Big data analytics in high-throughput phenotyping

by

Chaney L. Courtney

B.S., Kansas State University, 2014
M.S., Kansas State University, 2017

AN ABSTRACT OF A DISSERTATION

submitted in partial fulfillment of the requirements for the degree

DOCTOR OF PHILOSOPHY

Department of Computer Science
Carl R. Ice College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2020

Abstract

As the global population rises, advancements in plant diversity and crop yield is necessary for resource stability and nutritional security. In the next thirty years, the global population will pass 9 billion. Genetic advancements have become inexpensive and widely available to address this issue; however, phenotypic acquisition development has stagnated. Plant breeding programs have begun to support efforts in data mining, computer vision, and graphics to alleviate the gap from genetic advancements.

This dissertation creates a bridge between computer vision research and phenotyping by designing and analyzing various deep neural networks for concrete applications while presenting new and novel approaches. The significant contributions are research advancements to the current state-of-the-art in mobile high-throughput phenotyping (HTP), which promotes more efficient plant science workflow tasks. Novel tools and utilities created for automatic code generation, maintenance, and source translation are featured. Promoted tools replace boiler-plate segments and redundant tasks. Finally, this research investigates various state-of-the-art deep neural network architectures to derive methods for object identification and enumeration.

Seed kernel counting is a crucial task in the plant research workflow. This dissertation explains techniques and tools for generating data to scale training. New dataset creation methodologies are debuted and aim to replace the classical approach to labeling data. Although HTP is a general topic, this research focuses on various grains and plant-seed phenotypes. Applying deep neural networks to seed kernels for classification and object detection is a relatively new topic. This research uses a novel open-source dataset that supports future architectures for detecting kernels. State-of-the-art pre-trained regional convolutional neural networks (RCNN) perform poorly on seeds. The proposed counting architectures outperform the

models above by focusing on learning a labeled integer count rather than anchor points for localization. Concurrently, pre-trained models on the seed dataset, a composition of geometrically primitive-like objects, boasts improvements to evaluation metrics in comparison to the Common Object in Context (COCO) dataset. A widely accepted problem in image processing is the segmentation of foreground objects from the background. This dissertation shows that state-of-the-art regional convolutional neural networks (RCNN) perform poorly in cases where foreground objects are similar to the background. Instead, transfer learning leverages salient features and boosts performance on noisy background datasets.

The accumulation of new ideas and evidence of growth for mobile computer vision surmise a bright future for data-acquisition in various fields of HTP. The results obtained provide horizons and a solid foundation for future research to stabilize and continue the growth of phenotypic acquisition and crop yield.

Big data analytics in high-throughput phenotyping

by

Chaney L. Courtney

B.S., Kansas State University, 2014
M.S., Kansas State University, 2017

A DISSERTATION

submitted in partial fulfillment of the requirements for the degree

DOCTOR OF PHILOSOPHY

Department of Computer Science
Carl R. Ice College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2020

Approved by:

Major Professor
Mitchell Neilsen

Copyright

Abstract

As the global population rises, advancements in plant diversity and crop yield is necessary for resource stability and nutritional security. In the next thirty years, the global population will pass 9 billion. Genetic advancements have become inexpensive and widely available to address this issue; however, phenotypic acquisition development has stagnated. Plant breeding programs have begun to support efforts in data mining, computer vision, and graphics to alleviate the gap from genetic advancements.

This dissertation creates a bridge between computer vision research and phenotyping by designing and analyzing various deep neural networks for concrete applications while presenting new and novel approaches. The significant contributions are research advancements to the current state-of-the-art in mobile high-throughput phenotyping (HTP), which promotes more efficient plant science workflow tasks. Novel tools and utilities created for automatic code generation, maintenance, and source translation are featured. Promoted tools replace boiler-plate segments and redundant tasks. Finally, this research investigates various state-of-the-art deep neural network architectures to derive methods for object identification and enumeration.

Seed kernel counting is a crucial task in the plant research workflow. This dissertation explains techniques and tools for generating data to scale training. New dataset creation methodologies are debuted and aim to replace the classical approach to labeling data. Although HTP is a general topic, this research focuses on various grains and plant-seed phenotypes. Applying deep neural networks to seed kernels for classification and object detection is a relatively new topic. This research uses a novel open-source dataset that supports future architectures for detecting kernels. State-of-the-art pre-trained regional convolutional neural networks (RCNN) perform poorly on seeds. The proposed counting architectures outperform the

models above by focusing on learning a labeled integer count rather than anchor points for localization. Concurrently, pre-trained models on the seed dataset, a composition of geometrically primitive-like objects, boasts improvements to evaluation metrics in comparison to the Common Object in Context (COCO) dataset. A widely accepted problem in image processing is the segmentation of foreground objects from the background. This dissertation shows that state-of-the-art regional convolutional neural networks (RCNN) perform poorly in cases where foreground objects are similar to the background. Instead, transfer learning leverages salient features and boosts performance on noisy background datasets.

The accumulation of new ideas and evidence of growth for mobile computer vision surmise a bright future for data-acquisition in various fields of HTP. The results obtained provide horizons and a solid foundation for future research to stabilize and continue the growth of phenotypic acquisition and crop yield.

Table of Contents

List of Figures

List of Tables

Acknowledgments

This material is based upon work supported by the National Science Foundation (NSF) under NSF-Basic Research to Enable Agricultural Development (BREAD) Grant No. 1543958. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

This research is also funded by the Bill and Melinda Gates Foundation.

The ideas and design of the various applications for plant based high-throughput phenotyping came from collaboration with Mitchell Neilsen, Jessie Poland, Trevor Rife, Michael Gore, Lukas Mueller, Jenna Hershberger, Solomon Nsumba, Peter Selby, Peter Rosario, Venkat Margapuri, Shanshan Wu, Siddharth Amaravadi, Friday James, Joydeep Mitra, and Guillaume Jean Bauchet.

Hardware described in Chapter 4 Section Population of Kernel Sample and Data Source came from inspiration from Mitchell Neilsen, Masaaki Mizuno, Paul Armstrong, Dan Brabec, Siddharth Amaravadi, and Dan Wagner.

Deep neural network architectures and algorithms were also inspired by Mitchell Neilsen, Masaaki Mizuno, Dave Schmidt, Torben Amtoft, RDoina Caragea, Rodney Howell, Jessie Poland, Trevor Rife, Chendi Chao, Venkat Margapuri, Greg Erickson and Umar Gaffar.

Dedication

To my grandpa who inspired me to be a scientist. To my professors who are friends, and my family who showed me patience and unending support.

Preface

All research, including fabrication of tools used for experiments, was conducted in the Cyber Physical Systems Laboratory (CPSL) within the Computer Science Department at Kansas State University.

A version of Chapter 2 Section Intercross was published in Proceedings of 32nd International Conference on Computer Applications in Industry and Engineering.

Chapter 1 - Introduction

Applications in high-throughput phenotyping include both generic and specific tools for solving tasks typically related to data acquisition. *Data acquisition* is the memoization of actions, events, and observations. Classical data acquisition has been mainly a manual effort of writing information into a booklet; for field studies, the information includes statistics about plot-level details. A set of fundamental phenotyping tools developed for specific tasks in plant breeding improves the breeding work cycle [1]. Poland et al., encapsulate the process workflow of a plant breeder within three main categories: sampling, crossing, and evaluation. Field Book was one of the first Android Applications created by Poland/Rife for field-data-acquisition. Field Book's primary purpose is to eliminate the need for pen-and-paper methods for tracking plot specific phenotypes. A collection of related applications called PhenoApps has been developed at Kansas State University through collaboration between the Poland Wheat Genetics Laboratory, the Cyber Physical Systems Laboratory in the Computer Science Department, and the Gore Laboratory at Cornell University. A subset of these applications is discussed in this dissertation; a descriptive functionality is given, along with a motivation for each tool. Field Book has proven success on the Google Play Store. These applications are opening the rural and developmental world up to new tools of free access. Supporting grants for these projects are funded by the National Science Foundation and the Bill and Melinda Gates Foundation. The NSF Basic Research to Enable Agricultural Development (BREAD) project is helping small-holder farm development by funding the creation of these applications and has supported various workshops and inventory distribution for these tasks. Research and development for this dissertation is a product of the work at the Cyber Physical Systems Laboratory (CPSL) at Kansas State

University with close collaboration with Cornell University, Boyce Thompson Scott Institute, and the International Potato Center in Lima, Peru.

With the rise in cheap mobile devices, a new subcategory that we call *mobile high-throughput phenotyping* emerges. Lab technicians and breeders can now replace their spreadsheet processor-based workflow with free-of-charge applications, without the need for a desktop computer. There is an innate difference in the processing power of computers and mobile devices due to architectural differences, with the former being more powerful and power-consuming than the latter. Central processing unit (CPU) intensive tasks such as image processing perform fathoms greater on a mid-range desktop than most mobile devices. Mobility and power-consumption is a worthy trade-off to the lack of CPU processing compared to average computers. As mobile devices advance, we can perform more powerful operations designed for image processing and general data acquisition. There is a plethora of devices available for mobile phones, including printers, bar-code scanners, and even scales with sub-gram resolution. The usages of these devices are plentiful and essential for fieldwork in plant research. As more plant research centers populate plots with bar-code identifiers, it is essential to allow users to scan these tags for ease-of-use in their research. There is a general aspect of the usages of these applications without the overlooking topic of phenotyping. Working with mobile devices for such tasks allows the user to harness the Internet or 3G networks for cloud-based data storage and processing. The PhenoApps Application Prospector is one such application that looks at analyzing near-infrared-spectrometry on the cloud to classify traits of individual plant samples. Because the computational cost of evaluating the linear regression of these models is typically too much for a low-end mobile device, the use of clusters helps reduce training time. Therefore, as some applications utilize the ability of the Internet, others avoid it as a necessity. Most rural

farms and some plant-research facilities have little to no Internet and thus require pre-built Android packages (APK). While offline, data acquisition tasks occur. The *task of fieldwork* is offline work; however, when completed, there is the option of utilizing the Internet to upload results.

Implementing state-of-the-art algorithms into applications requires not only knowledge of the new algorithms, but expertise in building applications. An element of this dissertation is the implementation of tools and libraries that can ease this burden. Code generative plugins implemented using Gradle are proposed, along with design architectures for using Android Jetpack libraries. Along with libraries and code generative tasks for Android development, another objective of this dissertation is to discuss and create tools for generating annotations for training deep neural network models. With the rise in deep learning, there is a need to handle and generate large datasets. General object detection is an undecidable task [2]. This dissertation introduces a process to automate annotations, which is a bottleneck (or expensive task) for creating datasets to train such models. The annotations generator does not act as a universal object annotator. However, it requires some supervised task of tuning for its data. Specific project setups can improve the ease of creating object detection scripts. This dissertation includes headless and user-interface driven tools to support multiple tasks. A specified script for a particular object detection task can then power the automatic generator script, which parallelizes the annotating process.

A generalized object detector can be created and is discussed later for seed kernels based on a deep neural network trained on datasets generated from this methodology. The automatic annotations generator script creates labeled datasets where the labels include bounding boxes, images, area for each contour, and approximated contour arrays. TensorFlow is used for training,

evaluating, and the creation of various models. The use of TensorFlow allows for creating TensorFlow-Lite models, which optimize models for mobile devices. Although TensorFlow-Lite doesn't currently have the same capabilities as full-fledged TensorFlow models, this dissertation experiments with its use and searches for viable options on mobile devices. A benefit of using such deep neural network models is the ability to checkpoint pre-trained models. The general methodology for these experiments is to train networks on the local cluster at Kansas State University, Beocat, checkpoint the model, and test the model in different environments. Models include linear-regression based classification models and autoencoder models. Classification models do not predict localization or contour masks but output a confidence interval for the classification of such seeds. Currently, there are no defined usages for this type of model, as most applicable models for this research require localization and masking. Another such model is used to locate kernels; these models include SSD, FastRCNN, and FasterRCNN, which place bounding boxes around their classifications. MaskRCNN uses mask contour data to measure the area and perimeter data for seeds. However, an optimized version for mobile devices does not exist.

The open-source nature of these projects precludes the opportunity to extend this software. If future researchers wish to create features or forks of the current projects and have little knowledge of the Android ecosystem, it may be difficult to make simple changes. Work has simplified the process of coding for Android applications. However, there is always a need to have a basic understanding of programming languages and the Android operating system. As Kotlin has become more popular and an evident replacement to Java, it is the recommended programming language of choice. Two Gradle tasks begin the process of alleviating boiler-plate code, and a Java to Kotlin translator is also available. Android Studio has a built-in Java to

Kotlin translator that was developed in parallel to this research. The provided translator focuses on null safety and reduces run-time errors that are common in Android Studio's translations.

Various data flow analyses determine the initialization of variables. The Android Studio translation globally initializes variables as null when this is not necessary. This translator aims at creating null safe code by replacing null initializations with lazy initializers and lateinit variables. Lazy initializers require developers to write initialization code post-translation. Using data flow analyses such as liveness, the mutability of variables can be determined and, in this way, decide initialization strategies. Lateinit variables have non-null static types but are initialized on their first line of usage instead of at declaration time. Lazily evaluated objects are useful and predominately used for view objects. Typically, when a view object in the Android environment initiates, it is declared using the 'findViewById' function, which cast into an object of the view's type. These calls can be wrapped into top-level lazy declarations that are initialized when the value is called. These fundamental Kotlin syntaxes are used to create a more idiomatic code translation in comparison to the Java to Kotlin translator in Android Studio, which is more Java-like.

Gradle is a popular build language for Android applications. It has a flexible application programming interface (API) for building plugins and build tasks. The Gradle API is perfect for creating generative code tasks that operate on source files. One featured plugin is the Auto Shared Preferences plugin. This plugin aims at reducing code in Kotlin by dynamically creating a class populated with fields collected from a local preference file. Preferences files define the Shared Preference settings' key-value pair mapping. Another critical benefit to code generation is the ability to suppress and alleviate bug production in development. Developing succinct and

safe code generative tasks reduces the number of bugs in comparison to typically hand developed code.

The following dissertation is an accumulation of the research efforts, ideas, motivations, and future goals of improving the status quo of high-throughput phenotyping big data tasks. As the volume and velocity of data increases, including crop images from uncrewed aerial vehicles (UAV), genomic processing, and near-infrared spectroscopy (NIRS) data, it is vital to have a fundamental change in the way fieldwork is managed. The data quality of such tasks is described to ensure their transitivity and reproducibility. For example, the usage and extension of these tools and models may require similar experimental setup, such as the addition of new training data on a specific seed kernel which has not been previously trained upon. The data acquisition of such information will need to remain within the bounds of what is defined. The reason being is the complicated nature of contour clustering and identification. If data taken has a noisy background or is within a busy lighting scenario, the objects may be difficult to track using the given software.

Therefore, the experimental setup and *data quality* is a common element of this dissertation, which aims at settling a foundation for big data tasks in high-throughput phenotyping. Therefore, given the previously stated knowledge, this dissertation expects to solve three overarching goals, which include new research tasks that contribute to the society of computer scientists. First, the production of a novel dataset for sorghum, rice, rough rice, wheat, soybean, and canola. Secondly, the implementation of a semi-automated contour annotations generator for training deep neural networks. Finally, the evaluation and observations of various seed-based datasets using single-shot detection Mobilenet (SSDMobilenet), a density-based seed counter, and an autoencoder seed counter.

Furthermore, each implementation has an open-source repository where reviewers may download the code, dataset, or application necessary to reproduce or extend these results. This dissertation's contributions are not limited to the above-defined goals. They include several open-source applications, libraries, and algorithms.

Chapter 2 - Applications in High-throughput Phenotyping

Motivation

The Cyber Physical Systems Laboratory (CPSL) at Kansas State University has a focus on developing high-throughput phenotyping applications for rural agricultural communities. To improve research workflow and development, the CPSL works closely with PhenoApps, an open-source organization based in the Plant Science department at Throckmorton Hall on the KSU campus. CPSL has traveled to Hyderabad, India, for a workshop at the International Crops Research Institute for the Semi-Arid Tropics (ICRISAT). Here the CPSL worked on an Android application, Verify, for various bar-code tracking related tasks. PhenoApps has had multiple hackathons based at Cornell University in Ithaca, New York. CPSL worked on creating a new application, Intercross at PhenoApp's first hackathon, by working closely with plant science researchers at Cornell and the Boyce Thompson Institute. Intercross was later introduced at the International Institute of Tropical Agriculture (IITA) in Nigeria and the Centro Internacional del Papas in Lima, Peru. The CPSL was able to personally visit CIP for close work with its staff and beta-test against an estimate of one-hundred users. At the same time, members of the Boyce Thompson Institute introduced Intercross to plant researchers at IITA. Intercross is a submission-based and motivated on the workflow at CIP and IITA and proposes the use of mobile devices for their experiments. Verify is now a commonly used application within ICRISAT and other plant research facilities with thousands of device installations.

Verify

Relational databases have been a fundamental piece of software since the need to store large amounts of data in a structured way has arisen [3]. Structured query language (SQL) is a leading query language for designing the schema and accessing data within the tables of a relational database. Whether they be web, mobile, or any other type of application with a front-end and back-end, databases are essential to data management. It is a common understanding that a database accompanies most web applications and mobile applications. The need to store information about users or work is fundamental for any basic algorithm designed for an application. Verify is one such example where the front-end of the application is highly tied to back-end processing using an SQLite database. Android, by default, allows developers to utilize a 'lighter' version of SQL within applications, which allows users to manage data independently of the system and other applications. SQLite is "the most used database engine in the world.". It is a "small, fast, self-contained, high-reliability, full-featured, SQL database engine." [4]. SQLite ensures atomicity, consistency, isolation, and durability (ACID) fundamentals [5].

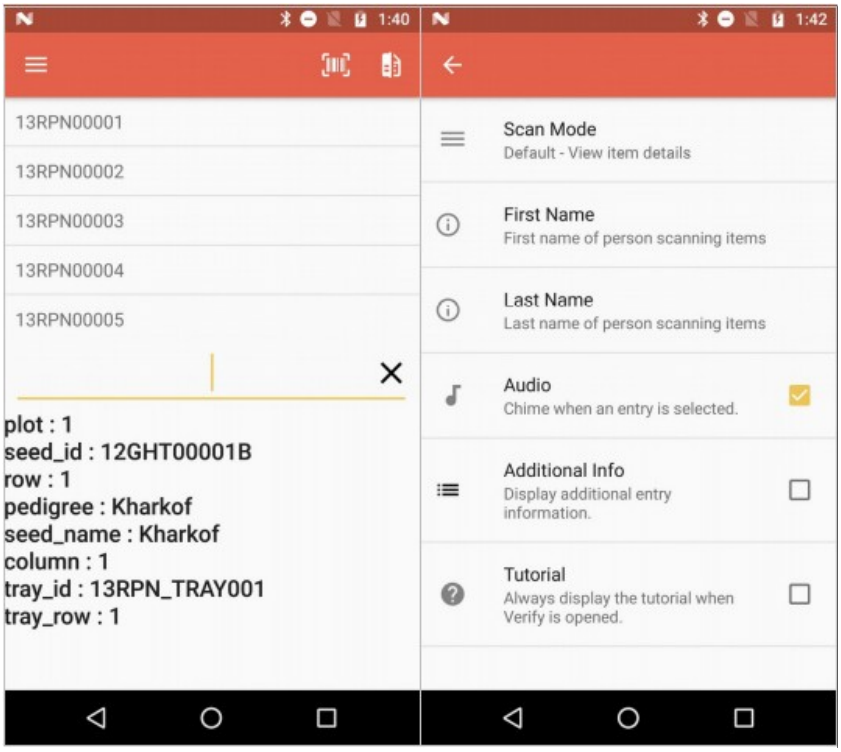

Figure 1: Examples of fragments from Verify.

Verify's primary purpose is to track collected samples within a database. Verify is an Android application that has different modes and functions that can be applied to the data and exported for iterative use. While Verify's purpose is relatively generic and is for many different tasks, the primary purpose is for plant breeders to track specimens within a greenhouse. Plant breeders that PhenoApps has worked with closely have been tracking samples by hand, which is a highly manual and worker-intensive task. Verify alleviates the process by allowing each worker to have a device that can scan bar-coded samples. Greenhouses populated with samples typically have a unique bar-code per sample. The workflow of Verify is to import a list of samples and use the various modes to mutate the database. Figure 1 shows the most common and basic usage of Verify, which is the ability to select an imported sample and view meta-data in the SQLite database. Users can optionally select a row by clicking an item in the enumerated list or

type a given name within the text view, which is below the list. The following are the defined modes that are within Verify: default, match, filter, color, and pair. All of which are described, along with performance analysis, time-complexity analysis, and space complexity analysis within Chapter 1 of 'Open source application development for phenotypic data acquisition.'

Figure 2: The bar-code matching feature in Verify.

Bar-code scanning happens in two different ways. The user may connect an external device; when an identifier is scanned, the decoded string registers into the text view for selection. Users may use their camera to capture the bar-code, which is decoded into a string and placed in the main text view. The API used for on-device bar-code scanning is very efficient and, by default, over scans items, by scanning them more than once.

Verify utilizes multiple novel algorithms that augment daily high-throughput phenotyping tasks. A suppressive algorithm nullifies the multitude of scans for ease to the user. Verify requires Android devices to have at least version 4.1 and currently has over 10,000 installs on the Google Play Store. Currently, there are no other applications for bar-code based acquisition of plant samples. With the previous features, Verify is a significant improvement and contribution to mobile high-throughput phenotyping. Many basic features of Verify, including handling the Android file system, are freely available and should guide new developers. Many features of Verify are embedded as fragments. Each fragment has its functionality and can be used within other applications with little interoperability overhead. Verify sets a new path for bar-coding samples by speeding up bar-code recognition and utilizing an easy user interface. Intercross, another application discussed in this dissertation, also utilizes Verify's bar-coding fragment. All application code and resources are available freely; the link is available in Appendix A.

Survey

With the need for precise latitude-longitude data acquisition on mobile phones, we developed a new PhenoApp, called Survey. Survey uses the default SQLite database on Android phones to store location data for plot-based phenotypic data acquisition. Fieldworkers can navigate fields of plots and not only determine their exact location, but they can also query which plot they are closest to or facing. Survey uses the Haversine method to calculate and base localized distances on exact geodesic paths [6]. With the release of the global navigation satellite system (GNSS) providers for Android, Survey can read and parse real-time GNSS messages [7]. Using GNSS messages, Survey can access a multitude of satellite constellation features, including the number of satellites the device is communicating with and their locations. The default accuracy of mobile devices is known and only accurate within 10 meters [8]. A novel feature of Survey is the ability to connect to an external global positioning system (GPS) provider for a higher level of accuracy and precision. Although the implementation is generalized for any such external device, PhenoApps has tested the Tersus GPS receiver along with the Emlid Reach. PhenoApps concluded that the Tersus lacked not only universal asynchronous receiver-transmitter (UART) capabilities, but their technical support staff was negligent. Although the Emlid Reach device has similar issues with technical support staff, it turns out to be more robust and easier to use for testing purposes. The Emlid Reach is capable of sending real-time kinematic (RTK) corrected National Marine Electronics Association (NMEA) messages from a base to a rover, which can pair with mobile devices. The Emlid Reach's initialization creates a web server hot-spot that mobile devices with a wireless module may

connect to and configure their rover or base [9]. After initialization is complete, Survey uses Android's JetPack architecture for viewing data in real-time.

