SMOOTHING OF CONVOLUTIONAL NEURAL NETWORK CLASSIFICATIONS

THESIS

Glen Ryan Drumm, 1st Lt, USAF

AFIT-ENS-MS-22-M-122

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENS-MS-22-M-122

SMOOTHING OF CONVOLUTIONAL NEURAL NETWORK CLASSIFICATIONS

THESIS

Presented to the Faculty

Department of Operational Sciences

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

Degree of Master of Science in Operations Research

Glen Ryan Drumm,

1st Lt, USAF

March 25, 2022

AFIT-ENS-MS-22-M-122

SMOOTHING OF CONVOLUTIONAL NEURAL NETWORK CLASSIFICATIONS

THESIS

Glen Ryan Drumm,
1st Lt, USAF

Committee Membership:

Lance Champagne, Ph.D
Chair

Bruce Cox, Ph.D
Member

Trevor Bihl, Ph.D
Member

AFIT-ENS-MS-22-M-122

Abstract

As technologies advance, the Air Force seeks to maintain air superiority. One way of achieving this revolves around having state of the art sensors capable of collecting critical information from live streaming video feeds. Data analysis must take place in real time, and quickly overwhelms human processing, neural networks are quite capable. Over the course of a few frames, neural networks may skip around in predictions in an image causing a stutter. This research develops and demonstrates a methodology that can be used to smooth model predictions to alleviate oscillating classifications for single action classification. While a simple problem environment shows proof of concept, obstacles remain for applying such a technique to a more operationally complex problem.

Contents

Page

List of Figures

List of Tables

SMOOTHING OF CONVOLUTIONAL NEURAL NETWORK CLASSIFICATIONS

I. Introduction

Modern warfare increasingly depends on up-to-date battlespace information to feed real-time decision making utilizing automated and human-in-the-loop machine learning techniques. As sensors and their associated capabilities flourish, the warfighter increasingly relies on automated systems to handle complex data. In this respect, artificial intelligence (AI) has a proven potential in being integrated into the battlespace [1].

Intelligence, surveillance, and reconnaissance (ISR) systems constantly record and compile optical data needed for immediate observation. Video classification is a technique used to autonomously categorize this data. Video data can be parsed into images where AI techniques such as neural networks are a popular choice for image classification. Neural network subcategories such as convolutional neural networks (CNNs) and recurrent neural network (RNNs) can be used to classify these images. This research extends Swize's thesis [2] in which she concluded that the inclusion of a Bayesian neural network was able to detail model prediction probabilities and also increase overall model accuracy [2]. This thesis alters her methodology and application of the Bayesian convolutional neural network (BCNN) – RNN blended network, to test if other methods can increase overall model accuracy and attempt to smooth classification prediction stuttering.

1.1 Problem Statement

This research was conducted to reduce, or eliminate, seemingly random, intermittent, and false classifications from neural network predictions taken from live video.

1.2 Background and Motivation

A primary objective of ISR systems is to gather and react to a substantial volume of collected data. A vast number of sensors produce massive amounts of data which requires timely analysis. Human review may not be feasible given the amount of data, therefore machine learning (ML) and AI techniques are being relied on to bridge the gap [1], [3], [4]. During video classification, network results are gathered via continuous video frame-by-frame. This process naturally considers each frame as an independent observation; however, by independently categorizing each frame, the linkage between past and current frames is severed. This broken link prevents the model from using all of the information available.

The result of this broken connection, or the incorrect treatment of linked data as independent, could contribute to stuttering video classification. A highly accurate network boasting even 95% accuracy on 100 frames would missclassify five frames on average. Such stutters result in questions related to the reliability and trust-ability in AI algorithm results [1] and resolve around users of the algorithm invariantly asking the typical questions about the automation: *What is it doing? Why is it doing that? What will it do next?* which related to the inability of the algorithm to address fundamental "ilities" [1]. On top of these interpretation issues, this can cause many issues in the application of a neural nets. Fixing this stutter by smoothing the output is highly important for applications that require the output for identification and detection in real time. Reducing the amount of misclassifications reduces the time needed by a human in human-in-the-loop techniques. Examples of these techniques

may be rectifying misclassifications or handling scenarios in which the network is unable to be used. This extends to applications that use the classification for executing other tasks such as targeting or detection. This research investigates whether or not smoothing the network output can increase accuracy of the model overall and can give the system more surety in the prediction as it can leverage the last n predictions to make its classification for the current frame.

1.3 Uncertainty in Machines

In classic stochastic processes, we are able to determine cases that are more likely and less likely to happen. Bounds of our confidence intervals are directly related to the system we are studying. Current neural networks do not let us interpret the way in which they make their predictions. In 2016, Gal raises the point of needing to be able to determine how certain a model is on its prediction [5]. When computers are becoming the decision makers on critical questions such as " is there a bystander walking in front of the car?" or becoming central to facilitating decisions in man-machine teaming operations, such as determining if a sample has cancer, we want to know just how certain they are when making decisions. This is a crucial point of investigation for this research.

II. Background and Literature Review

This chapter reviews past and current literature regarding neural networks and how they apply to image classification. The initial subsection begins with an overview of computer vision and transitions to discussing applying computer vision with general neural networks and closes by reviewing the specific networks administered for this model's application.

2.1 Computer Vision

Computer vision is a rapidly growing field of computer science that applies to a plethora of disciplines across the world. It seeks to be able to apply ML and AI to simulate a human agent for pattern recognition and item/task classification. Common tasks include processing visual data for finding differences and commonalities amongst a large dataset or live feed. Due to significant progress in graphics processing unit (GPU) computing power and recent deep learning developments, performance of computer vision techniques have began to rapidly excel [6]. Popular computer vision applications includes object detection, object tracking, segmentation, item classification and more [8]. One way computers "see" an object are by analyzing features that

Figure 1: An example of edge detection, a type of image processing method. A) Original image, b) and c) images with two different max (1 and 4) gradients for detecting edges [7]

are stripped from the image using many various image processing forms as displayed in Figure 1. This is precisely what the blended network of neural networks seeks to do in this research. Research has shown that neural networks can be successfully applied to computer vision tasks for issues such as drug development and biological research [3]. Other fields such as security have made use of computer vision to detect pedestrians or anomalies that may pose a threat [8]. Thus, this research makes use of a series of neural network as the tool to accomplish computer vision techniques.

Computer vision has three challenges. First, training for computer vision takes a large amount of data to become robust in its predictions [6], [8]. Second, these algorithms are often only applicable to data that is similar to the training data and often of limited utility to data far removed from training [1]. Third, training a system capable of robust computer vision can be difficult [6]. Large networks can have an extremely large number of parameters that effect how well a system can learn [6]. All of these challenges can play a role in how well computer vision is ultimately used.

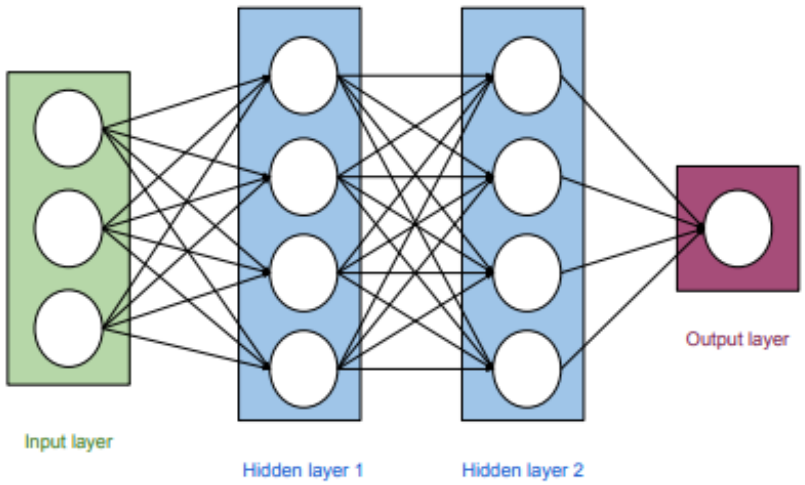

Figure 2: Basic layout of a neural network. [9]

2.2 Neural Networks

First introduced in 1943 by Mcculloch and Pitts , Artifical Neural Networks (ANNs) are a structure of machine learning that was conceived to model the way neurons function within the human brain [9], [10]. Neural networks have been proven to be applicable in a wide breadth of fields. They can take on large data sets and conduct predictions without having any prior knowledge of the task [3]. Neural networks are a broad definition for the process in which they conduct their algorithm through the nodes. Figure 2 shows the basic layout of a NN consisting of an input layer, hidden layer and output layer. The bulk of the algorithm happens in the hidden layer in which data is algorithmically digested to learn patterns that represent the data [11].

2.2.1 Convolutional Neural Networks

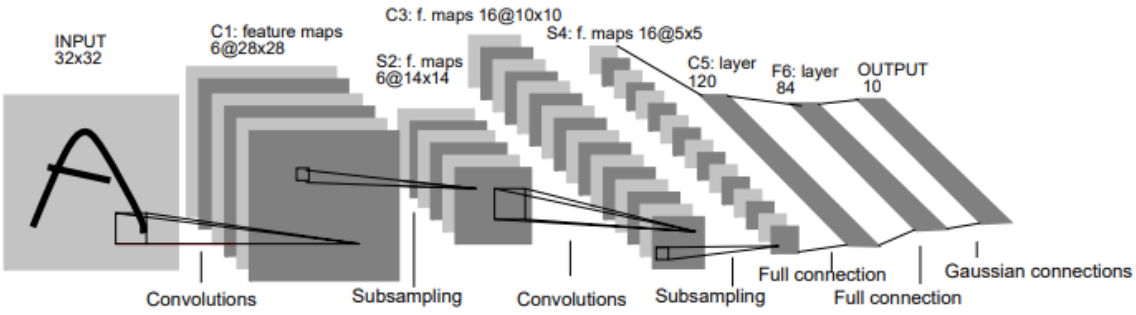

Figure 3: Composition of CNN LeNet-5. Convolutional steps for feature map processing. [4]

First introduced in 1980 by Fukushima as the "Neocognitron", Convolutional Neural Networks (CNNs) are a branch of neural networks that analyze the invariances in 2D shapes by finding patterns and further implementing constraints to the node weights [4], [12]. Examples with applications of CNNs are used in 1998 with LeNet-5 [4] to analyze handwriting and recently in autonomous driving [13], [14]. Patterns in

the larger image are searched for using kernels, which are a type of small filter used to iterate over the image in smaller chunks [15]. This is helpful for a closer examination of 2D images to recognize patterns.

Figure 3 shows how a connection of convolutional layers is depicted and how a digital image is processed throughout LeNet-5 for handwriting analysis. Starting with a 32 x 32 image, a convolution happens with six filters of size 5 x 5. This creates six feature maps of size 28 x 28. In the next layer, average pooling occurs which emphasizes the important filtering and reduces dimensionality. This continues for the next few layers until dimensionality has been reduced and 84 feature maps have been created. These final feature maps are connected to 10 classifications at the end of the algorithm, used to classify numbers 0-9.

During the convolution process, kernels extract important features, outputting them into feature maps. The feature maps are thus a collection of important qualities from the layers previous image. The network then shifts and distorts the map using a method called pooling. Pooling is mostly done through max pooling or average pooling [11], which is a way of identifying the most prominent features in the size of the filter. This allows the network to learn the common feature qualities while forgetting about the location of the feature in the full image [4]. The pattern is the important quality to learn and if the algorithm were to rely on the location of the feature in the larger image, that could be potentially harmful to the overall robustness of the network. Figure 4 shows that learned patterns can be used to predict obscured images because of the distortion conducted by the algorithm during the learning process. Pooling helps in reducing the dimensionality of the input image giving CNNs the ability to scrutinize the image in smaller chunks [13].

