Playing card detection

PlayCDC

Object recognition and image understanding lecture

Frank Gabel and Daniel Gonzalez

15.07.2018

1 Abstract

With the capabilities of upcoming small video capturing devices in, for example, smart contact lenses with built-in cameras, whole new ways of cheating in cardgames emerge. In order to help facilitate these cheating endeavours, we implement an algorithm that detects the suits and ranks of playing cards in the field of view of a camera using the latest iteration of the YOLO object recognition algorithm.

Figure 1: Stock photos of an envisioned camera-equipped contact lense (left) and a mysterious black-jack player (right)

2 Introduction

A big topic in computer vision is the search of methods that are, given a query image, capable of answering questions like: What is present in an image? Is there a particular object in it? Where exactly in the image is this object located? Is it possible to semantically segment objects of interest in the image? Object detection deals with detecting instances of semantic objects of a certain class (such as humans, buildings, or cars) in digital images and videos. Typically, object detection deals with two sub-taks: **object localization** using bounding boxes and **multiclass object classification** within said bounding boxes. Sometimes, a third sub-task of **semantic object segmentation** is performed, i.e. the process of labelling objects on pixel level.
In this project, we use the YOLOv3 object detection algorithm [8] for object detection (without additional segmentation) in order to tell apart playing cards of a standard 52-part deck.

3 Dataset

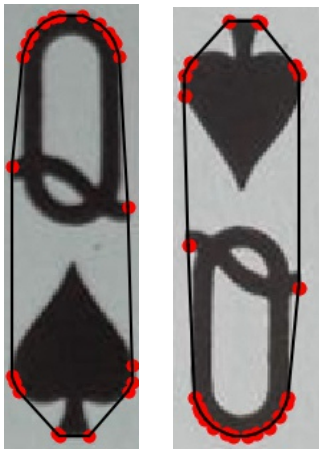
Machine learning is based on the idea of learning patterns in training data in order to build a model that can later perform predictions on unseen test data. For our detection model, we needed to have a large dataset of cards, together with the bounding box information for each of the suit/rank

combinations of cards (subsequently referred to as **logos**). Unfortunately, such a dataset is non-existent, therefore, we decided to create it on our own. We did this in two big steps, finally ending up with 750 training images for each card, as well as the bounding box information necessary for the training step.
The first important step was destined to the data preparation. We had to create data and rearrange it in a way that could be easily used for data generation. Beside this, it was also necessary to detect the convex hulls of the card logos, as they were crucial to become the Bounding Boxes information. The reason why we worked with convex hulls from the beginning on, will be explained in the later subsection.
The second big step was focused on data generation. For this, we pasted the cards on different textures, applied blurring and linear transformations on them and changed the lightning and sharpening of the cards. We apply all this transformations in a randomly way, keeping track of the convex hulls positions.

3.1 Data preparation

We decided to work with 50 cards of a standard deck of 52 cards, where each card contained two times its logo. First, we took for each card 2 different photos, manually cropped the cards to a selection and then rescaled the result by 600x900 pixels. We chose this resolution arbitrary, because we though that it was a good size to begin working with. For this step, we used the selection/rotation/cropping/rescale -tools provided by GIMP[1].
Concluding this, we proceed by detecting the convex hulls of the card logos. We used the python libraries SciPy[2] and OpenCV[3] to obtain the desired detections in a semi-automatic way (Figure 2 illustrates this process). After verifying manually each selection, we saved the two convex hulls of each card as an NumPy array.

Queen of spades with a resolution of 600x900 pixels.



Convex hulls of the queen of spades. The red points were saved as a NumPy array.

Figure 2: Semi automatic detection of the convex hulls.

3.2 Synthesize a general dataset

With all the images in the same resolution and having detected the Bounding Boxes for each card, the next goal was to generate a big amount of new data destined to train our model.
First, we randomly blurred the cards using Gaussian, Average and Median filters. We did also randomly perform sharping and lightning operations on the image. After this, we pasted the cards in the middle of 3000x3000 pixels scaled canvases provided by DDT [1]. The reason why we chose to work with such big canvases, will be more clear while reading this subsection. Beside this transformations, we also had to modify the previous detected convex hulls by a translation.

[1]GIMP 2.8.22 - GNU Image Manipulation Program
[2]SciPy: Open Source Scientific Tools for Python
[3]OpenCV: Open Source Computer Vision Library

In total, we pasted each card in 75 different textures destined for the training and in 15 different textures for testing. Figure 3 illustrate some examples and textures.

