Automated Failure Detection in 3D Printing Using Artificial Intelligence

Kase Johnson
Advised by: Amr Hilal
Tennessee Tech University
Computer Science
Fall, 2024

Abstract

3D printing has become more and more common in the past decade, with its success and growing popularity attributed to it's rise in availability and lowering prices. Manufacturers, designers, and hobbyists that undertake 3D printing in their homes and shops often have no time for monitoring their day long, or sometimes even longer, prints. Fused Deposition Model (FDM) printing in particular, is subject to excessive print failures that can go undetected for many hours if no person or system is monitoring them, which can lead to wasting filament, time, and money. Monitoring and detection of FDM-3D printing failures is an important step towards ensuring process efficiency and reducing the waste associated with 3D printing. This paper introduces an approach to get over this hurdle. This project explores the application of image based failure detection through deep learning with a YOLO architecture. The focus is on utilizing a dataset comprised of images captured from the printers video camera during the process of printing. While other papers have researched this issue in similar ways, there are gaps in each that we try to fill. Using a YOLO ML model, this study will isolate important visual indicators of printing failures in order to accurately identify these errors early in their life. We use ResNet-50 and EfficientNetb0 as baseline models to compare the performance of binary classifiers against the broader detection capabilities that YOLO models can offer, resulting in a model that gives results that are both trustworthy and accurate. Our objective is to develop a model capable of real-time analysis and detection of failures in 3D prints early in the failure lifecycle, resulting in more efficient use of money, filament, time, and power.

Project Goal(s)

The goal for our project is to create a machine learning model capable of detecting failures in 3D prints by processing images captured by a camera built into the printer that captures the entirety of the print area. This goal can be split into many smaller goals that need to be achieved under certain parameters. All goals are outlined in more detail in the following subsections:

Accuracy

As mentioned, our main goal is to create an accurate model. *Accurate* can be an open-ended word, so we define it here. The most important metric for our model is the recall score, defined below.
Recall: "Of all the images that are actual failures, how many did the model correctly flag as failures?"

$$\text{Recall} = \frac{\text{True Positives (TP)}}{\text{True Positives (TP)} + \text{False Negatives (FN)}}$$

So, a recall score of 0.6 would mean that, out of all real failures, 60% were accurately captured. Precision is another important metric to consider.
Precision: "Of all the images the model flagged as failures, how many are actually failures?"

$$\text{Precision} = \frac{\text{True Positives (TP)}}{\text{True Positives (TP)} + \text{False Positives (FP)}}$$

This means that precision measures the ability of our model to avoid false alarms. It evaluates how many of the predicted failures are actual failures. So if we have a precision score of 0.6, that means that out of all the images the model identified as failures, 60% are actual failures, whereas the other 40% are not failures, and can be considered false positives. Precision is an important metric, however, in our case, a higher recall score (even at the expense of some precision) is the more desirable metric. Precision would be most valued in scenarios where the consequences of false positives (misidentifying something as a failure) would be worse than those of false negatives (missing a failure). In our project, if the model notifies the user that a failure has occurred, but it hasn't, this may be a minor inconvenience for the user to check the camera or to check the model themselves, but with the goal of catching as many failures as possible when they happen, it is more important to over predict failures than under predict. That being said, there is only a certain trade off associated with this that is acceptable. For example, if we were to have a recall score of 90%, but a precision score of 25%, this is not acceptable. Since our model may be testing thousands of images per printing session, the user would be notified almost any time an error occurred, but 75% of the notifications would be false positives, which would be extremely

inconvenient for the user. Thus, it is important to find a balance of precision and recall that maximizes both as much as possible. This is where the F1-Score comes into play.
F1-Score: "The harmonic mean of Precision and Recall, providing a balance between the two metrics."

$$\text{F1-Score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

The F1 score is the harmonic mean of Precision and recall, and is meant to give a balance between the two metrics. It is useful in our project, because it gives a good one-look "*accuracy*" rating for our model, however, it is still important to consider the other metrics and review their scores. For example, a Recall of 0.9 and Precision of 0.2 would result in the same F1-score as a Precision of 0.9, and a Recall of 0.2, however, we may favor one of these results over the other, so F1 alone is not a good metric, but paired with results from precision and recall, we get much better insights into how our model is performing.

Computational Power

Computational Power is not necessarily a goal, but rather a condition that we need to meet. Part of our goal is for our model to have the ability to run *inference* on small micro computers such as Raspberry Pi's. Of course, training our models on these devices is not a requirement, but the ability to run inference on them is a must. For our model to be useful to the everyday consumer, it will need to be able to be deployed to a lightweight computer that is both cost efficient and has a small physical footprint, as most people will not want to run a full desktop computer that pulls 600+ Watts of power at all times in order to keep an eye on their prints while they are away. We will not need a high FPS (Frames Per Second) rating for our model when deployed, since our prints change very little over the course of 2-3 layers. More frames per second would be beneficial for something like a self driving car where inference speeds are crucial for safe operation, but in our case, running inference on 4-5 images per layer of the print will be plenty due to the slowly changing nature of the prints. This also gives more opportunity to use more complex models that may perform better. We choose the number of (at least) 4-5 images per layer because in some images the extruder may be blocking parts of the print, so we would like to have more than a few images per layer to infer on, so that we get the full breadth of the print. The fastest commercial printers right now are able to print at around 500mm/s, whereas the lower tier printers usually cap around 120mm/s. With the fastest commercial printer on the market, an average print, 5 inches tall, 0.2mm layer height, and 2 hours long will take around 15 seconds per layer. Thus, an inference speed around 4 seconds per image (0.25 FPS) will suffice.

According to Ultralytics, YOLOv11N with pytorch on a raspberry pi 5 runs inference in around 500 milliseconds, whereas YOLOv11S takes around 1200 ms running on an image size of 640x360. Most of our images are 960x540, so this time will be lowered if we use our image size of 960x540. These times are based on YOLOv11, rather than YOLOv10. According to real time inference speeds [16], YOLOv11 is only around 2% faster than YOLOv10, so these numbers can accurately represent the speeds YOLOV10 will find as well, although off by a small margin.

No data is given on the inference times for Raspberry Pi's on the medium and larger models, however, given the latency, we can estimate the speeds it might achieve based on the speeds of the small and nano models. [21] YOLOv10-S is around 90% faster than YOLOv10-M, and anything larger than medium model size will most definitely be too large according to the creators of the YOLO models Ultralytics, so we focus on the nano and small modules mostly. We do run some of our training on medium, since there are some ways we can speed this up, such as switching from pytorch to other formats such as ONNX or NCNN which give lower inference speeds[21]. ONNX, or Open Neural Network Exchange, is an open source model format meant to make deep learning models interoperable and more efficient [7]. NCNN, or No-name convolutional Neural Network, does much the same as another open source framework that reduces the size of a model to speed up inference times [27]. Unlike Pytorch, you cannot train and run inference in ONNX or NCNN, as they are both exclusively inference frameworks. Converting a model to ONNX or NCNN usually greatly improves inference time for devices such as Raspbery Pi's, however, there can be some challenges in converting from one model to another [14], so we do not rely on conversion in our project, but rather leave it as an option. From all deployments we have seen of YOLO to Raspberry Pi's and other small devices, medium has been much less common than small and nano [3] [2] [9].

Adaptability

The last goal is for our dataset to be adaptable to different environments. With our goal of these being able to be deployed to various different 3D printers, regardless of brand, size, open or enclosed, or other factors, our model needs to work well across many environments. It should not perform much differently on an Ender 3 V2 than a Bambu Labs P1S. Some environmental factors might make it harder to detect failures, such as if the background behind the printer contains lots of noise, but the change in printer or environment should have very little impact to our models performance.

Justification

3D printing technology has revolutionized, and continues to revolutionize manufacturing. The price of a consumer grade printer in 2012, on average, was between $1,000 and $2,000 USD , now, in 2024, the average price of a consumer grade printer has lowered all the way to between $100 and $400. With the lowering price, everyday designers and hobbyists are able to make physical creations of any digital 3D file they can think up and bring them into fruition in the comfort of their own home. The ability to create replacement parts, tools, household accessories, or even art pieces from home is a benefit that can both save time,

money, and decrease CO2 emissions that are incurred by driving to the store or getting those items delivered to your house. Although, its not all advantages, there are still some disadvantages associated which can't be ignored as they bring time, money, and financial consequences. Printing errors and failures still plague every printer on the market. Whether a $200 Ender 3 or a $1,500 Bambu Labs X1 Carbon is used, you will still run into some errors in your day to day prints. Some of the more expensive printers have gone above and beyond to ensure higher print success by implementing a plethora of somewhat reliable error detection steps that are useful for some errors, but not all printers are created equal in this regard, the common household printers, such as Creality's Ender line, have very little monitoring/error compensation since they are targeted to be on the more affordable side and adding these features would raise their price. An error in your print wastes both your time, your machines time, money, filament, and electricity.

It is estimated that there are between 300,000 and one million 3D printers in North America alone [18] [19]. On the conservative side, an average print is around 50 grams of filament, assume an average print time of 3 hours, average failure rate being 10% [4] (which it is typically much higher), and the average cost of a kilogram of filament is $18. This means that one gram of filament is worth about 1.8 cents, and the average print worth 90 cents. If every single person in the United States alone printed one failed print in their entire 3D printing lifetime, that would result in a loss of $270,000 in filament alone, 15,000 kg/ 33,070 pounds of filament, and 900,000 hours which is 102 years 8 months and 12 days of print time. While this is spread across 300,000 users, this is, once again, an estimate of if only one failure happened per person. After years of printing even inconsistently, hundreds of prints will fail, resulting in hundreds of dollars, dozens of kilograms of filament, and hundreds of hours wasted, not to mention the general frustration the user will find. This is why alleviating these errors is so important to us, to save the user frustration, time, money, and the earth while were at it. We will be working specifically with printing errors associated with (FDM) printers, which intake a small strand of filament on a spool, and build the item/model tiny layer by tiny layer until it is finished.