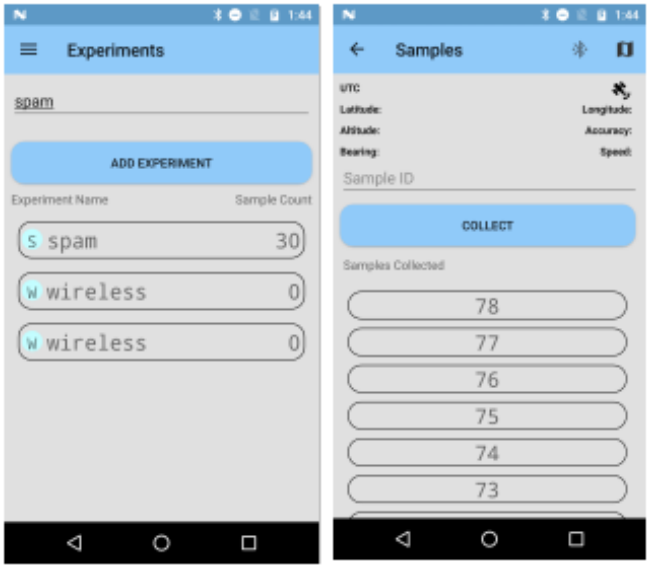

Figure 3: Examples of fragments from Survey.

Survey is not yet released on any online market and has gone through two significant version changes. Initially, Survey attempted to create a programmable interface for the Tersus GPS; this was soon replaced by Survey V2, which uses the reasonably new Android JetPack architecture libraries [10]. A field survey verifies the accuracy of such devices on one of Kansas State University's research farming plots. Two Emlid Reach RS devices are used for the survey. The base device was placed on a tripod and accumulated base coordinate data for five minutes before the survey. The base device outputs corrections data at a frequency of 868.0MHz, with an air data rate of 18.23 kb/s and an output power of 12dBm on its long-range radio (LoRa) module. For RTK settings, the elevation mask angle was at 15 degrees, and the SNR mask was 35, max horizontal and vertical acceleration was 1 m/s^2. The field study is named Single Plant

Associated Mapping (SPAM). It is an accumulation of 30 data points to test the accuracy of Emlid Reach. Fixed point RTK coordinates have 0.5cm accuracy. The rover can either be in single precision, float precision, or fixed precision mode in order of increasing accuracy, respectively.

In the experiment, there existed 2'x2' in shape with four plant samples per plot. A subset of the plots was chosen to visualize if the output coordinates could informally distinguish between different plant samples within each plot. The experiment was split into two different sections to test the amount of time needed per coordinate acquisition and how it would affect the accuracy of the results. The first section acquired fixed points by correcting rover coordinates with the base. Although this was at times not possible due to weather conditions. Each sample was a collection of fixed point data for ten seconds. If the fix was not possible, then the point was taken with float coordinates. The second section attempted a faster acquisition of points by collecting the data immediately after the rover survey pole was planted into the ground; most points achieved float. Some samples reached a fixed point within five seconds. The output geographical JavaScript Object Notation (geoJSON) files are available in Appendix A.

The development of Survey contributes multiple algorithms for state-of-the-art high-throughput phenotyping. Survey's impact zone algorithm utilizes geographic data coupled with an on-device accelerometer and magnetometer to visualize the direction and location of a user and their surveyed points. This computation is novel and has vast usability for Android developers. Along with the impact zone algorithm, a parameterized location listener is introduced from Survey. The parameterized location listener is an alternative PhenoApps Android library, which creates further fine-tuning on Android's default GPS class.

RTCM3 Message	Constellation	Frequency Updated
1002	GPS L1 Observations	1Hz
1006	ARP Station Coordinates	0.1Hz
1008	Antenna type	1Hz
1010	GLONASS L1 observations	1Hz
1019	GPS Ephemeris	1Hz
1020	GLONASS Ephemeris	1Hz
1097	GALILEO	1Hz
1107	SBAS	1Hz
1117	QZSS	1Hz

Table 1: RTCM3 message update parameters for the SPAM survey.

Furthermore, there are no modern GNSS parsers for Android development. The development of Survey has spawned many of the previously mentioned features, which can all be used independently for other applications. This contribution of libraries and algorithms is a significant contribution of Survey, which has been a fundamental aspect of research in high-throughput phenotyping. Readers and developers are encouraged to explore Survey's repository in Appendix A and extend these algorithms and libraries for other new applications.

Intercross

Intercross is an Android application for tracking hierarchical crosses in a database. The motivation for this application comes from two different plant research facilities. The International Institute of Tropical Agriculture uses hand-written labels for tracking their crosses and requires a modern solution to create redundancy and simplify the plant research workflow. A cross is the child of a set of parent plants. *Bi-parental* plants have a unique male and female parent. *Open-pollinated* plants have two female parents. *Self-pollinated* plants have a single female parent. Finally, a *poly-cross* has a set of male parents with one female parent.

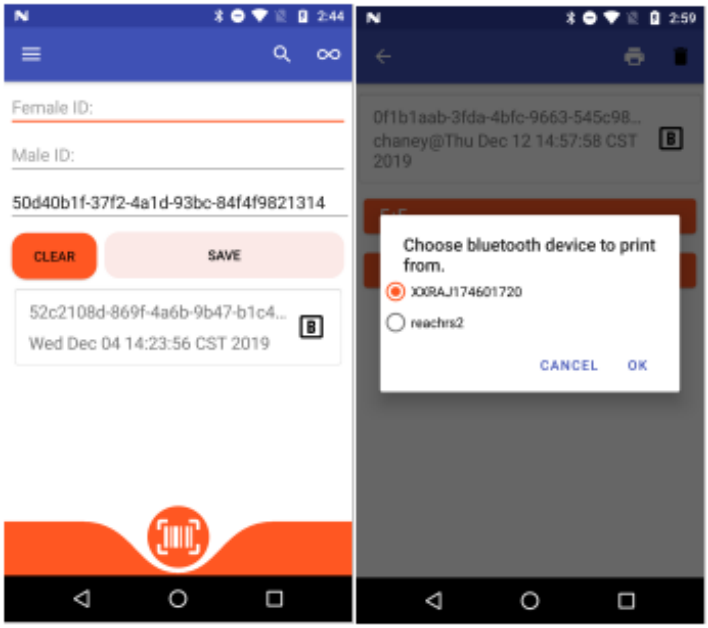

Figure 4: The main page and connection prompt in Intercross.

Intercross uses these classifications at the data-entry time to populate a view-able list to the user, which describes the time of crossing, the user who made the cross, and the pollination classification. Along with IITA, the Centro Internacional del Papas is another group that needed a new solution for crossing plants. Previously, CIP was using Microsoft Motorola phones, which have embedded bar-code scanners. The research group at CIP claims that the software was difficult to change on these devices when they needed certain features. Along with this, the devices are expensive and should be replaced with cheaper Android devices; furthermore, Microsoft no longer produces or supports Windows mobile devices. As many people in the world use Android devices, they are more familiar with their universal design and usages. Android devices also contain modern components, whereas the Motorola Windows mobile devices were using resistive-touch screens, which require a physical press on the screen. These resistive-touch screens are less accurate than modern Android phone screens.

The usage of Intercross involves three necessary steps: acquiring, querying, and printing data. The user has many different ways to acquire data. Most naively, the user may input data by typing a male, female, and cross name manually into three text boxes shown on the main page. Intercross also has the option to scan data using the rear camera on the device. The scanning mode has three different options, search, single, and continuous. The search mode brings up a cross-specific page for the bar-code. Single-mode scans a bar-code and places the scanned id onto the next relevant text view. For example, if the male and female text view is filled previously, the single scan fills in the cross text view. Finally, the continuous mode takes three consecutively delayed pictures, one for male, female, and cross. It enters this into the database without transitioning back to the homepage. This cycle continues until the user manually returns. The camera scanning uses Android SharedViewModels to send data between fragments within

the application. SharedViewModels use the observer pattern; in this case, the camera fragment submits bar-code data to the model. At the same time, the main page listens for changes to the model and populates the user-interface (UI) with the bar-code data.

Intercross was made for ease of data entry. There are individual automatic contextual switches when entering data. For example, when a user enters data into the female text view and submits enter, the text view automatically transitions to the male text view. The user input acts as a finite state automaton, where each state of the machine is either in empty, first, second, third, and filled mode. Where each mode correlates to the text view, which is currently filled-up to, user settings that allow for flexibility of data entry do change this model. Users have the option to enter data for males first, whereas Intercross complies with female-first data entry by default. Another such setting is the ability to allow for blank-males, where male data skip entirely. Meta-data entry is also a non-default option for each cross, where the user can track the count of seeds, flowers, and fruits of a plant. With this setting, the cross-entry page prompts the user with information about the sample. Users have the option to fill cross entry text views automatically. There is an option to automatically generate cross id's using universally unique identifier (UUID) strings or patterns. Pattern generation in Intercross allows the user to define a simple regex-type pattern where they use a prefix, suffix, and middle number. The middle number can have a preset value or start at zero and has the option for padding at any arbitrary length. As cross ids are needed, the middle number increments for a unique id. Intercross also has the option for audio notifications; these notifications occur when a new cross enters into the database, and if a wish-list fulfills.

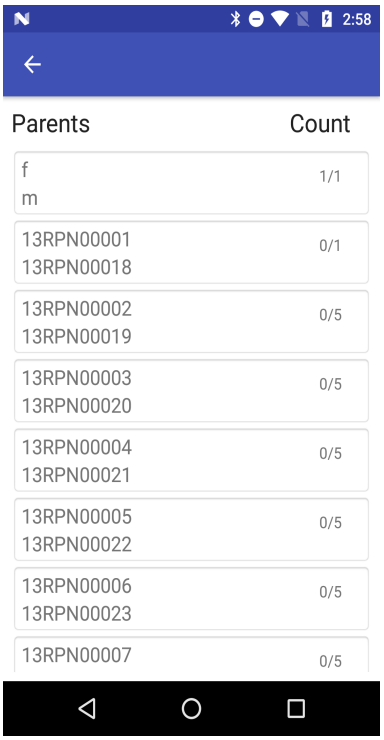

Figure 5: A wish-list summary in Intercross.

The main features of Intercross include wish-lits creation, parental summaries, cross counts, a pollen manager, and an import method for initializing parental samples. The summary page displays a list of parental pairs and a number for each, representing the unique crosses for that pair. The user can go to the cross-entry specific page for each child by clicking the parental pairs. The UI creates a list of children, which are enumerated using their buttons. An import file defines wish-lists with the following headers: fid, mid, fname, mname, type, min, max. Fid and mid are the bar-codes for the female and male parent. Fname and mname are text aliases for each bar-code. Type, min, and max are specific to wish=list imports. A wish-list defines a list of integer values for certain possible events in Intercross.

The events include cross entries, fruits, and seeds. When a sure cross id submits to the database, its parents are checked if they exist within the wish-list table. If the parental pair exists,

then a counter is incremented. The user is then notified when a minimum and a maximum number of these events are counted. Pollen bag management defines the functionality for dealing with poly crosses, which are cross entries that have a set of males with a magnitude greater than one. Poly crosses are managed similarly to Unix groups; the user interacts with two separate fragments.

The first fragment defines a list of groups. The user enters a text name for the group, which generates a bar-code-based on a UUID. A table updates with the group name and the randomly generated UUID. The user can click a row in the list of group names to move to the pollen fragment. The pollen fragment screen allows the user to enter data manually or via the camera. These populate the set of males for a given group. This page also has a print function that prints a bar-code label for the group. The group label can then be used instead of a typical parental scan. When the cross entries export to a file, the males contained with the group set are semi-colon delimited. Finally, for bar-code based plant research, there must be a way to initialize a plot or greenhouse with labels. Otherwise, users would be constrained to manual entry. The parental import feature uses the headers defined above to import a list of parents. On this fragment, males and females separate into two lists. The user can highlight items to select each parent. When the user presses the print button, a batched job prints each parental label. Because these initial parents are not crosses, they are not tracked within the main table of Intercross, and they hold no individual cross-entry page.

Figure 4 shows the main data-entry page of Intercross. This fragment has three text boxes for user input, a button to clear the entered data, a button to submit data to the database, and a list to view crosses entered. Once data is entered, the new cross is appended to the top of the list on the page. When a user selects a precise cross entry, a new fragment is viewed, which is the cross-

entry page related to the selected cross. The cross entry page is populated with data relevant to that cross and is the page that allows the printing of bar-codes. Specifically, the cross entry page allows users to travel to the parent cross entry pages of the current cross, and enter the number of fruits, flowers, and seeds acquired from the plant.

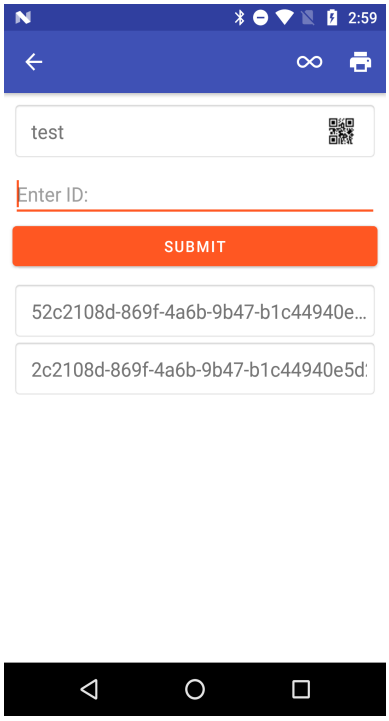

Figure 6: The pollen manager fragment in Intercross.

Intercross has played a significant role for breeders in plant science. Evaluation, crossing, and sampling are the primary steps for breeders; each step is a fundamental part of high-throughput phenotyping. Evaluation is a generic step of accumulating data and processing results using various algorithms intended for the subject. The sampling step filters the population of plant experiments based on the evaluation. Finally, the crossing step is essential for pairing previously sampled parents for breeding. The output breed, or cross, is the input for the next iteration of evaluation. This process life-cycle of sampling, crossing, and evaluating is essential

for plant scientists. Previously, this dissertation discussed Verify, which is used for sampling. Many Android applications can be used for evaluation, including FieldBook, OneKK, and Prospector. The only application that currently solves the crossing step is Intercross. This fundamental step has previously been accomplished by hand or out-dated-devices. Intercross contributes to high-throughput phenotyping by developing a solution to this task of crossing. Multiple novel algorithms and design architectures are used in Intercross. These contributions are highlighted in Chapter 3, Development Tools. These tools are created to ease the development of Android applications and reduce the number of run-time errors. Originally, Intercross was a Java application that did not use the Jetpack libraries. We developed the Java to Kotlin translator to explore the possibilities of Kotlin and to reduce code-base sizes while decreasing maintenance time of applications. With the use of Kotlin, the Shared Preference Fragment and Typed Data Bindings can be utilized, which are discussed in Chapter 3. Intercross is an accumulation of the novel tools and libraries developed for Android development. It serves as a template or guide for future developers.

OneKK and Abacus

One-thousand Kernels (OneKK) and Abacus are two Android applications for analyzing various features of seeds. While Abacus focuses on counting seeds in videos and images, OneKK aims at mining features of seeds such as their areas. OneKK and Abacus are both still in development but have feature-rich versions. Both applications use the Watershed segmentation algorithm to separate clusters and use mined ground truth kernels to estimate cluster sizes. The data input into Abacus for video is required to be a flow of seeds in a downward direction.

The seed tracking algorithm was created for desktop environments and solved the issue of seeds colliding during the tracking process [11]. The original algorithm worked well on desktop environments but performs poorly on mobile devices due to various computational bottlenecks; for example, part of Watershed requires a per-pixel search for labels. Efforts alleviate this bottleneck while still using Watershed. Optimizations include pixel skipping when labeling Watershed markers. Larger pixel skips can lead to inaccuracies, but basing pixel skip length based on the average kernel area of objects in the scene proved impressive performance boosts. Another bottleneck included searching for the maximum distance transform value, which is a pre-Watershed step for automated marker creation. Another algorithm was created based on cluster estimation, which relies on ground truth data in the image. The critical assumption of the cluster estimation method is to have some seeds disconnected from each other in the image. The cluster estimation method can mine contours that are disconnected, and then estimate cluster counts based on the averages of the ground truth area, perimeter, and inflection point count.

With the rise in deep learning and real-time object detection capabilities of RCNN networks, Abacus and OneKK are now planning to use such models to identify objects. More

information on the experimental setup, datasets, and algorithms used for detection is available in the Computer Vision chapter.

Processing video on mobile devices requires a significant amount of time and is computationally intensive. Along with this, at the time of creation, the Open Computer Vision (OpenCV) Android library did not have video processing functions. Abacus uses pre-built fast forward Moving Pictures Experts Group (FFMPEG) libraries to extract frames from videos. Then each frame is used within a detection algorithm to count the flow of seeds in the video. Two algorithms are used for frame processing. One algorithm processes frames in a batch as they are being extracted from the video. In contrast, the second algorithm extracts the entire frame sequence before running the image processing algorithm. The former algorithm shows slight improvements in run-time, but both are not real-time systems. These were both preliminary solutions to test whether real-time object detection was feasible with OpenCV and FFMPEG on Android devices. General RCNN models are necessary for real-time object detection. The cluster estimation algorithm requires parameters for an accurate result. A full-fledged seed classifier and counter would require multiple implementations of the cluster estimation algorithm. For example, one algorithm tuned on wheat seeds would not perform well on corn or soybeans theoretically.

A solution to this problem is to generate contour data for each seed kernel type and feed the contours to a RCNN model for real-time detection and classification. As OneKK requires contour mask data, MaskRCNN would be suitable, while FasterRCNN or SSDMobilenet is suitable for Abacus. Contour mask data is simply the array pixel values that bound a given contour; this information can be used to calculate the features required in OneKK such as area. While Abacus requires real-time object counting, SSDMobilenet is a good alternative as the

output is a localization of a bounding box with a class label. Future applications that involve simple classification of seed kernels can use any of the typical linear regression CNN models such as Visual Geometry Group Net (VGGNet), Resnet, and Mobilenet.

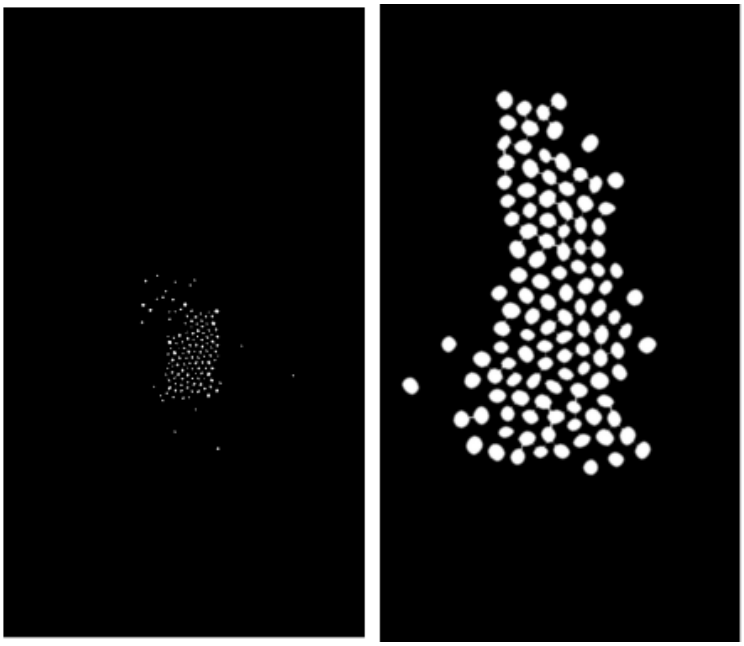

Figure 7: Two different distance transform outputs on canola and soybean.

There are limitations and trade-offs to using Watershed, the commonly used segmentation algorithm in image processing. Figure 7 shows a pre-processing step to Watershed, which is the distance transform. The distance transforms, also known as Euclidean mapping, creates a new image that replaces pixels with a value representing their distances from a border. Therefore, the pixels representing the center of gravity for objects is the highest value. Applying the same method to canola, Figure 7 shows promising segmentation. However, looking closely, some canola seeds are lost from the original image. The distance transform creates a sure foreground image which represents the markers of objects, a parameter to Watershed. Typically, when the markers are too small, this leads to over-segmentation of objects, a typical result of

Watershed. Therefore, the final count of images is an over-count if post-processing cannot account for over-segmentation. Fundamentally, a higher resolution image would be more precise for this measurement but overall would take longer to process, especially when working on a mobile device. Therefore, this dissertation proposes a simple trade-off guideline for creating image processing algorithms reliant on Watershed.

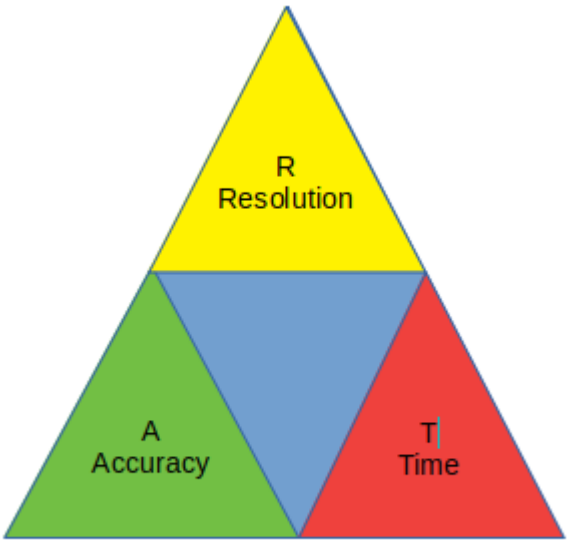

Figure 8: RAT diagram of image processing trade-offs.

The resolution, accuracy, run-time diagram (RAT), illustrated in Figure 8, specifies this connection: the run-time and accuracy of an algorithm are dependent on the resolution of the image. Therefore, at higher resolutions, image processing algorithms take longer to run but may achieve higher accuracy in results, and fidelity in measurements. Subsequently, more precise requirements require a higher-resolution input but have a longer run-time. Because Abacus requires real-time processing in the video, further iterations of this algorithm focus on maximizing accuracy while maintaining real-time requirements.