A frequent issue with CNNs is overfitting. Overfitting is when the network is very accurate at predicting the training data but is not robust enough to pivot into

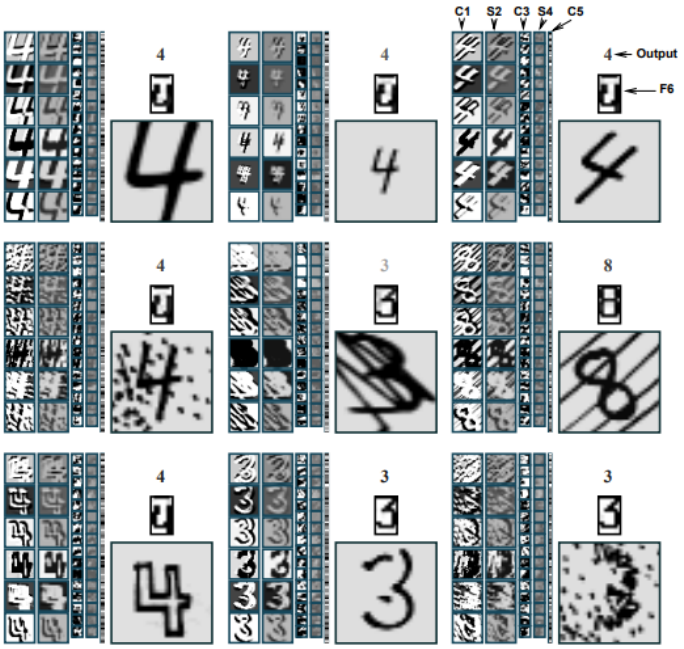

Figure 4: Examples of unusual, distorted and noisy characters correctly recognized by LeNet-5 [4]

classifying data that is significantly different from the training data. A few common origins of overfitting is not having enough diverse training data or having too many connections to nodes in the network. A proven way to alleviate this problem is through dropout [16]. Dropout is usually randomly conducted in which connections to nodes are temporarily not used. This introduces noise into the training set. Dropout is visualized in Figure 5. With this noise, the method attempts to make connections

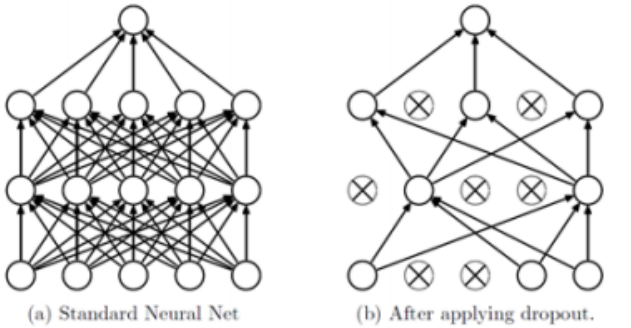

Figure 5: Dropout visualized in a basic neural network [4]

throughout the model account for more of the relationship between patterns than the algorithm and seeks breaks up learned complex co-adaptions between the layers [16]. One way dropout is further utilized is through Bayesian networks. Bayesian networks handle dropout as an approximation of the true probability of predictions. This gives the analyst a means of judging model prediction certainty. These networks are investigated in section 2.2.3 of this chapter as a possible way to handle classification stuttering.

2.2.2 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are again another branch of neural networks with a specialty in handling sequence-based systems. They differ from generic NNs by having an internal memory state that allows them to process sequential inputs. This directly explains why they are used for video sequencing and why an RNN is chosen for the back end network in this research. These algorithms are solved through moving between nodes of the network until at the end, the gradient of descent is calculated [4]. This is found via the partial derivative of the weights and the error term. By systematically tuning the weights to find the minimum value of the error term function, the accuracy of the model is maximized. This is known as Backpropagation Through Time (BPTT) and is commonplace in RNNs. Similar to CNNs, dropout is used during BPTT to remove connections between nodes in a network, reducing overfitting [14].

RNNs originally had issues with long term memory. This issue would happen when backpropagation failed and a gradient was tuned to be zero or infinity [11]. However, the method used to overcome this was pioneered by Hochreiter and Schmidhuber in 1997, called long short – term memory (LSTM) and works in conjunction with gradient based learning (Hochreiter and Schmidhuber, 1997). LSTM units consist of

a cell and multiple gates in which the cell "remembers" information passed to it and is regulated by the accompanying gates. LSTM fixed issues RNNs had with oscillating weights, time lags and cases where BPTT would not work. However, with BPTT and LSTM working in partnership they alleviate these shortfalls. Due to RNNs proven record in handling sequential data, it would be very viable to handle predictions for similar issues that this research faces. However, due to the operational speed constraints for live data, this type of network is not chosen.

2.2.3 Bayesian Neural Networks

Since other networks are unable to assess uncertainty in their predictions, they often over confidently make predictions [17]. Bayesian neural networks are implemented to address this issue. Specifically, they use the dropout of the CNN as a Bayesian approximation [5]. Computing the true probability in the prediction would be taxing on system performance. However, as shown by Gal and Ghahramani in 2016, the approximation of the true probability prevents a penalty to model complexity and computational costs [13]. BNNs are relatively new compared to other types of neural networks. Shridhar et al. showed in 2019 that they could eliminate dropout entirely from the structure by incorporating a method called "Bayes by Backprop", which applies a convolutional operation for both the mean and variance [9].

Gal found that dropout applied before the weighted layers of a CNN can be understood as approximate Bayesian inference [5] and that these uncertainty estimates can be obtained via the variance given on multiple predictions [9]. In 2019, Shridhar et al. showed an additional way of implementing Bayesian methods for uncertainties. They claim deficiencies with Gal and Ghahramani, stating that one should start with prior probability distributions and not calculate the probabilities from dropout. Shridhar et al, tested their methods with Image Super-Resolution and Generative

Adversarial Networks where they received good results [9]. However, these methods proved to be more computationally exhaustive and only slightly improved on model accuracy. While Bayesian methods could be a potential viable candidate for improving stuttering of network classifications, Bayesian methods pose computational costs. Operational viability must be maintained and thus Bayesian methods are excluded from this research.

2.3 Neural Network Usage

2.3.1 Multiple Object Detection

For this research, Unmanned Aerial Vehicle (UAV) footage will be used. Scans from the aerial sensors could include multiple objects in one video frame, so a network designed to detect more than one object in a frame is required.

Object detection happens in two phases, the first phase is locating the demanded item in the image using learned features, while the second phase is verifying the suggested item using a classifier [8]. Different forms of object detection exist such as appearance-based, motion-based and deep learning. Appearance-based approaches have trouble with recognizing images that are obstructed or obscured [8]. Motion-based approaches can fill the gap where appearance-based approaches lack, however, these approaches have their own flaws. Motion-based approaches have a higher likelihood to fail when scenarios are complex and contain objects that are moving in chaotic ways that are difficult to predict [8].

Model accuracy is measured through a metric known as Average Precision (AP). AP is calculated by finding the area under a precision-recall curve [18]. Precision and recall are core statistics in the measurement of neural network performance. Precision is a measurement of the proportion of identified positives of a dataset that were actually correct, or how many objects detected that were actually there, a value

closer to one is a better score, meaning the model has fewer false positives. Recall is the measurement of the proportion of actual positives that were correctly identified, or how good the model was at identifying all of the actual positives, a value closer to one is a better score, meaning the model has fewer false negatives. Equations for precision and recall are as follows:

$$Precision = \frac{(TruePositive)}{(TruePositive + FalsePositive)} \tag{1}$$

$$Recall = \frac{(TruePositive)}{(TruePositive + FalseNegative)} \tag{2}$$

Unique to object detection, Intersection Over Union (IoU) is used to measure overlap of the object truths bounding box and the object detectors predicted bounding box. Figure 6 displays the IoU process. The amount of overlap in these boundaries is measured and used to classify a positive classification. Standard IoU thresholds are 0.5 and 0.75 [19], [20]. For example, $AP_{50} = 35$ would indicate that the model had an AP of 35% at an IoU threshold of 50%.

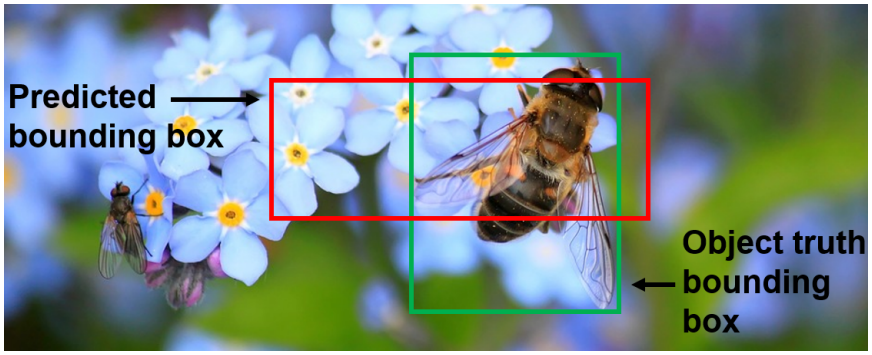

Figure 6: Overlaping bounding boxes of truth versus prediction where IoU is calculated from

Deep learning object detection has been the interest of researchers in the past few years [6]. Results from algorithmic advancements as well as technological advancements have sprung this form of object detection forward. These approaches have two

separate ways of proceeding: one-stage detectors and two stage detectors [8]. Two stage detectors attempt to approximate object regions that are predicted, based on features learned, from the architecture [8]. These regions are then used for classification as well as bounding box regression for the proposed object [8]. This form of object detection usually results in higher accuracy because of the added step of having a proposed region. One stage detectors lose the region proposal stage, thus predicting bounding boxes over the entire region. One-stage detectors are able to do predictions faster but do so at the loss of accuracy. This makes one-stage detectors increasingly favored for real-time devices [8]. Pal et al. list a few notable networks that utilize deep learning for object detection such as AlexNet, ResNet and VGG16 [8].

2.3.2 Two-Stage Detectors

Two stage object detectors are given their name because of their two phases to identify objects: region proposal and object classification [8]. A region is suggested by the model by using convolutional layers to identify object features [8]. Since there are two phases to these models, higher accuracy is commonly seen because the models can take their time to hone in on the first pass of the convolutional filters. Pal et al. explain that two stage detectors are commonly comprised of four modules: "The first module proposes object regions in the image frame. In the second module, a fixed-length feature vector is extracted from these regions. Third module deals with object classification task. In the last module, bounding boxes are fitted over the classified objects" [8].

Within the first module, the network is fitted with a search method that will give suggestions for future modules to use as a proposed location of sought after objects. These locations are images that are fed into a deep CNN where inputs are

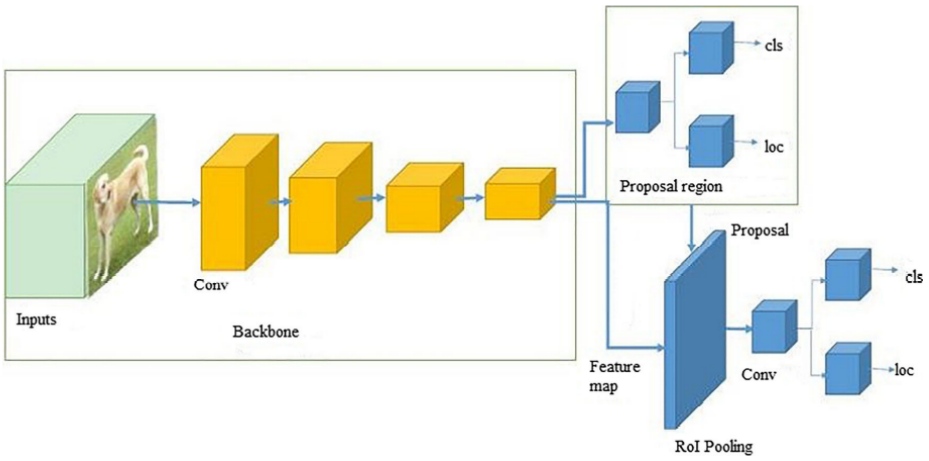

Figure 7: Basic architecture of a two-stage detector [8]

fixed length vectors used to classify the objects within the proposed zones [8]. These vectors are the same size so that the features extracted are able to extrapolate to any sized object in the original image. This means that the objects can be different sizes amongst the training/testing split within the images, therefore, the region proposals can be different sizes [8]. The feature generation is handled by the CNN component of the network. A basic understanding of CNNs is discussed further on in this chapter.

Not all two stage detectors are alike. Two stage detectors further break down into architectures such as region convolutional neural network (RCNN), Fast RCNN, Faster RCNN and many more. The details of these networks are outside the scope of this research; however, the understanding that a two stage detector sacrifices speed for accuracy is important. The demand for an accurate network architecture that is also operational on live video pushes this research to explore options that are faster than two stage detectors.

2.3.3 One-Stage Detectors

What One-stage detectors lack in detection accuracy they make up for in speed [8]. Here, the regional proposal stage is skipped causing bounding boxes creation

and object classification to be made in one step [8]. Figure 8 shows that there is no region proposal to aide in object classification that is included in Figure 7. The allure of using a one stage detector over a two stage detector is for detection of real-time images. These type of detectors are the basis for this research since speed is required to keep up with operational usage.