There are other types of 3D printing, such as SLA printing, which uses a laser to cure a liquid resin solution to build prints. We have chosen FDM printing failure detection over SLA for several reasons. The first is that FDM printers are more common and affordable than SLA printers. This is because SLA printers use resin, which is much more expensive than filament. SLA printers also require more space than FDM printers because they usually need a curing station (a separate machine) to cure the resin when the print is finished, they also need to be in a room of their own or have proper ventilation outside because breathing in the fumes from the resin is toxic. Many of the newer FDM printers are much faster and nearly all of them have larger bed volume compared to SLA printers, which is appealing to a wider audience since they can create much larger prints. All these factors lead to FDM printers being more widely used and accepted compared to SLA printers, which in turn contributes to our choice of failure detection in FDM pritning over SLA. Other than creating printers that are fully reliable and have no chance of error occurring, which is highly unlikely to happen at all, much less in the near future, the only other way to ensure time, filament, and electricity are not wasted on failed prints is to implement continuous monitoring and catch the errors/failures early on in their lifespan. Continuous monitoring will help to ensure precious filament, money, and time is not wasted on undetected failed prints and printer time is maximized for successful prints, rather than continuing with unsuccessful ones.

Literature survey

3D printing technology has many papers and articles written and published in various journals, by various institutions, over various topics. However, most of these studies focus on theory and how 3D printing can be used in specific fields in the future, such as the paper by Tracy et al. [20], which provides an overview of 3D printing in drug development and highlights the use 3D printing has, but gives no insight into how the process can be improved. There are also some "hopeful" suggestions of how 3D printing can be used in the future, such as for use in food/meat printing [26]. Some papers also introduce and make the case for other bio-chemical compounds for use in 3D printing [12] to reduce the environmental impact and increase sustainability, which is more along the lines of our goal, but not quite there. While theory and use cases of 3D printing have been researched extensively, there are much fewer papers relating to the error detection and monitoring of prints, or further development of fail-safes, and even fewer of these have experimental results rather than theory and conjecture. There are, however, some papers that give very valuable insights into our specific implementation and attempt the same goal. "Automatic Error Detection in 3D Pritning using Computer Vision" [10] by Langeland, Stig Andreas Kronheim, from the University of Bergen gives an extremely in-depth overview of their project, however, they take a very different approach to their detection and the failures detected are limited.

They created two new modules, used as extensions to YOLO (but not part of the core YOLO framework), to test for errors at different points in the prints, both of which use cameras placed at different angles to capture images of the print bed. The first module (called First Layer Verification) tests the first layer of the print for inconsistencies (the first layer of prints is often one of the largest deciders of whether a print will succeed or fail, at least in lower end commercial printers) by comparing it to a simulated 3D model of the specific print. This seems limited in scope, since this module is not nearly as scalable as an image detection neural network might be, since this module would need to be fed the 3D model of every print before that model is sent to the printer. The second module, called the

Nozzle Analyzer, monitors the movement of the nozzle and is meant to detect some errors, such as detached layers or obstructions, but it struggled with certain movement direction and cases where the filament got stuck and altered their segmentation accuracy. Their detection ”modules” are interesting, and have promise for the future, however they do not provide numerical results for their detection accuracy, recall, or precision, and their testing environment was limited to few tests being done. The paper states that a deep learning computer vision model would be more accurate and more realistic to deploy into the real world, however, they decided against this because

> ”Collecting thousands of images for training an image classification model of defects in 3D printing will be very time consuming and, to best of our knowledge, no such data sets seems to exits.”

. The techniques employed in this paper hold future promise, specifically for the adaptation of YOLO by adding new modules that increase overall accuracy, the goal of this paper does not completely coincide with our goal, and as such, is not ideal for our use case. This is also not able to scale at the rate we would prefer, with our implementation having the capability of being used by most any commercial user since it implores a deep learning model. The evaluation metrics and results are not shown, and it requires more hardware (2 cameras versus 1 in our case) than we would prefer to implore in our approach.
Paraskevoudis et. al [15] also attempts a failure detection model specifically for ”stringing” failures by using deep learning computer vision. Unlike our model, which uses YOLOv8 and YOLOv10, they decided to use a similar model called Single Shot Detector (SSD) [13]. While both models are derivations of a Convolutional Neural Network (CNN), they have differing strengths and weaknesses. Single Shot Detector’s are meant to localize much better than regular binary neural networks that do not intend localization, such as AlexNet or ResNet, and they do this in a single forward pass. Doing this in a single forward pass means that it performs both object localization and classification in one step, predicting bounding boxes and probabilities directly from feature maps. This makes it usable for real-time use, but still slower and takes more computational power than most YOLO models, especially the newer YOLOv8-11’s. With the goal of our model being able to be deployed to microcomputers like raspberry Pi’s, this is something to keep in mind. YOLO is also considered more scalable and customizable since it is anchor free and has a modular architecture, whereas SSD does not. With these things in mind, YOLO would be the best option for our needs.

Paraskevoudis et. al also have limitations in its dataset. Their dataset consists of 500 images, which is not a trivial amount, but is still a bit low. While they do 5 different image augmentations to bring that number to 2,500, there is also a limitation in the prints being considered. They only trained their model on one single 3D design, which consists of two identical towers being printed a certain distance from one another. This print is specifically to test stringing, which usually occurs when the nozzle tries to pull back the filament when it is done with one area, then it moves to another and pushes the filament back out. During that process, the filament is normally still warm, so some of it may drag along that area in between the towers. This print type is usually used to fine tune the printer to remove or alleviate some of the stringing problems. Because they train and test on only this design, it cannot be assumed that their model will work as well with more complex or different prints without the data being shown. The results returned from their experiment are also limited, with their highest precision (percentage of correctly identified failure images out of all images the model labeled as failures) over varying hyper parameters being 0.44 and their highest Recall (percentage of correctly identified failure images out of all actual failure images in the dataset) being 0.69, giving an F1-Score of 0.55. Without testing their model on a foreign dataset or different designs, it is unclear whether these numbers will be able to scale to an environment with differing variables such as the background, lighting exposure, filament color, and 3D model used. Table 1 shows a comparison of YOLO and SSD.

Another paper, Karna et al. [9] attempts to achieve a goal even more similar to ours, by detecting 3D print failures using a YOLOv8 model. There are many insightful findings that can be drawn and concluded from their work. [9] Karana et al. tests their data training on multiple sizes, from YOLOv8n (the smallest model) to YOLOv8x (the largest), with their best performance coming from YOLOv8s. *A comparison of these models with their counterparts in YOLOv10 can be seen in Table 2* [23] They tested their model on various hyperparameter customization’s and gave an overview of the results that each of these displayed on different models, giving us an insight into how each model performed, as well as how each hyperparameter change affected the results of each test. Their model was meant to train on 3 failures, which they label stringy, obstacle, and unstick. Stringy, more commonly called stringing or spaghetti printing, occurs when the filament starts extruding over a blank or unsupported space (can happen for many reasons) and the filament just starts ”stringing” with no form. *Obstacle* occurs when something prevents the extruder head from going where it needs to go. *Unstick*, more commonly called a bed adhesion failure, occurs when the model ”unsticks” itself from the base of the print bed and starts to warp up or completely come unglued from the print bed. These three failure types cover the breadth of all 3D print failures very well, and the best model they trained exhibits an impressive 89.7% precision in their detection.

Research Gaps

However, these experiments, and many like it, have gaps that we have identified and seek to fill in. The research gaps we focus on mostly come from [9] Karana et al. since this paper is the state of the art and best performing solution.

Feature	SSD	YOLOv10
Architecture	Multi-scale feature maps, anchor boxes	Single network, anchor-free
Speed	Real-time capable, but slower than YOLOv10	Very fast, optimized for real-time
Accuracy	Good, struggles with small objects	Higher accuracy, especially with small objects
Customization	Limited, less modular	Highly modular, supports custom tasks
Ease of Training	Stable, but limited flexibility	Adaptable, easier to fine-tune
Community Support	Established, but less active	High, with robust tools and libraries

Table 1: Comparison of Single Shot Detection and YOLOv10

Print Shapes Firstly, their model only trains on 3 printing shapes, a rectangle, cube, and a cylinder. This does not capture the breadth of models that will be printed by your everyday consumer and thus will be ill-fit for the goal we have in mind, being scalable and adaptable to any printer and any print. Most everyday consumers have two desires in mind when printing: the first is to create functional parts on demand, such as tools, latches, hinges, storage pieces, etc. the second is for fun, such as cosplay items, Dungeons and Dragons miniatures, Warhammer 40k miniatures, and game/movie memorabilia. Only training on these 3 shapes limits our ability to detect failures in more complex structures and pieces.

The Data Another potential drawback to their experiment was the data. Their data was captured in much the same way as ours, with a camera in a fixed position pointing towards the printers bed. However, their dataset consists of 20 time-lapses with a little over 5,000 images total. It is never stated how much of that data is failure data and how much is success data, which could skew the model in the wrong direction if we are trying to detect failures. Based on images provided in the paper and the lack of a dataset to view, the data is at least not completely failure data, since some of the images provided in the text are successful prints. Information on how the data is split between successful time-lapses and failure time-lapses is not provided, and as such, it is hard to validate how well their model can be trained without the fear of generalizing or overfitting. The authors were contacted in order to find out the exact split as to not make assumptions, but no response has been received. [10] Langeland et al.

