * height)
                try:
                    label = labelMap[detection.label]
                except:
                    label = detection.label
                cv2.putText(frame, str(label), (x1 - 75, y1 + 20),
cv2.FONT_HERSHEY_TRIPLEX, 0.4, (255,255,255), 3)
                cv2.putText(frame, str(label), (x1 - 75, y1 + 20),
cv2.FONT_HERSHEY_TRIPLEX, 0.4, 255, 1)
                cv2.putText(frame, "{:.2f}".format(detection.confidence*100), (x1 - 78, y1 +
35), cv2.FONT_HERSHEY_TRIPLEX, 0.3, (255,255,255), 3)
                cv2.putText(frame, "{:.2f}".format(detection.confidence*100), (x1 - 78, y1 +
35), cv2.FONT_HERSHEY_TRIPLEX, 0.3, 255, 1)
                cv2.putText(frame, f"X: {int(detection.spatialCoordinates.x)} mm", (x1 - 78,
y1 + 50), cv2.FONT_HERSHEY_TRIPLEX, 0.3, (255,255,255), 3)
                cv2.putText(frame, f"X: {int(detection.spatialCoordinates.x)} mm", (x1 - 78,
y1 + 50), cv2.FONT_HERSHEY_TRIPLEX, 0.3, 255, 1)
                cv2.putText(frame, f"Y: {int(detection.spatialCoordinates.y)} mm", (x1 - 78,
y1 + 65), cv2.FONT_HERSHEY_TRIPLEX, 0.3, (255,255,255), 3)
                cv2.putText(frame, f"Y: {int(detection.spatialCoordinates.y)} mm", (x1 - 78,
y1 + 65), cv2.FONT_HERSHEY_TRIPLEX, 0.3, 255, 1)
                cv2.putText(frame, f"Z: {int(detection.spatialCoordinates.z)} mm", (x1 - 78, y1
+ 80), cv2.FONT_HERSHEY_TRIPLEX, 0.3, (255,255,255), 3)
```

```
                cv2.putText(frame, f"Z: {int(detection.spatialCoordinates.z)} mm", (x1 - 78, y1
+ 80), cv2.FONT_HERSHEY_TRIPLEX, 0.3, 255, 1)

                cv2.rectangle(frame, (x1, y1), (x2, y2), color, thickness=2)#,
cv2.FONT_HERSHEY_SIMPLEX)

                centroidx = (x1+x2)/2
                centroidy = (y1+y2)/2

                if(centroidx <= (screen_centerx + geotag_deadzone) and centroidx >=
(screen_centerx - geotag_deadzone)):
                    if(centroidy <= (screen_centery + geotag_deadzone) and centroidy >=
(screen_centery - geotag_deadzone)):
                        detAxis = [detection.spatialCoordinates.x,
                            detection.spatialCoordinates.y,
                            detection.spatialCoordinates.z]
                        detDict = gpsm.get_detection_dict(detAxis)
                        print("Box screen coords: ("+str(x1)+","+str(y1)+") |
"+"("+str(x2)+","+str(y2)+")")

                        detCoordStr = str(str(label) + ', ' + str(detDict['lat2']) + ', ' +
str(detDict['lon2']) + '\n')
                        txtFile.write(detCoordStr)
                        PicturefuncFile = Location_Save(picFolder).savePicture()
                        cv2.imwrite(PicturefuncFile, frame)

            start_time = time.monotonic()
            shouldSave = False

        cv2.putText(frame, "NN fps: {:.2f}".format(fps), (2, frame.shape[0] - 4),
cv2.FONT_HERSHEY_TRIPLEX, 0.4, color)
```

```
        # enable this to show the depth color map window
        # cv2.namedWindow('depth', cv2.WINDOW_KEEPRATIO)
        # cv2.imshow('depth', depthFrameColor)
        # cv2.resizeWindow('depth', 416,416)

        cv2.namedWindow('Bale Detection', cv2.WINDOW_KEEPRATIO)
        screenFrame = frame
        cv2.rectangle(screenFrame, (cx1, cy1), (cx2, cy2), color, thickness=1)
        cv2.imshow('Bale Detection', screenFrame)
        cv2.resizeWindow('Bale Detection', 960,960)

        if(cv2.waitKey(1) == ord('q')):
            print("q pressed")
            break
        elif(cv2.getWindowProperty('Bale Detection', cv2.WND_PROP_VISIBLE) < 1):
            print("closed")
            break
```

A2 GPS_Manager.py

```
#!/usr/bin/env python3
"""
Benjamin Vail
Kansas State University
BAE Department
```

Continuously polls the GPS for location data and calculates the heading direction from previous coordinates.