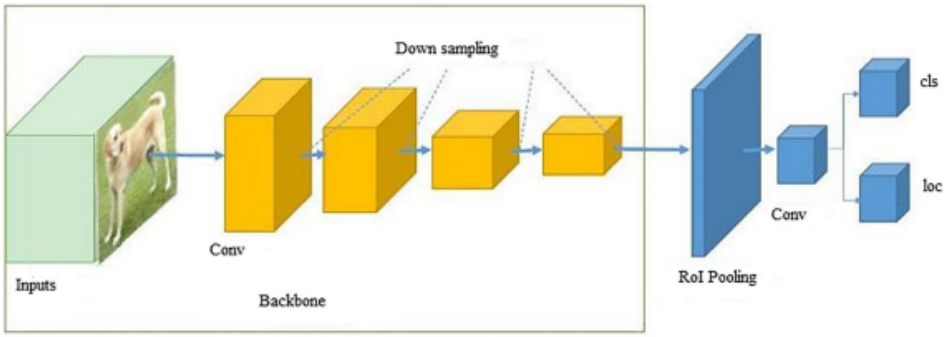

Figure 8: Basic architecture of a one-stage detector [8]

2.3.4 Object Tracking

Object detection is naturally followed by object tracking. The goal of tracking the object is so that the network is able to link to the detected object through progressive frames and follow it by holding on to past information [8]. Similar to object detection, there are singular and multiple versions of object tracking; this research is only concerned with Multiple Object Tracking (MOT). A primary concern of MOT is to create unique identifiers for the separate objects as they move around frame by frame [21].

Highly sophisticated networks are needed to provide these links to be applied in real time scenarios [8]. Such a network (AlexNet) was created by Krizhevsky's team for the ILSVRC 2012 competition, showing that object tracking is indeed useful for increasing accuracy of a deep network [20], [15]. Specific object tracking results among object tracking literature is hard to derive since interrelated disciplines such

as feature tracking and object detection must be done also [8]. Object tracking has three difficulties that newer networks are always trying to adapt to and overcome: tracking speed, background distractions and spatial scales [21].

Prepping the network is crucial for it to be able to track fast enough to handle real-time data. These issues are caused and fixed by the choice of the type of detector utilized [21]. If track speed is too slow, then the network may cease to track the object on screen. Likewise, if accuracy is too low, the network may again lose its tracking of the object. A good blend in the network will be the deciding factor of how well your model can track.

Background distractions are caused by the available training and testing data. These are not factors that may always be able to overcome. These distractions greatly affect the accuracy of the model. Tracking objects in a cluttered environment or tracking small objects among large objects can be problematic for the detectors. One specific type of network distractions is called occlusion [21]. This occurs when objects that are being tracked overlap for some amount of frames, and this can confuse the model. Not only can this affect model performance but when the objects cease to overlap, the model may predict the objects as new objects, seemingly out of thin air [21].

When performing object tracking, targets shown are often different sizes[21]. The different aspect ratios of the targets from one image to another can further confuse the detectors, which will negatively affect model performance. One way that this issue is combated is through "anchor boxes", which are a series of bounding boxes with a fixed size that are used by the network to size up the targets [21].

A technique using MOT is explained in the methodology section with an attempt to utilize a similar application for detection smoothing.

2.3.5 Similar Applications

2.3.5.1 Thesis Predecessor

The predecessor of this thesis work showed that the addition of a BNN into a CNN-RNN combined network saw gains in performance and the additional capability of monitoring prediction probabilities [22]. Swize tested the differences between a BCNN-RNN for classifying video data. The dataset employed by Swize was from The University of Central Florida (UCF), which comprised of 13,320 videos spanning 101 total classes [2].

The research of Swize [22] was similar to Simonyan and Zisserman who used a very deep CNN to great success for the 2014 ImageNet Challenge [23]. Their application of a very deep CNN is more than is applied in this research but their initial setup of the CNN framework is useful. Their model was used to classify a dataset of 1.3M videos for training and 1000 classes with roughly 25% misclassification [23].

2.3.5.2 SqueezeDet

SqueezeDet is a fully convolutional single-shot detector (SSD) used for object detection, written about in June 2019 [24]. SqueezeDet was created with object detection specific to autonomous driving in mind [24]. The inspiration for this network was to focus on a small model size that was energy efficient to aid in implementation for system deployment [24]. Many paths taken for design choices in SqueezeDet's creation were closely tied to the You Only Look Once (YOLO) architecture. SqueezeDet utilizes SqueezeNet as their CNN backbone for its model size and energy effeciency for detection [25]. The SqueezeNet Model size is smaller than other tradionally larger models because of it being built upon "Fire Module", which is made up of a "squeeze" layer for input and is output to the "expand" layer for output shown in Figure 9 [25]. This module allows for a faster, less parameter backbone at the expense of some ac-

curacy. The SqueezeDet team consciously chose the decision of speed over accuracy to allow for a more operational design.

SqueezeDet is the starting point for this research. The background of its usage will be elaborated on in the methodology. While not using SqueezeDet directly, using the inspiration for SqueezeDet, YOLO, will be the main architecture in this research.

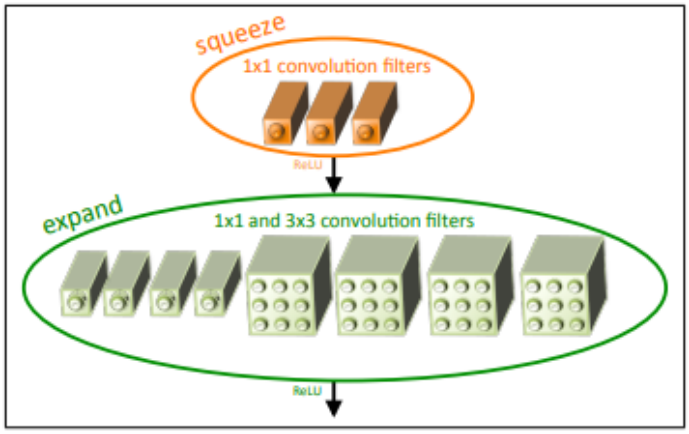

Figure 9: Organizational layout of convolutional filters in the Fire Module [25]

2.3.5.3 You Only Look Once (YOLO)

YOLO-net is tailored to object detection of real-time images, making it a very fast design. The YOLO network can predict up to 100 regions per image at a speed of 45 FPS, not considering batching when used with a Titan X GPU [8]. The way YOLO-net is able to achieve these speeds is partially due to the way it considers the object detection problem. Using regression, YOLO picks features for class probabilities and bounding box locations [26]. YOLO first separates the initial image into a S x S grid. Then, creates B bounding boxes with a confidence level per bounding box. The network creates a class probability map in parallel to the bounding boxes map that labels each cell with a probability of containing a class. The two maps are then cross analyzed to produce the final detections [26].

YOLO has had many iterations in the past. There have been smaller forks of main version iterations to achieve certain model parameters or constraints, however, there are four distinct versions. YOLOv1 in 2015 [26], followed by YOLOv2 at the end of 2016 [27] and YOLOv3 in 2018 [28] are the most commonly known. With the withdrawal of Joseph Redmon from the computer vision scene, Alexey Bochkovskiy picked up where Redmon left off. In April 2020, Bochkovskiy published a paper on YOLOv4 detailing significant improvements and a community fork to the publicly accessible code [19]. YOLOv4 continues to use Darknet as the CNN backbone similar to its predecessors and performs admirably with real-time detection speeds exceeding 60 FPS [19]. As an aside, there is a "YOLOv5" that has had claims of faster speeds and even more lightweight. This version has not been published in this iteration with replicable results as of this date and thus was not considered for implementation in this research.

Darknet is an open source neural network framework that is written in C and uses CUDA [29]. CUDA is an API that helps existing libraries connect to compatible GPUs for accelerated processing. This speeds up computations by giving these GPUs the capability of parallel processing on their available CUDA cores [30]. C is extremely useful as a compiled programming language meaning that it has an edge for efficiency and speed. Darknet being coded from C and CUDA means that it has the speed for clean computations as well as being open source gives Darknet many positives to be used for the YOLO CNN backbone.

YOLOv4 showcased many appealing qualities that this research seeks to implement. Most appealing was a lightweight architecture capable of performing at operational speeds. Given the above reasons, YOLOv4 was the chosen architecture for object detection.

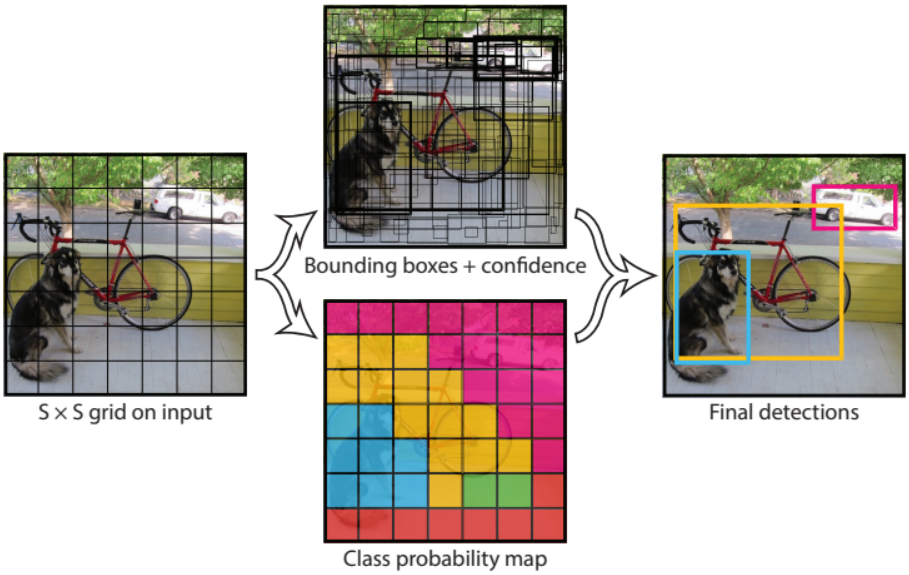

Figure 10: Basic overview of the YOLO process [26]

III. Methodology

The methodology described in this chapter illustrates the path taken from using a toy problem to illustrate the usage of a classification smoothing technique to employing smoothing for a robust object detector. The first section details the toy problem setup for single video classification. Then, a prospectus of the model trained for multi-object detection using Bochkovskiy et al. newly published object detection network, YOLOv4 [19]. This section finshes with a synopsis of the implementation efforts for object detection smoothing within a possible multi-object space.

3.1 Smoothing for Single-Video Classification

3.1.1 UCF101 Dataset

The toy problem uses the University of Central Florida's (UCF) dataset for action recognition [31]. The dataset boasts 13,320 videos spread over 101 classes. The videos are collected from YouTube videos that showcase an action. The recordings are given as 320 x 240 pixel images shot at 25 frames per second (FPS). Classes are sectioned into 25 groups that consist of around five videos each lasting approximately five seconds per video. The dataset was slightly modified from being an entire YouTube video to only being clips from the video that showcase the action listed as the truth value for that video. This just means every frame of the video should result in the truth value and there are no B-role shots or background noise.

To scale down the problem, the original dataset had to be modified. The 101 original classes were modified to only include 10 classes: ApplyEyeMakeup, ApplyLipstick, Archery, BrushingTeeth, CuttingInKitchen, Fencing, GolfSwing, HeadMassage, HulaHoop, and PizzaTossing. These classes were carefully picked to introduce some

Figure 11: Snapshot of UCF101 classes.

hard to decipher actions, such as ApplyEyeMakeup, ApplyLipstick, HeadMassage, and BrushingTeeth; as well as some clearly distinguishable actions such as CuttingInKitchen, Archery, and Fencing. 1-2 videos were extracted from each of the first 20 groupings to be used in training and validation. This totaled 270 videos.

3.1.2 Technique

CNNs perform video classification by treating each frame as an independent test frame. CNNs are a tried and true method and are especially useful for showcasing the video stutter at the frame level [11]. Since CNNs use no temporal information in classification, this research investigated a technique to bring together past information to assist in making the prediction on the current frame. However, this technique still allows for the networks independent prediction for the current frame to hold some weight. In order to do this, an average smoothing technique is used. Where

smoothing is done over some queue size, length $\boldsymbol{x}$. Every frame is predicted by the CNN as normal. This prediction is placed into a first in first out (FIFO) queue. The current frames prediction is calculated using the past x frames independent predictions, including the current. The frames prediction is stored as the pseudo probability distribution output from the network. The entire queue is summed across the axis of each class and divided by the count of predictions. The highest average across the classes is selected to be the smoothed output prediction from the model. This technique effectively seeks to suppress the stuttering effect at the frame-by-frame level. Importantly, the smoothed prediction is never used for any future model smoothed predictions, only for classification. Hence, the model will not get stuck in a reoccurring prediction of the same class because of past smoothing. Thresholds can be changed as well as the queue lengths to produce different results. Shorter queues result in less past information being used, but provides for a more responsive network, while the opposite is true for longer queues.

3.1.3 Model

Resnet-50 is used as the base model for this experiment and transfer learning is used to freeze part of this network [32]. From Resnet-50's 50 layers, 49 of them are frozen. The head of the network is removed for the application of our layers that are tailored to the modified UCF dataset. Pooling is done to condense features and then the images are flattened and fed into two dense layers used to streamline the outputs into the classes of concern.