For OneKK, it is possible to have multiple modes of processing. Some users may be interested in fast results and lower accuracy. Other users may not mind the wait for higher accuracy.

OneKK is a soon-to-be-released application with a novel hierarchy-based cluster counting algorithm. The complete algorithm is not described in this dissertation. However, preliminary results describe the efficacy of these methods. Figures 9-14 show the visual results and counts for each image. Because this algorithm relies on a distinct difference between the kernel's color and the background, white rice performs very poorly. These are preliminary results; therefore, the algorithm may be modified to account for white objects in future iterations. The count ranges are output from two different algorithms. One uses ground truth areas to estimate cluster counts. In contrast, another uses the difference between contour curvatures and their convex hulls to count clusters. Therefore, for the area-based estimations, a sample size of ground truths with a reliable standard deviation (accounts for the total population) in the area is required to maximize accuracy. Along with area-based measurements, the kernels which have been appropriately segmented can be measured using minimum area rotated rectangles to calculate their minimum and maximum axis.

Figure 9: Counting 115 soybeans in OneKK.

Figure 9 shows the output of OneKK on a sample of 115 soybeans. The algorithm estimates that this picture contains 110-115 beans using its convexity estimation method. The area-based estimation range is 112-115. The lower bound inaccuracies are due to cluster estimation.

Figure 10: Counting 168 wheat kernels in OneKK.

Figure 10 shows the OneKK output on wheat kernels. The convexity-based counting range is 150-170 while the area-based counting range is 150-169. Both methods give very similar results in this case. Wheat seeds are more difficult to segment than more spherical objects. There is apparent over-segmentation within the wheat kernels, these over-segmentations aren't counted but would affect the overall area measurements.

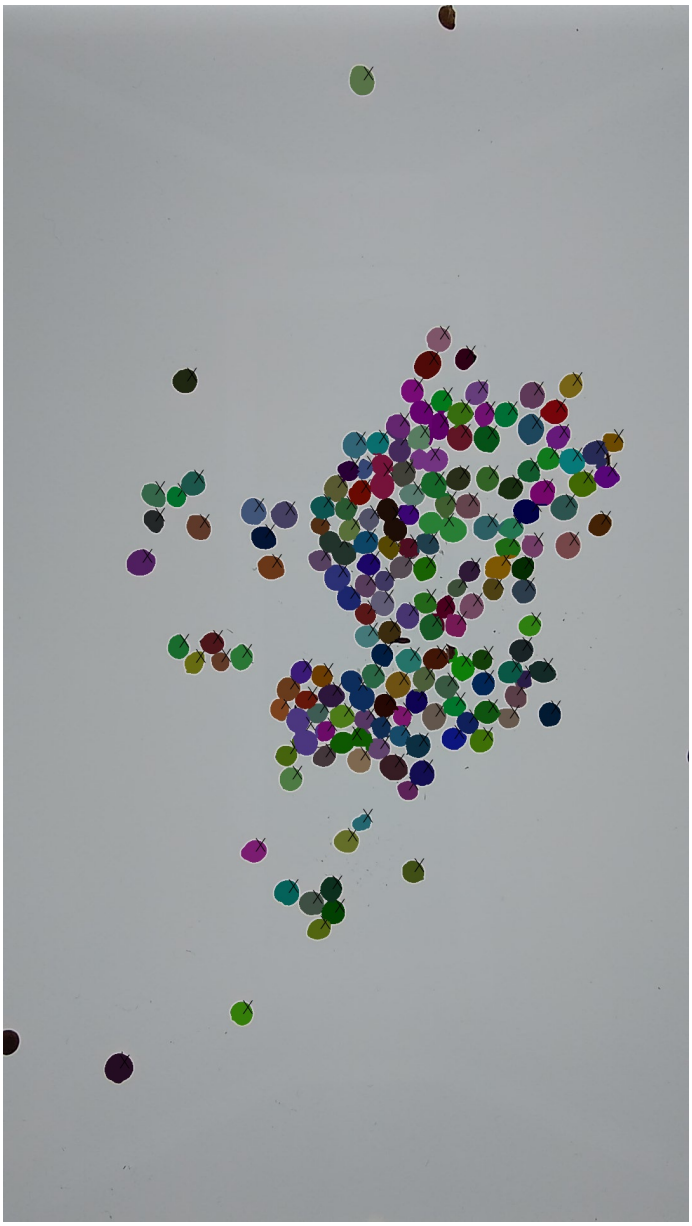

Figure 11: Counting 172 sorghum in OneKK.

For sorghum, in Figure 11, the convexity count is 170-173 and the area-based count is 170-177.

Figure 12: Counting 133 rough rice in OneKK.

Rough rice kernels, shown in Figure 12, have the most apparent over-segmentation of kernels. As shown in the image, some inner kernels are not shaded. The shaded contours represent the counted objects. The results give a slight undercount, but the final area and dimensions are inaccurate. The final count using the convexity-based estimation is 125-127,

while the area-based estimation is 121-127. Figure 13 shows the results for white rice. These results are by far the worst of all the kernel dataset. The image contains 317 white rice kernels, but the final count range is only 12-12 for both methods. Finally, Figure 14 shows the output for canola seeds. The total count for this image is 136, but the convexity range is 141-142, and the area-based range is 141-149. Both methods show an over-count for these kernels for a unique aspect. Between canola kernels, white space is detected as its kernel. Because the canola seeds are tightly clustered, the disjoint background between connections become their detected contours. These false positives are alleviated by searching all pixels within each contour for a certain intensity of white. The contours are filtered by checking if it contains a white intensity similar to the background. Filtering helps reduce the over-counting problem.

The mobile application of counting multiple types of seed kernels in a natural laboratory environment has never been accomplished before. OneKK has the unique ability to count five different seed types effectively. OneKK accomplishes this by requiring the use of a lightbox, and multiple novel estimation algorithms for accurately identifying clusters based on ground truth mined data. Watershed has been widely researched and accepted as the best for object segmentation in image processing. Problematically, the traditional usage of Watershed is infeasible on mobile devices for applications that want real-time results. OneKK proposes a solution to counting multiple class objects in a reasonable amount of time, which is a major contribution to mobile high-throughput phenotyping. The image algorithms and area estimations are also novel and contribute to the society of computer scientists. With the future moving toward extended use of deep neural networks, applications like OneKK, should be further studied to explore the utility of image processing. The deep learning community has full application and is promoted for its generality and performance. However, zero-data-needed

image processing algorithms are powerful. The use of such algorithms requires no overhead of training models, reflecting good practice for solving rudimentary computer vision tasks. Chapter 4, Computer Vision, discusses these applications further and aspires to build a new hybrid deep learning image processing pathway.

Figure 13: Counting 317 white rice in OneKK.

Figure 14: Counting 136 canola in OneKK.

Rangle

Rangle is an in-development project that focuses on calculating root-angles of various plants. Currently, there is an Android implementation for manually calculating the root angles by using Canvas drawing. The user may upload an image, draw lines over the roots, and produce an angle as output. Multiple Python preliminary scripts are in development to quantify root angles automatically. The scripts typically have a pre-processing segment to threshold the roots from the background. One script uses an iterative Hough Lines transform to model the roots as lines. Another script uses convex hull calculations to find the furthest points from the root. The low quality of the test images creates inconsistencies for the scripts, which is the motivation for creating an Android application. Roots may have an unpredictable growth network such as interleaving roots, making automatic angle calculation difficult.

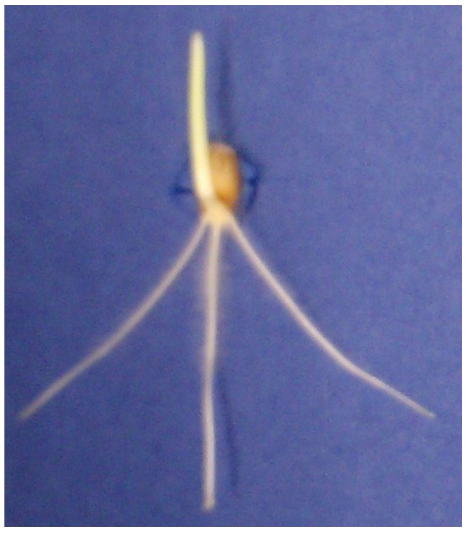

Figure 15: An example of the Rangle dataset.

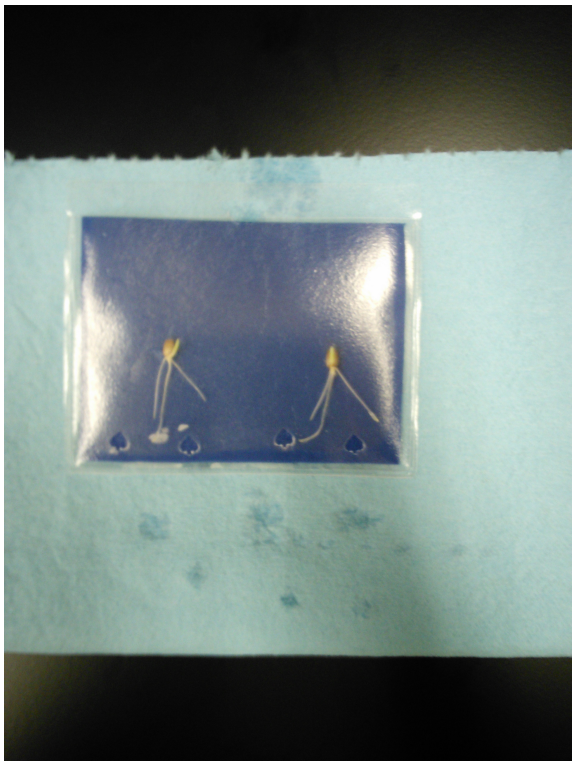

Figure 16: A Rangle example in plastic.

Figure 17: Example of the Rangle dataset.

Figures 16 and 17 show the various original images taken. The main goal of this project is to find a reproducible way to calculate root angles. However, the original images can be manipulated, and there is no requirement for using the original dataset. The original dataset is fundamentally challenging to create generic angle inference because of the varying scales (pictures taken at different heights) and variation in whether the roots were taken in a container or not. Finally, some samples have no roots or interleaving roots. The first step of this project was to recreate the dataset so that each sample image had one root sample with a uniform blue background. Figure 15 shows what the new dataset looks like, which is a cropped version of the original; each image should have a single sample. With this given, the following algorithm can be applied using OpenCV libraries. First, a color threshold occurs to eliminate the blue background from the foreground object. Secondly, Otsu's thresholding creates a binary image

representing the background (white) and the foreground object (black) [12]. Finally, the foreground object is processed and found as a contour. The contour can then be represented as a convex hull. The convexity defects are then found (the difference between the hull and the contour) to calculate the root angle. However, there are many convexity defects on naturally occurring manifolds; therefore, a greedy algorithm search is used to find the left and right-most contour points. These contour points then form a line with the contours center-of-mass (calculated using Hu moments), and an angle can be calculated [12].

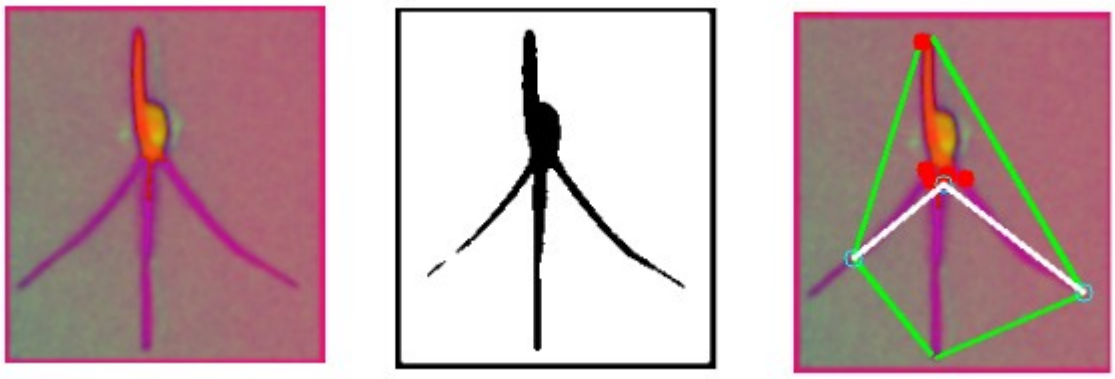

Figure 18: The three step-process: threshing, extract contours, calculate angle.

From the limitations of image processing algorithms, it is not always possible to infer a root angle given any image. For this reason, an Android application was created and available for reference in Appendix A. This application utilizes a drawing canvas surface for user-annotated root angles. Lines are then generated, given these user-inputted points to calculate a root angle.

```
Rangle(img, contour):

    #finds center of contour in row, column format within the image
    center = centerOfMass(contour)

    #find the hull and differences between the hull and the contour
    hull = convexHull(contour)
    defects = convexityDefects(contour, hull)

    #initialize left-most and right-most points (row column format)
    left = (0, img.width)
    right = (0,0)

    #search through all defect points to find left and right-most points
    for x,y in defects:

        if left.x > x and left.y < y:

            left = (y,x)

        if right.x < x and right.y < y:

            right = (y,x)


    return rads*180/pi * calculateRads(center, left, right)
```

Figure 19: Greedy algorithm used to calculate root angles.

Automatic root angle measuring is a new field with no current alternative to the Rangle process. The significant contribution from Rangle is the ability to automatically calculate the angle between the two most distance root paths. Previously, plant scientists were required to manually calculate angles on the plant itself, which introduced human error. Rangle simplifies the process by only requiring the user to take a picture, and speeds up the calculations

significantly. The greedy algorithm and image processing code are also novel, which show promising results for evaluating contours of simple blob detected objects.

Chapter 3 – Development Tools

Motivation

As the high-throughput phenotyping community grows, there is a need to expand available tools and libraries to accommodate users solving similar problems. Chapter 2 discussed a plethora of applications for this community. Not surprisingly, application architecture and fundamental modules such as file input and output operations are relatively static and can be used ubiquitously throughout all applications.

The tools described in this chapter are libraries and scripts that reduce development time and data generation. The Auto Preference Fragment and Type Data Binding Fragments sections are specifically for Android development. They both aim at reducing boilerplate code with newly proposed architectures and code generation. The Automatic Cluster Generator (ACG) section describes a script and user-interface tool for ease of annotating datasets. The ACG is a supervised tool for generating Microsoft COCO annotations [13]. The latter script aims at reducing the time it takes to annotate large datasets of images with class labels. Currently, there are many different annotation tools in existence specific to COCO, but none offer a semi-automated approach. Most annotation tools focus on multi-class labeling, while this approach leverages the assumption that users are annotating one class at a time and can infer potential regions based on pre-processing user input.

Auto Preference Fragment

The Auto Preferences Fragment aims at reducing boilerplate code by wrapping typical Android Shared Preferences code into class members that are extended by fragment specific code. Developers can efficiently utilize the Auto Preference Gradle plugin by modifying their Gradle build script file in a given Android project. The only parameters required to the Gradle Auto Preference extension are the package name and path to the preferences file that defines the Shared Preferences setting layout.

```
<PreferenceCategory android:title="Profile"
    app:iconSpaceReserved="false">
    <EditTextPreference
        android:title="Person"
        android:icon="@drawable/ic_setting_person"
        android:key="org.phenoapps.intercross.PERSON"
         />

    <EditTextPreference
        android:icon="@drawable/ic_setting_experiment"
        android:key="org.phenoapps.intercross.EXPERIMENT"
        android:title="Experiment" />
</PreferenceCategory>

<PreferenceCategory android:title="Naming"
    app:iconSpaceReserved="false">
    <ListPreference
        android:defaultValue="0"
        android:entries="@array/blank_male_array"
        android:entryValues="@array/blank_male_array_values"
        android:icon="@drawable/ic_setting_male_blank"
        android:key="org.phenoapps.intercross.BLANK_MALE_ID"
        android:title="Allow blank male ID" />
    <ListPreference
        android:defaultValue="0"
        android:entries="@array/cross_order_array"
        android:entryValues="@array/cross_order_array_values"
        android:icon="@drawable/ic_setting_cross_order"
        android:key="org.phenoapps.intercross.CROSS_ORDER"
        android:summary="%s"
        android:title="Cross order" />
    <Preference
        android:icon="@drawable/ic_setting_pattern_create"
        android:key="org.phenoapps.intercross.CREATE_PATTERN"
        android:title="Create cross ID pattern"
        android:summary="None"/>
</PreferenceCategory>
```

Figure 20: An Android preferences layout file.

The Android resources folder contains extensible markup language (XML) definitions that define views used in the source code for the application. Typically, applications define a

preference file that contains the relative view hierarchy for the settings page. Developers typically use the PreferenceFragment Android class to load preference resources into a template view based on the XML definition in the file. This template view is ubiquitous across Android applications and contains different widgets for persisting user-specific data. The XML file always has a PreferenceCategory root tag. It embeds a tree of other tags such as PreferenceTextView, and PreferenceRadioBox. Each preference view contains a unique user-defined key and other optional attributes. When developers need to access the persisted data from the settings, they must query a SharedPreferences service with the key defined in the settings file. Developers who define such a preference file can utilize the Auto Preference Fragment plugin. The plugin uses XML processing libraries to parse and translate each XML tag into the fields of a class. Figure 20 shows an example of such a file where multiple settings are defined in an application. The rendered version of this settings page is viewed in Figure 21.

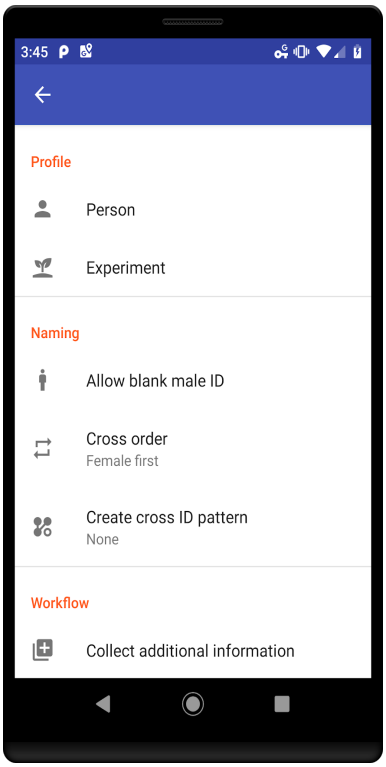

Figure 21: A PreferenceFragment example.

The plugin begins by recursively searching the resources folder of the project. When a file with the parameterized name is found, it is parsed and finds the first Preference Category tag. The children nodes of the category tag are parsed, and a file is created with the generated code for each parsed tag. The current supported Preferences are EditTextPreference, CheckBoxPreference, ListPreference, DropDownPreference, MultiSelectListPreference, SwitchPreference, and SeekBarPreference. Figure 23 describes the idiosyncratic nature of using Shared Preferences; however, this is an example of an automatically generated preference class that eliminates the need to program such details. To query for variables, the user needs to know the key defined within the preferences file. Furthermore, to edit that setting, the developer is required to create a Shared Preference Editor object using a call to the Shared Preferences object.

The generated code aims at replacing this idiosyncratic code with typical variable assignment statements. If the user chooses to extend the generated Auto Preference Fragment, the generated fields from the superclass are inherited. The generated fields correlate to the preference tags defined in XML. By generating getters and setters for each field, the user then has access to each setting as a standard variable in Java or Kotlin without having to know the key of that specific settings tag. The AutoPreferenceFragment code can be extended or modified by the user to define getters and setters optionally.

```
open class AutoPreferenceFragment: Fragment()
{
    internal companion object CompanionPrefs {

        val prefs = PreferenceManager.getDefaultSharedPreferences(requireContext())
        var person: String
        get()=prefs.getString("org.phenoapps.intercross.person", "") ?: ""
        set(value) {
            prefs.edit().putString("org.phenoapps.intercross.person", value).apply()
        }
        var experiment: String
            get()=prefs.getString("org.phenoapps.intercross.experiment", "") ?: ""
            set(value) {
                prefs.edit().putString("org.phenoapps.intercross.experiment", value).apply()
            }
        var allowBlankMaleId: Boolean
            get()=prefs.getBoolean("org.phenoapps.intercross.blank_male_id", false) ?: false
            set(value) {
                prefs.edit().putBoolean("org.phenoapps.intercross.blank_male_id", value).apply()
            }
        var crossOrder: String
            get()=prefs.getString("org.phenoapps.intercross.cross_order", "0") ?: "0"
            set(value) {
                prefs.edit().putString("org.phenoapps.intercross.cross_order", value).apply()
            }
```

Figure 22: Automatically generated code based on a user's preference file.

Figure 23 is an example of a piece of code that queries and edits a Shared Preference setting with the typical key-value pair defined in the preferences file. The assert statement shows the generated Auto Preference variable being used to verify equality between the two implementations. The person variable can also be updated using typical assignment statements, which is a wrapper to the typical idiosyncratic code. Auto Preference Fragments default to asynchronous updates using the apply method. A future implementation may extend this class to allow the developer to decide how the variables are updated in the settings.

```
val pref = activity?.getSharedPreferences("PREFS", MODE_PRIVATE)
val tPerson = pref?.getString("org.phenoapps.intercross.PERSON", "")
val experiment = pref?.getString("org.phenoapps.intercross.EXPERIMENT", "")

assert(person == tPerson)

val edit = pref.edit()
edit.putString("org.phenoapps.intercross.PERSON", "Chaney").commit()
```

Figure 23: A comparison between preference querying.

The Gradle files and project source code is available in Appendix A. In conclusion, this plugin is ideal for new and experienced Android developers. They want to replace Shared Preferences based boiler-plate code with class fields automatically. The setup for this project is simple and has a concise parser for further extensions. Currently, the plugin supports all available Preference views and is a recommended plugin for application development. This plugin is novel to Android developers and is a simple code generative project for reducing development time.

Typed Data Binding Fragments

As the Android ecosystem grows, more developers are starting to learn how to create mobile applications. The ability to freely submit applications and monetize features has pulled in developers. The Android software development kit (SDK) is continuously updated to alleviate specific problems with old architectures and replace them with new and more comfortable or more flexible versions. The newest of which is the Jetpack libraries. The introduction of Jetpack libraries defined a new system for binding variables to view object definitions. The system in Jetpack is called data binding, which allows the user to define a given variable for a view within the fragment layout XML file [10]. Jetpack dynamically generates a class file that represents the views defined in the layout, and the variables defined in a particular top-level data tag. The proposed Typed Data Binding Fragment is an extension to the data binding plugin of the Jetpack libraries. This proposed methodology replaces layout querying functions with references to the view objects built by the dynamically generated data binding class. Currently, when developing Android applications, the user is required to call the 'findViewById' function to query their layout files for view ids. These view ids then create a local object for the developer to modify the UI or other front-end elements of their application. This section provides a replacement for 'findViewById' calls. It decreases the learning curve for starters while also eliminating boiler-plate code.