```
"""

import utils.GPS_Poller as GPS_Poller
import time
from geographiclib.geodesic import Geodesic
from threading import Thread
import math

class GPS_Updater(object):
    def __init__(self):
        super(GPS_Updater,self).__init__()

        self.current_coords = None
        self.last_coords = None
        self.GPS_thread = None
        self.should_stop = False
        self.geod = Geodesic.WGS84
        self.distance = 0
        self.inv_dict = None
        self.dir_dict = None

    def run(self):
        self.GPS_thread = Thread(target=self.gps_thread, daemon=True)
        self.GPS_thread.start()
        return

    def get_GEO_dict(self):
        if(self.last_coords != None):
            self.inv_dict = self.geod.Inverse(self.last_coords[0],
                                self.last_coords[1],
                                self.current_coords[0],
```

```
                        self.current_coords[1]
                        )

    else:
        print('No geo dictionary found')
        pass

def get_detection_dict(self, detAxis):
    x = detAxis[0]/1000
    y = detAxis[1]/1000
    z = detAxis[2]/1000

    direction_azi = self.inv_dict['azi1']
    if(direction_azi < 180):
        direction_azi += 180
    else:
        direction_azi -= 180

    cam_distance = math.sqrt(x**2+y**2+z**2)
    # cam_plunge = math.degrees(math.asin(z/cam_distance))
    try:
        cam_object_camera_azimuth = math.degrees(math.atan(x/y))
    except ZeroDivisionError:
        y += 0.1
        cam_object_camera_azimuth = math.degrees(math.atan(x/y))

    dar = math.radians(direction_azi)
    car = math.radians(cam_object_camera_azimuth)
    world_object_azimuth = math.degrees(dar + car)

    self.dir_dict = self.geod.Direct(self.current_coords[0],
```

```
                                    self.current_coords[1],
                                    world_object_azimuth,
                                    cam_distance
                                    )
        # print(str(self.dir_dict['lat2']) + ',' + str(self.dir_dict['lon2']))
        return self.dir_dict

    def gps_thread(self):
        global GPS_current_coords, GPS_last_coords
        while(self.should_stop == False):
            self.last_coords = self.current_coords
            GPS_last_coords = self.last_coords

            self.current_coords = GPS_Poller.run()
            lat = self.current_coords.lat
            lon = self.current_coords.lon
            self.current_coords = [lat,lon]
            GPS_current_coords = self.current_coords
            self.get_GEO_dict()

            time.sleep(1)

    def stop(self):
        self.should_stop = True
        self.GPS_thread.join()
        return

# if __name__ == '__main__':
#   GPS_Updater
#   print("Imported")
```

A3 GPS_Poller.py

```
"""
Benjamin Vail
Kansas State University
BAE Department

Polls the GPS module for location coordinates and returns them to the GPS_Manager

"""
from ublox_gps import UbloxGps
import serial

def run():
  port = serial.Serial('/dev/ttyACM0', baudrate=38400, timeout=1)
  gps = UbloxGps(port)
  try:
    coords = gps.geo_coords()
    port.close()
    return coords

  except (ValueError, IOError) as err:
    print(err)

  # finally:
  #   port.close()

# if __name__ == '__main__':
#   run()
```

```
"""
Benjamin Vail
Kansas State University
BAE Department

Utility script to determine where to save the location and picture files and manage
file/folder names.

"""

import os
import datetime
import sys

class Location_Save():
  def __init__(self,saveFolder):
    super(Location_Save,self).__init__()
    day = datetime.datetime.today().day
    month = datetime.datetime.today().month
    year = datetime.datetime.today().year
    dateVar = str(month) + "-" + str(day) + "-" + str(year)
    self.sF = saveFolder
    self.finalFolderPath = self.sF + dateVar + "/"

  def saveLocation(self):

    if(os.path.isdir(self.sF)):
      pass
    else:
      os.mkdir(self.sF)
```

```
    if(os.path.isdir(self.finalFolderPath)):
      pass
    else:
      os.mkdir(self.finalFolderPath)
    i=0
    while(os.path.isfile(self.finalFolderPath + "Baling Run " + str(i) + ".txt")):
      i = i + 1
    FileToReturn = self.finalFolderPath + "Baling Run " + str(i) + ".txt"

    return FileToReturn

  def savePicture(self):
    i=0
    while(os.path.isfile(self.sF + "Bale " + str(i) + ".jpg")):
      i = i + 1
    FileToReturn = self.sF + "Bale " + str(i) + ".jpg"

    return FileToReturn

  def createPictureFolder(self):
    if(os.path.isdir(self.sF)):
      pass
    else:
      os.mkdir(self.sF)

    if(os.path.isdir(self.finalFolderPath)):
      pass
    else:
      os.mkdir(self.finalFolderPath)
    j=0
    while(os.path.isdir(self.finalFolderPath + "Baling Run " + str(j))):
```

```
      j = j + 1
    os.mkdir(self.finalFolderPath + "Baling Run " + str(j))
    returnDIR = self.finalFolderPath + "Baling Run " + str(j) + "/"

    return returnDIR

# if __name__ == '__main__':
# if(len(sys.argv) < 1):
#     exit("Missing args")

# saveLocation(sys.argv[0])
```