3.2 Order of Operations

This process will start from the classes and videos selected in the "Dataset" chapter. These 270 videos are first compiled into frames. During this phase, the videos

are broken down one frame at a time and converted from BGR (due to the library Open-CV) to RGB. Each frame is then given a specific label and sorted into a directory folder with its truth value as the folder name. This phase creates 43,826 frames which are used for training and validation.

Before the frames are able to go through the network, they must undergo preprocessing. They are loaded into a data frame with their truth values being extracted from folder names, and are reduced to a resolution of 224 x 224 so they can be used in Resnet-50's architecture. The images are augmented so the network can be more robust. Training and validation occurs in batches of 32 for three epochs. The resulting model is saved as an "h5" model and saved for use in testing.

Testing is finalized by performing a few different methods listed in the section below. Video prediction is run using the saved model. The video is compiled again into frames, run through the CNN and smoothed using the average smooth queue method. The smoothed prediction is placed at the top of the frame and saved. Once all frames from the test video is run through the network, all frames are stitched back together and saved as a *.avi* format. The code has been used to predict overall video classification as well as output an array of every frame in the video for closer inspection on where the frames being misclassified lay, as well as computing the number of total misclassified frames. This is helpful in judging the performance of differing numbers of sizes for the smoothing queue.

3.3 Measures of Performance

The measures of performance of this technique come in three distinct ways. First, testing model accuracy with training and validation. This comes from recall, precision, loss, and overall accuracy. This measures how well the model learned at the frame level for model implementation. Secondly, looking at a confusion matrix among

the model as it is classified to an unseen test set of videos. This is also examined further at an overall video classification level with no smoothing frame-by-frame and then also smoothed at a 30 and 120 frame smoothing level. Lastly, video inspection is conducted on how well smoothing was preformed and supplemented by a frames analysis in section 4.1.2.

3.4 YOLOv4

3.4.1 Dataset Description

The dataset used for this research, consists of 23,067 frames of annotated high-altitude drone images. The frames were initially contained in 33 different folders which indicated the video or same landscape they came from, hence all 23,067 frames are not distinctly different. The original images are annotated in KITTI format within their accompanying ".txt" files. The images are pre-augmented, with each frame being rotated, mirrored, resized and having the color inverted. There are nine distinct classes: MV, van, car, truck, suv, bus/rv/motor_home, cargo, dismount and other. Most of the images are taken of highway and rural landscapes from a top-down viewpoint. The images vary in quality with some being very clear and others hazy. All images are at 1600 x 1200 resolution, however, they are inlaid within a black surrounding box. The image size within the boarder varies as showcased in figures 12, 13, and 14.

3.4.2 Data Annotations

The research dataset was initially in *KITTI* format. All text files needed to be converted from *KITTI* to *YOLO Darknet TXT*. YOLO Darknet TXT contains one .txt file per data image. These annotations are normalized within a $[0, 1]$ range which makes it easier for Darknet to stretch and scale them. Each line in the .txt file

Figure 12: Ex 1. Buildings and Vehicles, slight haze and large

Figure 13: Ex 2. General Highway Image, high quality, clear and medium inlay

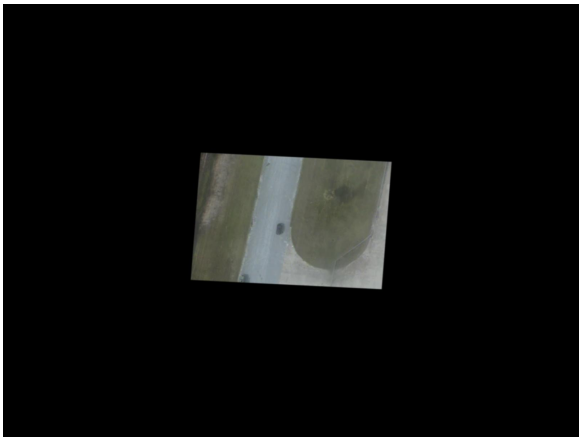

Figure 14: Ex 3. Rural Image, hazy and small inlay

corresponds to an object in the related image and contains four peieces of information:

$$< objectclass >< x\_center >< y\_center >< width >< height > \tag{3}$$

Where:

- $< object\_class >$ — integer specifying the object's classification from 0 to $(classes - 1)$
- $< x\_center >< y\_center >< width >< height >$ — float values relative to width and height of image

Associated with the .txt files are an accompanying master file which is a .NAMES file. This file is the key to decoding the names of the objects. Each name of a detection object is on a newline of this file and tells the network the number associated with the *objectclass*. Figure 15 is an example of a sample image with three objects annotated in Darknet format. Figure 16 is the dataset's .NAMES file, showing that two of the objects in figure 16 are *car* and the last object is a *suv* (indexes are off by one because of zero indexing).

Example001.txt

```
1  2 0.788 0.206 0.065 0.034
2  2 0.863 0.165 0.022 0.058
3  4 0.772 0.246 0.051 0.032
```

Figure 15: Darknet .txt Format

obj.NAMES

```
1  MV
2  van
3  car
4  truck
5  suv
6  bus|rv|motor_home
7  cargo
8  dismount
9  other
```

Figure 16: Dataset .NAMES file

3.4.3 Dataset Challenges

There are some important features from the dataset to detail in the dataset and to highlight their potential effect on gathered results. First, two of the 33 folders were removed from training. The two folders consisted of images not from the drone but of stock image frames that were excessively watermarked, thus were removed to not contaminate the training. This removed 2,492 images from the original dataset, leaving 20,575 frames.

Second, the most of the original folders contained frames that were very similar, or exact images only augmented. Frames were commonly sequential frames stripped from videos. These folders varied immensely in size, from a max of 4434 frames to a minimum of 33 frames. Further, the largest four folders (just four different landscapes) make up 61.5% of the total data. A robust model requires multiple landscapes and differing images. Some difficulty in predicting in other landscapes could be stemmed from the dataset being skewed toward these four backgrounds for testing.

Third, all of the data is taken from a high altitude. Therefore, future usage for low altitude object classification may not benefit from this training set. If model usage were to be considered on lower altitudes, additional training data is required.

Lastly, there is an imbalance within the classes of the data set. In figure 17, a breakdown of classes is detailed. Classes such as "MV", "dismount", and "bus/rv/-motor_home" are severely lacking representation in the dataset while "cargo" and "other" also show potential issues in representation. These five classes individually have less than 2% representation of total instances. Together, these bottom five classes account for 6% of all instances. When looking at file representation, these classes only slightly gain better representation. For example, this means that most of the files that include a "MV" only contain one, whereas files that contain "car" are much more likely to contain multiple instances of the class. These classes could

be predicted to a better quality if the model could train on more instances of these lower represented classes.

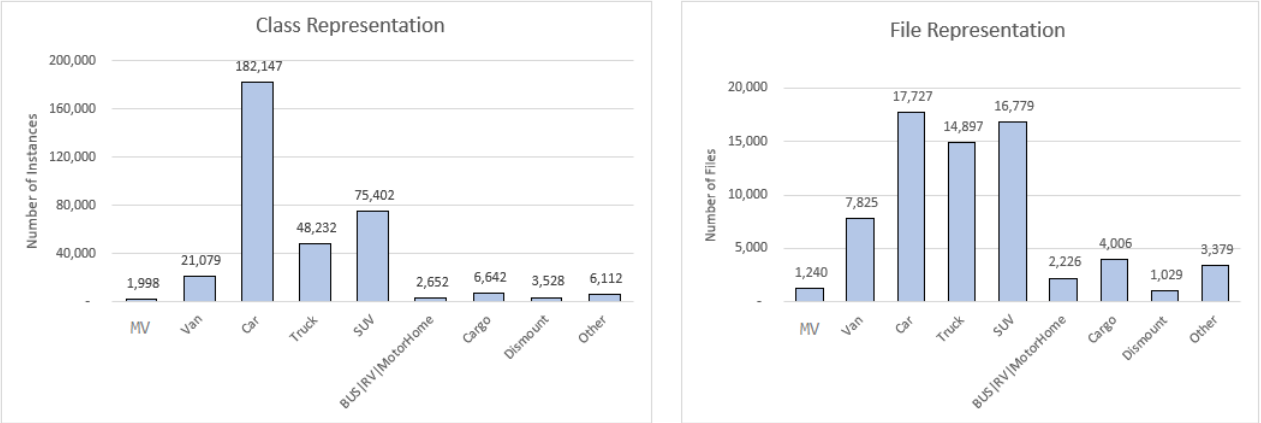

Figure 17: Research dataset class statistics

3.5 Model Architecture

This research began with code from SqueezeDet, with the intention of further applying this architecture for investigating smoothing of object classifications between frames. The code was delivered in a repository that came with the original SqueezeDet code [24], as well as a wrapper configured to detect based on the provided data. The code was abandoned for the YOLO architecture after encountering package depreciation issues, legacy version complications and a lack of possible model retraining excursions.

3.5.1 Choosing YOLOv4

Since SqueezeDet was based off of YOLO and YOLO is a leading one stage detector for competitive object detection, YOLO was a clear choice. The newest version of YOLO is currently YOLOv4. YOLOv4 was introduced on 23 April, 2020 [19]. It boasted impressive AP scores while also maintaining a classification speed of approx-

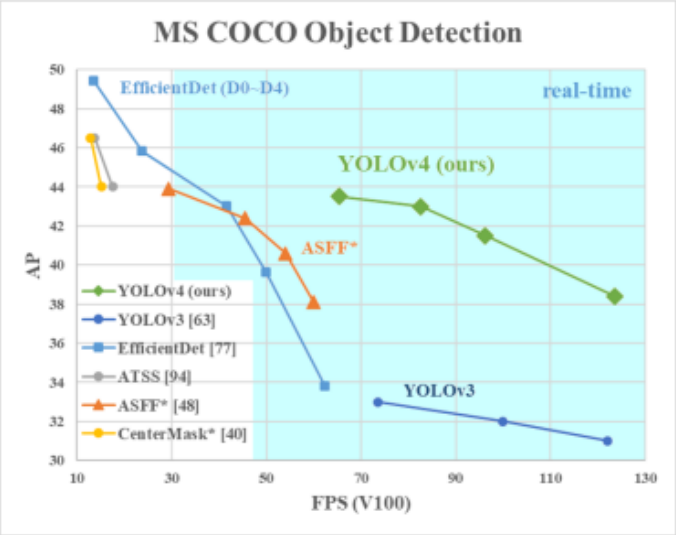

Figure 18: YOLOv4 Object Detection comparison

imately 65 FPS [19].

Basic YOLOv4 is trained on the Microsoft COCO dataset [33]. This is a dataset consisting of 80 easily identifiable classes from in their natural context [33]. The dataset has a train/validate split of 118k/5k [33]. Of the classes trained for COCO, classes such as car, truck and bus are beneficial for potentially classifying ISR footage taken from a lower altitude than what the provided dataset supports.

3.5.1.1 Windows 10 - Python

The hardware used for running the network was done on a windows 10 machine using an AMD Ryzen 7 5800X CPU, NVIDIA GeForce RTX 3080 and 16 GB DDR4-3600MHz RAM. CUDA was configured for the RTX 3080 to make use of the 8704 CUDA cores and 10 GB of memory to enhance computations. One of the libraries that CUDA commonly interacts with is cuDNN. NVIDIA cuDNN is a GPU accelerated library made specifically for deep neural networks. cuDNN's library is customized to provide highly efficient applications of commonly found neural network process such as convolutions, pooling, normalization, and other similar processes [34].

3.6 Training YOLOv4

The YOLOv4 custom configuration file had to be changed to train competently on the UAV footage. Training for this new dataset is not done completely from scratch. For example, weights on all network nodes are not random. The weights are taken from the first 137 layers of YOLOv4 and used as a starting point for training. Therefore, the file "yolov4.conv.137" is used to initiate training.

3.6.1 Hyper-parameters

The file "yolov4-custom.cfg" houses all of the preliminary setup information for the training of the network. The network's final hyper-parameters are summarized below:

Table 1: Hyperparameters for YOLO Trained on UAV Footage

$batch = 32$	$subdivisions = 16$	$width = 416$
$exposure = 1.5$	$hue = 0.1$	$height = 416$
$channels = 3$	$momentum = 0.949$	$learning\_rate = 0.001$
$decay = 0.0005$	$saturation = 1.5$	$max\_batches = 20000$
$burn\_in = 1000$	$steps = 16000, 18000$	$policy = steps$

Bochkovskiy *et al* gives specific parameter guidelines for training datasets on YOLOv4 [19]. A GitHub repository is shared by Bochkovskiy *et al* to aid in training YOLOv4 networks on new datasets [19]. The following are the changes from the "yolov4.conv.137.cfg" network configuration, which is the network structure Bochkovskiy *et al* used in training on the COCO dataset [19]. The changes are made at the reccomendation of Bochkovskiy *et al* because the extra features needed for the COCO dataset found in the previous model, such as higher batching, subdivisions, and steps, are not needed in this dataset since it is remarkably smaller. Batching was changed

from 64 to 32, multiple of 16 is kept. The reduction in total images were dramatically reduced compared to the COCO dataset. Since batch is set to 32, subdivisions are set to 16 to determine the mini batches, again multiples of 16. $max\_batches$ should roughly be $2000 * (\text{number of classes})$, yet larger than number of total images. So $max\_batches$ is changed to 20,000 to be above 18,000 but close to max number of images. $line\_steps$ is set to be roughly 80% and 90% of $max\_batches$. Before each of the three YOLO layers in the network, the $filters$ parameters need to be hard coded to be $filters = (classes + 5) * 3$.