The classes trained Another gap identified in this paper is the classes that have been trained on. We mentioned they train on 3 different failures, however, this is not all of their classes. They also seek to identify each print shape in their paper, so they have classes for Cylinders, Rectangles, and Squares, which gives 6 classes total, 3 to identify the shape of the print, and 3 to identify failures. Since they are training on all 6 of these classes rather than just failures, and we do not know the split of failure vs successful time-lapses, it is impossible to validate the results on only failure bounding boxes, which is what we are focused on. This could be giving a higher F1-score, precision, and recall value's than if they were to train on failures alone, since we do not know their failure-success split, and since failures would be smaller and harder to detect than the print shape itself.

Evaluation Metrics While not necessarily a gap in their research, they focus more on the mAP(50) and mAP(50-95) values than precision and recall. mAP(50) stands for the Mean Average Precision of the bounding box compared to the ground truth, in the mAP(50) case, the prediction is correct if the predicted bounding box overlaps with the ground truth bounding box by at least 50%. mAP(50-95) is the average precision when the prediction overlaps with the ground truth over multiple Intersection Over Union (the metric used to test how much of the predicted bounding box overlaps with the ground truth) values, starting at 50 and increasing in 5% increments until it reaches 95, therefore 50-95 is the much more stringent metric. The model proposed in Karna et. al [9] focuses on these values and works well at predicting the bounding boxes within those ranges extremely accurately. Since their proposal includes a more advanced YOLO model with an extra module inserted, the mAP values were increased as a byproduct of their additional module. Although their model does a good job at predicting the mAP values extremely accurately, their results never state their precision scores. Since our printed pieces are more complex, often with many pieces on the print bed at a time and complex shapes, it is not a necessity to get absolute precision with our bounding boxes and is much less likely, since our images are less uniform than theirs. Their goal is more aligned with predicting the bounding boxes mAP values accurately, yet our goal is to detect and localize failures as much as possible in order to alert of failed prints early in their life-cycle.

Our Contribution With this research, we seek to fill in some of the gaps from the papers mentioned as well as gaps left by other papers. We plan to do this by training on a larger dataset, on many different shapes and sizes of prints, and training only on failure classes with our YOLO model, which we believe gives more comprehensive and trustworthy metrics for detecting failures in 3D prints.

Design Discussion

Originally, we had decided to use YOLOv8 as our main model in our experimentation. Much of our training and hyperparameter optimization was done early on with YOLOv8, and results from that model will be displayed in the results as well, but we eventually decided to use the to the newer

YOLOv10 [25] as well to compare our results, and also since v10 uses less computational power. YOLO stands for You Only Look Once, with the v8 and v10 being version 8 and version 10 respectively (YOLOv11 was officially released in October of 2024). YOLO is an object detection model which is best known for its speed, accuracy, and ability to be used in real-time applications with little processing power compared to other object detection models. Its speed was an important aspect in our consideration of what model to choose since we would like for the model to have the ability to be deployed to a microcomputer such as a raspberry pi with no problems. With this ability, it would be both inexpensive and convenient for hobbyists or even manufacturers to set up and use in their homes.

Image Lifecycle

Our goal is to connect a camera in our printer faced towards the printing bed, with full view of each print, to a microcomputer running our AI model. This camera would send pictures to our model, and our model would process them in real time to tell whether an error has occurred or not. Firstly, we must train our model on the failures it will need to look out for, and we start with Preprocessing these images.

Preprocessing

Backbone/Feature Extraction The backbone is a section of the architecture in both YOLOv8 and YOLOv10 where feature extraction occurs. The backbone is based on an enhanced CSPNet variation called CSPDarknet [6]. While the documentation and papers on YOLOv8 specify the CSPDarkNet variation as CSPDarkNet-53 (CSPDarknet with 53 convolutional layers), nowhere is it specified if there is a difference in the amount of convolutional layers in v10 versus v8. The only difference claimed in documentation is that further optimization has been introduced to v10 with additional layer modifications, better gradient flow, and reduced computational redundancy [22] [25]. CSPNet stands for Cross Stage Partial Network and it is used to split the ”feature map” into two pieces and uses a strategy called split-and-merge to save computational power and time. Split-And-Merge reduces the computations needed by splitting each feature map into two pieces, and sends one of those feature maps into additional convolutional layers while the other bypasses those layers. The model applies series of convolutional ***filters*** (or ***Kernels***) [5] [11] to each image, with each filter detecting a specific feature, whether that be edges, textures, shapes, colors, etc. One convolutional layer is usually made up of dozens to even hundreds of filters, with each of those filters creating a feature map, that are then stacked on top of each other to create the output of that layer, which will then be passed to the next layer for further *convoluting*. These *filters* are in the form of a grid/matrices, usually a 3x3 or 5x5 matrix (mostly 3x3 and 1x1 in our case for capturing of more fine grained details). This matrix slides across the image at the rate of the ***stride*** specified and creates feature maps by performing *element-wise multiplication* [11]. Figure 1 shows an example of how a filter would slide across an input image with a stride of 1 and matrix size of 3x3.

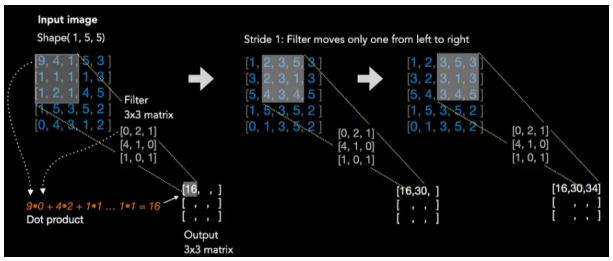

Figure 1: Example of a 3x3 filter sliding across an input image with a stride of 1

After being convoluted, all feature maps will be transformed non-linearly using some activation function (the most common being ReLU, which stands for Rectified Linear Unit). ReLU's function is to replace all negative values in a feature map with 0's, which helps the network model complex patterns. The activation function most commonly used in the hidden layers in YOLO architecture is Leaky ReLU, which replaces *most* negative values with 0's, leaving some of the negative values closer to 0 untouched. While ReLU is used for hidden layers, output layers use Sigmoid activation for the confidence scores and bounding box predictions, since it mAPs every value to a range between 0 and 1, it is often used for binary classification tasks. After the activation function has been applied, normally (especially in binary classifiers) a pooling layer would be used to reduce the spatial complexity of feature maps in output layers. However, that is not the case with YOLOv8 and v10. Instead, they use *strided convolutions* to reduce the spatial complexity, so instead of a convolution using a stride of 1, like in our example in Figure 1, it would use a higher number, usually 2, rarely ever over 3. This is another attribute that contributes to the minimal computational power that YOLO achieves and its ability to be deployed real-time easily.

Neck Features from different stages of the backbone are then *aggregated* to enhance the models ability to detect different sized objects and objects with differing complexities. It does this by using something called a Path Aggregation Network (PAN). The PAN [22] in YOLOv10 aggregates features from multiple levels of the model to allow information from lower level layers that have finer details to merge with high level layers (called up-sampling and down-sampling) with more coarse details to create a set of feature maps that are more robust.

Head The Head, as the last layer of the model, is responsible for making the predictions we see. This includes the confidence scores, bounding boxes, and the classification of the objects in each box and image. The head's are where the main differences in YOLOv8 and YOLOv10 are found. YOLOv8 has a single-head structure where there is only one set of prediction layers that generates multiple

Model Variant	Parameters (M)	FLOPs (G)	mAP val 50-95 (%)	Latency (ms)	Conv Layers
YOLOv8-N	3.2	8.7	37.3	6.16	64
YOLOv8-S	11.2	28.6	44.9	8.52	64
YOLOv8-M	25.9	78.9	50.2	12.3	84
YOLOv8-L	43.7	165.2	52.9	18.4	104
YOLOv8-X	71.7	257.8	53.9	26.8	104
YOLOv10-N	2.3	6.7	39.5	1.84	108
YOLOv10-S	6.7	21.6	46.3	2.49	112
YOLOv10-M	16.0	59.1	51.1	4.74	142
YOLOv10-L	30.6	120.3	53.2	7.28	180
YOLOv10-X	43.5	160.4	54.4	10.7	198

Table 2: Comparison of YOLOv8 and YOLOv10 Model Variants

predictions for each location on the feature maps. Each location is still able to give multiple bounding boxes, but Non-Maximum Suppression (NMS) [22] is used to filter out the duplicates. To make the predictions, it divides the input image into a grid and assigns each grid cell as positive or negative. Positive means that the grid is close to the center of a bounding box, whereas negative means it does not contribute to a bounding box at all. The main improvements with YOLov10 are that it uses a dual-head structure, one of which being a one-to-many head (the same as seen in v8) that generates multiple predictions for each location (which should in theory give better recall scores), and the other being a one-to-one head that produces one single prediction per location. The addition of a One-To-One head removes the need for Non-Maximum-Supression, which greatly reduces the computational capabilities needed and is the main novel addition to YOLOv10.

While the image lifecycle in YOLOv8 and YOLOv10 is very similar, there are some differences in the complexity of each model and the usefulness each may have in real-world real-time scenarios. YOLOv8 has a more established and supported community, since it was used widely for much longer than v10 since v10 was just released 5 months prior to the writing of this report, with v11 released 1 month prior. v8 had a lifespan of 1 year before v10 was released, while v10 has only been released for 4 months and v11 is officially released. YOLOv9 was mentioned because the literature on YOLOv8 is more prominent since it is an older version and its community support is more stable and flushed out, making it a much easier model to begin training on. According to some sources [17] YOLOv8 also performs better in real-time scenarios, and oftentimes has a higher true positive count with the risk of detecting non-existing objects more often than YOLOv9. This is a trade off we are willing to make here, since it is more important for us to detect errors more often than they happen than to miss some and risk wasting more time, money, and filament on failed prints that continue unchecked.

In order to confirm our claim that YOLO is the best model to use for our project, we have also decided to train and test on two binary classification models, ResNet-50 for a more computationally intensive example, and EfficientNet, for a more lightweight example that uses closer to the same computational power as YOLO.