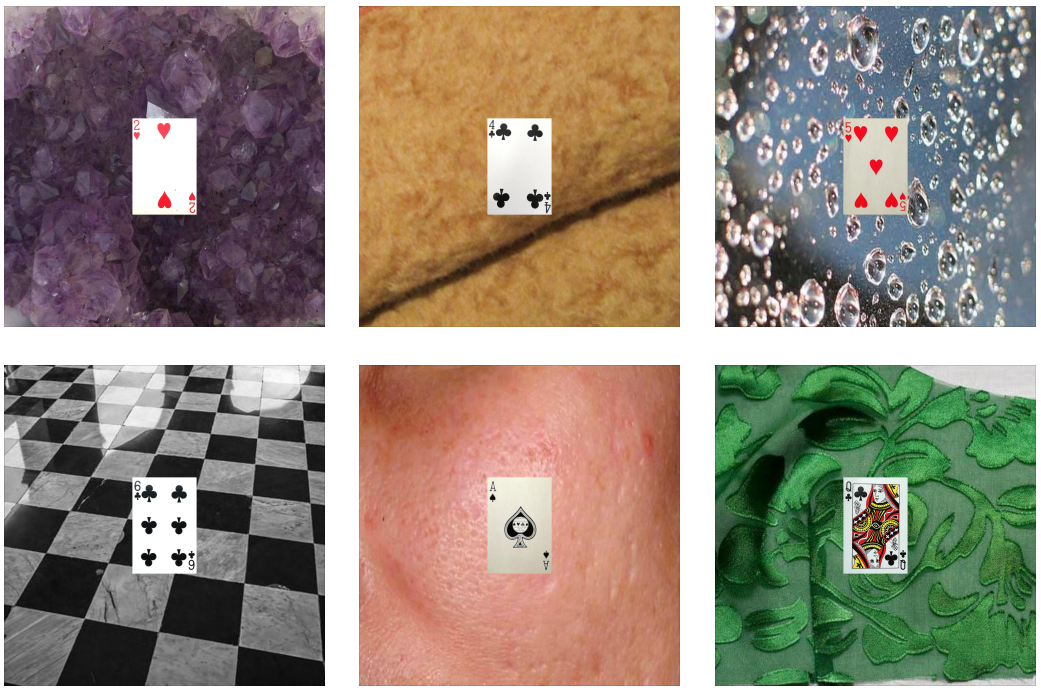


Figure 3: Some cards pasted into canvases of 3000x3000 pixels.

The next step for the data generation was to perform linear transformations on the images on canvases. We randomly performed scaling, translation and rotation on the images using the imgaug[4] python library. Concluding this, we crop the images to the middle, reducing their pixel resolution by 800x800 pixels. This operation allowed us to obtain a more accurate dataset to train our model, as in a real-life problem, some cards might be only partially visible. But cropping led also the partially/complete lost of some convex hulls, reason why we had to modify our NumPy arrays. We fixed this, letting some boundary points take the role of the affected convex hulls, but only, if we considered that there were enough points left and not direct at the boundary. See Figure 4 to have a clearer image of what might have occur.
Notice, that having pasted the images on big canvases, allowed us to have nice backgrounds around the images after rotation and cropping. This explain the choice that we notice before.
As mentioned before, we decided to use the imgaug python library for some transformations, because it allowed us to keep track of the convex hulls.