3.7 Smoothing for Multi-Object Classification

This section details the methodology for the transition between the toy problem for smoothing single video classification to an operational problem of multi-object detection smoothing.

3.7.1 Obstacles

Key differences between the toy problem and the operational problem need to be identified so that a way forward can be determined.

First, in the toy problem we made use of the simplicity of a video *always* having some classification. In the operational problem, there is always a changing number of possible objects in an image. We no longer have one static queue, but a dynamically changing number of queues. This could result in having dozens of queues at once or conversely zero queues if no objects are in the frame. A procedure needs to be in place that constantly creates or terminates queues related to when objects enter or leave the frame. This dynamic queue has to also inherently *track* the object such that the queue knows what object it is smoothing. This could possibly rectified with other types of algorithm additions, such as Deep SORT [35].

Second, the model could fail to detect the object at all. This is a moot point for the toy problem where there is always some classification, even if it is the incorrect one. Here, procedures must be detailed in order to handle the correct, incorrect, and lack of classification. Upon doing this, the model must be able to continue tracking the object such that the queue is not totally reset after failing to detect for one (or possibly greater than one) frame.

Third, is the inverse to the classification stutter we are trying to solve. We have previously detailed the stutter from correct detection classification, to false classification, and back to correct detection. However, the issue of correctly identifying that the object is *not* there, to incorrectly identifying an object as being there, to back to correctly showing no object is detected is harder to solve. This is harder to solve because using this method, there would have to be a queue for every object on the screen in order to smooth. If a object isn't detected, there is no queue to smooth. A proposed solution to this is to smooth all new identified objects by a smaller queue amount. For example, if all new objects identified by the model are not shown until 6 of the last 10 frames are identified to be a class object. This would prevent cases in which objects flicker onto the screen that should not be class objects in the first place. This would impose a delay onto all new objects entering the image but would greatly reduce the stutter of false positive predictions from the model.

IV. Results and Analysis

This chapter summarizes research results from the toy and operational problems.

4.1 Toy Problem

Table 2: Model Training Analysis Summary

	Precision	Recall	Accuracy
ApplyEyeMakeup	0.97	0.97	0.97
ApplyLipstick	0.92	0.95	0.94
Archery	0.87	0.95	0.94
BrushingTeeth	0.97	0.97	0.97
CuttingInKitchen	1.0	1.0	1.0
Fencing	1.0	0.96	0.98
GolfSwing	0.94	0.94	0.94
HeadMassage	0.88	0.98	0.93
HulaHoop	0.91	0.69	0.79
PizzaTossing	0.87	0.75	0.81
Total Accuracy			0.93

Table 2 shows summary statistics for the model used for the toy problem. The model statistics showcase notably high accuracy and recall amongst most of the classes individually. These results were promising because it showcased that the model was learning correctly and operating well. However, a potential issue lays in these accuracy measurements. It doesn't affect model usage, but does artificially inflate these accuracy numbers. When the train and validation splits occur, these images are still too close in temporal space. Take for example, a snapshot in time during the training

process; imagine that the Archery class has a video of a man pulling back a bow. It could be the case that frames 1,3,5,7 are used for training, while validation is conducted on frames 2,4,6,8. The difference in information in these frames are very minimal, so it would be expected that the accuracy would be higher than if the model was trained on a complete set of independent images. In the below subsection, impact from the smoothing technique on the overall video's classification is tested, as well as test accuracy derived from videos that the network has never seen such that there is no impact from closely related frames as mentioned above.

4.1.1 Classification Impact

Testing was conducted to measure the impact that the smoothing technique played on overall video classification. Theoretically, if the technique should converge more individual frames to the greater predicted class, then the videos overall classification should stay the same. A confusion matrix was created based on doing no frame smoothing (Figure 19), 30 frame smoothing (Figure 20), and 120 frame smoothing.

Overall classification for 275 videos resulted in 90 misclassifications when no smoothing was implemented. This results in 67% test accuracy. Many classifications seem to be a result of having classes that were close together, which was intended as this helped to see greater benefit in smoothing at the frame level. The result of smoothing 30 frames did not change the test accuracy, however, this was only a coincidence because it did change a couple of the videos overall classification, an unexpected result.

Upon reflection, it seems that overall classification could change if the network had many oscillating classes frame-by-frame or the total number of frames that predicted the original class were spread over the whole videos classification, which would result in some of the frames instead being smoothed to a different class than was predicted

Figure 19: No smoothing - Classification heatmap

Figure 20: 30 Frame smoothing - Classification heatmap

in a denser area of the video. One other potential reason could be that the frames at the beginning of the videos are not fully smoothed. What this means is that the queue is not full until the first x frames have been classified. So if the queue length is 30 and frame 15 is being classified, only 15 frames are currently in the queue so the smoothing for that frame is 15, not yet 30. Therefore, the beginning frames where $(frame\_number) < x$ have less past information used to smooth than the remaining frames in the video.

Testing done with 120 frame smoothing was identical to 30 frame smoothing except that one additional video was misclassified. Increasing the queue length to incorporate more frames to average against eventually leads to too much smoothing. The longer the queue, the more delay is artificially introduced into the architecture's response time. So, since 120 frame smoothing was frequently similar to 30 frames of smoothing, choosing a smaller queue is more valuable for keeping a highly responsive system.

Additionally, the smoothing technique has very little affect on overall classification, which is a positive trait.

4.1.2 Frames Analysis

Testing was conducted on how the frames classification was spread along the video. Figure 21 shows a rather typical result for many of the videos in the set. Most of the time, having a queue length of 25 frames was enough for total image smoothing on a video that was relatively well predicted. Since this research expects a relatively good network to be used on the front-end, these are promising results. In Figure 21, the pattern showed very consistently where zero frame smoothing showed many frames to be misclassified into a large number of bins (Figure 21 showed 6 different bins that frames were categorized into). Then using 25 frames of smoothing, this would usually push the number of misclassified bins to be just one, two or three. Even though this

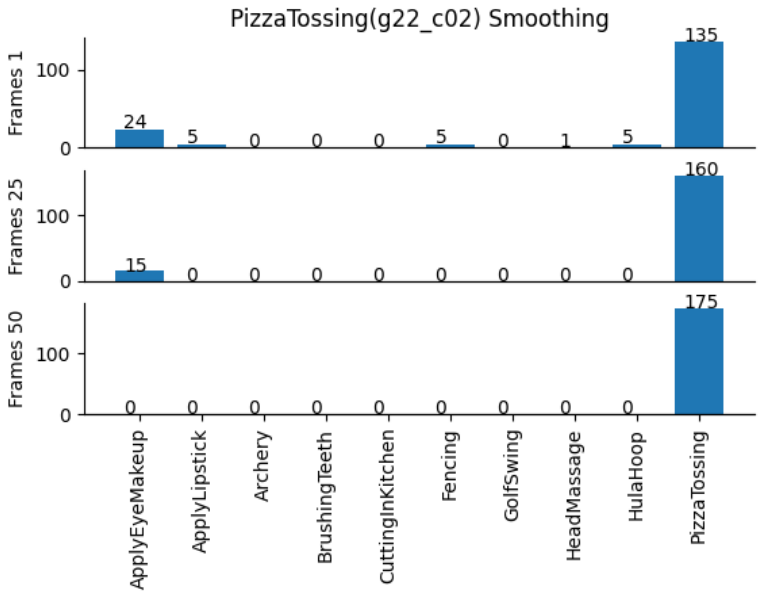

Figure 21: Multiclass frame smoothing for no smoothing, 25-frame smoothing and 50 frame smoothing.

has a change in categorization while the video plays, it is already much more easy to understand visually. With 50 frame smoothing, the image was smoothed to be only one bin, which was the correct bin for classification.

Figures 22 and 23 show the same video from Figure 21. These figures show how the sequential frames on how they are classified relative to their neighboring frames. With no smoothing stuttering can be clearly seen from the intermittent red bands. If the red bands are more than a couple in a sequential order, this can be attributed to poor model performance instead of a model stutter. After smoothing for 25 frames, all small stuttering occurrences have been smoothed. Now instead of stutters, there are blocks of classifications.

Overall, smoothing does its job well, but certainly will suffer from having a bad CNN front end. There were instances where smoothing can coerce an image classification to the wrong classification depending on how uncertain the base CNN model is at predicting certain classes. If the model is truly misclassifing the bulk of frames

Figure 22: No smoothing - Sequential frames (Red are misclassifications)

Figure 23: 25 Frame smoothing - Sequential frames(Red are misclassifications)

for a video then the smoothing technique will smooth correct classifications to the wrong classification.

Given these results, the smoothing technique does work effectively for smoothing model predictions. The smoothing technique still relies heavily on the base model predictions. The technique works best in cases where stuttering is intermittent. The technique performs poorly when many model misclassifications are made in sequence, which leads to the technique smoothing away correct classifications.

4.2 YOLOv4

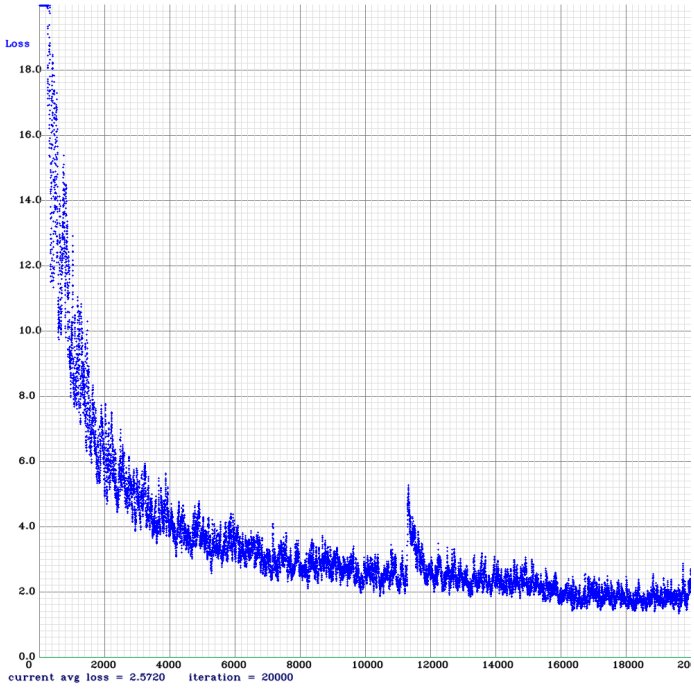

Figure 24: Model training loss statistics

YOLOv4 was trained on the dataset for 20,000 iterations. Upon completion the model had a loss of 2.6%. Since the entirety of the dataset was used for training, additional drone footage was requested for testing. Non sequential frames were extracted from the requested footage and hand-annotated. These frames were used in mAP testing and for summary statistics of the model. Using a confidence threshold of 50, the model had a precision of 33% and recall of 31%. The model mAP_{50} was 23.7%. Model precision suffered from issues resulting in false positive detections as can be seen in 3 below:

Table 3: Model Precision broken out by class

Class	True Positive	False Positive
Cargo	0	11
Other	0	5
Car	21	32
Truck	6	5

Table 3 shows that the model is cable of detecting these classes, but tends to falsly identify some objects. It is important to observe the models ability to correctly identify "car" and "truck" roughly 50% of the time, while incorrectly identifying the "cargo" and "other" classes. This shows that the model is able to be used to detect the former but has not learned the later classes. The overall mAP_{50} is pushed down greatly from these false identifications in classes that are not correctly learned.

V. Conclusions

Solving neural network misclassification is an issue at the leading edge of AI research. Much research has been conducted in enhancing the capabilities of the underlying model. This research scopes this problem into the lens of classification stuttering, where misclassifications are the minority in an expanse of correct classifications. Therefore, we take a different approach of making additions to a network and acquiring smoothed results with a potentially low impact to the speed at which the model runs.

In this thesis, the problem of video stuttering was investigated. A toy problem was used to test a potential way of smoothing classification stuttering in single action videos. This gave promising results showing smoothed classifications. Benefits and consequences of the technique were discussed such as keeping the system highly responsive yet creating an artificial delay.