Implementation Summary

Our dataset was procured through two FDM printers, both with a camera placed in the left corner of the printer enclosure closest to the access door. The two printers in question are the Bambu Labs P1S, which the majority of our images come from, and the Bambu Labs X1-Carbon, which is used by my University's, Tennessee Technological University, education department. It is important to mention that these printers are very reliable and advanced compared to most other consumer printers that hobbyists might use, such as the Ender 3 V2 or some of the Prusa printers. So, with that in mind, it takes much longer to get failure prints with this printer than with some others, with the average failure rate of these printers being around 8-10%, whereas with the Ender 3 line, it is closer to 20%. That being said, these printers still have many of the same problems, just less often. The most common error in FDM printing is stringing, which we explained earlier. Stringing commonly occurs no matter what print error originally occurs. So if the print has a bed adhesion problem and comes unglued from the bed, stringing will most likely occur since the print moved and there may be nothing to extrude onto. Stringing occurs even more commonly in *obstacle* fails, where the extruder box bumps or runs into something unexpected while printing that prevents it from moving to the position it was designated to be at. With this in mind, we will be testing for stringing exclusively, since it is a byproduct of most other failures and is the easiest to accurately identify. While this may result in later detection of failures for some prints, since stringing would need to occur before we detect most errors, many of the other failures are not visually prevalent enough in the photos to be able to train effectively on them, so our focus is on stringing.

Dataset Description

The cameras within both printers capture images in the same way, each taking one image after every layer of the print has been laid. If you have a print that is 10 millimeters tall with a layer height of 0.2 millimeters, then 5 images will be generated per mm, and 50 total in the timelapse. For every print in our datasets, we use a layer height of 0.2 mm, which is the standard for FDM printing, although the layer

Figure 2: Example Image taken from Bambu Labs P1S

height will not change the results of the detection, it is still a good note to make. Once each print finishes, the timelapse is saved to an SD card, which is then copied the computer and a python script (which can be found in my github for this project) [8] is used to split the timelapse videos into individual images. Each timelapses individual images are placed in its own folder, numbered 1-n, where we then must manually check each print to see where the first error occurred. Once the first failure is identified, we split the images into GOOD and BAD folders, where every image from the first failure onwards is placed in the BAD folder, and all images before that (successful layers) is placed in the GOOD folder.

Preprocessing

Our images are preprocessed differently depending on the model we use to train. Our YOLOv8 and v10 models have a few pre-processing steps that must be done manually and take a lot of time, whereas the ResNet-50 model tested is more of a plug-and-play model where we point it in the direction of the dataset and adjust hyper-parameters as needed.

For the YOLO models, which we focus on mostly in this experiment, we must first make sure that our images are all placed in the same directories together, rather than having them split by timelapse. We then have two folders, one for training and one for validation. We decided to go with an 80-20 training and validation split, where testing is done later on rather than during the training process. Originally, we had placed these images into their respective folders with no thought as to where they were coming from, and we received great results from our training, but these results were skewed to say the least. We had been randomly splitting our data into train and validation sets, regardless of what print timelapse they came from, so an image from layer 50 of a print could have been used for training whereas the image from layer 51 of the same print may have been used for validation. Since these images are only one 0.2mm layer apart, there is a very small difference in their visual makeup, so our model was correctly predicting the validation image almost always, since the training image had a minuscule difference from the validation, it had already seen 99% of the image before, with that extra 1% being the extra one layer. In order to fix this, we make sure that, if one image from one timelapse is used in training, the entirety of that timelapse is used in training, and if one image was used in validation, the entirety of that timelapse was used in validation.

Once we had split the images correctly into their respective folders, we then had to label the images one by one. Labeling the images consisted of using a python program called LabelImg, which gives a graphical interface in order to create boxes around the specific feature in an image you are looking for, in our case, failed parts of a print (stringing). These boxes are called *Bounding Boxes* and are used in YOLO to more accurately localize features to be train on. While the model still sees the whole image, it is focusing mostly on the bounding boxes to train on the failures. Since we only create bounding boxes on the failure images, and since the only class we are testing for is strining, there is little need for the GOOD images in our YOLO training (successful prints), because YOLO needs bounding boxes to calculate the localization and classification losses. As mentioned before, it treats bounding boxes as positive and everything else as negative [1]. An image with no bounding box would be shown as completely negative. This may be helpful in some cases, because it would help the model to lower the false positive score, hence increasing precision. However, since we have a somewhat large dataset already, and every part of an image that is not a bounding box is already treated as a negative, it seems unnecessary to train on every image and increase our training time by a large margin for a minimal gain in precision and no gain in recall. Allowing bounding boxes is the key reason YOLO is able to train on localized features more accurately and not generalize on the whole image. This will also help when using this model with other printer types, since the other printer types environments may look different from this one, (ex. More lighting, exposed rather than covered, different surroundings, different camera angle,etc.) but the failures themselves will not.

Using LabelImg, we iterate through every image manually and create a bounding box around most of the failures we see. Originally, bounding boxes were placed on every single failure, no matter how faint, blurry, or visually obvious it seemed in the image, but this caused some problems. With a trained eye on what to look for, it is much easier to see a small blur that can barely be noticed by those without, much less a camera. It is easier to tell whether that happens to be a failure or not, but the model does not know many of the signs, so when it was trained on EVERY failure, no matter how small the detail, it performed more poorly. Once the problem was identified, we went back into the images and re-label them, being more conservative with what was labeled as failures and what was not. Figure 3 shows an example of one of these such images.

In this prints case, there are **many** different models being

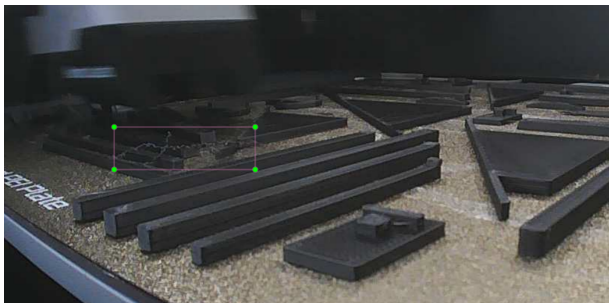

Figure 3: Hardly apparent failure example

printed on the print bed at one time. The bounding box shown contains an anomaly, while this anomaly did not cause the print to ”fail” per se, it might be something we would want to look out for since some filament came loose and caused a bit of stringing. However, this is barely noticeable to a humans eyes, much less to a camera trying to differentiate between each piece of the strand and the background. So failures that were not apparent enough to be confidently recognized were removed from the dataset before training began.

Another problem had with the bounding boxes, is that there are many failures in some images that would span a large portion of the print, but be very minimal in size. An example of a failure of this kind can be seen in Figure 4. We can see that multiple bounding boxes have been drawn, but the large bounding box just below the extruder holds very little information that can be useful, and lots of noise is present. While this is where the stringing originally occurs, it is conglomerating on the print bed, which we have another bounding box around since that is the more obvious failure area. When an anomaly occurs and we are not able to place a bounding box around it in a way that minimizes its environment/background and maximizes the actual failure, we also removed these because they were too hard to detect. While it is an option to create lots of smaller boxes around it to achieve this, it is hard to keep that consistent across a dataset with thousand(s) of images, so our goal in making the bounding boxes was to cover entire failure areas rather than having small bounding boxes around different parts of the same failure.

Hyperparameter Optimization

Hyperparameters were changed manually based on a training plan that changed dynamically depending on the results received from previous runs. The main hyperparameters and customizations that were often changed were: Learning Rate, Batch size, Epochs, Image size, Confidence, Augmentation, and Intersection Over Union. The goal of our tuning was to receive a high recall score, with an acceptable precision score, since we have decided that Recall is our most important metric, since we want to identify as many failures as possible, even if it may lead to slightly more false positives.

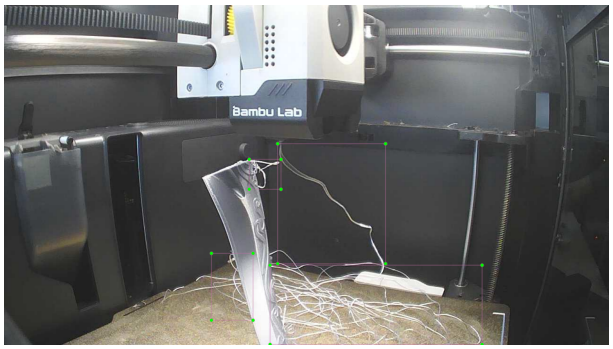

Figure 4: Bounding box with very little information

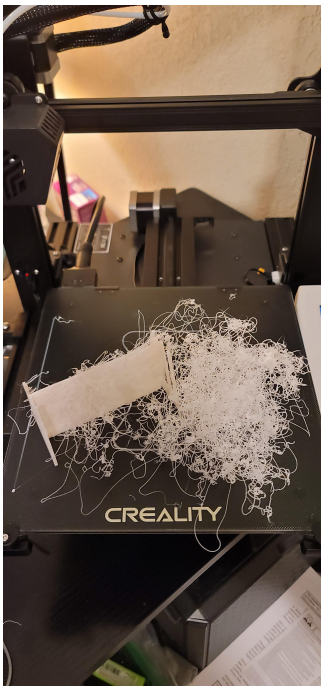

Figure 5: Secondary testing dataset failure example

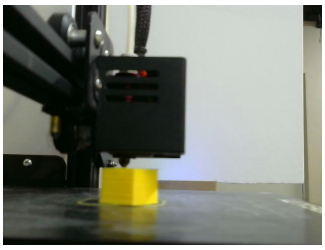

Figure 6: Secondary testing dataset success example

Training and Validation

The dataset was split into training (80%) and validation (20%) sets. A testing dataset was not employed on every training set, but rather done after the fact by an inference script with the models with the best results in order to be more efficient with time and computational power. Testing has been done on two sets of data, one consisting of images from the same printers. The second dataset consisted of many image types conglomerated into a Kaggle dataset [24]. Some of these images come from other individual's personal printers, also with a camera mounted on the side (but in a different angle and position, while some seem to have been taken directly off of google, with many different orientations and image qualities. This dataset consisted of 1440 total images to test on (over 30 different prints), 640 of which were failure images, and 800 of which were successful prints. Some examples of these images are shown in Figures 5 and 6. Choosing this dataset as one of our test sets demonstrates the adaptability and reliability of our results, since the images are taken from different angles on different printers, with different colors, lighting, aspect ratios, qualities, and backgrounds. We aim to make our model accurate and trustworthy across many environments and settings, not only the environment the training data has been taken from.