The suggested methodology for using Type Data Binding Fragments is to use the generated data binding class as a reified type to a fragment's class signature. A use-case reference is the application Intercross defined in Chapter 2. Within Intercross, a base fragment class defines a type that extends the base class ViewDataBinding [10]. The base fragment also

takes a layout id as a parameter to the fragment constructor. Finally, the base fragment extends the fragment class. However, it can utilize the Auto Preference Fragment open class. The view for the layout file inflates in the onCreateView method, which is a standard life-cycle method for fragments. The inflation uses the DataBindingUtil reified inflate method from the Jetpack libraries and calls the layout id given as a parameter to the fragment class. Using the dynamically created binding class, the developer can now create fragments with the type of the generated binding class and use Kotlin extension functions to access any variable available in the binding class. For example, a developer has defined a data binding layout for sending a message and thus is named message_fragment.xml, Android generates a class dynamically named MessageFragmentBinding. The developer can use this MessageFragmentBinding class as a reified type to their MessageFragment class implementation. In Kotlin and other object-oriented programming languages, functions can statically extend the reified type reference, which is MessageFragmentBinding. These extended functions access the members of the binding class allowing access to defined data-bound variables and views within the message_fragment.xml layout file. Android has recently released a view binding extension, which is a similar approach but requires maintaining a reference to the binding object after inflation, which requires more work from the developer. Android Typed Data Binding fragments is a proposed replacement for view binding, an announced project during the writing of this dissertation.

The Android Typed Data Binding fragments introduce an easy and straightforward way to join layout files with class variables. This generative code step utilizes the new JetPack libraries to bind layout views to an abstract class automatically. This architecture allows users to extend this abstract class to any fragment in their application, and automatically populate member variables from their layout hierarchies. This creates a ubiquitous coding environment for

cross layout-application development. With this implementation, word completion and code correction will automatically pick-up class member variables, such as the view hierarchy id's, which previously was impossible. This class hierarchy is a novel approach to code development. It can be used across areas of computer science using the model view controller architecture (MVC), where views are necessary for human interfacing.

GNSS Message Parser

The GNSS Message parser is an open-source NMEA string interpreter written in Java. NMEA messages define the communication protocol between satellites and GPS devices [14]. Messages contain different information given the connected constellation, including the number of satellites, timestamps, latitude, longitude, horizontal dilution of precision, vertical dilution of precision, altitude, and mean sea level. Previously, Android devices relied on a Location Manager object to connect with internet or GPS services to receive location data as a black box [15]. As of Android version 9, Pie, some devices can receive raw GNSS measurements. According to the documentation, some devices are limited to certain GNSS features: pseudorange and pseudorange rate, navigation messages, accumulated delta range, and hardware clock [15]. The purpose of this library is to parse navigation messages and provide more precise and flexible location measurements than the default Location Manager. Related work includes the Networked Transport of Radio Technical Commission for Maritime Services (RTCM) via Internet Protocol (NTRIP) and Google applications for viewing GNSS information. This NMEA parser aims at helping open-source developers implement new applications, where NTRIP and the Google GNSS Measurement tool are for visualizing GNSS data.

Currently, parseable navigation messages include global positioning system fix data (GPGSV, GPGGA), geographic positioning data (LCGLL, GPGLL), and Recommended Minimum (RMC). RMC messages are required for RTK communication between a base and a rover. RMC messages are used to send corrected data to rover devices. The GNSS is used within the application Survey from Chapter 2 and, when connected to an RTK device, listens for corrected location messages to get Float RTK to fix quality. Survey also utilizes the satellite

positioning messages to visualize constellations based on the receiver's position. As discussed in Chapter 2, the GNSS message parser is a central library for Android developers wanting to utilize their on-device GPS unit. The ability to parse GNSS messages gives the developer a more fine-tuned control on NMEA messaging. Currently, there are no other modern GNSS message parsers, this repository link is available in Appendix A, in the Survey section.

Snackbar Queue

A common theme throughout this chapter has been the introduction of various libraries to increase the productivity of developers. This section describes yet another approach that augments Android development by extending an available class. Although, this section focuses on a visual driven object that notifies the user of messages. Toast and Snackbar messages are a common feature in Android that all just this.

The Toast messages object for Android applications allows developers to display a rectangular message pop-up to the user with a defined text [15]. Snackbar is a variant of Toast messages that allows the pop-up to have an action defined as a button. For example, when an item is deleted from a database, the developer may queue a message saying that a certain item was deleted and add an action that allows the user to undo the deletion. Toast and Snackbar messages are built to be quick and unobtrusive to the user to display a simple message. Currently, there is no default way to queue messages to display sequentially.

An implementation called Snackbar Queue correctly queues and displays Snackbar messages. Snackbar queues use a data class called SnackJobs. The view defines a SnackJob, the message to display encoded as a String, the action text to display, and a function that is applied if the action button is clicked. The Snackbar Queue object uses a mutex to display messages if a message is not already displayed safely. The object also uses a TimerTask to check if any jobs in the queue are running. The only time a SnackJob is not displayed is if the user transitions from the view it is attached to. Because the SnackJob must bind to a given view to use the Snackbar functions, it is killed if the view is destroyed. A scenario being when an arbitrary amount of SnackJobs queues for a given page, and the user transitions to a new page. It would be possible

to continue the queuing on the next page, but the current implementation kills the jobs that were on the source page. Therefore, all queued jobs are viewed in the fragment in which the job is submitted.

The functionality of a Snackbar queue is essential for developers who need to ensure message visibility. If the UI allows asynchronous actions that are coupled with these types of messages, then the current state of Snackbar messages overwrites previously queued messages. This current Android implementation claims in the documentation to be a queue, but it does not function like one. This implementation of SnackJobs is another example of the various libraries and tools used to simplify and augment Android development. This contribution to Android developers is an essential tool for handling Toast messages and is available for use in Appendix A.

Automatic Contour Generator

This section proposes a new way to generate contour annotation files for state-of-the-art RCNN models such as MaskRCNN and FasterRCNN. Chapter 4 defines an experimental setup for training deep neural network models, the script used for creating annotation data is this semi-supervised contour generator. This annotator focuses on environments with a uniform background to label merely geometric objects. Simply geometric objects are a population of objects with a mesokurtic area. These assumptions are essential for how the script automatically bisects contours and assumes clusters of more than 2 objects.

The ACG is a Python script that has both a headless and user-interface oriented mode. The general idea is to fine-tune an image processing script with the user-interface mode until contours generate correctly. The headless mode then automates the computation. The UI mode contains some linear algebra processing features to split contour data and screen-picking for selecting and deleting contours. The ACG creates COCO datasets, including contour approximation arrays, areas, locations, and bounding box coordinates. However, most of the UI and automated code could be reused to create other annotation-specific-formats. Plant scientists typically use major and minor axes to estimate the area of various types of seeds using the formula for an ellipsoid. It is not true that seed kernels are perfect ellipsoids; therefore, this previous methodology introduces error in measurements. This data generator aims at using pixel-mask area acquisition to reduce errors from such methods.

The Python script uses OpenCV for image processing. It is aimed at creating an environment for testing object-specific contour generators [12]. Manual, and sometimes expert-level techniques are required to generate precise contour data. Therefore, instead of manually

annotated data, the user must create a script that can segment the objects needed for localization. For many domains, scripts can be tweaked to work for different but similar objects. This script works well for the seed domain. Most kernels are similar in size, typically very small, and the pictures are all generated with similar backgrounds. With a specific domain such as the seed kernel dataset, the ACG saves significant time as compared to manual annotation. As the size of data increases for deep neural network models, we see a need to prove and verify the training data. The ACG is a step forward to automating such tasks by allowing the user to approve or delete generated contours, and apply other various functions. The following defines the usage of the ACG script: python acg.py –dir A –target target > annotations.json &. These parameters work on a dataset directory that includes multiple category folders of training, testing, and validation images. Specifically, the directory format includes a root directory folder, N class folders inside the parent root folder, where each class folder contains three folders. These three folders must have the naming: train, test, val, as is standard for this type of analysis. Each of the three sub-folders contains a set of images to be processed. Typically, train, test, and validation datasets and created separately. Therefore, the target parameter to ACG specifies which dataset to process. For example, running ACG with target 'train' processes each class folder's train directory and accumulates an annotations JavaScript Objection Notation (JSON) object for each image. They are labeling each image with the class folder name as its tag. This process initiates three times to create three separate JSON files to be used to train a deep neural network.

To fully evaluate a supervised model, a test, validation, and training dataset is needed. The training set is typically batched and fed into the network for training. The validation dataset is assessed per epoch of the training to ensure the model is not over-fitting to the training dataset. Finally, a test dataset evaluates the generalization of the model's training.

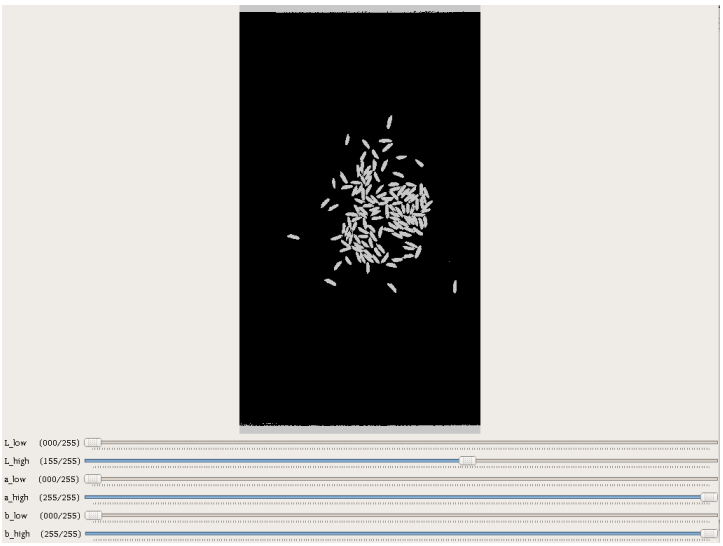

Figure 24: The LAB threshing UI for ACG.

There are three main steps to this semi-supervised annotator. The first step is a pixel-level threshing replacement using the International Commission on Illumination LAB (LAB) color space [12]. The threshing input has slide-bars to control the minimum and maximum threshold for each LAB vector. The annotator uses NumPy's logical and operator for visualizing threshold updates in real-time. If the user is unable to partition the objects of interest from the background, then it is recommended to adjust the experimental setup or use a manual annotator. Once the objects are correctly threshed, the next step is to adjust the minimum and maximum area variance of clusters. Three equations eliminate noise, identify pairs and clusters. In this step, a precise area is calculated for each contour. Eliminated contours have an area less than the minimum area specified in the UI. Otherwise, they are split in half using the mid split algorithm and considered a pair of kernels if they do not satisfy Equation 1.

$$area \leq avgArea + a \vee area \geq avgArea * b$$

Equation 1: Formula to detect clusters of two.

The user inputs a and b by using a seek bar in the UI. The input is an effective and straightforward way to annotate clusters of simple-objects semi-automatically. The methodology is aimed at seed kernels but should work for other simple objects. Another assumption is that the image only contains one class of objects or objects with similar average areas.

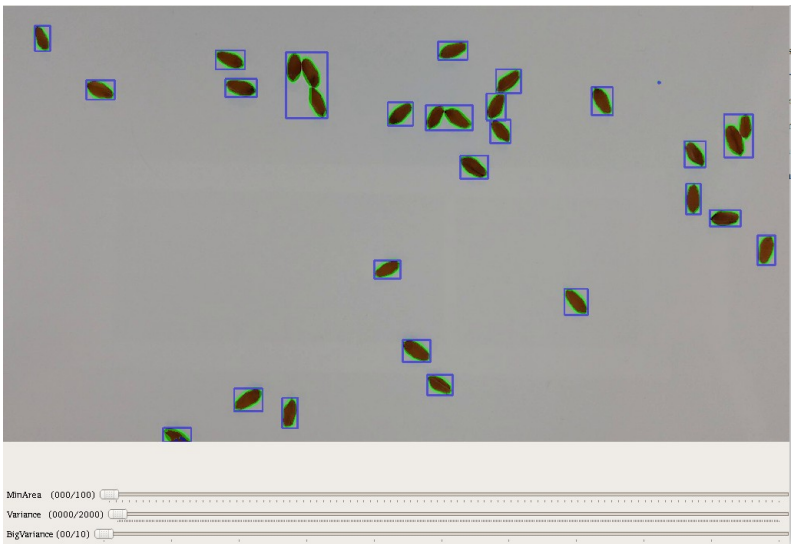

Figure 25: The cluster estimation and noise reduction step using Equation 1.

Contours viewed that are less than the average area display as single clusters with a bounding box. Contours that are greater than the average area multiplied by a user given scale are clusters, as they are intuitively a multiple of the average area. Otherwise, the contour is a pair of kernels and uses the mid split algorithm to split the contour into two pieces automatically. The split contours are less than or equal to the average area and have boxes automatically rendered.

The clusters are then manually bisected by the user, as currently, there is no automatic cluster slicing implemented.

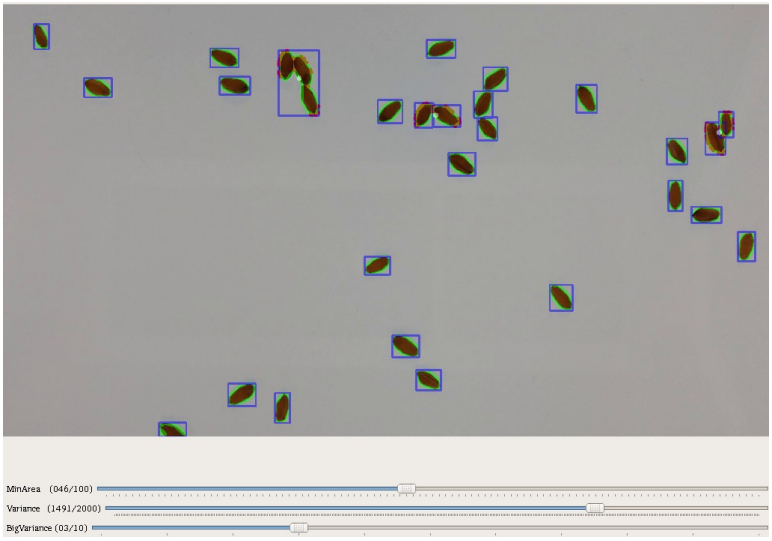

Figure 26: The second step after clusters of two have been automatically sliced.

After the user is satisfied with the non-cluster bounding boxes, the third step initiates. The final step in the process is manual input of line bisections to cluster contours. The user focuses on clusters of three or more kernels and holds right-click and drags to create a line to bisect and render new bounding boxes in real-time. The threshing step of this process is immutable from the start and only occurs once during a single run of the script; the subsequent steps occur for each image in the dataset folder. When the user is in UI mode, they have the option to print the current annotation JSON file, which shows in the terminal, the option to skip the current image in the dataset folder. Finally, they can delete with mouse left-clicks. For line bisection, the created line uses the determinate to separate pixels of a contour into two groups, one side of the line or the other. The determinate algorithm is an iterative approach that loops through contour approximation arrays. A matrix is created with the two entered points, the

currently analyzed point on the contour, and a one's vector to create a square matrix. The value of the determinate will either be positive or negative, which defines which side of the line that point is on. This feature is useful when a contour is a cluster of objects; this way, a user can slice the cluster into multiple contours. This manual bisection concludes the three-step process. This annotating tool is available and open-sourced on Github, and it aims at reducing manual annotation time.

Future editions to this script may include semi-automated Watershed operations to attempt cluster segmenting. Problematically, Watershed has the issue of over-segmenting; therefore, further algorithms should prevent such results. Various visualizations can be improved, such as viewing wire-frame depictions. This type of visualization may be a useful option for users who consider the current version to be busy. Multi-object labeling is another improvement to this script but was not the original goal of this project, which focused on a pipeline for similar-object-annotating. Finally, rather than solely using LAB threshold pixel replacement, using other color spaces would be useful to help segment objects of similar color to the background.

With the increasing usages of models that require vast amounts of data, there is a rising need for more sophisticated approaches for creating labels. Network architects are either forced to pay for online community-based labeling services such as Amazon's Mechanical Turk or use a limited dataset for their model fitting. The tool described in this section alleviates such burdens by automating image processing tasks. The usage documentation and download links are in Appendix A. The algorithms and ideals used within this application are novel. The mid split algorithm and other criteria for automatically splitting contours are contributions to computer scientists for alleviating the manual annotations process. The novel algorithms have been

encapsulated into a Python script for batch use or use with a UI. As more data annotations are needed to accompany deep learning models, this tool can be utilized or extended.

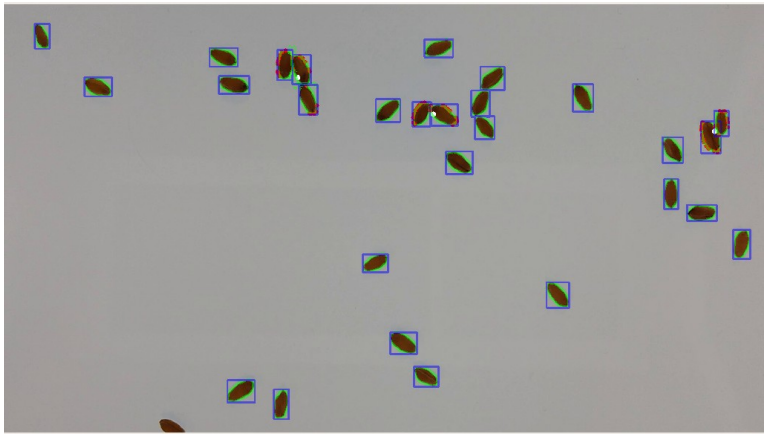

Figure 27: Final manual bisection step.

```
MidSplit(contour):

        #finds the center of mass of a contour
        centerPoint = calculateCenterPoint(contour)

        #finds the convex hull
        hull = convexHull(contour)

        #finds the points within the contour different from the hull
        defects = convexityDefects(contour, hull)

        #finds the furthest defect points from the center of mass
        extremas = getExtrema(defects, centerOfMass)

        #sorts defects by distance
        splitKey = EuclideanDistanceSort(extremas, centerPoint)

        #create pairs of defects to consider possible bisections
        P = permutations([splitKey[0], splitKey[1], centerPoint], 2)
        splitMap = {}
        actualArea = contourArea(contour)

        #loop through all possible lines and find equal-area-split
        for line in P:
                split1 = []
                split2 = []
                for point in contour:

                        #use determinate to classify which side of the line
                        #each point on the contour is on.
                        if determinate(line[0], line[1], point) > 0:
                                split1.append(point)
                        else:
                                split2.append(point)

                #calculate the areas of each split, add to a mapping where the
                #key is the difference between the splits
                splitArea1 = contourArea(split1)
                splitArea2 = contourArea(split2)
                splitMap[abs(splitArea1 - splitArea2)] = (split1, split2)

        #sort the splits to find the splits with similar areas
        split1, split2 = splitMap[Sort(splitMap.keys()).pop()]

        return split1, split2
```

Figure 28: The mid split algorithm, used to split a contour into two separate pieces.

Chapter 4 - Computer Vision

Motivation

With the evolution of mobile high-throughput phenotyping, farmers and plant scientists can utilize state-of-the-art deep neural networks to solve various tasks. Leveraging deep neural network libraries such as TensorFlow is the future for augmenting HTP. For example, state-of-the-art localization and classification of objects in images use RCNNs, which are readily available as pre-trained models. These models can be used to classify various plant species or even detect diseases. PhenoApps developers have worked on models for detecting whiteflies and necrosis in plants using such models. The applications of these models are boundless for plant researchers.

In terms of using these models, they are burdensome to train. However, deep neural networks (DNN) are significantly more costly to train than running a model for inference predictions. Most models need to utilize a cluster or a dedicated machine for training. When training completes, the model is check-pointed and loaded onto a local device to use in real-time. For mobile computation, models created with TensorFlow are using a subset of the original graph operations. TensorFlow Zoo contains various models that have TensorFlow-Lite versions. Some have already been optimized for mobile computation. Different models have managed to achieve higher accuracy in trade-off to throughput; therefore, there is a discussion on what is essential for specific tasks in this domain. TensorFlow Zoo is an oracle to decide which model is best when time or accuracy is of necessity. Whether they be for desktop environments or mobile devices, HTP methods can utilize various models to leverage modern computer-vision libraries

in their research workflows. Significant efforts are made within this dissertation to identify, implement, train, and deploy various deep neural network architectures. The idea of counting objects in an image is focused on in this chapter.

LeNet-5 was the first shallow convolutional network that performed reasonably on image classification, and the ImageNet large scale visual recognition challenge was created [16]. Since then, many networks have deepened similar models and have introduced new techniques to improve performance [17, 18, 19, 20]. MobileNet is a type of lightweight convolution-based network built for efficiency on embedded devices, precisely phones [21]. All of these previously mentioned networks are mainly used for feature recognition and classification of data based on training datasets. The RCNN is a far more sophisticated type of network that has the ability of semantic segmentation, image localization, and image masking [22, 23, 24]. RCNNs have longer inference time compared to image classification, and some networks are still inefficient or unoptimized for mobile devices. Other than MobileNet, there have been efforts to increase the efficiency of models on embedded systems, dependent on a given device [25, 26, 27, 28, 29]. Along with network-based optimizations and searches, multiple application-specific integrated circuits are being developed for improving network performance on embedded systems such as the Edge Tensor Processing Unit (EdgeTPU), Neural Compute Stick (NCS2), and NVIDIA Turing Tensor Core. The newest MobileNet, MobileNetV3, used with Pixel 4 on its new EdgeTPU application-specific integrated circuit (ASIC), has shown impressive improvements for real-time object detection. However, this research has shown that V3 and V2 have issues where V1 works per the documentation. Similarly, this research aims at finding low-cost devices and not necessarily the state-of-the-art equipment on modern phones. Therefore, SSDMobilenet is

used and tested on a Motorola Z running Android 7.1.1 with a Quad-core ARM 64-bit 2.2 GHz processor and 3Gb of RAM.

Specific examples of computer vision in HTP include classification, feature analysis, and counting models. PhenoApps works on various concrete implementations using OpenCV to analyze traits of seed kernels. OneKK is an Android application for acquiring weight measurements. Weight measurements with contour data give one-thousand kernel weight approximations, a standard metric for plant scientists to see the distribution of weights over their seed kernels. This ratio helps approximate the final yield of the crop given a set of seeds. Specifically, OneKK uses a cluster estimation method to locate seeds and uses constant-known indicators on the image to estimate the area in pixels of each seed.