Following the toy problem, the dataset consisting of UAV footage was analyzed. Dataset challenges were detailed and areas for improvement were listed. Using this dataset, a YOLOv4 network was trained on 20,575 frames. Using this trained network and lessons learned from the toy problem, the smoothing technique was applied to a multi-object detection problem. After continuous unsuccessful implementations, important obstacles were documented and potential solutions are outlined for future work below.

5.1 Future Work

This thesis paves the way for additional research to be conducted. Future work should seek to implement the smoothing technique into a multiple object detection network. This research has showed proof of concept with a toy problem but was unable

to resolve the technique for multiple objects. A detailed breakdown of important obstacles to overcome are documented in the methodology section. A method for smoothing needs to be introduced for each object in the frame as well as smoothing objects before they are introduced as a positive detection in the frame. If this can be done, I believe this technique should be used as a end component for any competent object detection network. If implemented to an accurate network, stuttering should rarely be seen.

Appendix A. Toy Problem Additional Results

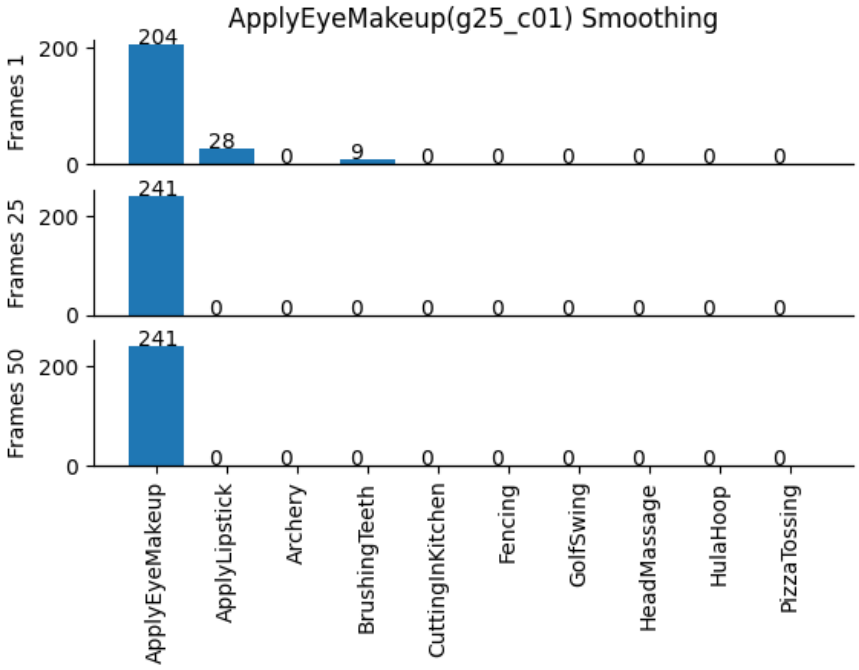

Figure 25: Frames analysis for video ApplyEyeMakeup_25_01.

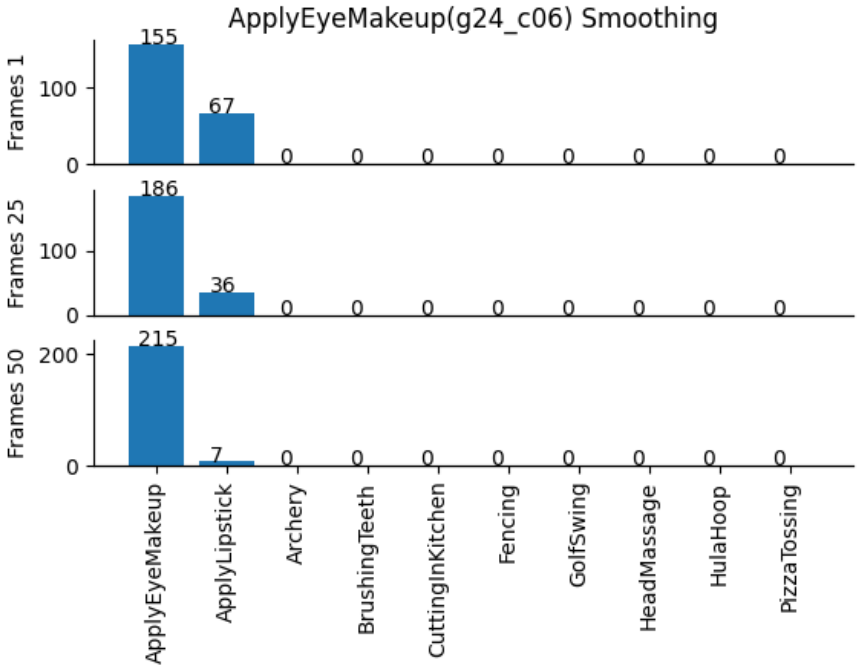

Figure 26: Frames analysis for video ApplyEyeMakeup_24_06.

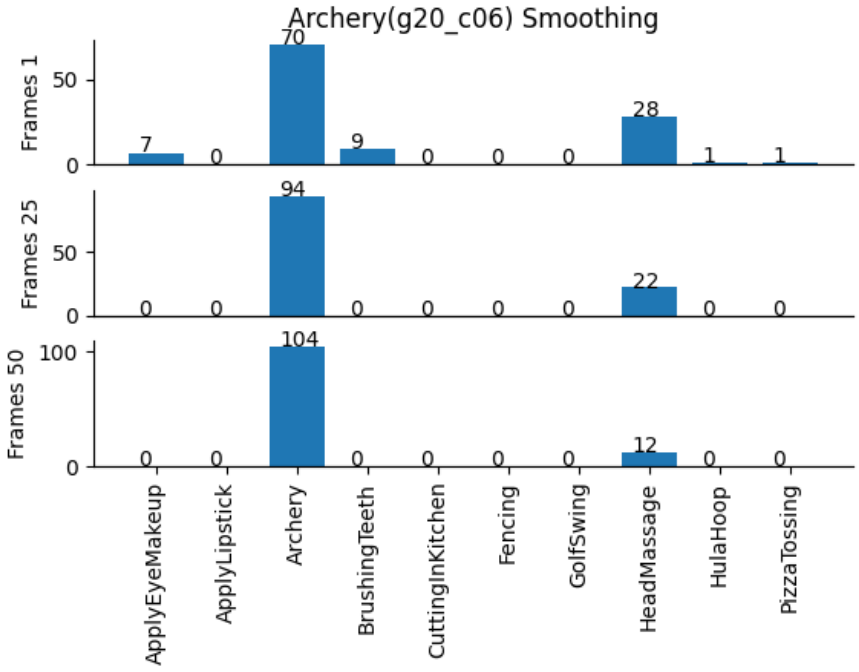

Figure 27: Frames analysis for video Archery_20_06.

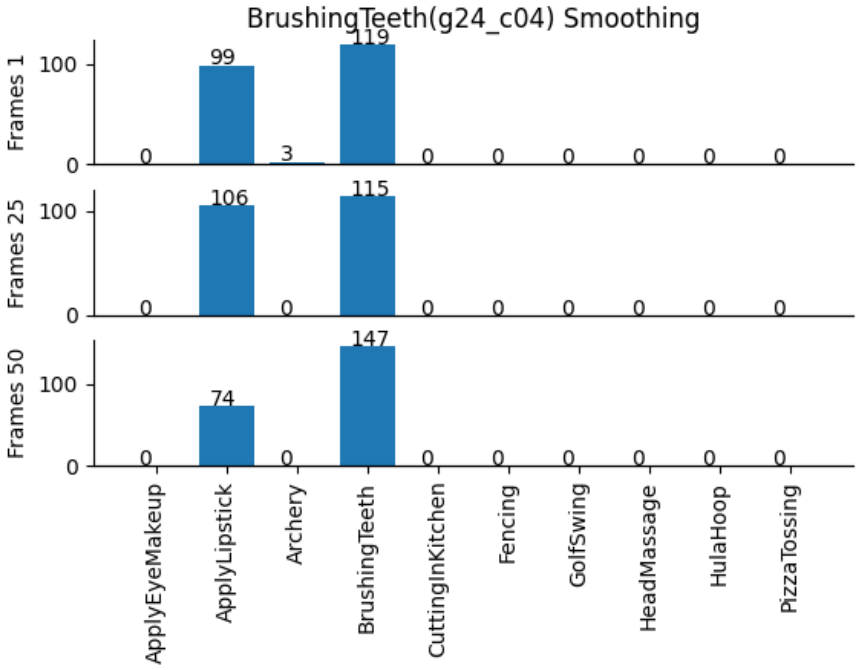

Figure 28: Frames analysis for video BrushingTeeth_24_04.

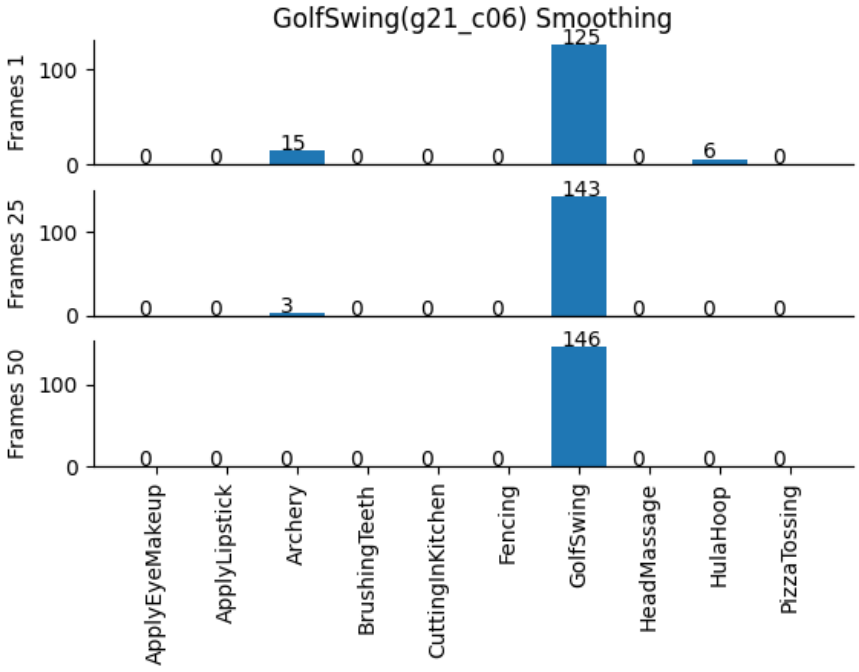

Figure 29: Frames analysis for video GolfSwing_21_06.

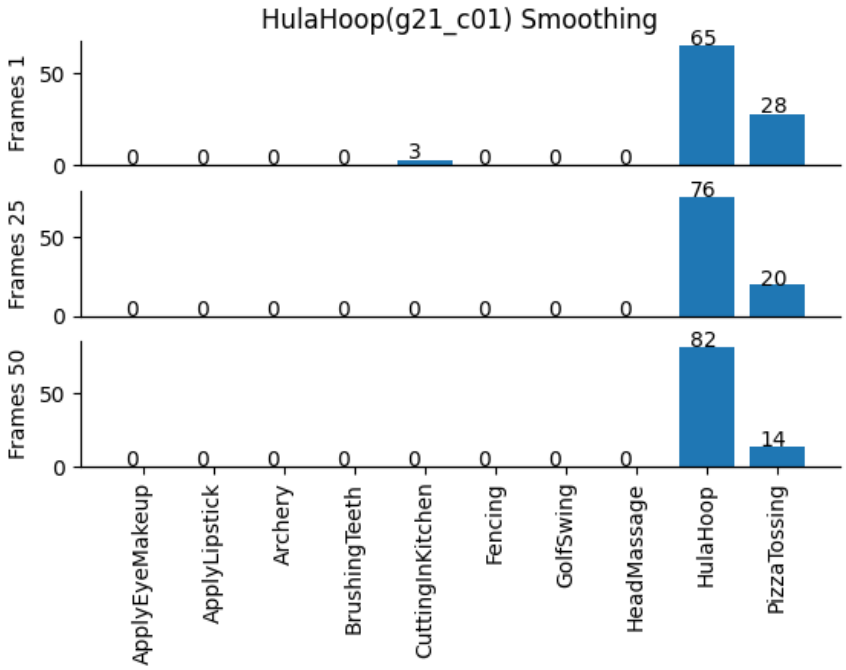

Figure 30: Frames analysis for video HulaHoop_21_01.