Assessment Methodology and Results

The metrics used to evaluate our model are just as,if not more important than the model we used and the training methods employed. We stated that we would like to detect when an error has occurred, so that the user can minimize the wasted time, filament, and money associated with that error. We have also stated that the YOLO model can show the user exactly where the error occurred and detect multiple error for each image. We do not necessarily need to know exactly where an error has occurred, or even how many have occurred. What difference might it make to the user if the model says it classified 3 errors versus 1? The 3 errors could be much smaller than the 1, or they could be much larger. Simply giving an alert that an error may have occurred is sufficient.
Then, the question comes up of why we would use a YOLO architecture rather than a more traditional binary classifier such as a CNN, or one of its derivations like ResNet, EfficientNet, or some other deep neural network. To justify the use of YOLO versus one of these other models, we have tested the performance of YOLO against ResNet-50 and EfficientNet. ResNet-50 is a deeper and more computationally expensive binary classifier, while EfficientNet comes closer to the performance of our YOLO models in terms of real-time inference. The hypothesis adopted going into this evaluation was that YOLO would outperform both these classifiers due to its ability to localize and focus in on specific features more accurately because of the weight given to bounding boxes. When training, YOLO looks more intently and gives more weight to the features within the bounding boxes than those outside of it, allowing it to see the entire image, but focus more heavily on those specific features.
The performance of our model is evaluated using precision, recall, and the F1-score, as defined earlier. The mAP(50) and mAP(50-95) values are shown in our results tables as well, also defined previously.

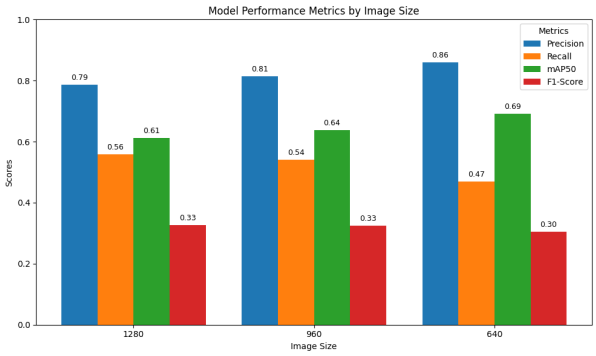

Figure 7: Metric based on Image size change

Results

In this section, we will include results for YOLOv8 and YOLOv10. We will then compare those scores to ResNet-50 and EfficientNet scores to determine which performs better and whether or not we have succeeded in creating a model capable of detecting the majority of 3D print failures in our datasets in order to save a large amount of time, filament, and money in the long run. A table of all the data shown in the following figures will be given at the end of the paper for closer looks at exact numbers.

YOLOv8 and v10 The first few hyperparameters tested were tested because of their relatively independent nature.

One parameter that should have little to no dependence on others is the Image Size. While lower image sizes make training much faster and would also increase inference speeds, lowering the image size too much would result in a trade off in precision and recall that was not necessary or desired.

Figure 7 shows three YOLOv8-N runs with the exact same batch size (16), confidence score (0.3), learning rate (0.001), and epoch count (150), with our image size being the only variation. The change from 1280x720 to 960x540 resulted in a Precision gain of 2.9%, a Recall loss of 1.8%, an mAP50 loss of 2.5%, and a F1-Score loss of only 0.18%. The change from 960x540 to 640x360 resulted in a Precision gain of 4.5%, a Recall loss of 7%, an mAP gain of 6.5%, and an F-1 score loss of 1.9% . These numbers were consistent across all models and, we decided to consistently use the image size of 960x540 for the majority of the rest of the training since, while 1280x720 has the highest recall score, the added computational power needed for this size is not enough to persuade us to use it rather than 960, which has the same F1-Score. On the other hand, 640x360 exhibits a higher precision score, but a 7% lower recall score. The trade

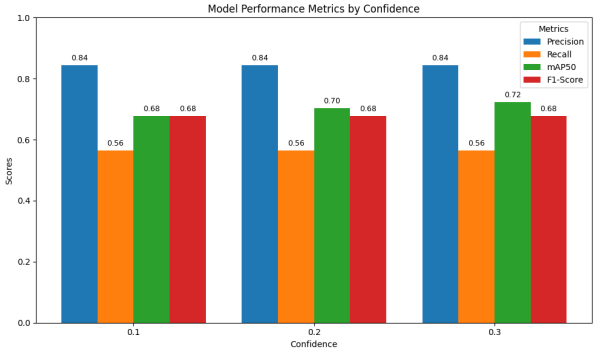

Figure 8: YOLOv8-N Metric based on Confidence change

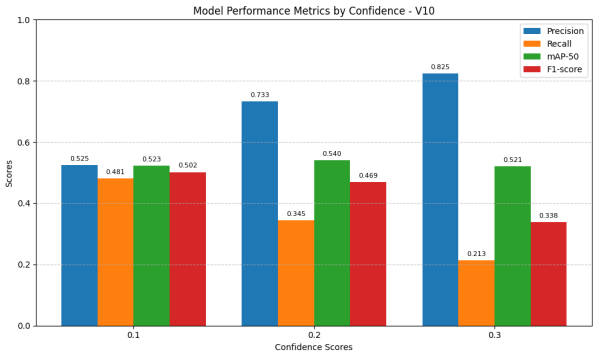

Figure 9: YOLOv10-N Metric based on Confidence Change

off between these, even at the lower computational complexity, is not enough to persuade use of 640x360.

Next, we tested the change in the Confidence score, which should (and did) stay consistent across many models, even if the more complex models may have slightly higher confidence scores for the same bounding boxes. These tests are shown in Figure 8, where we used YOLOv8-Nano to test confidence scores between 0.1 and 0.3. These tests were done with a batch size of 16, epoch count of 10, and image size of 960x540. Figure 9 shows the same tests done on YOLOv10-N.

Figure 9 shows that, a change in confidence score at these thresholds did not change our precision, recall, or F1-score at all with the v8 model. This indicates that every single bounding box is being predicted with a confidence score of 0.3 or higher, meaning that lowering the confidence score will not produce better results in the v8 model. The mAP score differs slightly between each confidence threshold, but the mAP score differs slightly between two runs of the same hyperparameters, so a changing mAP value is expected across any run. However, when we run the v10 model, we see different results, with v10 losing 30% precision as the confidence lowers from 0.3 - 0.1, but gaining 26.8% recall. Because of this, we cannot be confident that 0.3 will always

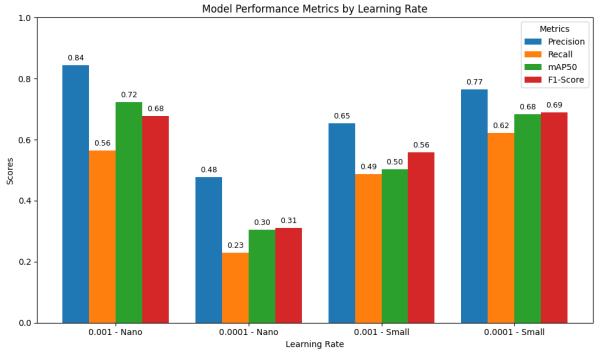

Figure 10: YOLOv10-S Metric based on Image size change

give better results than 0.1, so both confidence scores are tested throughout the rest of our testing as well, although 0.1 seems to give better overall scores with v10 and 0.3 seems best fit for v8.

Next, it gets more tricky deciding what hyperparameters are important to change, as stated before, the batch size, epoch count, and learning rate are all dependent on one another, so a change in one could give drastically different results. Many different values for learning rates were tested, and it was found that lower learning rates were more beneficial for higher epoch counts, with the Small and Medium models getting more benefit from lowering learning rates than Nano. After this point it becomes a bit of trial and error to see what parameter configurations give the best results. Table 3 show some of our best results from the rest of our training.

We can see that a very low learning rate on V8-S and V10-M paired with a large epoch count of 150 and a small batch size of 8 gives us our best results. When we go above an epoch count of 150, to 200, on either Small or Nano, we receive a higher precision score but usually lower Recall scores, so after a few tests, 150 is the highest we decided to go for epoch count, even with lower learning rates for the v8 model. V10-M correctly predicts 87.5% of the bounding boxes it labeled as failures, and predicts 64.8% of all failures. V8-S correctly predicts 88% of the bounding boxes it labeled as failures, and predicts 67% of all failures from the dataset.

We can see in Figure 11 the results of our best model, YOLOv10-M with 150 epochs, 960 image size, 8 batch count, 0.1 confidence, and a learning rate of 0.00001-0.0000001.

Since our Medium model may struggle to run on some raspberry pi's, our best Small model results are also included in Figure 12. These results come from our v8-S model with only a 3% decrease in F1-Score when compared to v10-M. The hyper-parameters used for this were the exact same as those for v10-M.