The indicators are typically coins. The application uses a database to remember the diameters in millimeters for hundreds of different coins. The seeds are weighed using a scale, and the average weight per seed is calculated. Problematically, a new algorithm needs to be tweaked for each possible seed type. In Chapter 2, the section on OneKK shows the non-generality of pre-Watershed processing. Foregrounds of objects are threshed for input markers to the Watershed algorithm. The thresholding parameter is not constant for all types of objects. A specific image processing algorithm is used to accurately count a set of five seeds. This algorithm does not generalize to new seed types. It takes a significant amount of time to run on mobile devices, and it has a strict environment required for accurate results. Secondly, Abacus is an Android application for generic counting of seed kernels. Abacus has similar image cluster analysis algorithms using OpenCV to analyze high-speed video and static images. These techniques are often slow on mobile devices and do not meet real-time requirements. It is necessary to transition research to deep neural networks that have promising results for real-time

classification and localization of objects. Machine learning models lack generalization to new data, in this case, new types of seed kernels. In contrast to deep learning models, which have the potential to infer from previously learned data.

TensorFlow Zoo measures mean average precision as 21 for SSDMobilenet and 33 for MaskRCNN at the time this dissertation was written. However, the website shows that these values are slightly higher and differ from reproducible experiments due to eliminating detections that are below a certain mean average precision threshold. Mean average precision (mAP) is a standard metric used by data scientists. Classically, it is the mean of the average precision for all queries in a given dataset. With the introduction to the COCO dataset's metrics, mean average precision is now the mean of the average precision over all intersection over union (IOU) values [13]. Therefore, for the COCO dataset, SSDMobilenet classifies 21% of objects correctly on average over all IOU values. The experiments for SSDMobilenet in this dissertation use these pre-trained models to train on a new seed dataset. The seeds in question are generally simple objects like ellipsoids and spheres. It is hypothesized and shown that the mAP values are higher in comparison to the complex objects within the COCO dataset, such as cars, people, and bicycles.

This research aims at creating concrete applications for state-of-the-art computer vision models and comparing novel approaches with such models. Specific questions are related to the performance of object detection and how well objects are counted using these models. Counting objects is a computationally non-trivial task. Intuitively, how would a computer determine to count the background versus the objects within the image [2]? Classically, these problems have been solved by semi-supervised tasks where the developer knows the color of the object being counted or the color of the background. The models mentioned above have been foundational in

image processing and have attempted to reach human-level identification. Various metrics are evaluated on datasets of different seed types. Novel model approaches that output a number for a given image are different from standard RCNN models that output bounding boxes and labeled values for their detected objects. This thesis hypothesized that pre-trained models have higher accuracy for counting than the proposed models. The results section shows that the proposed counting models perform very accurately. While the pre-trained models also perform better than on the COCO dataset, they are not perceived to work as well as the proposed models. Although there are differences with how pre-trained models evaluate with the proposed models. Pre-trained models give recall and precision scores. While the proposed models output an average ratio of actual to expected count, mean squared error, and mean absolute error.

Models do exist for estimating counts based on density mass probabilistic functions, where the images are trained on labels, which are heat-maps of the original image [30,31,32,33,34]. There is a large community of research in-crowd and cell counting. However, such models require a density mapping label before model training. In contrast, the novel aspect of the proposed models is the ability to use only an integer label for training or post-processing on the image output.

Autoencoders train a consistent function where the input is an image, and the output is an encoded and then decoded version of the image. Autoencoders are known to work well with noisy data and take significantly less time to train than state-of-the-art localization models. Noisy data is an obvious factor in image classification. For the sake of plant science research, it can be assumed that a given concrete implementation may be solely used for laboratory-setting experiments where lighting and background factors are constrained. In such scenarios, auto-encoders may be unnecessary for noise reduction, whereas field-like scenarios would differ. In

terms of image processing, there is typically no ideal threshold value to generally segment foreground objects from the background. There are adaptive thresholding algorithms such as Otsu that find a possible value given a bi-modal distribution of pixel intensities. Although, some seed datasets have a non-bi-modal distribution, which furthers the need for more sophisticated image processing algorithms.

There is also the question of leveraging data augmentation to increase the performance of such models using generated contours. Also, how inter-seed classification works at inference time.

Three models are discussed and evaluated. Pre-trained RCNN models are evaluated to show their performance on mobile devices and prove whether or not they are capable of real-time object detection. Two different counting-based DNN models, with variants, are tested and evaluated for accuracy. Supervised models that take as input an image and its counted objects are compared to a similar unsupervised model. MobilenetSSD is evaluated to determine the following: whether different resolution datasets affect performance, whether overlapping seeds can be identified, and if combined datasets of different seed types perform as well as singular class identifications. Finally, compare the best of these results with the supervised density and unsupervised counter.

Supervised Density Count

The supervised density count network takes images as training data and actual counts of the objects within as labels. Various density count networks exist but utilize heat maps of the original image [31]. These networks apply a similar methodology for counting cellular structures, and dense crowds [32,33]. These are prevalent networks. However, they lack in total data evaluated. The microscopy-based density count utilizes only two-hundred images and requires manually labeled positional data of each cell [32]. The crowd-based density counting paper used a single video where different time-steps are assigned to training, testing, and evaluation [33]. Finally, the novel aspect of this network is the ability to train on an integer count label instead of a heat-map image.

This proposed network is a stack of alternating convolution and max-pooling layers followed by a min-max normalization layer and a density counting layer. The model utilizes separable convolutions, which are shown to be more efficient on mobile devices [21]. The stack of alternating convolutions and max-pooling reduces the size of the input image into a feature map that contains highlighted regions of seeds. The min-max normalization layer creates pixel values between 0 and 1, where the values close to one represent pixels that are seed kernel contours. Finally, the density counting layer sums the axes of the image to estimate a count. An updated model is testing whether or not adding trainable parameters to the summation step augments the learning process. By default, the TensorFlow mathematical reduce sum function does not have weights. A new layer creates trainable parameters for row and columnar data although this has not shown significant improvements in results.

This model uses TensorFlow, which loads, pre-processes, trains, and evaluates the images given a folder. Images are resized to 300x300 resolution for memory efficiency. Each image path that is loaded includes the count for the given image. The labels are parsed from the input paths and zipped with the corresponding image before batched into a TensorFlow dataset object.

The Pixel Normalization layer for this model uses min-max normalization:

$$\frac{X - Min(X)}{Max(X) - Min(X)}$$

Equation 2: The min-max formula used to normalize pixels data.

$$J = \sqrt{\sum_{i=1}^{n} \left(\frac{\left(y_{true} - y_{pred}\right)^2}{n}\right)}$$

Equation 3: Objective function for training the density count model.

The objective function, shown in Equation 3, uses the root mean squared error. The model is compiled using stochastic gradient descent (SGD). Density counting is accomplished by taking the summations of channels, rows, and columns in order.

$$N = \sum_{i=0}^{C} \sum_{j=0}^{R} \sum_{k=0}^{3} X_{i,j,k}$$

Equation 4: Arithmetic summation of an image.

Equation 4 uses subscripts to define column, row, and channel indices. For example, X_i represents an entire column of an image. $X_{i,j,k}$ gives a pixel value of a given channel.

Colored images typically have three channels representing red, green, and blue. Some images even have an alpha or depth channel, but for this research, basic red green blue (RGB) images are used. Therefore, Equation 4 first aggregates all RGB channel values for a pixel in a row and column. Secondly, $X_{i,j}$ sums all of the previous aggregated pixel channel values within a row. The final sum over the columns will create a single integer that represents a count for a given image. In Equation 4, C represents the number of columns while R represents the number of rows. By using the actual count as the labeled value for a given image batch, the pixel norm layer fits seed kernel pixels to a value close to one, and the density layer aggregates the total pixel values to estimate a count for an image. This model aims at estimating a count based on the feature map of several convolutions and max-pooling, which is similar to a heat map or density map but with a smaller dimension.

$$N = \alpha \sum_{i=0}^{C} \beta \sum_{j=0}^{R} \delta \sum_{k=0}^{3} X_{i,j,k}$$

Equation 5: A variant of Equation 4 with trainable weights.

$$argmin_w \sqrt{\sum_{i=1}^{n} \left(\frac{\left(y_{true} - w^T N \right)^2}{n} \right)}$$

Equation 6: Back-propagation step for counting weights.

Equation 5 shows a variant to the pixel counting summation step that adds trainable parameters to each step in the aggregation. The parameters alpha, beta, and delta form a weight vector that is applied to the input image. Each parameter focuses on different regions of the

image, the alpha parameter is applied to the columns, the beta parameter is applied to the rows, and finally, the delta parameter applies to the channel summation. Equation 6 represents the vector of alpha, beta, and delta as a single weight vector, w. Equation 6 shows the update process during stochastic gradient descent. Each parameter is updated to minimize the objective function. This minimization step attempts to decrease the difference in the actual and expected count. Intuitively, these parameters adjust the count based on the spatial properties of the image. Results show that these parameters did not provide significantly better results than Equation 4. A visualization of the density maps is produced by stripping the final density count layer off of the network. What's left is the stack of convolution and max-pooling layers followed by the pixel min-max normalization layer. Therefore, with an image as input, the stripped network produces a density map like in Figure 29.

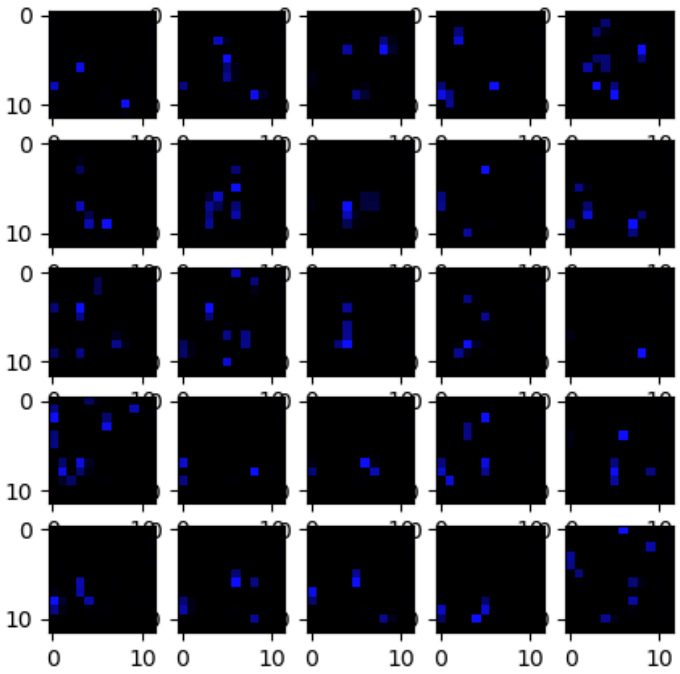

Figure 29: Density maps of several images of wheat.

The supervised density counter network is tested on the six-category seed dataset defined in the next section. Experiment performances are in the results section. An epoch takes about 40 seconds for 256 images. With 20 epochs, the training finishes in about 13 minutes. This training time is significantly faster than the state-of-the-art RCNN models.

The models were fine-tuned by searching hyper-parameters based on minimizing metric model differences from one. Hyper-parameters are randomly searched to find improved values for each dataset.

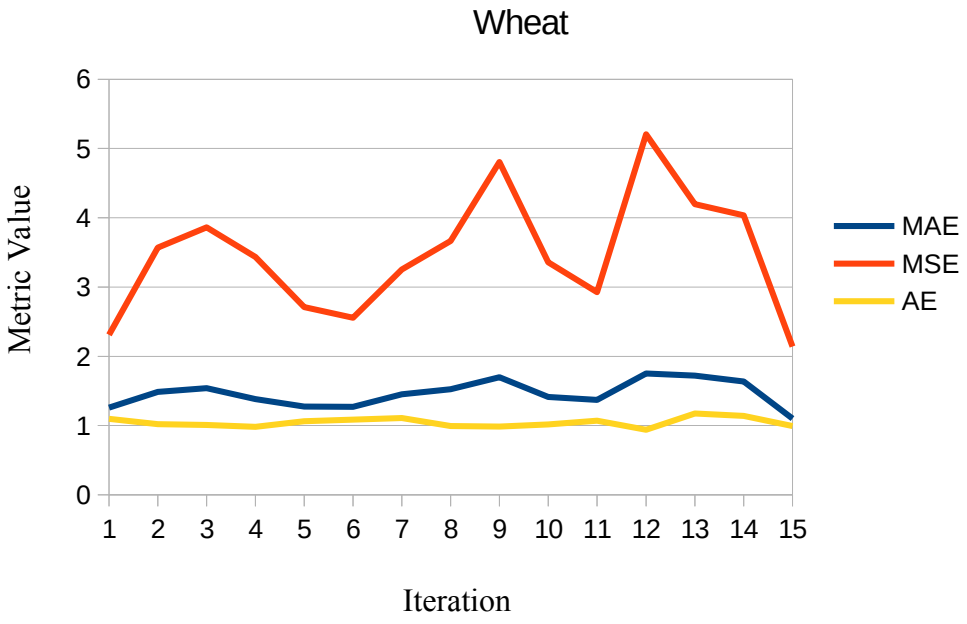

Figure 30: Hyper-parameter search of the wheat dataset.

There are many different parameters within this model, including learning rates and filter sizes. Figures 30-35 graph a randomized parameter search of around fifteen steps to find iterations with minimal error. Three different metrics are used, mean squared error, mean absolute error, and actual-over-expected (AE) count. Mean squared error (MSE) is an upper

bound to mean absolute error (MAE) and is a good metric for finding out-layers that affect the total error. As seen throughout the figures, MSE is always greater than MAE.

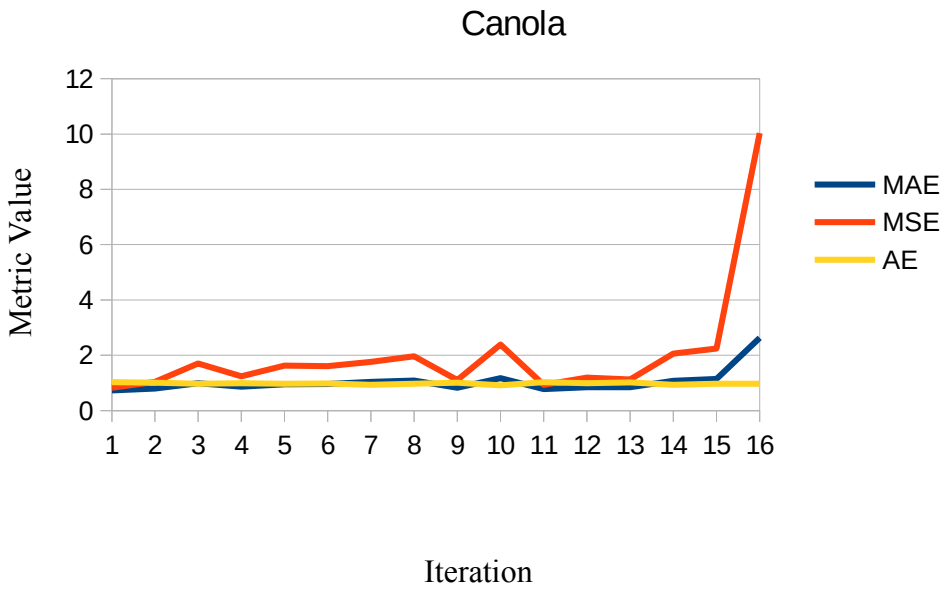

Figure 31: Hyper-parameter search of the canola dataset.

The metric, AE, is generally lower than other metrics but lies close to the MAE line. As the AE metric is a ratio between the expected and actual count while the MAE is the absolute difference, they are closely related.

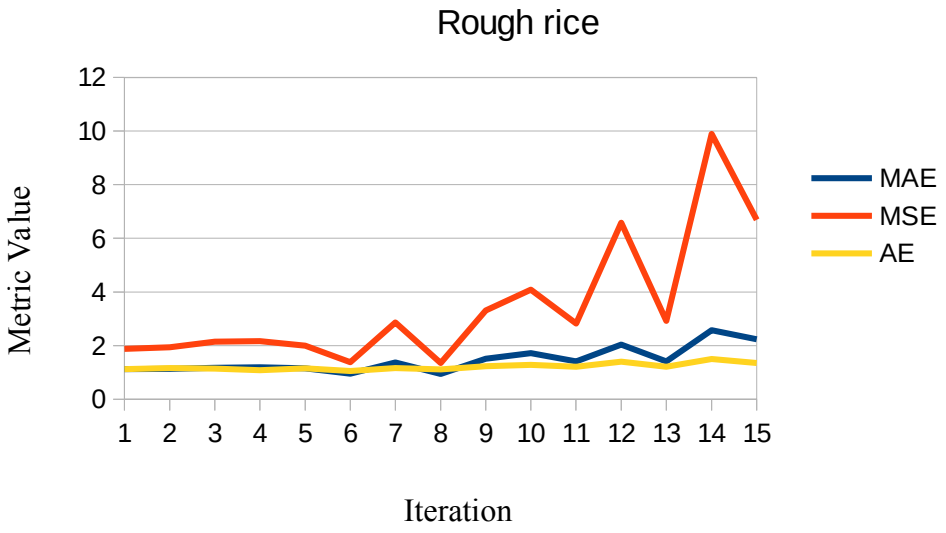

Figure 32: Hyper-parameter search of the rough rice dataset.

Mean squared error is used for the objective function in Equation 3 because it is differentiable, while MAE is not always differentiable. Although, MSE is affected greatly by outliers while MAE is not. Actual-over-expected is mainly used to evaluate models and express results. The closer AE is to 1, the more accurate the count is. The greater the AE is, the more the model overcounts. Similarly, the closer the AE is to 0, the more the model is undercounting.

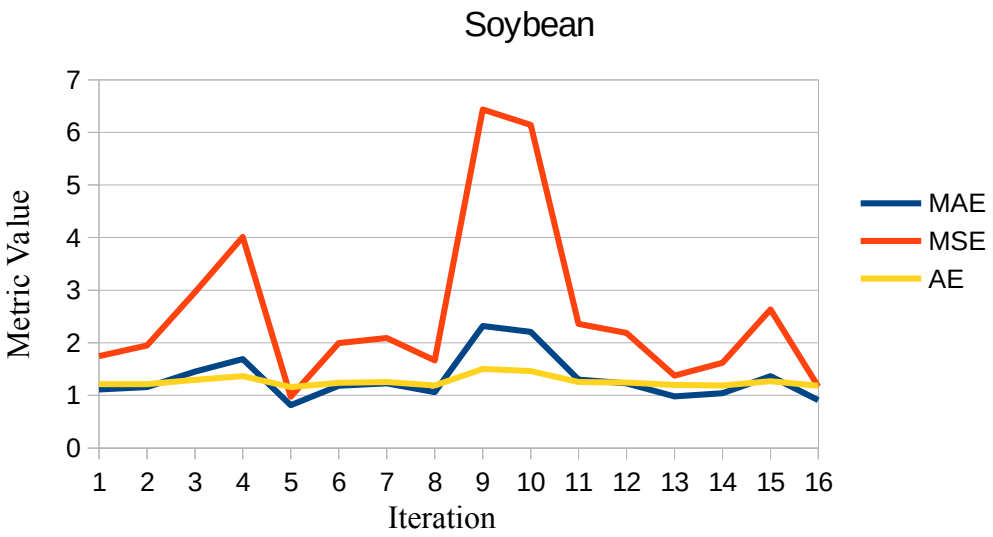

Figure 33: Hyper-parameter search of the soybean dataset.

Figures 30-35 have iterations on the x-axis and respective error values on the y-axis. A script was used for the random search, parameters that performed the best define the model that was check-pointed. For example, in Figure 33, iteration 5 shows promising results in comparison to the other iterations. The random search uses a simple greedy approach to save parameters which performed the best. In the case of Figure 33, soybeans, iteration 5 was the final saved model.

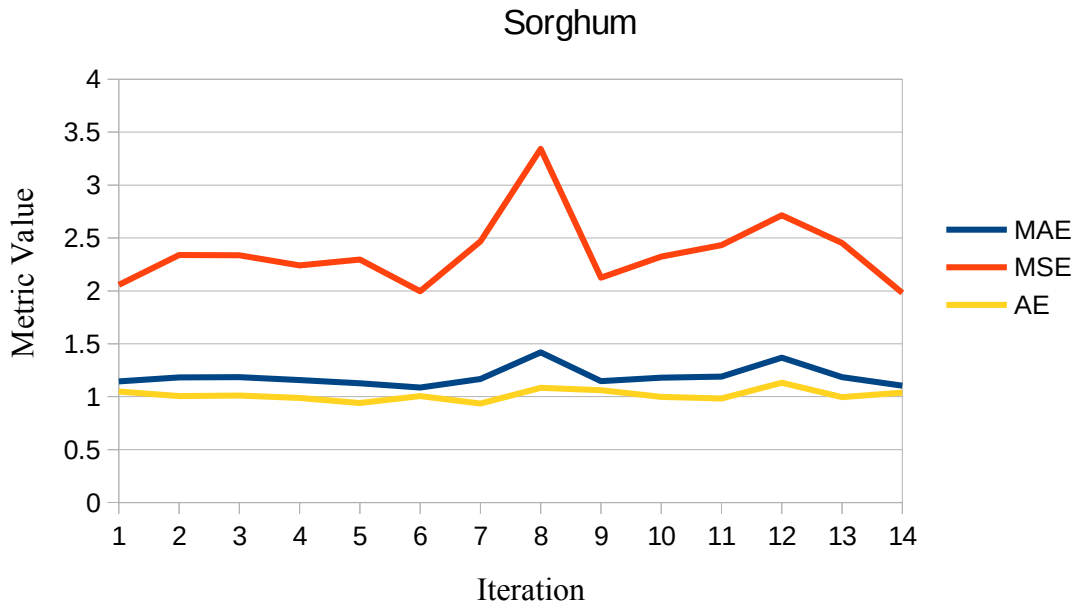

Figure 34: Hyper-parameter search of the sorghum dataset.

