MTG CARD GRADER USING IMAGE CLASSIFICATION AND DETECTION

Juan Carlos Sosa Hernandez

Jsosa913@stanford.edu

Abstract

Given Magic the Gathering's (MTG) incredibly profitable secondary market based on given cards and their subsequent quality, I was inspired to build a card grader to make accurate prices on individual cards easily available to would-be sellers. To that end, I built a number of object classifiers meant to give users a better prediction as to how much any given card will actually sell for and an object detector to point out flaws and damage on those cards to justify the card grade given with some mixed but promising results.

Introduction:

Magic the Gathering, henceforth referred to by the acronym MTG, is a very popular trading card game from the early 1990's that has only become more popular with recent digital releases to the game. In addition to a strong Esports presence, MTG has birthed a new generation of player and rekindled lost interest in older enfranchised players. With this new popularity, players who are now or have been playing on tabletops (i.e. with real cardboard cards) for tournaments or casually play have to deal with the existence of the game's secondary market to obtain older and more competitive cards that are currently out print. The scarcity and rarity of these cards has birthed the need for resellers to obtain and provide stocks of these cards to anyone looking for these cards, one such, which I will be using for reference, is TCGPlayer. These types of marketplaces and their prices are defined by the specific cards being sold, with rare, useful and limited print cards being considerably more costly, and the quality of those cards, with higher quality 'mint'-condition cards being more desirable. Of these, specific cards with their respective quality grade have a well-defined price range in each marketplace, however the actual quality grading is much more loosely defined, more often than not being decided by the reseller when you sell a card. While there does exist card grading services, these can cost as low as 50$ for less expensive premium cards and scale upwards with the card's price range. For cards like Black Lotus, one of the most premium and rare cards in MTG history being famously sold for $166,100, a difference in a single card quality grade can cost upwards of $2000 so these services are justifiable in these kinds of cases. However the majority of the market is dominated by cheaper to mid-costed cards (ranging from a couple of cents to $100) whose card grades can dictate differences in a couple of dollars, easily not worth the $50 price tag for grading. However, when larger collections are sold or

the price difference is enough to warrant a seller's attention, these small dollar price changes can provide significant information as to whether it's worth it for a seller to do so at that point in time. For this purpose, I decided to fill in this hole by providing an easy-access card grader that given a user image of a card can output it's quality grade to better inform would-be sellers as to how much they can expect in profits to better plan ahead. I separated this project into two categories, image classification which would simply meet my afore mentioned goal, and object detection, which would point out flaws in cards that can be used to justify the grade given and help inform users as to what exactly constitutes each grade category. Both of these would take a simple image input and for classification output its subsequent grade, and for detection output a box enclosing acknowledged card flaws borrowed from the TCGPlayer grade classification rubric. While the focus of this project seeks only to grade MTG cards, this can easily be transferred to other trading card games or markets that necessitate any form of accessible card-grading.

Related Work:

For this specific purpose I was unable to find any kind of previous work, so my entire approach was based off concepts and models explored both within the context of my CS230 class and on other professional uses. For image classification I based two of my models (Fully Connected and Convolutional Neural Networks) directly from the concepts learned in CS230, and one (ResNet50) from [1] which inspired me to try this model to compare to my hand-crafted models. For object detection I used the publicly available Tensorflow Object Detection API [2] to define and construct my model in addition to this article [3] on the inception model I used for transfer learning. Given that this task would normally be done by eyeing up card damage, there exist multiple rubrics which are used to grade cards in different contexts. For the purpose of easing potential transition into a market use, I opted to base my rubric from TCGPlayer [4], noting their classifications and what constitutes each classification.