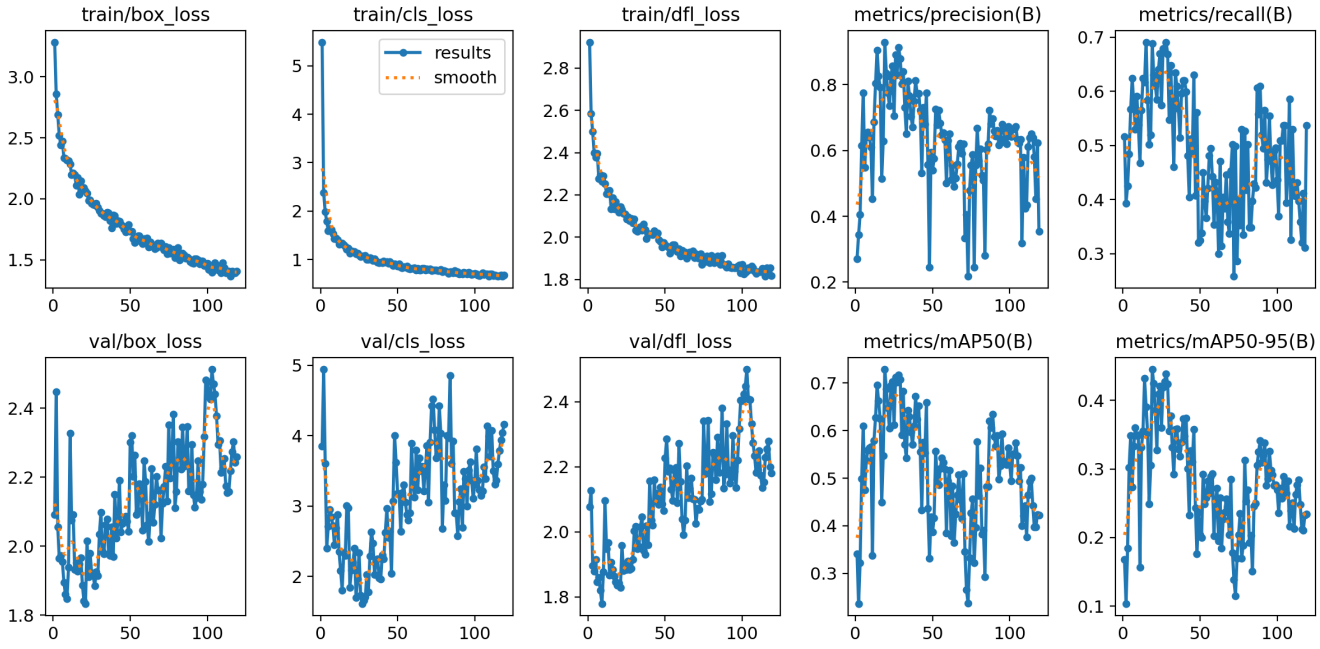

Figure 11: YOLOv10-M Best Run Metrics

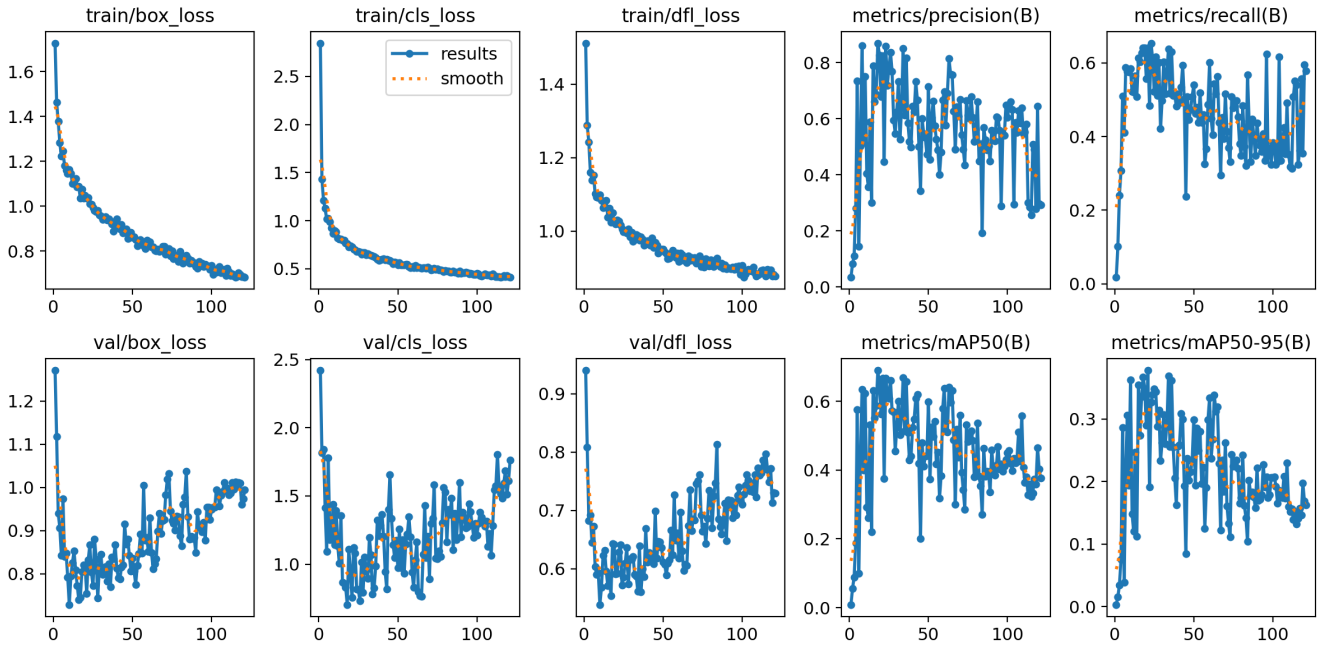

Figure 12: YOLOv8-S Best Run

Table 3: YOLOv8 and v10 Hyperparameter Optimization and Bounding Box prediction scores

Model	Epochs	ImgSize	Batch	Confidence	Learning Rate	Precision	Recall	mAP50	F1 Score
YOLOv8-S	100	960x540	8	0.3	0.00005-0.000001	0.824	0.613	0.734	0.703
YOLOv8-N	**100**	**960x540**	**8**	**0.1**	**0.001-0.00001**	**0.808**	**0.662**	**0.711**	**0.733**
YOLOv8-N	200	960x540	8	0.3	0.001-0.00001	0.925	0.429	0.681	0.586
YOLOv8-S	200	960x540	8	0.3	0.001-0.00001	0.908	0.513	0.711	0.655
YOLOv8-M	100	960x540	8	0.3	0.00005-0.000001	0.874	0.579	0.727	0.697
YOLOv8-S	**150**	**960x540**	**8**	**0.3**	**0.00005-0.000001**	**0.88**	**0.67**	**0.779**	**0.761**
YOLOv10-N	150	960x540	16	0.1	0.001-0.00001	0.882	0.492	0.682	0.632
YOLOv10-N	200	960x540	8	0.1	0.00001	0.83	0.57	0.69	0.674
YOLOv10-S	250	960x540	8	0.1	0.0001-0.000001	0.74	0.526	0.669	0.615
YOLOv10-S	150	960x540	8	0.1	0.00001-0.0000001	0.624	0.512	0.573	0.563
YOLOv10-M	150	960x540	8	0.1	0.00005-0.00000001	0.875	0.648	0.773	0.745
YOLOv10-M	**250**	**960x540**	8	**0.1**	**0.00001-0.00000001**	**0.902**	**0.677**	**0.508**	**0.774**
YOLOv10-M	**150**	**960x540**	**8**	**0.1**	**0.00001-0.0000001**	**0.919**	**0.693**	**0.803**	**0.79**

Box loss measures the error in predicted bounding box coordinates compared to ground truth, which shows how well the model is learning to localize each failure. CLS loss measures the classification loss, which is the ability to classify the objects correctly. Lastly, dfl loss stands for distribution focal loss, and is used to refine bounding box regression and focus on learning more precise box coordinates, this helps the model to understand small deviations in bounding boxes better and adjust as needed. The consistent downward trends in box loss, cls loss, and dfl loss shows that each model is converging and learning effectively, with both models loss graphs being nearly identical, however, v8 seems to converge slightly faster on box and dfl loss than v10.

The validation box loss, cls loss, and dfl loss shows a slight increase in all, with the numbers in v10 being quite a bit higher than those of v8, and the increase slightly more dramatic. Increases in validation losses usually indicate over-fitting, which was worrying at first. However, while this may be slightly overfitting and could benefit from implementing early stopping or regularization techniques, the testing scores we receive later from both our own new testing dataset as well as a foreign testing dataset show that there is little concern in this regard.

Our v8-Small version seems to show slightly less variation in precision, recall, and both mAP metrics, but this is not the case. The graphs on v8 start at 0, whereas v10 graphs start higher than 0, making the v8 variability slightly less obvious since it is flattened slightly. The somewhat high variability in both precision and recall is likely the result of the diverse dataset used, with varying shapes, sizes, and types of 3D prints and print failures, which could make it challenging for the model to generalize over all scenarios. We can see that the graphs are very similar, with many of the "outliers" being displayed in the same areas between all graphs for both v10 and v8. This shows that there are some batches that likely included images with higher variability than others. With the low "outliers" likely being batches that included images where the failures were less obvious, and the higher outliers with failures that were more prominent.

ResNet-50 and EfficientNet The most notable results of ResNet and EfficientNet training are provided in this section and Table 4. As we can see, ResNet often performs very well in precision, but with a very low recall score. Increasing the epoch count and decreasing the learning rate, much like most AI models, gives better overall scores than those with low Epoch counts and high learning rates. Increasing the epoch count from 10 to 20 with the same learning rate gives an increase in precision by about 5% and an increased recall by an impressive 16%. Resnet, surprisingly performed its best with an epoch count of only 25, batch size of 8, learning rate of 0.0001, with an overall F1-score of 64.7%, recall of 50.7%, and precision of 89.85%. This outperforms our anticipations, with a higher precision score by almost 2% than our best YOLO model, although still with a recall 16.3% lower than our best YOLO architecture.

Testing results Testing has been done on two different datasets, the first of which is from the printers that the training data came from, and is the same format as the images that the models were trained on, although unseen. This consists of 1413 images total, across over 40 timelapses, 694 of which were images with failures in them, and 719 without. The other dataset consists of 1440 images from a Kaggle dataset as mentioned earlier in the Train and Validation section, with 640 failures and 800 successful images.