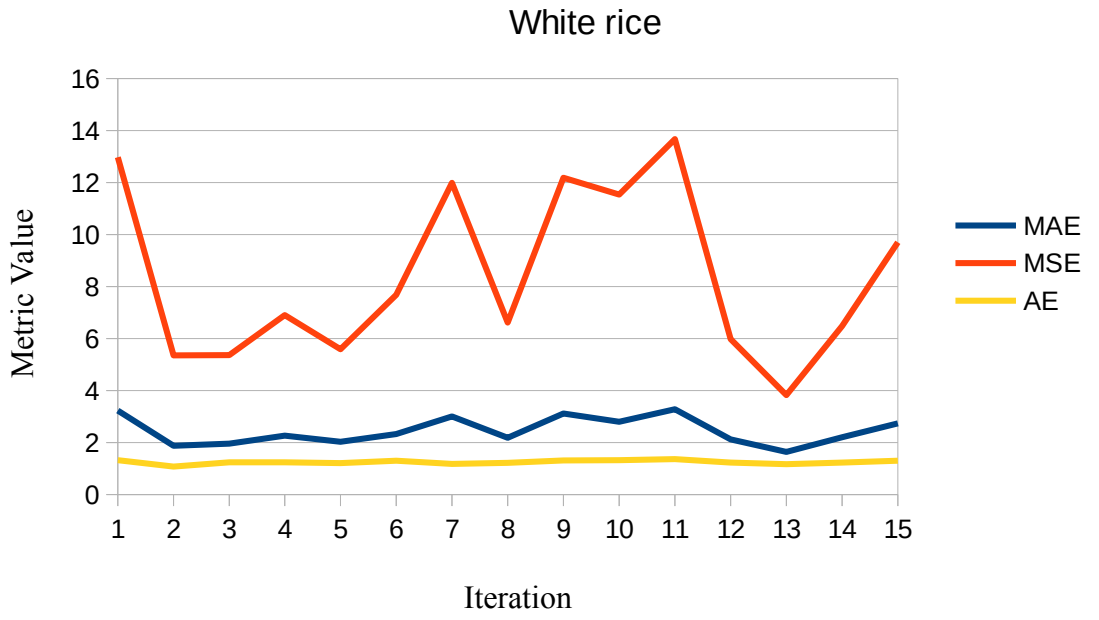

Figure 35: Hyper-parameter search of the white rice dataset.

Unsupervised Tiled Seed Counting

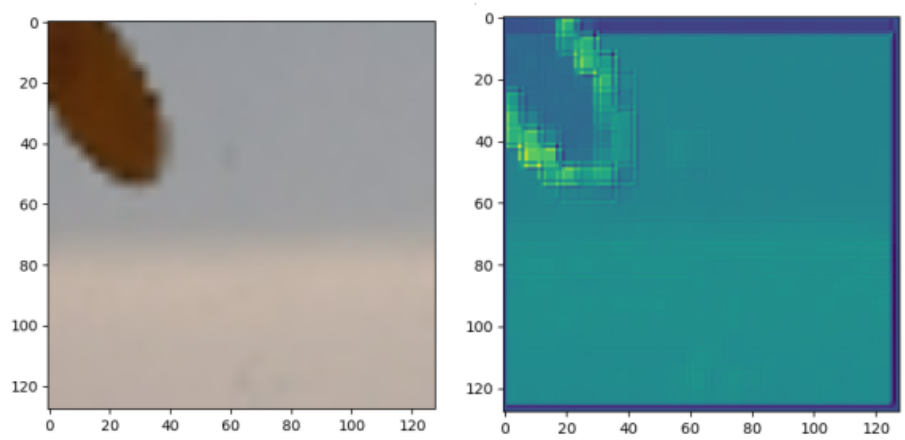

Figure 36: An input image and autoencoded feature map of a wheat tile.

This section explains an unsupervised autoencoder model that is trained on the six-class seed dataset. Similarly, for the supervised density count, this model is trained independently on all seeds along with a combined model where it is trained on all seed types. The metrics used are equivalent to the supervised density count model. The key differences between this model and the supervised density count model are the absence of labels, the pre-processing of images, and the post-processing count. The TensorFlow dataset adapter pre-processing resizes each image to 300x300 before tiling the image into 100 different tiles of 30x30 pixels from the original. These 100 tiles are resized to 128x128 images. They are batched together so that batches do not contain tiles of different images.

The encoder model reduces the dimensionality of the data given. In contrast, the decoder model uses up-sampling to recreate the data back to its original dimension. In TensorFlow, up-sampling is a simple nearest-neighbor approximation. Extensions to this network might use convolutional transpose layers to create a trainable version of up-sampling. Both sides of the

network utilize separable convolutions. After decoding, a zero-padding layer ensures the output returns 128x128 images.

Figure 37: Tiled depiction of a wheat image.

After the auto-encoding, the process of running an autoencoder, a post-processing step, calculates an estimated count based on min-max normalized local maxima of the model output. Correctly, after dropping the zero-padded borders, a greedy algorithm is used to search for the maximum vector. If there are no seeds in the tile, a background row may still be counted as a seed kernel. A more sophisticated model alleviates this. This model utilizes an intermediate Gaussian noise layer to produce noise between encoding and decoding steps; this introduces noise to the input and helps the model evaluate noisy data [35]. Figure 36 shows how the model output appears to dampen the lighting noise around the seed. Each batch contains one image but

100 tiles, the count is summed throughout a batch to represent the count of a given image. The metric evaluation is then calculated based on the input pair's label and the output of the post-processing. The model has two apparent potentials for inaccuracies in counting. The post-processing method has a few apparent inaccuracies. When one of the tiled images contains many seeds or splits a seed into sections, the model undercounts and over-count, respectively. Therefore, undercounting occurs more often for seeds with a major axis less than one hundred pixels, like canola. Overcounting occurs for seeds with a major axis greater than the one hundred pixels, which does not occur for this dataset. Depending on the positioning, the kernel may be split at a tiled boundary, such as in Figure 37. This boundary split may equally likely occur for any seed type. Because this model has a relatively distinct partitioning between the autoencoder and the counting algorithm, there are many avenues to explore when modifying this network. For example, another post-processing count algorithm was created based on the statistics of each tile.

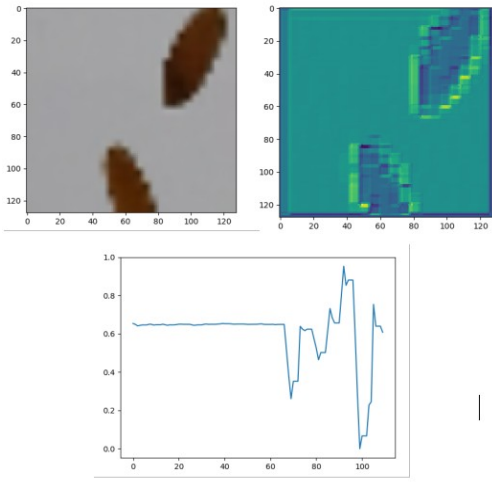

Figure 38: Intermediate outputs of the autoencoder model.

The post-processing algorithm is an essential part of the unsupervised autoencoder model. The fact that the model has no labels to train on means there needs to be a sophisticated method for processing the output image of an autoencoder to calculate a deeper meaning. Intuitively, the outlying peaks and crests of a given image are the objects, assuming the background is uniform. Autoencoders are known to recreate the most salient features of an image, which should, in this case, be the foreground seed kernels. As shown in Figure 38, the autoencoded image highlights the edges and inner contours of the seed kernels, while leaving the background relatively uniform. These sudden changes in the image produce a frequency graph, shown in Figure 39, where the baseline represents the background intensity. The local minima and maxima represent changes in the image due to the edges and inner contours of the seed kernels. Naively, one may process this vector and calculate the difference between the minimum point and the maximum point; using a predefined threshold, the user can estimate a given count. Although using a static threshold does not generalize well, some seeds or background types might require different thresholds. Specifically, if the background does include noise, this difference in noise within the background might satisfy the count threshold, and the model begins to over-count. There are multiple venues to improve the performance of this post-processing. One would be to eliminate the post-processing and attempt to create a secondary counting architecture. This approach is defined in the next section of this dissertation. Secondly, one could use the Laplacian to approximate the second derivatives of the frequency distribution and find the saddle points that are created from the differences in the contour and the background. Although, this may end up over-counting due to innate saddle points and differences on the seed kernel's contour.

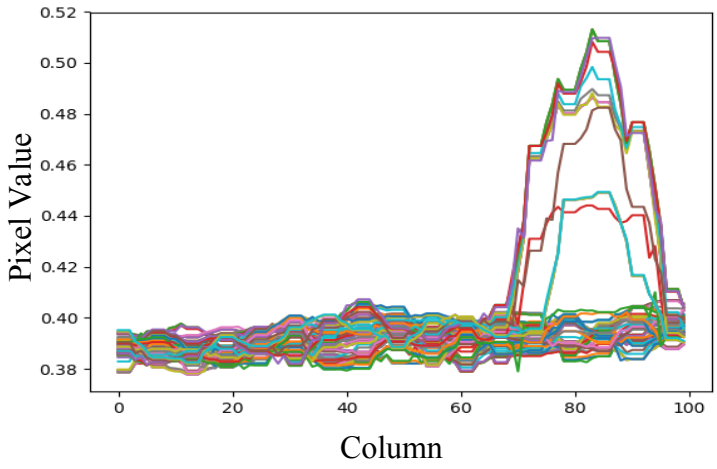

Figure 39: A spike visualizing the boundaries of a seed kernel.

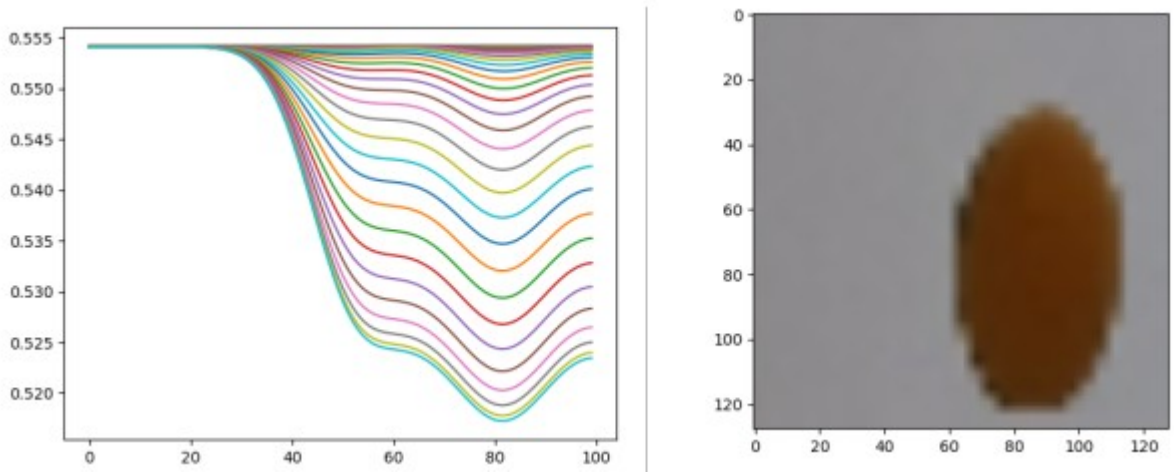

Figure 40: Gaussian blurring used on the edges of an image to reduce background noise.

A second post-processing algorithm, based on Gaussian blurring, reduces the noise in the row projection, visualized in Figure 40. A threshold based on the standard deviation is used to assume a count. If the standard deviation is above the threshold, then it is assumed there is a seed kernel in that tile. This model has similar issues to the previous post-processing methodology. However, a third model alleviates the need for a threshold to find the local minima and maxima.

The next section reveals a combined model that uses both an autoencoder and a density counting layer to train on integer labels.

Current state-of-the-art deep neural network models lack diversity in algorithmic approaches. Typically, models are scaled or tuned from previous models to increase performance on a given dataset. This section has delved into the possibilities of combining image processing techniques with deep neural network models. This combination of such techniques shows promising results that require less training time than state-of-the-art RCNN models. Training on one hundred tiles for a single image takes about 2.5 seconds. Therefore, training on one thousand images takes approximately 42 minutes. Appendix A includes a link to a downloadable pre-trained model for each kernel type. Along with pre-trained model checkpoints, there exists a script for training the model from scratch and loading the model from the given checkpoints.

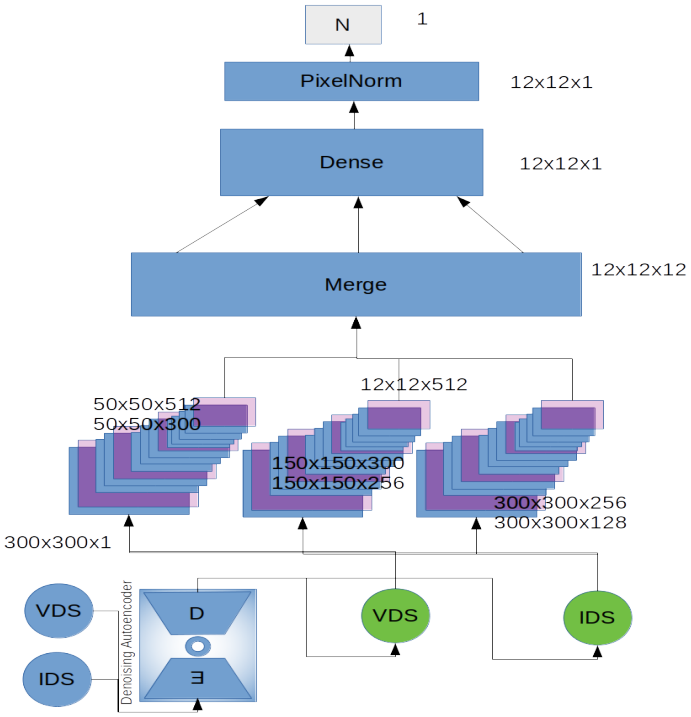

Figure 41: COUnt architecture, purple layers show max pooling.

This subsection introduces the Convolutional Online Undercomplete Autoencoder (COUnt) architecture. The Deep Learning book coined autoencoders to be undercomplete when the internal representation of the encoding has a smaller dimension than the input, which forces the autoencoder to learn 'the most salient features of the training data.' [35]. The COUnt model is a two-tier architecture that first fits a denoising undercomplete autoencoder, and is then fed into a convolutional counting model, which has similar characteristics to the supervised density counting model. This architecture aims at eliminating the need for a uniform background when detecting foreground objects at inference time, and the post-processing for unsupervised counting. Although, this model is not entirely unsupervised and does use labels on the second architecture tier. The previous section shows that autoencoders can count objects by post-

processing the output with an image processing algorithm. In replacement of an image processing algorithm, it is possible to use another network which is fed the output of the autoencoder to count. The architecture for this model first processes input images and trains an autoencoder. Both the validation and input training set are then converted into their autoencoded representations, which are fed into a new supervised density counting model. This secondary model is then trained and produces an integer count.

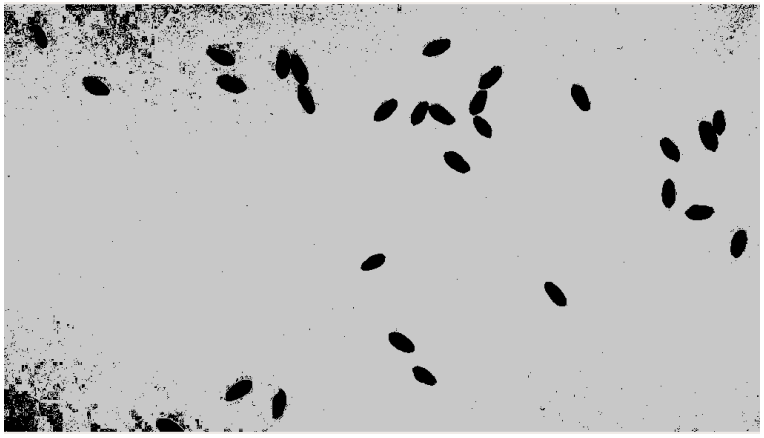

Figure 42: Ambient light detected after basic thresholding.

A well-established problem in image processing is the need to separate foreground objects from the background. This problem can be increasingly difficult with various lighting scenarios. Creating binary images tends to produce noise where light has scattered. Otsu's thresholding technique aimed at reducing this tendency by assuming a bi-modal distribution of pixel intensities and arithmetically finding the best threshold value to separate the two. Although, this assumes the background is of uniform color. Figure 42 shows an example of such lighting scenarios when not all scattered light can be differentiated from the foreground objects. Various denoising techniques were tested for the first-step autoencoder. The current network uses

dropout on the input layer to introduce noise. Similarly, one could use Gaussian noise or other noise layers. Dropout is a fundamental layer that prevents the over-fitting of models. When dropout applies to the input layer, neurons that represent pixels of the image convert to zero, dropped out of the image. The intuition behind this idea is to create disturbances in the image and train an encoding that can be decoded into the original image. Studies show that this auto-encoding is then more robust to noisy inputs. Batch normalization layers are used after convolutions to reduce internal covariate shift in parameter learning and decrease training time.

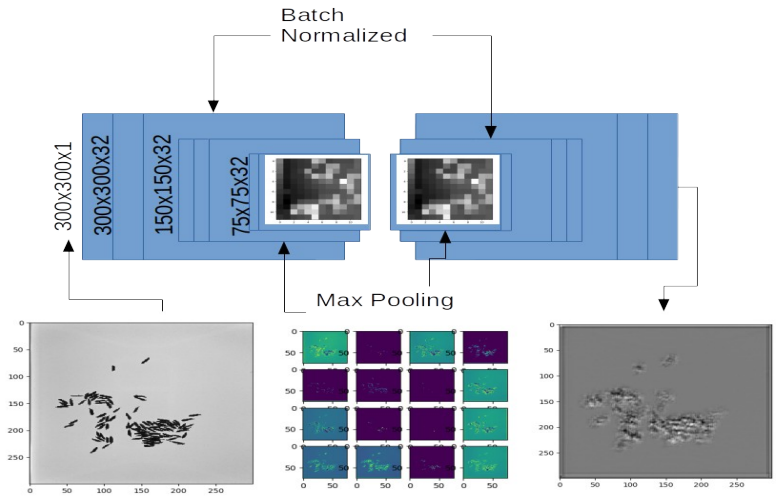

Figure 43: Architecture diagram of the autoencoder used in COUnt.

A novel idea behind COUnt is to use two different datasets of the same type of object. The first dataset has a uniform background, which allows the autoencoder to learn the salient features of the objects while introducing noise to strengthen feature recognition on noisy data. The counter network is then trained on the autoencoded uniform dataset and validated on the autoencoded noisy dataset. COUnt breaks the objective into two different architectures. The first model learns on a clean dataset and transforms the noisy dataset and the clean dataset into two new datasets that are autoencoded. The intuition behind this step is to reduce the noise created by

ambient lighting or various background patterns by using an undercomplete autoencoder trained on clean data. Assuming the undercomplete denoising autoencoder can learn the salient features of the objects in the clean dataset, the same objects within the noisy dataset are found.

The counter model is an updated version of the density count model. This second model is the supervised online step, which takes as input the autoencoded images with their labeled integer count values. The second model consists of three different convolutional pyramids that have varying stride shapes. Each pyramid is four levels of alternating separable convolution followed by max-pooling. Two variations of counting layers were tested for efficacy. First, a dense layer followed by a pixel norm and density counting layer were used to produce a count. Subsequently, the dot product was used to 'kill two birds with one stone' by calculating the similarity of the output feature maps of each pyramid while also normalizing the data (in replacement of the pixel norm layer). Each output feature map ends with a shape of 12x12x512. The next step is to calculate the dot product between each possible pair of pyramids. Therefore, the three outputted dot product layers have a shape of 12x12x12. Finally, these three layers are merged by averaging each value and applying the above density count layer to produce an integer. From preliminary results, it showed that the former layer had a smaller validation loss and is used in the results section.

By combining the supervised density counting model and the unsupervised autoencoder, this model creates an online learning approach that feeds and fits a model based on the autoencoded input data. The results section shows improvements in this model in comparison to the previous two.

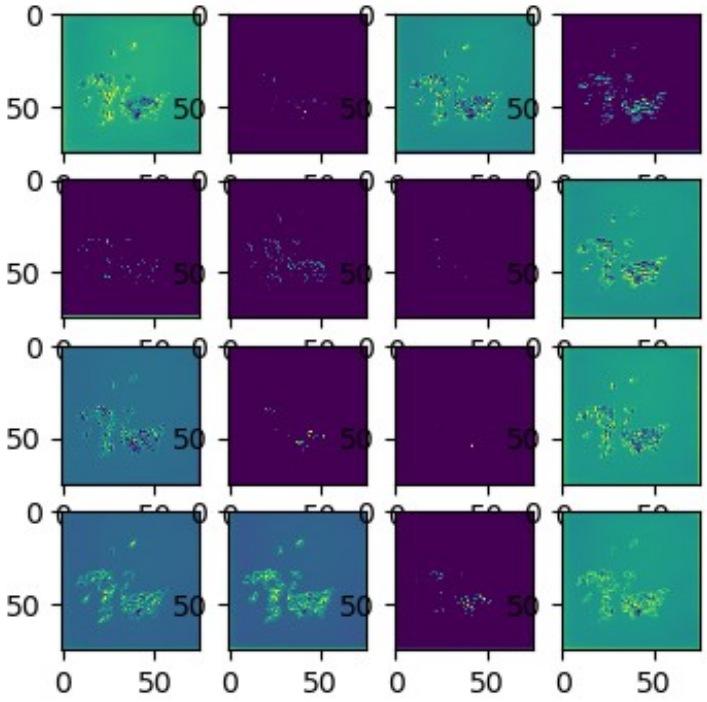

Figure 44: Encoded feature map images from the autoencoder.

Population of Kernel Sample and Data Source

Various grains were provided by local co-ops and the Plant Science department, including wheat, canola, grain sorghum, soybeans, white rice, and rough rice. Each seed class required three datasets for test, training, and validation. Each dataset is acquired from 60 frames-per-second (FPS) videos. Frames are extracted from the videos to be used as input to the model's training. However, because there is a low variance between frames the extraction happens per second instead of per possible frame. Links to the original videos and extracted frames are available in Appendix A.

A contour generating script annotates image data based on frames extracted from a high-speed video of seeds moving on a vibrating platter. The CPSL 3d-printed a novel phone-holder design for taking video of a lightbox [36]. A lightbox is a paper-tracing device for giving consistent lighting to a surface powered by a light-emitting diode (LED). Seed kernels are scattered onto the lightbox, which is secured above a vibrating grain feeder. The vibrating grain feeder has a potentiometer for controlling the amplitude of vibration and introduces noise and motion into the video. This type of setup reduces the human-effort required to reposition seeds for data-acquisition manually. Annotations are generated and stored onto Beocat, Kansas State University's local computing cluster.

The variance between population samples is essential, as there are many different types of kernels within each seed class. Each high-speed video randomly samples a varied count of seeds, from 10-50, from a respective grain bag. The varied count and a random sample of kernels (instead of reusing previously annotated kernels) is hypothesized to help the model generalize.

About the no-free-lunch theorem, a model cannot predict accurately on data it has not trained on [35]. A synthetic dataset is used to augment the amount of data fed to networks arbitrarily.