Dataset and Features:

I was unfortunately unable to find a pre-made dataset of MTG cards with varying amounts of quality to use for learning purposes. As such, I instead opted to gather my own by purchasing 500 Mint/Near Mint Basic Forests (264, Kaladesh) from TCGPlayer and 5 from my personal collection to make up a total of 505 individual samples. Given this rather small dataset, I opted to separate these cards into the following 4 of the 5 categories available which provided the most notable differences between grading; Near Mint (NM), Lightly Played (LP), Heavily Played (HP) and Damaged (D). While it would have been preferred to obtain naturally worn cards, I was unable to obtain a large enough stock of this card to purchase, so instead I artificially damaged these cards to match their assigned category. The damage caused per class is as follows. The 125 cards in (NM) were simply taken from the original 500 (NM) card order as it can be safely assumed that TCGPlayer would abide by their own grading standards. The 125 (LP) cards were then taken from the (NM) original order and shuffled 300 times in different ways (Riffle and Overhand Shuffling) with varying levels of intensity with the goal being to cause some natural wear and tear. The damage caused for this was mostly light corner and edge wear (i.e. whitening on the black border). The 128 (HP) cards, in addition to suffering the same damage as (LP), had between 1 to 5 scratches made by using an X-acto knife in different places on the face of the card. The final 127 (D) cards, in addition to being (HP) also had one of the following major forms of damage; a large scuff (peeled off area in the face of the card), a tear near the edge of the card, a

crease going across a large/notable portion of the card, a missing portion of the card near the edge or a hole on the face of the card. Having then created my dataset, I then moved to digitizing it to feed into my models. These cards were placed in a uniform surface and using a Samsung Galaxy S8's camera, I proceeded to take pictures, swapping in each card trying to maintain as uniform of images as I possibly could. These images were originally 3024 x 4032 pixels, but due to hardware and time constraints, I scaled these images down by 12.6 to 240 x 320 pixels, the largest size I could reliably feed into my models while maintaining important card and damage details visible. These cards were then separated into a training set of 405 cards (100 (NM), 100 (LP), 103 (HP) and 102 (D)) and testing set of 100 cards (25 of each) with their classification encoded into a one-hot format. Given early results showing a notable amount of variance between testing and training accuracy, I also applied some data augmentation by varying the brightness present in the image to better represent what a user's actual photo might look like as well as in the hope that it'd help the models generalize what a card and it's damage could look like. I also created a smaller set of 50 images, the largest amount I could feed into the model without crashing it, to annotate with boxes around damage to feed to the Object Detection inception model. Image data and examples can be seen in the Appendix.

Methods:

For the Image Classification problem, I created 2 models, a Fully Connected Neural Network, and a Convolutional Neural Network, and borrowed a pre-made proven to work model, a ResNet50 model that comes built into the Keras library. For the Object Detection problem, I borrowed a pre-trained inception model from the Object Detection API and applied some transfer learning given that I had neither the data nor the time to train a wholly new object detection model. The image classification models were built in Tensorflow 2.0 [5] and the ResNet50 model in Tensorflow 1.15 and using Keras for modelling and loading the dataset. These models each took as input an array of shape (batch size, 240, 320, 3) given that these images were RGB and outputted a one-hot encoding array that signified the class predicted. These predictions were then compared with a similar 4-class one-hot encoding that made up the labels for these sets.

The FCNN model is made up of a Flatten layer followed by 6 Dense layers of decreasing number of nodes culminating into a 4-node layer outputting with a softmax activation to denote the classification of the images. This model uses a simple Stochastic Gradient Descent optimizer with learning rate of $5*10^{-4}$ and uses Nesterov Accelerated Gradient with .9 Momentum, to help it avoid oscillations and quickly converge to minimize runtime, and calculates loss using Categorical Cross Entropy. This model seeks to maximize accuracy and minimize loss over the course of 100 epochs and usually finishes training within 15 minutes.

The Convolution Model is made up of 4 Convolutional layers each followed by a Max Pool layer to sharpen the image as much as possible for the next convolution and a Flatten layer followed by 6 Dense layers similarly culminating into a 4 node softmax layer to output class. The first convolution layer starts with eight 5 x 5 filters and the remaining convolutions use double the previous number of filters (16, 32 and 64 respectively) of size 3 x 3, each keeping the same padding. Similar to the FCNN, this model uses the same SGD optimizer with a learning rate of $1*10^{-2}$, CCE loss function and also optimizes for high accuracy and low loss. This runs for 60 epochs and runs within an hour and half (1:30).

The final Image Classifier model is the pre-built ResNet50 model imported from Keras. I opted to have this model start empty for the sake of seeing how it would learn from the ground up using only my dataset. This model uses the architecture described in [1] and trains with the same optimizer with a learning rate of $1*10^{-3}$, loss function and metrics as the other two. This model trained for 50 epochs and finished in about 5 hours.