Testing of these models in the foreign dataset in Figures 13 and 14 (as well as tables 5 and 6 if you'd like a closer look at the numbers) show that they are performing in much the same way, with both giving high recall, but low precision scores. This shows that they are detecting failures far more than they should. The fact that their performance is so different from the performance they had in training shows that these may not be the best choice for adaptability to different datasets and environments. The high recall is hopeful, since you will catch most failures, but low precision would give the user high false positive rates and may not be worth the trouble of implementing in the real world at all. Testing of

Table 4: ResNet-50 & EfficientNetb0

Model	Epochs	ImgSize	Batch	Learning Rate	Precision	Recall	F1-Score
ResNet-50	10	960x540	16	0.0001	0.875	0.109	0.194
ResNet-50	20	960x540	16	0.0001	0.9231	0.2609	0.407
ResNet-50	25	960x540	8	0.0001	0.8985	0.5072	0.647
ResNet-50	50	960x540	16	0.00001	0.862	0.176	0.29
ResNet-50	100	960x540	8	0.00001	0.875	0.375	0.525
ResNet-50	150	960x540	8	0.000001	0.832	0.402	0.544
EfficientNet	25	960x540	16	0.001	0.765	0.315	0.44
EfficientNet	50	960x540	16	0.0001	0.819	0.291	0.43
EfficientNet	100	960x540	16	0.00001	0.381	0.372	0.383
EfficientNet	100	960x540	8	0.00001	0.6552	0.3878	0.487

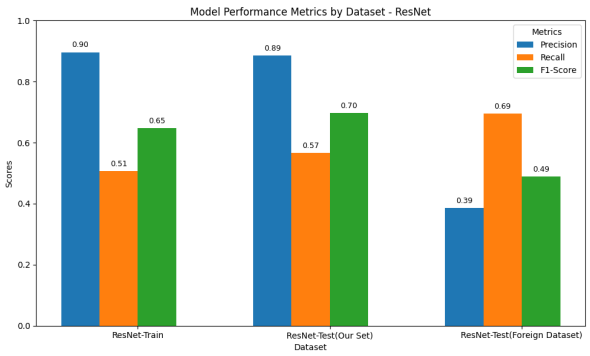

Figure 13: Best ResNet model results from all three datasets

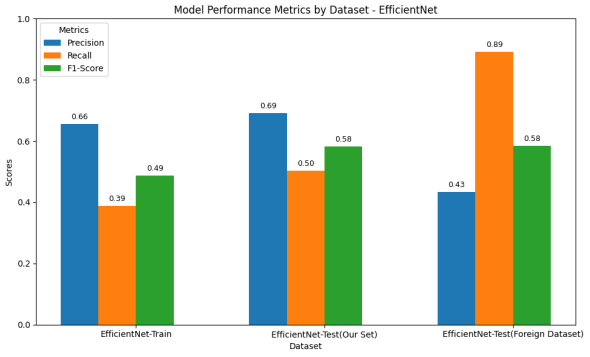

Figure 14: Best EfficientNet model results from all three datasets

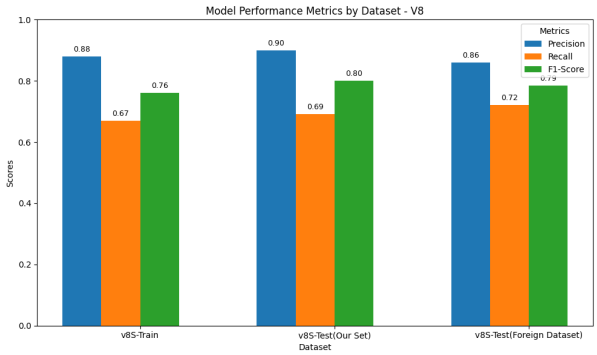

Figure 15: Best YOLOv8-S model results from all three datasets

these models in our own dataset, procured from the same printers, gives more hopeful results. Resnet achieves a precision of 88.5% with a Recall of 56.7%, showing a slight increase in the recall score of around 6% in our test dataset and a precision decrease of 1.5%. EfficientNet shows much the same, with an increase in recall of 11% and an increase in precision of 3%. Overall, ResNet performs well in our dataset, but shows little promise for being adapted to a different environment, and the same for EfficientNet, with both of their Precision scores taking a large hit when transferred to another dataset.

Figures 15 and 16 (as well as tables 7 and 8 for a closer look at the raw data) show our two best YOLO models performance compared across datasets, with the first being training results, the second being our test dataset results, and the third being the foreign dataset test results. Our two best YOLO models performed very differently when testing on the foreign dataset. While our *best* model in training results, Yolov10-M, exhibits a 96.5% precision score in testing on the foreign dataset, it falls behind the training score by far, with only 46% recall, and an overall F1-Score of 62%. Its performance on our own test dataset shows that it performs

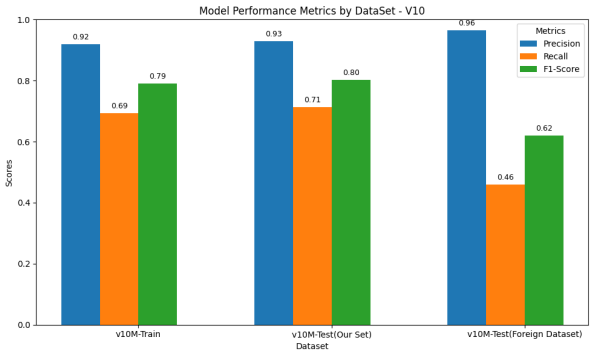

Figure 16: Best YOLOv10-M model results from all three datasets

nearly the same as training, with a precision increase of 1% and a recall increase of 2%. This shows that this model might not be best fit for adaptability to different printers and data, but performs very well under the constraints of our environment. On the other hand, our V8-Small model performed very well, with an 86% precision score and 74% recall score in the foreign dataset, and an increase in both precision and recall of 2%. The increase in accuracy on our test dataset is likely due to our test dataset having more prominent failure areas. This is also the explanation for the foreign test dataset having slightly higher recall in our v8 model, one such image from the foreign dataset is shown in Figure 17, whereas fewer of our training set images had failures this large. This indicates that our current best v8-S model is more versatile and adaptable than our current v10 models, and as such, would be our model of choice between these two for this reason and that it is guaranteed to perform well on a Raspberry pi, whereas the medium is much more computationally expensive and could perform overwhelmingly on the Pi. This does not mean that the v8 model itself is more versatile, only that our current training configurations deem the v8 model more versatile than our current training configurations on our v10, it is likely that with more hyperparameter tuning the v10 would outperform the v8 in this way as well.

Limitations and Future Work

There are a few ways this work can be improved upon. Firstly, most of our hyperparameter optimization was done with YOLOv8, since YOLOv10 had not been released when this project first started and v8 had more literature behind it. v10 already outperformed v8 in the medium model size, with more tuning, YOLOv10-Medium or even Small could be a better fit for our application, since it has a lower number of parameters per model and gives faster inference times compared to YOLOv8, so it would run faster on the smaller machines we would like this to be available on. YOLOv11 has recently been released and also shows promise, as it has even less parameters and runs faster than YOLOv10 on inference. Secondly, While our dataset was plenty large enough to give us good scores, a more robust

Figure 17: Failure Example in Foreign Dataset

dataset with data from many different printers could be even more beneficial in giving higher scores since it would be trained on multiple environments. While it is still believed that YOLO would outperform ResNet and EfficientNetb0 with a larger dataset, these two models would benefit even more from a more robust training dataset since they are currently under-performing in foreign environments and this might alleviate some of that under-performance.

To deploy this model to the real world is a future work in itself. The model can be found and downloaded from the GitHub page [8], where it can then be used alongside a python script and a raspberry pi or some other microcomputer for automated 3D print error detection.

Conclusions

As we can see from our results, YOLO outperforms the binary classifiers by a significant margin, even when localizing rather than using binary classification. The best ResNet model exhibits an average precision score across our three datasets of 72.6%, with a recall average of 59%. EfficientNet shows around the same metrics, with an average precision score across all three datasets of 59.3% in our dataset, with its recall score also being 59.3%. Both ResNet and EfficientNet, however, show little promise being adpated to other datasets with different environments with their high recall but very low precision scores in the foreign dataset alone. YOLOv8S surprisingly has given us the best metrics across both our dataset and the foreign dataset implored in testing, with a testing precision score of 86%, and a recall score of 74% in the foreign dataset, giving us a model that is robust, adaptable, accurate, and needs low computational input in order to perform. While the model may benefit from further hyperparameter optimization and training on multiple datasets, this sets a good foundation for what is to come

in this field.

The results of this study prove that a live deployment of our 3D print error detection model with YOLO is possible, accurate, and trustworthy. The approach we take demonstrates improvements in accuracy and reliability of this model through hyperparameter tuning, as well as gives a compelling argument for using YOLOv8 and v10 as real-time analyzers by showing the differences between the performance of YOLO and the performance of ResNet and EfficientNet in the application of 3D print error detection. We believe this is because of the enhanced ability for YOLO to localize small failures more accurately than other models because of its architecture and the use of bounding boxes and the weight given to them in training. In addition, while YOLO outperformed these other models, further hyperparameter tuning may narrow the gap, or widen it, but based on this research, it is unlikely for a binary classifier to surpass YOLO in its abilities in this application. This makes YOLO suitable for real-world applications that require precise monitoring of 3D prints on microcomputers such as a Raspberry Pi. In future work, hyperparameter tuning could be expanded across newer YOLO models to further advance the evaluation metric scores and also give way to computers with less computational power as YOLO continues to provide fewer and fewer parameters without sacrificing accuracy and effifiency. Ultimately, the work done here is meant to set a foundation for 3D print error detection using low-power cost effective devices in order to save time, money, filament, and reduce waste by catching printer failures as early as possible.