Appendix B. Code: Toy Problem Training

```
Training the toy problem and provides graphs of training.
Saves a h5 model and labels object for later use.
Transfer learning is used from Resnet-50.
(He et all, 2015),(Rosebrock, 2019).


import matplotlib
matplotlib.use("Agg")


import os
import cv2
import pickle
import numpy as np
from imutils import paths
import matplotlib.pyplot as plt
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Input
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Dropout
from tensorflow.keras.layers import Flatten
from tensorflow.keras.optimizers import SGD
from sklearn.preprocessing import LabelBinarizer
from tensorflow.keras.applications import ResNet50
from sklearn.model_selection import train_test_split
from tensorflow.keras.layers import AveragePooling2D
```

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from sklearn.metrics import classification_report, confusion_matrix


def plot_confusion_matrix(cm, target_names, title='Confusion matrix',
cmap=None, normalize=True):
    """
    given a sklearn confusion matrix (cm), make a nice plot

    Arguments
    ---------
    cm:           confusion matrix from sklearn.metrics.confusion_matrix

    target_names: given classification classes such as [0, 1, 2]
                  the class names, for example: ['high', 'medium', 'low']

    title:        the text to display at the top of the matrix

    cmap:         the gradient of the values displayed from matplotlib.pyplot.cm
                  see http://matplotlib.org/examples/color/colormaps_reference.html
                  plt.get_cmap('jet') or plt.cm.Blues

    normalize:    If False, plot the raw numbers
                  If True, plot the proportions

    Usage
```

```
-----
plot_confusion_matrix(cm           = cm,# confusion matrix created by
                                        # sklearn.metrics.confusion_matrix
                      normalize    = True,                # show proportions
                      target_names = y_labels_vals,     # names of the classes
                      title        = best_estimator_name) # title of graph

Citiation
---------
scikit-learn.org/stable/auto_examples/model_selection/plot_confusion_matrix.html
"""

accuracy = np.trace(cm) / np.sum(cm).astype('float')
misclass = 1 - accuracy

if cmap is None:
    cmap = plt.get_cmap('Blues')

plt.figure(figsize=(8, 6))
plt.imshow(cm, interpolation='nearest', cmap=cmap)
plt.title(title)
plt.colorbar()

if target_names is not None:
    tick_marks = np.arange(len(target_names))
    plt.xticks(tick_marks, target_names, rotation=45)
```

```
        plt.yticks(tick_marks, target_names)


    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]



    thresh = cm.max() / 1.5 if normalize else cm.max() / 2
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        if normalize:
            plt.text(j, i, "{:0.4f}".format(cm[i, j]),
                     horizontalalignment="center",
                     color="white" if cm[i, j] > thresh else "black")
        else:
            plt.text(j, i, "{:,}".format(cm[i, j]),
                     horizontalalignment="center",
                     color="white" if cm[i, j] > thresh else "black")



    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label\naccuracy={:0.4f};
        misclass={:0.4f}'.format(accuracy, misclass))
    plt.show()


epochs = 3
```

```
frame_folder = '/UCF101/Frames10/' # path to folder holding folder of frames
output_img_path = '/UCF101/Output/plot2.png' # path to output plot
output_model_path = '/UCF101/Output/activity2.model' # path to output model
output_label_bin = '/UCF101/Output/lb2.pickle' # path to output label binarizer

#Set of labels

LABELS = set(["ApplyEyeMakeup", "ApplyLipstick", "Archery",
  "BrushingTeeth", "CuttingInKitchen", "Fencing",
  "GolfSwing","HeadMassage", "HulaHoop", "PizzaTossing"])

imagePaths = list(paths.list_images(frame_folder))
data = []
labels = []

for imagePath in imagePaths:
# extract the class label from the filename
label = os. path. basename(imagePath.split(os.path.sep)[0])
# if the label of the current image is not part of of the labels
# are interested in, then ignore the image
if label not in LABELS:
continue
# load the image, convert it to RGB channel ordering, and resize
# it to be a fixed 224x224 pixels, ignoring aspect ratio
image = cv2.imread(imagePath)
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
```

```
image = cv2.resize(image, (224, 224))
# update the data and labels lists, respectively
data.append(image)
labels.append(label)

data = np.array(data)
labels = np.array(labels)
# perform one-hot encoding on the labels
lb = LabelBinarizer()
labels = lb.fit_transform(labels)

#Training and testing splits using 75% of data
#Testing is 25%
(trainX, testX, trainY, testY) = train_test_split(data, labels,
  test_size=0.25, stratify=labels, random_state=42)

#Data augmentation
trainAug = ImageDataGenerator(
rotation_range=30, zoom_range=0.15, width_shift_range=0.2,
height_shift_range=0.2, shear_range=0.15, horizontal_flip=True,
fill_mode="nearest")

valAug = ImageDataGenerator()
#mean subtraction
mean = np.array([123.68, 116.779, 103.939],
                dtype="float32")
```

```
trainAug.mean = mean
valAug.mean = mean

#ResNet-50 network
#Citation
#(He et all, 2015)
#(Rosebrock, 2019)
baseModel = ResNet50(weights="imagenet", include_top=False,
input_tensor=Input(shape=(224, 224, 3)))
#We use a new head tailored to our data
headModel = baseModel.output
#Create the pooling/full connection layers, with inclusion of dropout
headModel = AveragePooling2D(pool_size=(7, 7))(headModel)
headModel = Flatten(name="flatten")(headModel)
headModel = Dense(512, activation="relu")(headModel)
headModel = Dropout(0.5)(headModel)
headModel = Dense(len(lb.classes_), activation="softmax")(headModel)


model = Model(inputs=baseModel.input, outputs=headModel)

#Freezes the layers from Resnet-50 (The "base model")
for layer in baseModel.layers:
layer.trainable = False

#Compiling of the model hyperparameters
```

```
opt = SGD(learning_rate = 1e-4, momentum=0.9, decay=1e-4 / epochs)
model.compile(loss="categorical_crossentropy", optimizer=opt,
metrics=["accuracy"])

#The model is fit (training)  (Only the head is traininable)
H = model.fit(
x=trainAug.flow(trainX, trainY, batch_size=32),
steps_per_epoch=len(trainX) // 32,
validation_data=valAug.flow(testX, testY),
validation_steps=len(testX) // 32,
epochs=epochs)

#Statistics for the network are below
print("Network Stats")
predictions = model.predict(x=testX.astype("float32"), batch_size=32)
print(classification_report(testY.argmax(axis=1),
predictions.argmax(axis=1), target_names=lb.classes_))
#Loss and accuracy graphs
N = epochs
plt.style.use("ggplot")
plt.figure()
plt.plot(np.arange(0, N), H.history["loss"], label="train_loss")
plt.plot(np.arange(0, N), H.history["val_loss"], label="val_loss")
plt.plot(np.arange(0, N), H.history["accuracy"], label="train_acc")
plt.plot(np.arange(0, N), H.history["val_accuracy"], label="val_acc")
plt.title("Training Loss and Accuracy on Dataset")
```

```
plt.xlabel("Epoch #")
plt.ylabel("Loss/Accuracy")
plt.legend(loc="lower left")
plt.savefig(output_img_path)

trainPredictions = model.predict(x=trainX.astype("float32"), batch_size=32)
#Confusion Matrix for both train and test
cm_train = confusion_matrix(trainY, trainPredictions)
cm_test = confusion_matrix(testY, predictions)
#Plot Confusion matrix
plot_confusion_matrix(cm_train, lb.classes_, title='Confusion matrix: Training')
plot_confusion_matrix(cm_test, lb.classes_, title='Confusion matrix: Testing')

#Saving the model as an h5 so it can be used for classification later
model.save(output_model_path, save_format="h5")
#Pickle the labels for later use as well
f = open(output_label_bin, "wb")
f.write(pickle.dumps(lb))
f.close()
```

Appendix C. Code: Video Classification

```
Predicts videos classification and also saves output video
with all frames strung together for an output classifcation.
Used to visually inspect stuttering.


# import the necessary packages
import cv2
import pickle
import numpy as np
from collections import deque
from tensorflow.keras.models import load_model

model_path ='/Output/activity2.model'# path to trained serialized model
label_bin = '/Output/lb2.pickle'  # path to  label binarizer
input = '/UCF101/UCF-101/HulaHoop/v_HulaHoop_g21_c01.avi'# path to our input video
output_path = '/Output/makeup_output.avi'# path to our output video
size =  1#  size of queue for averaging (1-128) (1 means no averaging)

# load the trained model and label binarizer

model = load_model(model_path)
lb = pickle.loads(open(label_bin, "rb").read())

# initialize the image mean for mean subtraction along with the
# predictions queue
```

```
mean = np.array([123.68, 116.779, 103.939][::1], dtype="float32")
Q = deque(maxlen= size)

# initialize the video stream, pointer to output video file, and
# frame dimensions
vs = cv2.VideoCapture(input)
writer = None
(W, H) = (None, None)

video_classification = [0] * len(lb.classes_)

# loop over frames from the video file stream
while True:
# read the next frame from the file
(grabbed, frame) = vs.read()
# if the frame was not grabbed, then we have reached the end
# of the stream
if not grabbed:
break
# if the frame dimensions are empty, grab them
if W is None or H is None:
(H, W) = frame.shape[:2]

# clone the output frame, then convert it from BGR to RGB
# ordering, resize the frame to a fixed 224x224, and then
# perform mean subtraction
```

```
output = frame.copy()
frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
frame = cv2.resize(frame, (224, 224)).astype("float32")
frame -= mean


# make predictions on the frame and then update the predictions
# queue
preds = model.predict(np.expand_dims(frame, axis=0))[0]
Q.append(preds)
# perform prediction averaging over the current history of
# previous predictions
results = np.array(Q).mean(axis=0)
i = np.argmax(results)
label = lb.classes_[i]
video_classification[i] += 1

# draw the activity on the output frame
text = "activity: {}".format(label)
cv2.putText(output, text, (10, 20), cv2.FONT_HERSHEY_SIMPLEX,
0.75, (0, 255, 0), 2)
# check if the video writer is None
if writer is None:
# initialize our video writer
fourcc = cv2.VideoWriter_fourcc(*"MJPG")
writer = cv2.VideoWriter(output_path, fourcc, 25,
```

```
 (W, H), True)
# write the output frame to disk
writer.write(output)
# show the output image
cv2.imshow("Output", output)
key = cv2.waitKey(1) & 0xFF
# if the ‘q‘ key was pressed, break from the loop
if key == ord("q"):
break
# release the video
writer.release()
vs.release()
print(video_classification) # print list of how all frames were classified in video
print(lb.classes_[np.argmax(video_classification)]) # print classes associated with
#highest amount of predictions
```

Appendix D. Code: Video to Frames

```
Strips videos into frames. This gave more
control for training/validation/testing


import cv2
import time
import os

def video_to_frames(input_loc, output_loc,name):
    """Function to extract frames from input video file
    and save them as separate frames in an output directory.
    Args:
        input_loc: Input video file.
        output_loc: Output directory to save the frames.
    Returns:
        None
    """
    try:
        os.mkdir(output_loc)
    except OSError:
        pass
    # Log the time
    time_start = time.time()
    # Start capturing the feed
    cap = cv2.VideoCapture(input_loc)
```

```
    # Find the number of frames
    video_length = int(cap.get(cv2.CAP_PROP_FRAME_COUNT)) - 1
    print ("Number of frames: ", video_length)
    count = 0
    print ("Open Video\n")
    # Start converting the video
    while cap.isOpened():
        # Extract the frame
        ret, frame = cap.read()
        if not ret:
            continue
        # Write the results back to output location.
        cv2.imwrite(output_loc + '/' +name+"_%#05d.jpg" % (count+1), frame)
        count = count + 1
        # If there are no more frames left
        if (count > (video_length-1)):
            # Log the time again
            time_end = time.time()
            # Release the feed
            cap.release()
            # Print stats
            print ("Done extracting frames.\n%d frames extracted" % count)
            print ("It took %d seconds forconversion." % (time_end-time_start))
            break

if __name__=="__main__":
```

```
#input_loc = '/UCF101/Videos/ApplyEyeMakeup/v_ApplyEyeMakeup_g01_c01.avi'
#output_loc = '/UCF101/Frames/ApplyEyeMakeup/'
video_path = '/UCF101/Videos10/'
folders = os.listdir(video_path)

for classes in folders:
    video_sub = os.listdir(video_path + str(classes))
    for video in video_sub:

        input_loc = video_path+classes+"/"+video
        output_loc = '/UCF101/Frames10/' + classes
        video_to_frames(input_loc, output_loc,video[:-4])
```

Appendix E. Code: Heatmap

```
Saves a heatmap analysis graph using array rows that come
from video prediction output (must be copied over)


import seaborn as sn
import pandas as pd
import matplotlib.pyplot as plt
import pickle

model_path ='/UCF101/Output/activity2.model'# path to trained serialized model
label_bin = '/UCF101/Output/lb2.pickle'  # path to  label binarizer
output_img_path = '/UCF101/Output/heatmap_1frame.png' # path to output plot

lb = pickle.loads(open(label_bin, "rb").read())