Generated images of randomly displaced and rotated ground-truth contour annotations are considered a synthetic dataset. These images are created programmatically rather than extracted from a video source. Synthetic datasets are convenient and are more flexible than video-source datasets as they can introduce positioning differences and optionally multiple backgrounds. Contours may be procedurally superimposed onto a set of images to augment a given dataset classification to different domains. Similarly, contours could potentially be taken from multiple datasets to intermix classifications. For example, two separate kernels such as wheat and corn may have independent video sources. However, after contour extraction, a generated training image can include both in the synthetic dataset.

A synthetic dataset generation is a semi-supervised approach that has an initial manual annotation stage similar to what is described in the Automatic Contour Generator section of Chapter 3. Running the script requires a few parameters: dataaug.py --dir reals --background bg --target target. The ‘dataaug.py’ is a python3 script containing the code for a synthetic generation. The ‘reals’ parameter is a path to a directory containing video-sourced images with filenames referring to class labels. An example of such a directory may include the following images: ‘soybeansA.png,’ ‘soybeansB.png,’ ‘whiterice.png,’ ‘roughriceA.png’ where these images are the seeds of the random generation and represent the ground-truth kernels for contour annotation. The ‘bg’ parameter is a path to a given background image or directory that is a background to the modified contours. Finally, the ‘target’ parameter is a folder name where the images are created. Notably, this should be the test, train, or validation for COCO annotation datasets. Therefore, to create a typical COCO TensorFlow record dataset, this script is run in

parallel for the train, validation, and test datasets. The run-time of this script is highly dependent on the number of contours within the images. For the synthetic dataset, each image takes roughly 40 seconds on an x86_64 Intel Xeon CPU E5-2690 v3 with 2.60GHz, which contains 48 CPUs with two threads per core. One-thousand images are generated for each class in each target dataset. Arithmetically, with six categories, 18000 HD images were generated, each about 1.8Mb in size. Amounting to 32.4Gb of data total and covering 2.7 days of computation (sequentially, it would take about 8.3 days). For distributing random background images, a ring is created to partition the generated number of images, N into K partitions, where K is the number of background images. Subsequently, the modular division is used to create a partition where:

$$N_i \bmod K = background\ index$$

Equation 7: Generating indices that correspond to background images.

Equation 7 shows a typical application for creating a circular array or ring. The background index correlates to the index of an array populated from a directory of images. Therefore, with this algorithm, an arbitrarily sized dataset can be created, allowing the user to scale to the needs of their network. Many of the experiments with the seed kernel dataset used 5000 images in training, validation, and testing each.

For the actual script processing, there is an initial supervised step where the contours are annotated. The user does a pixel-level threshold using LAB color-space. This step is a first pass at identifying foreground objects from the background, although some noise is still apparent. The thresholding UI is seen in Figure 24.

The second step is the same as in the Automatic Contour Generator section of Chapter 3, where the minimum area for noise is filtered, and clusters are identified. Finally, the third step is to bisect clusters greater than three objects manually. The result is the ground-truth data for

synthetic data. For each generated image, there are three main steps, loading, translating, and drawing. The loading step reads the ground-truth annotation and reshapes the data into a point-formatted NumPy array, values with (x,y) coordinates. An associative array is created where the keys are the original coordinates to each pixel of each contour, and the value is the translated position of the respective pixel. It is essential to keep this mapping to ensure the validation of annotation data. The annotated bounding boxes can be accurately translated without any need for manual annotation. If the image were modified by randomly placing contours in different spots, the annotation data would have to be recreated, which defeats the convenience of such a method. In terms of Euclidean transformations, rotations occur in three hundred and sixty degrees at random, while translations are within a boundary defined as:

$$
\begin{aligned}
maxX &= (columns - X_i)/2 \\
minX &= (X_i - columns)/2 \\
maxY &= (rows - Y_i)/2 \\
minY &= (Y_i - rows)/2
\end{aligned}
$$

Equation 8: Bounds of translation for each contour.

In Equation 8, (X_i,Y_i) represents the center contour pixel coordinate of a given annotated ground truth contour. This translation keeps the contour from translating outside of the image, which may not be necessary for all scenarios. Future editions may modify or remove the division. The computational bottleneck to this algorithm is the following step. A mask of the contours is created where each contour is represented by a filled gray highlight, and for each contour the above translation is calculated; afterward, for each pixel on the mask, a point polygon test is used to check if the pixel coordinate is within the currently iterated contour. If the check returns true, the associative array is populated with the ground-truth contour pixel coordinate and the translated pixel coordinate. The annotation is then updated with the given

transformation. After each annotation has been updated, and the mapping is complete, a new image (with the loaded background parameter) is created by iterating over the key-value pairs of the point map and copying the original pixels (using the key coordinates) to the value coordinates of the output image. Finally, a unique filename is created for the synthetic image and it is written to disk. Filenames include the seed class and the total kernel count. For example, a picture of one hundred soybeans would have a filename such as soybean23_100.png, representing the 23rd generated soybean image. At this time, there is no guarantee that seeds will assume a certain cluster.

Figures 46 show various results of synthesized seed clusters. This algorithm can be used to create a *quasi-infinite dataset*, a dataset only limited by storage space. The efficacy of the dataset is also limited to the total variation in the population of data. If the ground truth contours are mined from a small set of kernels that do not encapsulate the total population, a certain deep learning model would not generalize well for the population. Figure 45 shows some of the various backgrounds used within the background dataset. Other backgrounds included variations in typical lighting scenarios. Future expansion to this tool could bring interleaving class data into synthesized images and augmenting the background dataset.

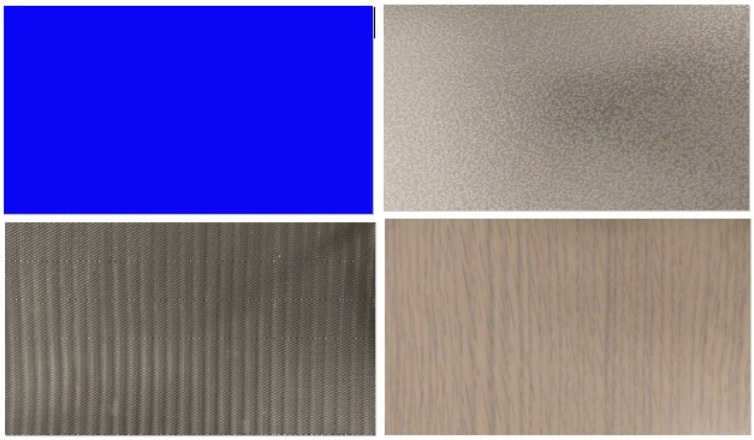

Figure 45: A subset of the various backgrounds used during data synthesis.

Figure 46: Examples of randomly generated rough rice clusters.

Results

Various metric definitions, neural network model results, and mobile application implementations are available in this section. Multiple experiments are shared to show the impact of these models on the seed kernel dataset. Figure 47 shows a sample of the different seeds in question. The tiled images are manually cropped and resized to fit the figure. Canola, being the smallest of kernels is almost spherical. White rice, an almost translucent grain, has proven to be difficult to detect in image processing algorithms. White rice is also relatively small for this dataset but still considerably larger than canola. What this dissertation calls rough rice is cypress long-grain rice, similar in size to the white rice but slightly larger due to the outer shell. Sorghum is also spherical yet much larger than canola. Soybeans are the largest kernel in this dataset and are also fairly spherical. Finally, along with white rice and rough rice, wheat is an ellipsoidal kernel that has a high area variability in its population. All of the described seed kernels have separate datasets that have been tested on all models. Similarly, a larger dataset was created that combines all of the data. This larger dataset is synthetically generated for all experiments. A noisy-background variant of this dataset is created to test the various networks on a non-back-lit background. All results were created using cross-validation by rotating the train, test, and validation sets and averaging the results. The All/Noisy-val row shows the evaluation of the pre-trained all network on noisy background data.

Originally, this research questioned the efficacy of object detection based on deep neural networks. It was assumed that state-of-the-art pre-trained networks would outperform

convolutional networks for counting. Surprisingly, the novel background invariant neural network model performs better in some areas than these models.

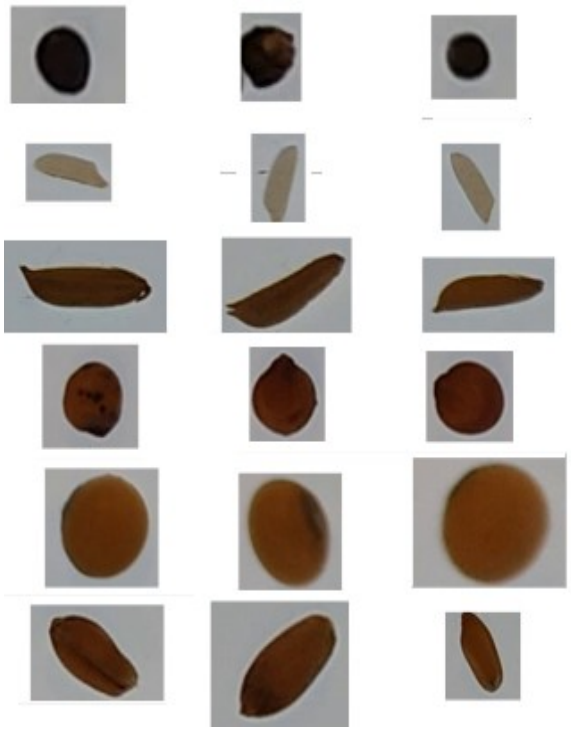

Figure 47: Single kernel examples from the dataset.

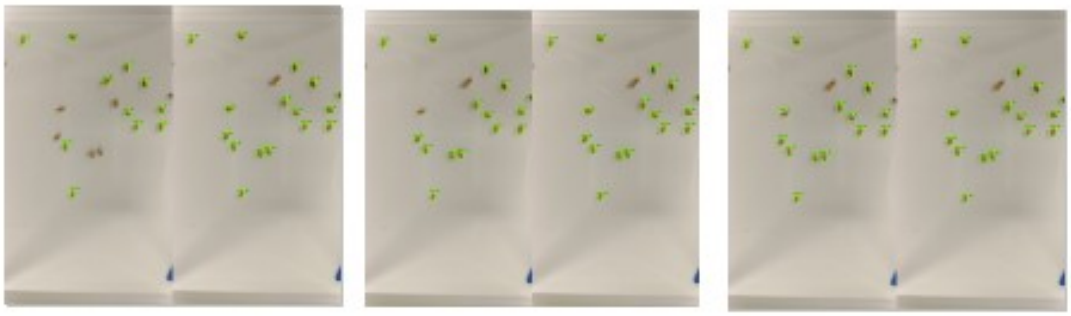

Figure 48: Examples of SSDMobilenet during training.

Figure 48 shows an example of SSD Mobilenet trained on only wheat kernels. These preliminary results determined the viability of SSD Mobilenet in terms of recognizing clusters.

For example, in the ground-truth images, it is shown that a cluster of two seeds is not labeled, and thus the network never learns to label this cluster. Therefore, a more fine-tuned labeling in the automatic annotator is adjusted for further results which included the bisection of clusters. This is an example of a model trained on wheat for high-resolution images (1920x1080). A similar experiment was run for low-resolution images (300x300) although there was not a significant difference other than the visualization of rendered boxes. Along with the comparison of low and high-resolution images, the labeling of crowds was tested. For example, in some cases, the labeled detection shows an annotation with two seed kernels, in the COCO dataset annotation format there is a key-value pair specifically for annotations with multiple objects, the results table for the low and high-resolution models contain this annotation when an annotated object contains multiple seeds. Although, preliminary results showed that at times the crowds were not detected or split. Table 2 shows the full results of the SSDMobilenet on the seed dataset. An auxiliary functionality of SSDMobilenet, in terms of this research, is the placement of bounding boxes. This research is focused on the actual numeric counting of objects, and thus the evaluation of such metrics is subtle. The metrics are described in the motivational section of Chapter 4. Mean average precision values with an intersection over union (IOU) of .50 shows that the model-placed bounding box intersects with only half of the annotated rectangle. Because the rectangles are generated using minimum area rectangles, there is little room for error caused by differences in margins. This index is reasonable for important counting because counting does not require the entire kernel to be recognized; these metrics are compared with the other deep neural networks. As the IOU value increases, the bounding box becomes more precise. The column that shows mAP is the average of all IOU indices and represents those bounding boxes that are perfectly fit, or very close to perfect. This column should be used for future evaluation of

minimum and maximum axis measurements. The more accurately bounding boxes can be placed, the higher confidence we gain on measurements.

Pre-trained model	mAP	mAP@.50	mAP@.75	AR@10	AR@100
Rough Rice	.587	.841	.414	.636	.668
White Rice	.333	.715	.212	.336	.423
Soybeans	.722	.960	.879	.738	.787
Sorghum	.535	.933	.561	.423	.583
Canola	.275	.741	.129	.397	.383
Wheat	.533	.939	.583	.395	.632
All	.381	.881	.538	.473	.616

Table 2: Result of SSD Mobilenet trained on various datasets.

From Table 2, it is clear that this model performed reasonably better on individual seeds such as Soybeans and Rough Rice. While seeds with smaller areas, like Canola, did not perform well. Also, much like the results from OneKK, white rice performed poorly. It appears that overlapping seeds and seeds near the boundary of the image perform poorly compared to completely segmented variants. Even after thirty thousand training steps, some independent kernels are not identified. Not surprisingly, kernels that are touching the boundary are not detected. When creating the dataset and preparing record files, annotations are skipped if outside the image. Skipping is a product of the TensorFlow record creator scripts, which filters all annotations that have bounding boxes outside of the domain or range of the annotated image. Figure 50 shows the results of various types of seeds run through this detector. These examples show when the model performs its best when seeds are typically not touching. It is apparent, such as in the example for wheat, that once seeds begin to touch, the detections perform poorly.

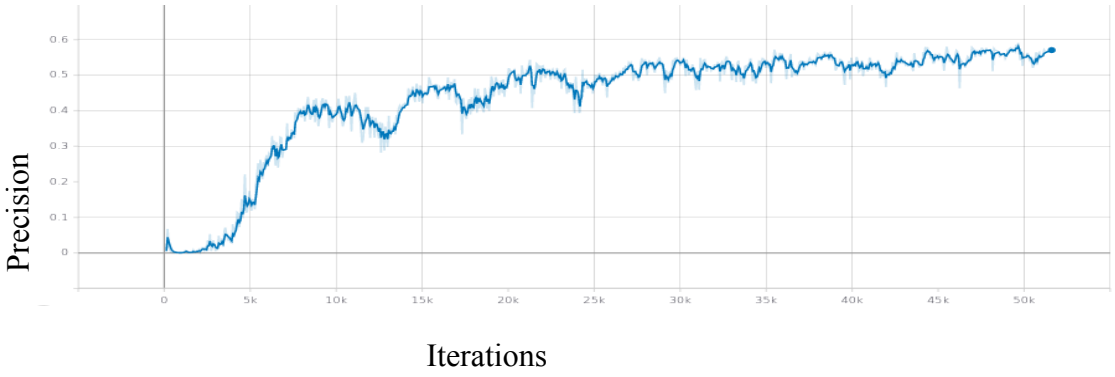

Figure 49: Graph of precision over time for the model.

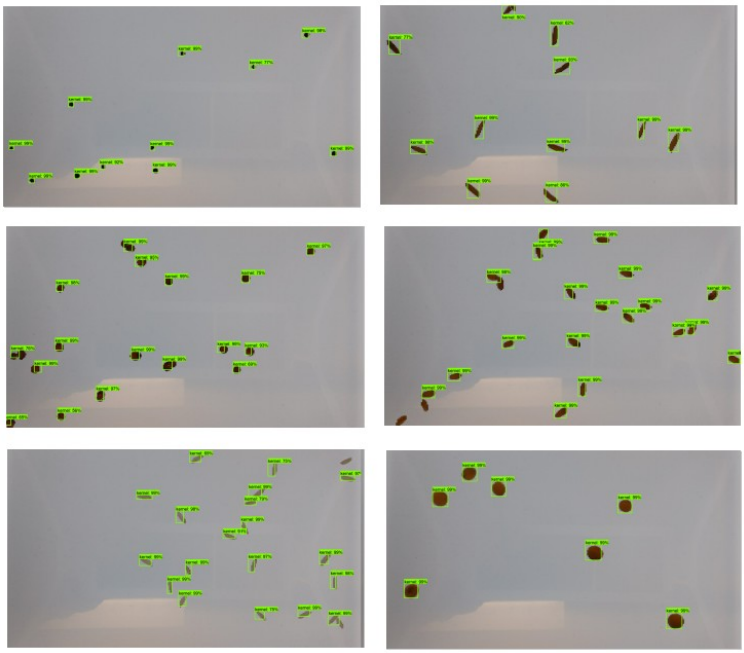

Figure 50: Results from the various independent models.

Metric	Formula
Actual-over-expected (AE)	$1/n(\sum_{i=0}^{n}(\frac{X^i_{actual}}{X^i_{expected}}))$
Mean Absolute Error (MAE)	$1/n(\sum_{i=0}^{n} X^i_{actual} - X^i_{expected})$
Root Mean Squared Error (RMSE)	$\sqrt{\sum_{i=0}^{n}(\frac{(X^i_{actual} - X^i_{expected})^2}{n})}$

Table 3: Various metrics used to evaluate count results.

Supervised Density Model	MAE	RMSE	A/E	Train Time	Final Validation RMSE
Rough Rice	1.69	2.11	1.25	6m41s	3.06
White Rice	26.7	27.3	4.23	7m28s	28.9
Soybeans	3.12	3.48	1.74	11m10s	5.83
Sorghum	2.07	2.47	1.19	11m1s	4.55
Canola	3.27	3.94	0.953	10m43s	12.35
Wheat	1.24	1.58	1.04	10m59s	2.81
All	2.33	2.85	1.17	2h38m	6.98
All/Noisy-val	12.2	18.3	2.82	-	-

Table 4: Results for the density counting network on all datasets.

For the supervised density counting model, it appears that almost all seeds contain an overcount, except for canola. The AE metric is defined as the actual count over the expected count. AE describes if the model over-counted or under-counted on average. Therefore, the best results should be close to 1, showing that the actual count is close to the expected count. Many of the results end up close to 1, except for the noisy dataset and the white rice dataset. Comparing this all-trained model to SSDMobilenet, it is not true that this model is significantly better. With

an average overcount of 1.17, we say that this model is, on average, counting 17% more kernels than the expected count.

Similarly, SSDMobilenet, on average, misses 12% of detections. Undercounting is not necessarily better than overcounting, and thus both results show a similar error, with the state-of-the-art models performing slightly better. After the training and evaluation of these models, the networks save via check-pointing for further tests. The subsequent experiments sought a solution for analyzing challenging datasets such as the white rice kernels. Correctly, the pre-trained counting-based models are used to evaluate on a disjoint dataset. Intuitively, when a model is trained on a dataset of primarily spherical objects, this model learns the salient features of a sphere. These features include the various edge orientations along an object's sides. Deep neural networks create feature maps that primarily capture these abstract traits by transforming the corresponding weight matrix during back-propagation of the loss function during training. Therefore, models that are trained on datasets of spherical objects are able to infer on other, yet similar, datasets of spherical objects. The following experiments use this knowledge to evaluate pre-trained networks on disjoint datasets of similar and dissimilar geometries.

Supervised Density Model (pre-trained validation)	MAE	RMSE	A/E
RoughRice-WhiteRice	2.63	3.1	1.16
WhiteRice-RoughRice	31.8	32.01	6.36
Sorghum-Soybean	5.29	5.49	2.04
Soybean-Sorghum	4.71	5.72	0.95
Soybean-Canola	2.56	3.005	1.32
Canola-Soybean	4.33	4.45	1.92
Sorghum-Canola	1.99	2.61	1.02
Canola-Sorghum	4.91	6.05	0.84
Wheat-WhiteRice	2.51	2.97	1.16
WhiteRice-Wheat	26.3	27.1	4.24
RoughRice-Wheat	1.58	1.98	1.11
Wheat-RoughRice	1.76	2.09	1.28
Soybean-WhiteRice	6.003	7.33	0.848
WhiteRice-Soybean	34.4	34.45	8.19

Table 5: Results of the cross-dataset evaluation of various seed datasets.

With the use of the supervised density count pre-trained models, the combinations of datasets are tested, and the results are shown in Table 5. Results show that the rough rice and wheat pre-trained network performed significantly better than the default white rice model. White rice, with an original overcount of 4.23, was reduced to 1.16 with the rough rice model. This result coheres with the original expectation that evaluation of ellipsoidal geometries benefits from pre-trained models that train on similar ellipsoids. While the soybean pre-trained model increased the performance for white rice, the other ellipsoidal models performed significantly better. Not surprisingly, the white rice model performed very poorly on other datasets. Another interesting observation is apparent overcounting on the sorghum-soybean model. The model that originally trained to count sorghum now overcounts soybeans. This overcount is potentially due

to the average area of both datasets, sorghum being a smaller kernel than soybean, the model counts higher due to the steep increase in pixels.

Supervised Density Model (pre-trained validation)	MAE	RMSE	A/E
RoughRice-WhiteRice	26.32	26.94	4.18
WhiteRice-RoughRice	**1.81**	**2.19**	**1.3**
Sorghum-Soybean	31.7	31.8	7.655
Soybean-Sorghum	3.77	4.47	1.43
Soybean-Canola	2.31	2.63	1.31
Canola-Soybean	**2.37**	**2.59**	**1.49**
Sorghum-Canola	1.49	1.88	1.01
Canola-Sorghum	20.7	21.4	3.62
Wheat-WhiteRice	20.9	22.0	3.55
WhiteRice-Wheat	**13.0**	**14.6**	**2.72**
RoughRice-Wheat	1.74	2.22	1.02
Wheat-RoughRice	1.78	2.13	1.28
Soybean-WhiteRice	5.1	6.1	1.02
WhiteRice-Soybean	30.5	30.9	7.39

Table 6: Results of the transfer learning on various seed datasets.

While the experiments in Table 5 show the results of a model trained and tested on two separate datasets, Table 6 is the result of feeding the tested seed type's training data as well. Therefore, the model learns on the dataset it is being tested on before evaluation. Surprisingly, this did not improve performance for all datasets. Those that were improved are highlighted in bold text.

Autoencoder Model	MAE	RMSE	A/E	Time	Final Validation RMSE
Rough Rice	2.39	2.95	1.25	41m32s	0.078
White Rice	8.71	9.56	0.249	39m21s	0.067
Soybeans	1.56	2.86	1.41	36m28s	0.073
Sorghum	1.16	1.58	1.01	36m37s	0.077
Canola	1.44	1.81	0.743	38m56s	0.064
Wheat	2.92	3.52	1.25	40m52s	0.069
All	7.69	10	0.315	40m12s	0.0813

Table 7: Results of multiple datasets run on the autoencoder model with various metrics.