Table 5: Binary Classification Testing (Foreign Dataset)

Model	Epochs	ImgSize	Batch	Learning Rate	Precision	Recall	F1-Score
ResNet-50	25	960x540	8	0.0001	0.386	0.695	0.49
EfficientNetb0	100	960x540	8	0.00001	.434	0.892	0.584

Table 6: YOLO Classification Testing (Our Dataset)

Model	Epochs	ImgSize	Batch	Learning Rate	Precision	Recall	F1-Score
ResNet-50	25	960x540	8	0.0001	0.885	0.567	0.698
EfficientNetb0	100	960x540	8	0.00001	0.692	0.503	0.583

Table 7: YOLO Testing (Foreign Dataset)

Model	Epochs	ImgSize	Batch	Learning Rate	Precision	Recall	F1-Score
YOLOv10-M	25	960x540	8	0.00001-0.0000001	0.93	0.71	0.8
YOLOv8-S	150	960x540	8	0.00001-0.0000001	0.86	0.72	0.79

Table 8: YOLO Testing (Our Dataset)

Model	Epochs	ImgSize	Batch	Learning Rate	Precision	Recall	F1-Score
YOLOv10-M	25	960x540	8	0.00001-0.0000001	0.96	0.46	0.62
YOLOv8-S	150	960x540	8	0.00001-0.0000001	0.90	0.69	0.80

Table 9: YOLOv8 and v10 Hyperparameter Optimization and Bounding Box prediction scores

Model	Epochs	ImgSize	Batch	Confidence	Learning Rate	Precision	Recall	mAP50	F1 Score
YOLOv10-N	150	1280x720	16	0.3	0.001	0.786	0.558	0.612	0.326
YOLOv10-N	150	960x540	16	0.3	0.001	0.815	0.54	0.637	0.325
YOLOv10-N	150	640x360	16	0.3	0.001	0.859	0.47	0.692	0.304
YOLOv8-N	10	960x540	16	0.1	0.001	0.844	0.565	0.677	0.677
YOLOv8-N	10	960x540	16	0.2	0.001	0.844	0.565	0.703	0.677
YOLOv8-N	10	960x540	16	0.3	0.001	0.844	0.565	0.724	0.677
YOLOv8-S	10	960x540	16	0.1	0.001	0.653	0.487	0.684	0.559
YOLOv8-S	10	960x540	16	0.2	0.001	0.653	0.487	0.481	0.559
YOLOv8-S	10	960x540	16	0.3	0.001	0.653	0.487	0.503	0.559
YOLOv8-N	10	960x540	16	0.1	0.001-0.00001	0.813	0.608	0.69	0.696
YOLOv8-N	10	960x540	16	0.2	0.001-0.00001	0.85	0.593	0.709	0.698
YOLOv8-N	10	960x540	16	0.3	0.001-0.00001	0.87	0.57	0.73	0.689
YOLOv8-N	10	960x540	16	0.3	0.0001	0.478	0.229	0.304	0.311
YOLOv8-S	10	960x540	16	0.3	0.0001	0.765	0.622	0.684	0.689
YOLOv8-S	100	960x540	8	0.3	0.00005-0.000001	0.824	0.613	0.734	0.703
YOLOv8-N	100	960x540	8	0.1	0.001-0.00001	0.808	0.662	0.711	0.733
YOLOv8-N	200	960x540	8	0.3	0.001-0.00001	0.925	0.429	0.681	0.586
YOLOv8-S	200	960x540	8	0.3	0.001-0.00001	0.908	0.513	0.711	0.655
YOLOv8-M	100	960x540	8	0.3	0.00005-0.000001	0.874	0.579	0.727	0.697
YOLOv8-S	150	960x540	8	0.3	0.00005-0.000001	0.88	0.67	0.779	0.761
YOLOv10-M	150	960x540	8	0.3	0.00005-0.00000001	0.875	0.648	0.773	0.745
YOLOv10-M	250	960x540	8	0.1	0.00001-0.00000001	0.902	0.677	0.508	0.774
YOLOv10-M	150	960x540	8	0.1	0.00001-0.0000001	0.919	0.693	0.803	0.79

References

[1] Christopher 5106. *Bounding Box Object Detectors: Understanding YOLO*. Accessed: 2024-11-29. 2017. URL: https://christopher5106.github.io/object/detectors/2017/08/10/bounding-box-object-detectors-understanding-yolo.html.

[2] Arana Corp. *Object Detection with YOLO on Raspberry Pi*. Accessed: 2024-11-29. 2024. URL: https://www.aranacorp.com/en/object-detection-with-yolo-on-raspberry-pi/.

[3] Core Electronics. *Getting Started with YOLO Object and Animal Recognition on the Raspberry Pi*. Accessed: 2024-11-29. 2024. URL: https://core-electronics.com.au/guides/getting-started-with-yolo-object-and-animal-recognition-on-the-raspberry-pi/.

[4] Michael Dwamena. *3D Print Failures – Why Do They Fail & How Often?* Accessed: 2024-12-03. 2022. URL: https://3dprinterly.com/3d-print-failures-why-do-they-fail-how-often/.

[5] GeeksforGeeks. *Kernels (filters) in convolutional neural network*. June 2024. URL: https://www.geeksforgeeks.org/kernels-filters-in-convolutional-neural-network/.

[6] Muhammad Hussain. "YOLOv5, YOLOv8 and YOLOv10: The Go-To Detectors for Real-Time Vision". In: *Department of Computer Science, Huddersfield University* (July 2024). *Correspondence: M.Hussain@hud.ac.uk.

[7] Tian Jin et al. "Compiling ONNX Neural Network Models Using MLIR". In: *arXiv preprint arXiv:2008.08272* (2020).

[8] Kase Johnson. *3DAiProject Github*. https://github.com/Kasej01/3DAiProject. [Accessed 26-02-2024].

[9] Nyoman Karna et al. "Towards Accurate Fused Deposition Modeling 3D Printer Fault Detection using Improved YOLOv8 with Hyperparameter Optimization". In: *IEEE Access* (2023).

[10] Stig Andreas Kronheim Langeland. *Automatic error detection in 3D pritning using Computer Vision*. Jan. 2020. URL: https://bora.uib.no/bora-xmlui/handle/1956/21450.

[11] Frederik vom Lehn. *Understanding the convolutional filter operation in CNN's*. Nov. 2023. URL: https://medium.com/advanced-deep-learning/cnn-operation-with-2-kernels-resulting-in-2-feature-mapsunderstanding-the-convolutional-filter-c4aad26cf32.

[12] Elena Herrera-Ponce de León et al. "Intelligent and smart biomaterials for sustainable 3D printing applications". In: *Current Opinion in Biomedical Engineering* (2023), p. 100450.

[13] Wei Liu et al. "SSD: Single Shot MultiBox Detector". In: *Computer Vision – ECCV 2016*. Ed. by Bastian Leibe et al. Cham: Springer International Publishing, 2016, pp. 21–37. ISBN: 978-3-319-46448-0.

[14] Moses Openja et al. "An Empirical Study of Challenges in Converting Deep Learning Models". In: *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2022, pp. 13–23. DOI: 10.1109/ICSME55016.2022.00010.

[15] Konstantinos Paraskevoudis, Panagiotis Karayannis, and Elias P. Koumoulos. "Real-time 3D printing remote defect detection (stringing) with Computer Vision and artificial intelligence". In: *Processes* 8.11 (Nov. 2020), p. 1464. DOI: 10.3390/pr8111464.

[16] Nandini L. Reddy. *Exploring YOLO11: Faster, Smarter, and More Efficient*. Accessed: 2024-11-29. 2024. URL: https://medium.com/@nandinilreddy/exploring-yolo11-faster-smarter-and-more-efficient-f4243d910d1e.

[17] Augmented Startups. *Is YOLOv9 Better Than YOLOv8?* Accessed: 2024-11-07. 2023. URL: https://www.augmentedstartups.com/blog/is-yolov9-better-than-yolov8.

[18] G2 Learn Team. *30+ Key 3D Printing Statistics Facts for 2023*. Accessed: 2024-12-03. 2023. URL: https://learn.g2.com/3d-printing-statistics.

[19] Nexa3D Team. *3D Printing Statistics: Key Numbers and Trends for 2023*. Accessed: 2024-12-03. 2023. URL: https://nexa3d.com/blog/3d-printing-statistics/.

[20] Timothy Tracy et al. "3D printing: Innovative solutions for patients and pharmaceutical industry". In: *International Journal of Pharmaceutics* 631 (2023), p. 122480.

[21] Ultralytics. *Deploying YOLO on Raspberry Pi*. Accessed: 2024-11-28. 2024. URL: https://docs.ultralytics.com/guides/raspberry-pi/#comparison-chart.

[22] Ultralytics. *Yolov10*. Sept. 2024. URL: https://docs.ultralytics.com/models/yolov10/.

[23] Ultralytics. *Yolov8*. Sept. 2024. URL: https://docs.ultralytics.com/models/yolov8/#performance-metrics.

[24] Padraig Valenti. *3D printing failure detection*. July 2024. URL: https://www.kaggle.com/datasets/padraigvalenti/3d-printing-failure-detection?resource=download.

[25] Ao Wang et al. “YOLOv10: Real-Time End-to-End Object Detection”. In: *arXiv preprint arXiv:2405.14458* (2024).

[26] Yaxin Wen et al. “Development of plant-based meat analogs using 3D printing: Status and opportunities”. In: *Trends in Food Science & Technology* 132 (2023), pp. 76–92.

[27] Shenghao Xu. *Ncnn-YOLOv3 Acceleration and Implementation*. Accessed: 2024-12-02. 2020. URL: https://runzexu.github.io/pic/DNN_project_report.pdf.