The Object Detection Inception model borrowed from the Object Detection API uses the same architecture described in [3] trained on the COCO dataset [5]. This model trains with an SGD model that changes learning rate given the epoch range it is in, for my purposes it was only $1.9*10^{-4}$ on an accuracy and loss metric. This model trained for 1641 epochs and was left to run overnight so I was unable to calculate the time it took to run.

The architecture for the hand-made Image Classifiers is available in the Appendix.

Results:

After training, these models achieved the following results. The FCNN achieved a maximum of 94% accuracy during training and on testing obtained 90% accuracy on the training set and 86% on the test set. The Convolutional Neural Network achieved an accuracy of 96% during training and when testing obtained 94% accuracy on the training set and 88% on the test set. The ResNet50 model achieved a maximum accuracy of 94% during training and obtained a 94% on training data and 84 on testing data. The Object Detection model achieved a 75% accuracy with IOU >= .5 and a general accuracy of IOU >= .9 of 33%. Graphs are available in the Appendix.

Analysis:

Overall, I'm satisfied with the results obtained from these models, given that the dataset I worked with was small and that the hardware I used to train and test these models was fairly limiting (no GPU and fairly small amount of free memory to use for calculations etc.). Regardless, I still want to point out several issues and disclaimers relating to this data and models. In particular, I want to note that the Image Classification data still had a notable amount of variance present in training and testing accuracy which could definitely be solved with a larger and more complex dataset and possibly with more augmentation (more damage variations and image quality etc.). The results from the Object Detection model are the most depressing given the general lack of accuracy, but it should be noted that the expected training epochs was within the 100000 mark of which I was a whole $1/100^{th}$ of. Given enough time it might have achieved more acceptable results, but given the limited training data given, I also expect a larger dataset to perform considerably better. The relevant hyperparameters for these models were mostly the number and types of layers, the optimizer, and its relevant values (learning rate and momentum) and the loss function. Of these the number and types of layers was largely based on the actual amount of processing my computer could handle so larger neural networks like the ResNet50 might perform considerably better, and probably do, when trained on other hardware. The optimizer I settled on, SGD, was chosen for its speed in converging. I also tried to apply an Adam optimizer but found it tended to crash my computer so it might be worth to try rebuilding these models with different optimizers on different hardware. Finally, I settled on using Categorical Cross Entropy for the loss as it was highly recommended to me for classification problems, but I

also did not try other loss functions. Ultimately, I would be curious to trying this again with better hardware to see if the issues are still largely present, otherwise I would simply like to obtain a larger and more diverse Dataset to train on.

Conclusion:

In closing, I present the above results and approaches as a tentative step towards showing how the problem outlined could be solved. The results here are promising and I hope this will help show the viability and application of these models for card grading.

References:

[1]: Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun. *Deep Residual Learning for Image Recognition*. Microsoft Research. 2015

[2]: Huang J, Rathod V, Sun C, Zhu M, Korattikara A, Fathi A, Fischer I, Wojna Z, Song Y, Guadarrama S, Murphy K. *Speed/accuracy trade-offs for modern convolutional object detectors.* CVPR. 2017

[3]: Ren, Shaoqing et al. *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks.* IEEE Transactions on Pattern Analysis and Machine Intelligence 39 (2015): 1137-1149.

[4]: *How can I tell what condition a card is in?* Retrieved June 07, 2020, from https://help.tcgplayer.com/hc/en-us/articles/221430307-Card-Condition-Guide

[5]: Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Rafal Jozefowicz, Yangqing Jia, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Mike Schuster, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

Appendix:

Dataset

D	HP	LP	NM
			
Hole Damage	Scratch	Edge Wear	No Noticeable Damage

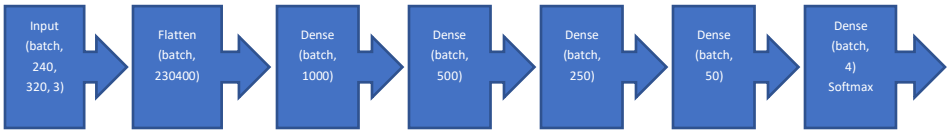
Other forms of (D) damage

			
Missing Part of Card	Large Wear and Tear	Tear from Edge	Crease

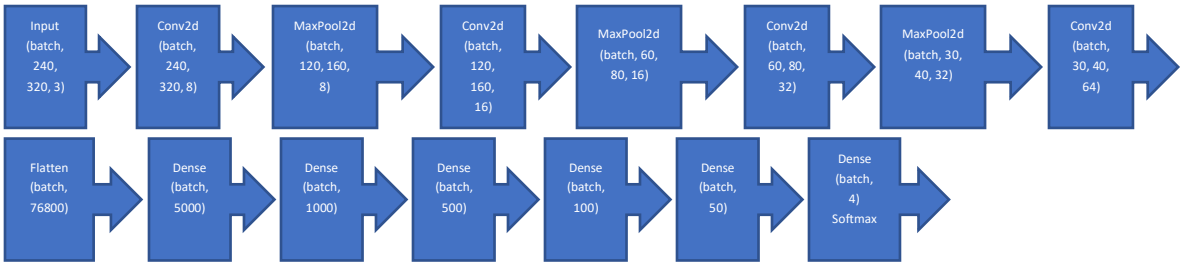
Brightness variations

Architecture:

FCNN

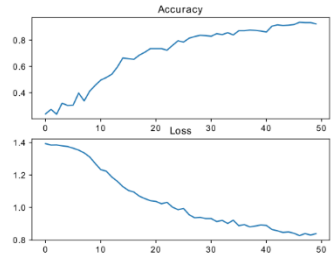


CNN

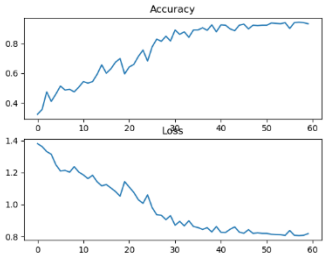


Results

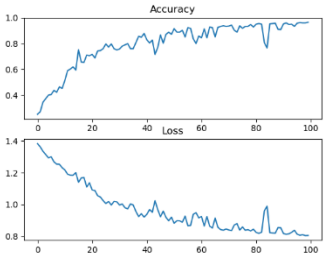
Final: ResNet50



Convolutional NN

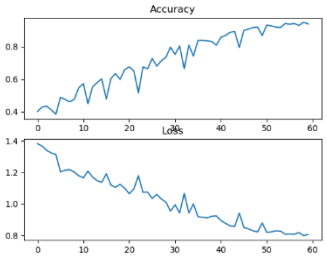


Dense NN

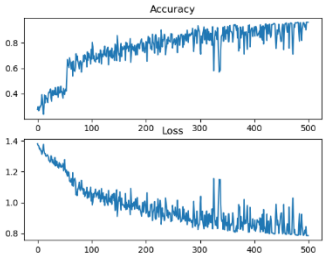


Older:

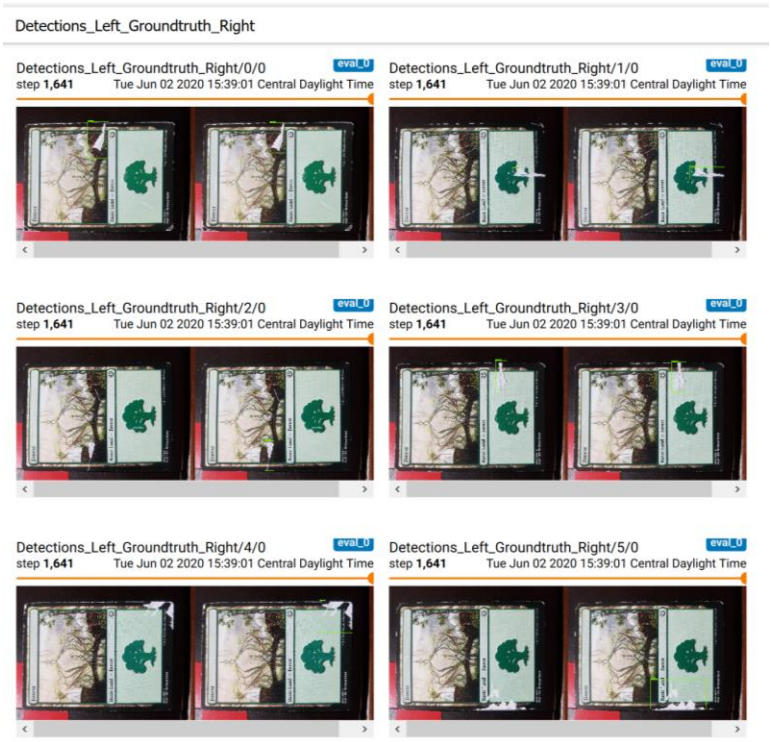
Convolutional NN

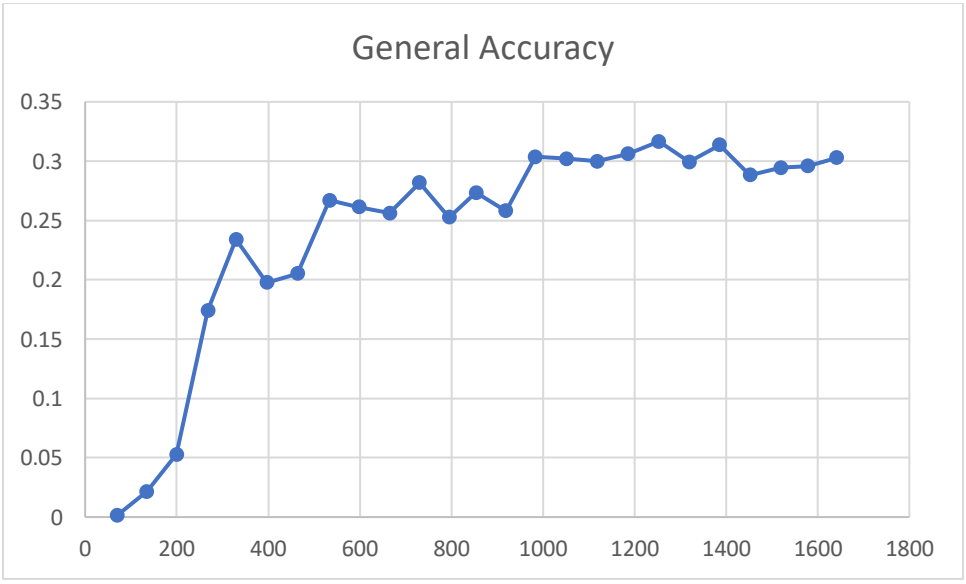


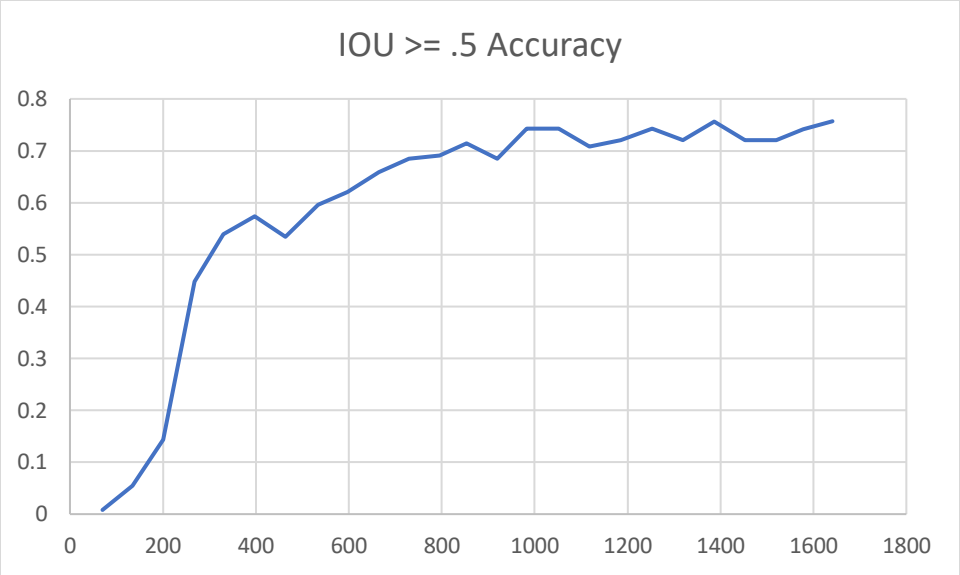
Dense NN

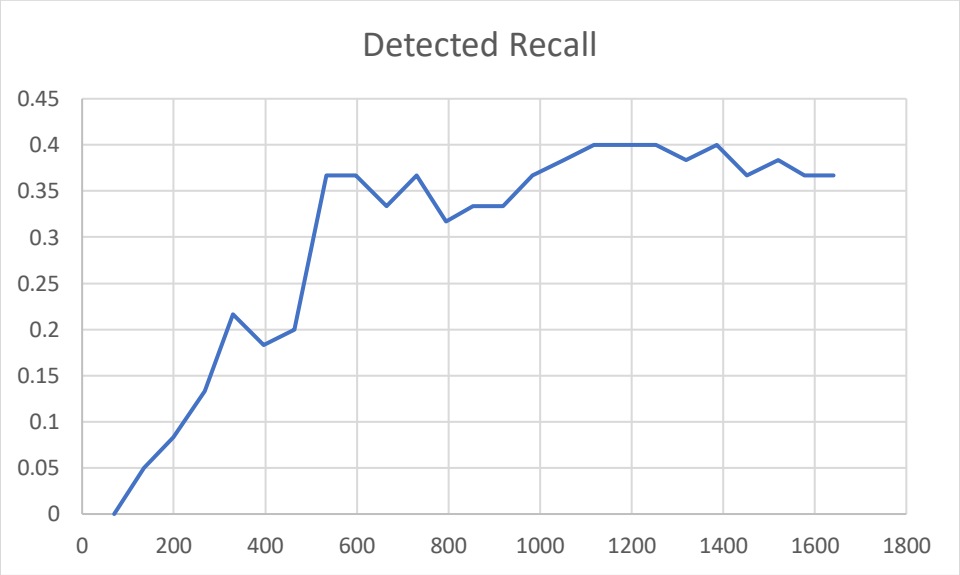


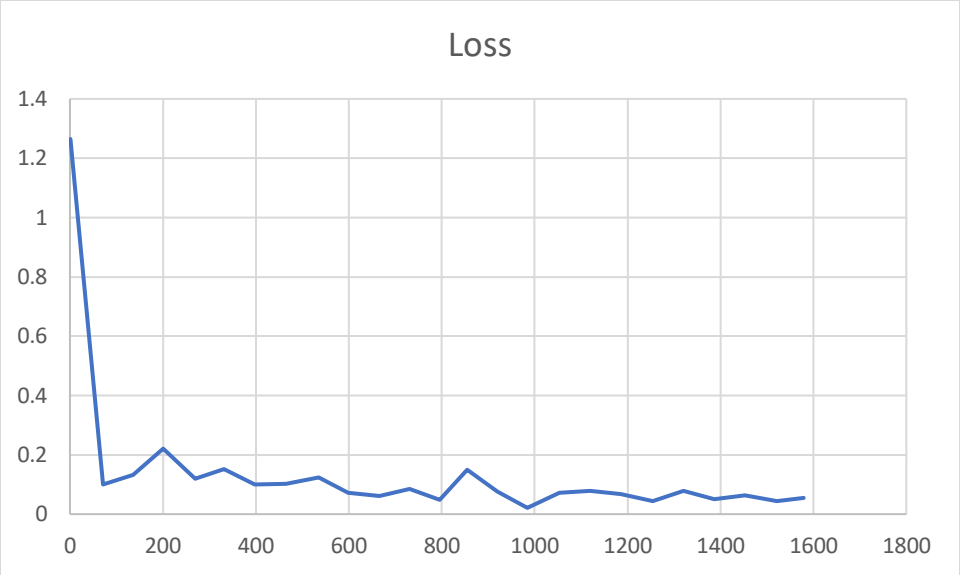
Object Detection






IOU >= .5 Accuracy
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
200
400
600
800
1000
1200
1400
1600
1800


Detected Recall
0.45
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
0
200
400
600
800
1000
1200
1400
1600
1800


Loss
1.4
1.2
1
0.8
0.6
0.4
0.2
0
0
200
400
600
800
1000
1200
1400
1600
1800