Table 6 shows the results from the base autoencoder model with post-processing based on the normalized max vector. Not surprisingly, this model performed much worse than the supervised counter due to the many established possible errors. Because this method used a tiled dataset, it is possible that over-counting, in soybeans, was due to seeds being split into multiple tiles. The white rice results are by far the worse out of all models. Both the unsupervised and supervised models do have the benefit of counting objects on the boundary of images; whereas this is not possible on the state-of-the-art RCNN models.

Autoencoder Model	MAE	MSE	A/E	Time	Final Validation RMSE
Rough Rice	4.71	5.02	1.64	38m40s	0.063
White Rice	3.31	3.69	1.43	39m23s	0.058
Soybeans	5.47	5.65	2.39	38m30s	0.085
Sorghum	2.39	2.73	1.33	38m43s	0.088
Canola	4.65	5.98	0.89	39m35s	0.061
Wheat	2.51	2.91	1.27	38m55s	0.073
All	4.67	7.02	0.77	39m41s	0.063

Table 8: Results of the autoencoder model with the statistical Gaussian-based count.

Table 8 shows the results for the Gaussian-based autoencoder model. The model performed significantly better than the original autoencoder algorithm on the all dataset. Although, it did not perform better on all individual datasets. The main purpose of the Gaussian-based model was to reduce overcounting on noisy images where differences in the background can cause seed detections. Surprisingly, the Gaussian algorithm showed more overcounting than the original algorithm. That being said, the original algorithm required a manual parameter search for the threshold used on detections. Each kernel had a different threshold. While the Gaussian-based model does not require a specific threshold. Therefore, the Gaussian-based model generalizes better but is less accurate in some datasets than the original algorithm. Future studies may expand these results to explore other post-processing algorithms on autoencoders which are very promising for thresholding and denoising of images.

COUnt Model	MAE	RMSE	A/E	Time	Final Validation RMSE
Rough Rice	0.81	1.03	0.92	33m15s	0.83
White Rice	1.21	1.54	1.03	36m31s	1.19
Soybeans	0.691	0.872	1.08	32m43s	0.69
Sorghum	3.4	3.77	1.45	33m05s	3.31
Canola	3.36	4.19	0.84	32m55s	3.53
Wheat	1.42	1.81	0.9	41m59s	1.36
All	2.44	3.64	0.97	2h40m	6.98
All/Noisy-val	7.99	12.55	2.25	-	-

Table 9: Results of the COUnt model ran on all datasets.

The final results for this dissertation are in Table 9. These results show the evaluation of the COUnt model on all datasets. On average, the AE shows a decrease over all datasets. The model takes significantly longer to train as well. The last column of Table 9 shows the final loss on the validation data on the counting architecture of COUnt. The primary purpose of this network is to increase the performance of white rice and noisy dataset.

Again, the network in question is built to count autoencoded images. For the case of the regular datasets, a training split is used to train the autoencoder. The validation split is then fed through the trained autoencoder to create a new autoencoded dataset. This autoencoded dataset is then used as training data for the counting section of the architecture. The final test split is then used for metric evaluation, which has also been run through the autoencoder.

Therefore, the autoencoder can pick out the salient features of the objects in question, and the counting architecture then learns to count these salient features with a labeled value. This network shows considerable improvement for the white rice dataset and the noisy dataset. The supervised density counting network, unsupervised network, and COUnt architecture are all examples of novel contributions to the computer vision domain of computer science. The

supervised density counting network is a novel and effective way to train a model to produce an integer count from an RGB input image. The unsupervised network introduces a new domain of hybrid deep learning and image processing algorithms. This introduction appeals to the power and efficacy of image processing algorithms and the generality of deep learning models. Finally, the COUnt architecture effectively combines the unsupervised autoencoder model and the supervised density counting model to count objects in noisy scenarios. Furthermore, the final architecture can effectively count objects that have a foreground similar to the background, which is a significant contribution to high-throughput phenotyping.

Chapter 5 - Conclusion

The efforts and accumulation of work described throughout this dissertation have been fundamental for various research groups and focus workshops. Contributions to the state-of-the-art in computer science are discussed in this chapter. Significant contributions include new design architectures for binding fragment view objects to class fields, improved tools for data set acquisition, a novel seed-based dataset, and finally, a novel deep neural network for counting objects. The counting models are compared to state-of-the-art RCNN models and show improvements for object counting. Similarly, these networks are novel in two different ways. One model attempts to count objects completely unsupervised using a post-processing algorithm. Secondly, another model trains the network using an integer-based count instead of using a density mapping image. The applications within this dissertation have been proven useful and utilized around the world. The International Center of Potatoes and the Agriculture and Agri-Food Canada / Government of Canada will use the various Android applications for their workflow. Various plant research institutions such as ICRISAT in Hyderabad India have utilized these applications for their research. Survey has been funded for a new grant and is still in beta development but will be used for plotting fields at Cornell in Ithaca, New York. This dissemination of knowledge to various parts of the world has been a goal and achievement.

Figure 51: Application using SSDMobilenet to detect kernels.

Android libraries, applications, and Gradle plugins have been submitted to Github and the Android Studio plugins market. These applications together form a new age of productivity and possibilities for high throughput phenotyping. All code, algorithms, and datasets implemented within this work have been published online through encrypted files so that reviewers and those with access can monitor them. Some applications such as Verify and Intercross are freely available on the PlayStore and already have over ten-thousand downloads with over-average ratings. With source code also available on GitHub. This project fosters the manifesto of open-source communities and hopes these methods and practices can be extended to enhance their usages.

There are multiple opportunities to use state-of-the-art computer vision techniques, especially for counting and region of interest localization. Specifically, this research studied the

performance of the networks above for seed kernels within an image. This task is imperative to plant scientists for bagging seeds, and classically requires substantial human effort. The Computer Vision chapter explored the state-of-the-art network, SSDMobilenet, which showed better performance on a refined dataset of seeds in comparison to TensorFlow Zoo's evaluation on the COCO dataset. This result was an intuitive finding. The COCO dataset contains hundreds of classes with very complex shapes and sizes; however, the seed dataset is composed of relatively primitive geometric shapes.

Furthermore, the background for each image is uniform, whereas the COCO dataset has various backgrounds. Subsequently, the network trained for seed kernels still has relatively low accuracy in comparison to density-based convolutional models. Although both networks predominately err for seed types with small sizes and colors similar to the background. Overall, comparing RCNN metrics with counting metrics is not precisely one-to-one. However, the significance is shown through the performance of each independently. Future studies that wish to extend this research may decide to increase the dataset size, use transfer learning, or expand the range of the hyper-parameter search. Even though the counting networks performed better and can be trained faster than the state-of-the-art pre-trained RCNN models, there is still a need for such networks in high-throughput phenotyping. By no means do these results intend to show that RCNN models are unnecessary. Unfortunately, there is much room for improvement when it comes to the mAP scores. Once these models improve, they will be beneficial for applications that require precise measurements of generic objects such as various seed kernels. At the time of writing, it has been shown that the requirements of OneKK are not satisfied by such available networks. Class-specific image processing scripts outperform these RCNNs. The future of mobile RCNNs is bright, as networks such as MaskRCNN output actual contours and can be

used to measure objects. The IOU thresholding for these metrics is essential to ensure precise measurements are computed. The original hypotheses to the computer vision chapter were the assumption that these state-of-the-art models would perform better than counting based architectures by enumerating their detection lists. The results of this research show that counting based models outperform RCNN models, which contradicts the former hypothesis. Overlapping of objects tends to decrease the performance of such models. However, in a real-world setting, it is not correct to assume seed kernels do not overlap.

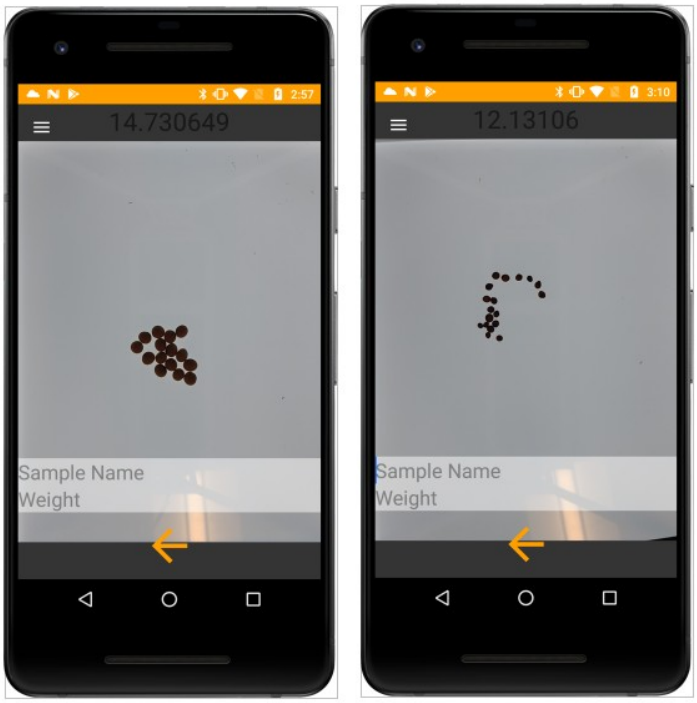

Figure 52: Supervised network used to count various kernels.

Critics may claim that such RCNN models are not trained long enough, or the data provided was specific for density-based models. Firstly, the results shown have thirty thousand steps of training, which is seemingly low, yet further training did not result in better performance in preliminary experiments. Because backgrounds are uniform, and the objects are relatively

primitive, the generated pre-trained models performed better than what is logged on TensorFlow Zoo. Secondly, the presumption that the dataset was tuned to work for density networks individually is lack-luster. The various images and datasets are created from lab-based settings, which reflect a daily process for plant scientists. Annotated images are created manually and synthetically; although, the difference between both was insignificant. An Android application in Figure 52 shows a necessary utilization of the SSDMobilenet model. Future developments might consider more sophisticated methods to analyze the detections of the model.

Currently, the detections are rendered to the screen but are surprisingly variant and unpredictable. Objects are bounded correctly at times, but the inference happens quickly, and a new, more imperfect, detection can be made soon after. A possible extension to using this detector is to wait for a steady-state, or if this is not possible, poll the detections for a certain amount of time and average the results to find a count. Figure 53 shows an Android application running the density counting model. There are no bounding boxes for this model as it is not RCNN based, but the top of the screen shows you a count, which is a float-represented value. The float value is off by .3, but rounding this number would give precise results. The floating-point representation is a result of the Android implementation. Surprisingly, this model was able to infer on metal screws, visualized in Figure 54, an object that it was never trained on. This result shows that the model can generalize well. Although, this is by no means a universal generalization, as the metal screw was somewhere within the bounds of the trained dataset. This application would not be able to count bananas or cassava roots. Because the metal screws are relatively ellipsoidal and about the size of rice or wheat, it performs as it was trained for these objects.

Figure 53: The application counting metal screws.

The data, models, and scripts are all available to reproduce and extend this research. Links may be found in Appendix A for respective downloads. In general, for each network, a training script is provided. The training scripts take three parameters: training directory path, validation directory path, and checkpoint name. For example, if the user wanted to retrain a model for wheat the command might look like: *python train_densitycount.py Data/train Data/val wheat.* A checkpoint folder will be created along with a .h5 model file. Evaluation of the model is similar, the evaluation script takes a single validation folder and a checkpoint name parameter. For example, *python test_densitycount.py Data/test wheat*. When training the model, the final validation loss can be compared with the results shown in this paper, finding a network or hyper-parameter set that improves the final validation loss would be helpful for this research. For training, the validation dataset is used to evaluate the model per-epoch, but this data is not

learned. Similarly, checkpoints and Keras model files have been included in Appendix A. If the user wishes to use these pre-trained models, they are available and can be loaded as standard models. The models can be loaded from a .h5 file, from .json, or .yaml. Pre-trained models include the hyper-parameters mined from the random search, although the training scripts will not have these parameters. Furthermore, TensorFlow lite models have been provided to run these models on an Android application. Base Android applications have been made available to load these particular models. The Android applications utilize the Camera2API to take pictures, the model is then loaded and run as a background process. The density-based network will output a float-value count result from the model, the autoencoder will return an image which is an encoded representation of the input.

Another aspect of extending these models is to explore the effectiveness of transfer learning. With the given script setup, the user can group training data and test on a single seed type. Therefore, the user may be interested in training on both soybean and sorghum before testing on soybean, which may help the model learn on circular geometries. In contrast, the user may train on all ellipsoidal kernels such as wheat and rice before testing on a data set with a similar structure. Previously, this dissertation made the finding that a model may be trained on a back-lit, uniform background-based dataset, and tested on a dataset with random backgrounds, increasing the performance in contrast to training on a noisy, random background, dataset. This is a fundamental improvement to image processing-based algorithms which require a noticeable difference between foreground and background objects. The results show that the base density supervised model performs slightly better on the noisy background model when trained on a uniform background all seed dataset. Similarly, the COUnt model in this dissertation has improved counting on objects that have a color similar to the background. Specifically white rice

in question has been a troublesome target for many image processing algorithms and even performs poorly on state-of-the-art RCNN models.

Major contributions within this dissertation pertain to the development of tools, maintainability of code, along with the quality of data and methods for data acquisition. To push the boundaries for the development of food and nutrition there is a need to establish research methods and tools to guide scientists. The methods and tools described in this dissertation accommodate the modern needs of the scientists developing such research techniques. Along with the developed tools, the ideologies of data quality and acquisition support future productivity in model training. The models and algorithms developed for computer vision create a new avenue of research for further development. Instead of focusing on RCNN based models, there needs to be a focus on density-based models and unsupervised approaches. Although unsupervised models essentially do not require labeled data, the need for annotations will always be prevalent. The annotation tool provided in this dissertation solves the issues of redundancy in typical manual object annotating. Altogether, these tools improve the modern research state-of-the-art and plan a future for automation and scalability of plant development to further nutritional growth and diversity.

Bibliography

[1] T. Rife, J. Poland. (2014). Field Book: An Open-Source Application for Field Data Collection on Android. Crop Science. 54. 10.2135/cropsci2013.08.0579.

[2] C. Courtney, M. Neilsen. (2018). A seed segmentation contour generator and counter. 31st International Conference on Computer Applications in Industry and Engineering, New Orleans, LA, Oct. 2018

[3] Codd E.F. (2002) A Relational Model of Data for Large Shared Data Banks. In: Broy M., Denert E. (eds) Software Pioneers. Springer, Berlin, Heidelberg.

[4] SQLite, www.sqlite.org/index

[5] Haerder, T. and Reuter, A. (1983). Principles of Transaction-Oriented Database Recovery. Computing Surveys 15, 4, pp.287-317.

[6] van Brummelen, Glen Robert. (2013). Heavenly Mathematics: The Forgotten Art of Spherical Trigonometry. Princeton University Press. ISBN. 0691148929.

[7] Android GNSS Raw Measurements Library https://developer.android.com/guide/topics/sensors/gnss

[8] C. Courtney, M. Neilsen, T. Rife. (2017). Mobile Applications for High-throughput Phenotyping. 31st International Conference on Computer Applications in Industry and Engineering, San Diego, CA.

[9] Emlid Reach RTK Documentation, https://docs.emlid.com/reach/

[10] Android Jetpack Documentation, https://developer.android.com/jetpack

[11] M. Neilsen, C. Courtney, S. Amaravadi, Z. Xiong, J. Poland, T. Rife. (2017). A Dynamic, Real-time Algorithm for Seed Counting. In 26th International Conference on Software Engineering and Data Engineering.

[12] G. Bradski. (2000). The OpenCV library. Dr. Dobb's Journal, 25(11), 120-125.

[13] Lin, T. Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., … & Zitnick, C. L. (2014). Microsoft coco: Common objects in context. In *European conference on computer vision* (pp. 740–755). Springer, Cham.

[14] NMEA documentation, https://www.gpsinformation.org/dale/nmea

[15] Android Documentation, https://developer.android.com/docs

[16] Y. LeCun, L. D. Jackel, B. Boser, J. S. Denker, H. P. Graf, I. Guyon, D. Henderson, R. E. Howard, and W. Hubbard. (1989). Handwritten digit recognition: Applications of neural net chips and automatic learning. *IEEE Communication*, pages 41-46, invited paper.

[17] A. Krizhevsky, I. Sutskever, and G. Hinton. (2012). Imagenet classification with deep convolutional neural net-works. In Neural Information Processing Systems (NIPS).

[18] M. D. Zeiler and R. Fergus. (2014). Visualizing and understanding convolutional neural networks. In the European Conference on Computer Vision (ECCV).

[19] K. Simonyan and A. Zisserman. (2015). Very deep convolutional networks for large-scale image recognition. In the International Conference on Learning Representations (ICLR).

[20] Szegedy, Christian, Reed, Scott, Erhan, Dumitru, and Anguelov, Dragomir. (2014). Scalable, high-quality object detection. CoRR, abs/1412.1441, 2014b. http://arxiv.org/abs/1412.1441.

[21] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang,T. Weyand, M. Andreetto, and H. Adam. (2017). Mobilenets: Efficient convolutional neural networks for mobile vision applications. arXiv:1704.04861.

[22] V. Badrinarayanan, A. Kendall and R. Cipolla. (2017). SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation. In IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 39, no. 12, pp. 2481-2495.

[23] K. He, G. Gkioxari, P. Dollár and R. Girshick. (2017). Mask R-CNN. In IEEE International Conference on Computer Vision *(ICCV)*, pp. 2980-2988. doi: 10.1109/ICCV.2017.322

[24] S. Ren, K. He, R. B. Girshick, and J. Sun. (2015). Faster r-cnn: towards real-time object detection with region proposal networks. In Neural Information Processing Systems (NIPS), pages 91–99.

[25] T.-J. Yang, A. Howard, B. Chen, X. Zhang, A. Go, V. Sze,and H. Adam. (2018). Netadapt: Platform-aware neural network adaptation for mobile applications. The European Conference on Computer Vision (ECCV).

[26] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, Quoc V. Le. (2019). In the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 2820-2828

[27] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, Song Han. (2018). In the European Conference on Computer Vision (ECCV), pp. 784-800

[28] Y. Chen, T. Krishna, J. S. Emer and V. Sze. (2017). Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. In *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127-138.

[29] Ramachandran, P. Zoph, B. Le, Q.V. (2017). Searching for activation functions. arXiv, arXiv:1710.05941.

[30] Lempitsky, V., & Zisserman, A. (2010). Learning to count objects in images. In *Advances in neural information processing systems* (pp. 1324–1332).

[31] Chen, K., Loy, C. C., Gong, S., & Xiang, T. (2012). Feature mining for localized crowd counting. In *BMVC* (Vol. 1, 2, p. 3).

[32] Weidi, X., Noble, J. A., & Zisserman, A. (2015). Microscopy cell counting with fully convolutional regression networks. In *1st Deep Learning Workshop, Medical Image Computing and Computer-Assisted Intervention (MICCAI)*.

[33] Lehmussola, A., Ruusuvuori, P., Selinummi, J., Huttunen, H., & Yli-Harja, O. (2007). Computational framework for simulating fluorescence microscope images with cell populations. *IEEE transactions on medical imaging*, *26*(7), 1010–1016.

[34] Chan, A. B., Liang, Z. S. J., & Vasconcelos, N. (2008). Privacy preserving crowd monitoring: Counting people without people models or tracking. In *2008 IEEE Conference on Computer Vision and Pattern Recognition* (pp. 1–7). IEEE.

[35] Deep Learning (Ian J. Goodfellow, Yoshua Bengio and Aaron Courville), MIT Press, 2016.

[36] Siddharth, Amaravadi. (2018). Mobile applications for high-throughput seed characterization. Master's Report. Kansas State University.

[37] Chaney Courtney and Mitchell Neilsen. (2019). Intercross: a breeding application for high-throughput phenotyping. In Proceedings of the 32nd International Conference on Computer Applications in Industry and Engineering
Vol. 63: pp. 72-79.

[38] Shanshan Wu and Mitchell L. Neilsen. (2019). Augmented reality for high-throughput phenotyping. In Proceedings of the 17th International Conference on Scientific Computing.

[39] Chaney Courtney and Mitchell Neilsen. (2019). Vetting anti-patterns in Java to Kotlin translation. In Proceedings of the 34th International Conference on Computers and Their Applications.

[40] Trevor Rife, Siddharth Amaravadi, Megan Calvert, Shravan Gangadhara, Sandra Dunckel, Mitchell Neilsen, and Jesse Poland. (2018). OneKK: A high throughput seed phenotyping Android application. In Proceedings of the National Association of Plant Breeders 2018 Annual Meeting, Guelph, Ontario, Canada.

[41] Mitchell Neilsen, Shravan D. Gangadhara, and Trevor Rife. (2016). Extending Watershed Segmentation Algorithms for High-Throughput Phenotyping" in Proceedings of the 26th Intl Conf on Computer Applications in Industry and Engineering, Denver, CO.

Appendix A - Repositories

The appendix is organized in repositories. All available code, downloads and other data are available online.

Intercross

Intercross is an Android application developed in the Kotlin programming language. This application is discussed within this dissertation and pertains to hierarchical data management of bar-coded plant samples.

Link: https://github.com/PhenoApps/Intercross

Verify

Verify was mentioned in this dissertation. Verify is an Android application for storing bar-code ID's with meta-data.

Link: https://github.com/PhenoApps/Verify

Survey

Survey is another free Android application available for download online. The current version is in beta.

https://github.com/PhenoApps/Survey

Furthermore, the experiment used within this dissertation is available in the following link.

Password: spamDataset

https://github.com/chaneylc/Data/blob/master/spam2.zip

Rangle

The root angle calculating application is available in the following link.

https://github.com/chaneylc/Rangdroid

Image processing algorithms using OpenCV are available in the following link.

Password: r4ngle

https://github.com/chaneylc/Data/blob/master/rangle_script.zip

Due to storage sizes, the rangle dataset is only available on request.

Preference Fragment

Password: prefManager

https://github.com/chaneylc/Data/blob/master/preferences.zip

Annotator

Password: annotations

https://github.com/chaneylc/Data/blob/master/annotator.zip

Supervised Density Count Models

All pre-trained models and .h5 files are available:

https://github.com/chaneylc/Data/

Their names are training_densitycount_kernelname3.

The password for each is the ‘kernelname’

Scripts for training and testing the supervised model, and transfer learning are here:

https://github.com/chaneylc/Data/blob/master/supervised_scripts.zip

Password: scripts

Unsupervised Models

All pre-trained models are available:

https://github.com/chaneylc/Data/

Their names are training_2_kernelname.

The password for each is the ‘kernelname’

Scripts for training and testing the models are here:

https://github.com/chaneylc/Data/blob/master/unsupervised_models.zip

Password: scripts

COUnt Model

All pre-trained models are available on request.

Scripts for training and testing the models are here:

https://github.com/chaneylc/Data/blob/master/count_models.zip

Password: scripts