#This array comes from classification of all videos
array = [[19, 9,  0,  2,  0,  0,  0,  0,  0,  0],
 [ 4, 21,  0,  0,  0,  0,  0,  0,  0,  0],
 [ 0,  0, 29,  0,  0,  0,  0,  0,  0,  0],
 [ 2, 21,  1,  7,  0,  0,  0,  0,  0,  0],
 [ 0,  0,  0,  0, 22,  0,  0,  0,  0,  0],
 [ 0,  0,  0,  0,  0, 24,  0,  0,  0,  0],
 [ 0,  0,  6,  0,  0,  0, 26,  0,  1,  0],
 [ 7,  0,  0,  0,  0,  0,  0, 24,  0,  0],
 [ 0,  0, 17,  0,  0,  3,  0,  0,  4,  0],
```

```
 [ 0,  0,  9,  0,  0,  4,  0,  3,  1,  9]]
df_cm = pd.DataFrame(array, index = lb.classes_,
                  columns = lb.classes_)
plt.figure(figsize = (11,12))
sn.heatmap(df_cm, annot=True, linewidths=.5, cmap='Greens',cbar=0)
plt.title("Test Data - 30 Frame Smoothing")
#ax.figure.tight_layout()
plt.savefig(output_img_path)
plt.show()
```

Appendix F. Code: Frames Graph

```
Saves a frames analysis graph using array rows that come
from video prediction output (must be copied over)


import pickle
import pandas as pd
import matplotlib.pyplot as plt

model_path ='/UCF101/Output/activity2.model'# path to trained serialized model
label_bin = '/UCF101/Output/lb2.pickle'  # path to  label binarizer
output_img_path = '/UCF101/Output/MultiFrame.png' # path to output plot

lb = pickle.loads(open(label_bin, "rb").read())

#Array rows come from video prediction output (Copied over)
array =[[0, 0, 0, 0, 3, 0, 0, 0, 65, 28],
        [0, 0, 0, 0, 0, 0, 0, 0, 76, 20],
        [0, 0, 0, 0, 0, 0, 0, 0, 82, 14]]


df_cm = pd.DataFrame(array, columns = lb.classes_,
                  index = ["Frames 1","Frames 25", "Frames 50"])


print(df_cm.iloc[0,])
```

```
plt.subplot(3,1,1)
plt.bar(df_cm.columns,df_cm.iloc[0,])
plt.ylabel("Frames 1")
plt.title("HulaHoop(g21_c01) Smoothing")
plt.gca().spines['top'].set_position(('data',0))
plt.gca().spines['right'].set_position(('data',-10))
plt.xticks([],[])

for i, v in enumerate(df_cm.iloc[0,]):
    plt.text( i-.25, v + 0.4, str(v))
#set_yticklabels(df_cm.columns, minor=False)

plt.subplot(3,1,2)
plt.bar(df_cm.columns,df_cm.iloc[1,])
plt.xticks([],[])

for i, v in enumerate(df_cm.iloc[1,]):
    plt.text( i-.25, v - 0.4, str(v))
plt.ylabel("Frames 25")
plt.gca().spines['top'].set_position(('data',0))
plt.gca().spines['right'].set_position(('data',-10))

plt.subplot(3,1,3)
plt.bar(df_cm.columns,df_cm.iloc[2,])
for i, v in enumerate(df_cm.iloc[2,]):
    plt.text( i-.25, v - 0.4, str(v))
```

```
plt.ylabel("Frames 50")
plt.gca().spines['top'].set_position(('data',0))
plt.gca().spines['right'].set_position(('data',-10))
plt.xticks(df_cm.columns, rotation='vertical')
plt.subplots_adjust(bottom=0.3)

plt.savefig(output_img_path)
plt.show()
```

Appendix G. Code: Sequential Video Frame Predictions

```
Prints two vectors.
1) The first vector is a breakdown of all the frame's classification binned
together for a specified video
2) A vector of all of the frames classification in a sequential order.




import cv2
import pickle
import numpy as np
from collections import deque
from tensorflow.keras.models import load_model




model_path ='/UCF101/Output/activity2.model'# path to trained serialized model
label_bin = '/UCF101/Output/lb2.pickle'  # path to  label binarizer
input =   '/UCF101/UCF-101/PizzaTossing/v_PizzaTossing_g22_c02.avi'#input video
output_path =   '/UCF101/Output/makeup_output.avi'# path to our output video
size =  25#       size of queue for averaging (1-128) 1:no averaging
```

```
# load the trained model and label binarizer from path
model = load_model(model_path)
lb = pickle.loads(open(label_bin, "rb").read())

# initialize the image mean for mean subtraction along with the
# predictions queue
mean = np.array([123.68, 116.779, 103.939][::1], dtype="float32")
Q = deque(maxlen= size)

# initialize the video stream, pointer to output video file, and
# frame dimensions
vs = cv2.VideoCapture(input)
writer = None
(W, H) = (None, None)

video_classification = [0] * len(lb.classes_)
sequential = []

# loop over frames from the video file stream
while True:
# read the next frame from the file
(grabbed, frame) = vs.read()
# if the frame was not grabbed, then we have reached the end
# of the stream
```

```
if not grabbed:
break
# if the frame dimensions are empty, grab them
if W is None or H is None:
(H, W) = frame.shape[:2]

# clone the output frame, then convert it from BGR to RGB
# ordering, resize the frame to a fixed 224x224, and then
# perform mean subtraction
output = frame.copy()
frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
frame = cv2.resize(frame, (224, 224)).astype("float32")
frame -= mean



# make predictions on the frame and then update the predictions
# queue
preds = model.predict(np.expand_dims(frame, axis=0))[0]
Q.append(preds)
# perform prediction averaging over the current history of
# previous predictions
results = np.array(Q).mean(axis=0)
i = np.argmax(results)
label = lb.classes_[i]
video_classification[i] += 1
sequential.append(i)
```

```
# draw the activity on the output frame
text = "activity: {}".format(label)
cv2.putText(output, text, (10, 20), cv2.FONT_HERSHEY_SIMPLEX,
0.75, (0, 255, 0), 2)
# check if the video writer is None
if writer is None:
# initialize our video writer
fourcc = cv2.VideoWriter_fourcc(*"MJPG")
writer = cv2.VideoWriter(output_path, fourcc, 25,
 (W, H), True)
# write the output frame to disk
writer.write(output)
# show the output image
#cv2.imshow("Output", output)
key = cv2.waitKey(1) & 0xFF
# if the ‘q‘ key was pressed, break from the loop
if key == ord("q"):
break
# release the file pointers

writer.release()
vs.release()
print(video_classification) # print list of how all frames were classified in video
print()
print(sequential)
```

```
print()
#print(lb.classes_[np.argmax(video_classification)]) # print classes associated
#with highest  amount of predictions
```

Appendix H. Code: YOLO

All things YOLO, as well as training and testing source code is provided at: `https://github.com/AlexeyAB/darknet`.

Bibliography

1. Trevor J. Bihl, Joe Schoenbeck, Daniel Steeneck, and Jeremy Jordan. Easy and efficient hyperparameter optimization to address some artificial intelligence "ilities". In *53rd Hawaii International Conference on System Sciences, HICSS 2020, Maui, Hawaii, USA, January 7-10, 2020*, pages 1–10. ScholarSpace, 2020.

2. Emmie Swize. Bayesian augmentation of convolutional neural network - long short term memory for video classification with uncertainty measures. Master's thesis, Air Force Institute of Technology, Wright-Patterson Air Force Base, Ohio, 3 2021.

3. Giacomo Deodato, Christopher Ball, and Xian Zhang. Bayesian neural networks for cellular image classification and uncertainty analysis. *bioRxiv*, 2020.

4. Yann Lecun, Leon Bottou, Y. Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86:2278 – 2324, 12 1998.

5. Yarin Gal. *Uncertainty in Deep Learning*. PhD thesis, University of Cambridge, Gonville and Caius College, 9 2016.

6. John E. Ball, Derek T. Anderson, and Chee Seng Chan. A comprehensive survey of deep learning in remote sensing: Theories, tools and challenges for the community. *CoRR*, abs/1709.00308, 2017.

7. Djemel Ziou and Salvatore Tabbone. Edge detection techniques - an overview. *INTERNATIONAL JOURNAL OF PATTERN RECOGNITION AND IMAGE ANALYSIS*, 8:537–559, 1998.

8. Sankar Pal, Anima Pramanik, Jhareswar Maiti, and Pabitra Mitra. Deep learning in multi-object detection and tracking: state of the art. *Applied Intelligence*, 51, 09 2021.

9. Kumar Shridhar, Felix Laumann, and Marcus Liwicki. *A Comprehensive guide to Bayesian Convolutional Neural Network with Variational Inference*. PhD thesis, 12 2018.

10. Warren Mcculloch and Walter Pitts. A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:127–147, 1943.

11. Aurelien Geron. *Hands-on machine learning with Scikit-Learn and TensorFlow : concepts, tools, and techniques to build intelligent systems*. O'Reilly Media, Sebastopol, CA, 2019.

12. Kunihiko Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36:193–202, 1980.

13. Yarin Gal and Zoubin Ghahramani. Dropout as a bayesian approximation: Representing model uncertainty in deep learning, 2016.

14. Alex Kendall and Roberto Cipolla. Modelling uncertainty in deep learning for camera relocalization, 2016.

15. Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, page 1097–1105, Red Hook, NY, USA, 2012. Curran Associates Inc.

16. Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014.

17. Charles Blundell, Julien Cornebise, Koray Kavukcuoglu, and Daan Wierstra. Weight uncertainty in neural networks, 2015.

18. Rafael Padilla, Sergio L. Netto, and Eduardo A. B. da Silva. A survey on performance metrics for object-detection algorithms. In *2020 International Conference on Systems, Signals and Image Processing (IWSSIP)*, pages 237–242, 2020.

19. Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. Yolov4: Optimal speed and accuracy of object detection, 2020.

20. Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

21. Vidushi Meel. What is object tracking? - an introduction, 2021.

22. Emmie Swize, Lance Champagne, Bruce Cox, and Trevor Bihl. Bayesian augmentation of deep learning to improve video classification. *Hawaii International Conference on System Sciences*, 2022.

23. Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2015.

24. Bichen Wu, Alvin Wan, Forrest Iandola, Peter H. Jin, and Kurt Keutzer. Squeezedet: Unified, small, low power fully convolutional neural networks for real-time object detection for autonomous driving, 2019.

25. Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and ¡0.5mb model size, 2016.

26. Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. *CoRR*, abs/1506.02640, 2015.

27. Joseph Redmon and Ali Farhadi. Yolo9000: Better, faster, stronger, 2016.

28. Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement, 2018.

29. Joseph Redmon. Darknet: Open source neural networks in c. `http://pjreddie.com/darknet/`, 2013–2016.

30. NVIDIA Developer. Cuda - gpu accelerated computing with python, Aug 2021.

31. Khurram Soomro, Amir Roshan Zamir, and Mubarak Shah. UCF101: A dataset of 101 human actions classes from videos in the wild. *CoRR*, abs/1212.0402, 2012.

32. Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

33. Tsung-Yi Lin, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hays, Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, and Piotr Dollár. Microsoft coco: Common objects in context, 2015.

34. NVIDIA Developer. Nvidia cudnn, Jun 2021.

35. Addie Ira Borja Parico and Tofael Ahamed. Real time pear fruit detection and counting using yolov4 models and deep sort. *Sensors*, 21(14), 2021.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
25 Mar 2022	Master's Thesis	Sept 2020 — Mar 2022

4. TITLE AND SUBTITLE

Smoothing of Convolutional Neural Network Classifications

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Drumm, Glen, R 1st Lt, USAF

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
Wright-Patterson AFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT-ENS-MS-22-M-122

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Trevor Bihl, DAF, DR-III, PhD
Sensors Directorate Air Force Research Laboratory
2242 Avionics Circle
Wright-Patterson AFB OH 45431
trevor.bihl.2@afresearchlab.com

10. SPONSOR/MONITOR'S ACRONYM(S)

AFRL/RY

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A:
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Smoothing convolutional neural networks is investigated. When intermittent and random false predictions happen, a technique of average smoothing is applied to smooth out the incorrect predictions. While a simple problem environment shows proof of concept, obstacles remain for applying such a technique to a more operationally complex problem.

15. SUBJECT TERMS

artificial neural network (ANN), convolutional neural network (CNN), deep learning (DL), computer vision (CV), machine learning (ML), artificial intelligence (AI)

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Dr. Lance Champagne, AFIT/ENS
U	U	U	UU	89	19b. TELEPHONE NUMBER *(include area code)* (937)255-3636x9999;Lance.Champagne@afit.edu