[4]http://imgaug.readthedocs.io/en/latest/

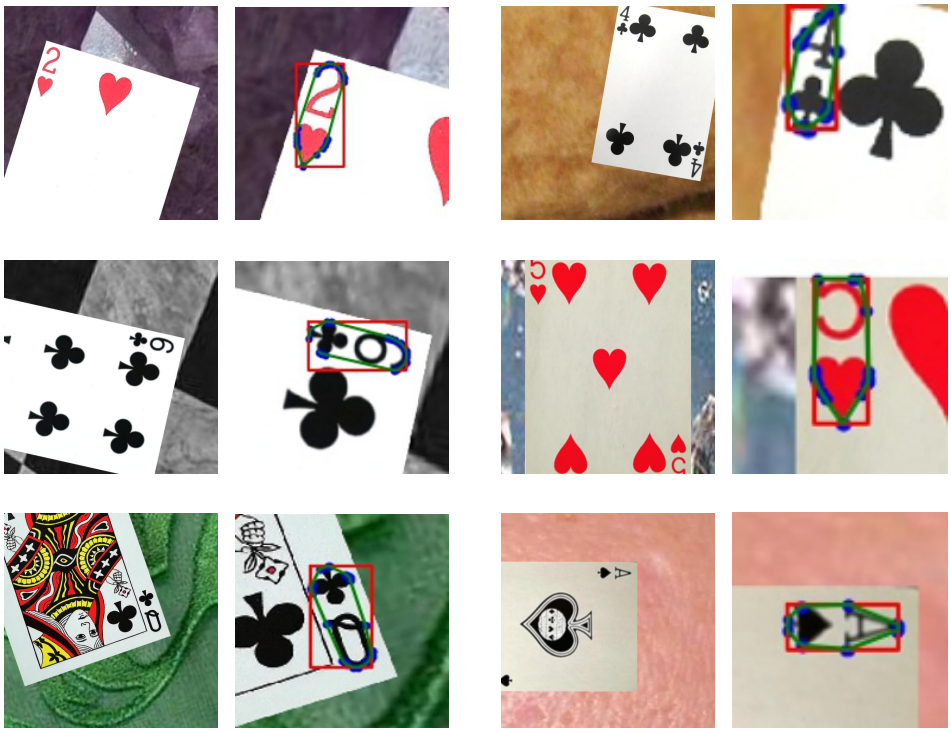


Figure 4: Images of the previous paragraph after transformations.

3.3 The convex hull approach

YOLOv3 expected for each image, a .txt file with a line for each ground truth object in the image that looks like: <object-class><x><y><width><height>, where x, y are the center positions of the Bounding Boxes (BBs) and the width and height, also of the BBs, are relative to the image's width and height. As is always the case, bounding boxes are perpendicular/aligned with image borders and can't be rotated. Nevertheless, we can rotate bounding boxes around.
This is the reason why, as mentioned before, we decided to work with convex hulls instead BBs from the beginning on. Otherwise we might have to increase the size of the BBs when doing rotations (Figure 5 illustrates this problem).

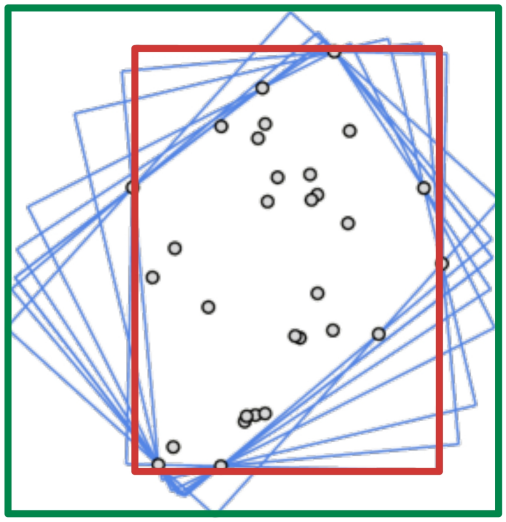


Figure 5: Green box represents the BB when applying some rotations, while the red box represents the BB of the convex hull.

4 Related work - The object detection landscape

The task of object detection in images encompasses both the "simpler" form of only localizing and subsequently classifying objects as well as the more "advanced" form of additionally segmenting objects pixel-wise.

For these tasks, different deep learning architectures have emerged, of which we will present the most important ones in the following. In 2015, **Faster Region-based Convolutional Networks (Faster R-CNNs, [9])** have come up as an enhancement of the existing R-CNN and Fast R-CNN methods that are based on both region proposal networks to find candidate bounding boxes and detection networks to perform classification. Faster R-CNN extend this by introducing weight sharing of the convolutional features between region proposal network and detection network, facilitating nearly cost-free region proposals.

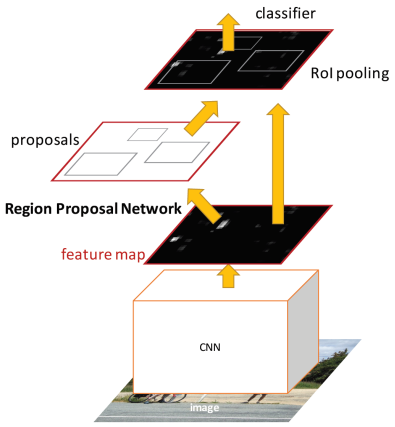


Figure 6: The Faster R-CNN architecture. Note the weight sharing between RPN Network and the classifier.

The previous methods of object detection all share one thing in common: they have one part of their network dedicated to providing region proposals followed by a classifier to classify these proposals. These methods are very accurate but come at a big computational cost (low frame-rate), in other words they are not fit to be used on embedded devices. Another way of doing object detection is by combining these two tasks into one network. We can do this by instead of having a network produce proposals we instead have a set of pre-defined boxes to look for objects.

One-stage detectors like **You Only Look Once (YOLO)**[6][7][8] or **Single-Shot Detector (SSD)**[5] that are applied over a regular, dense sampling of possible object locations have the potential to be faster and simpler, but have trailed the accuracy of two-stage detectors because of extreme class imbalance encountered during training. In 2016 and 2017, researchers from FacebookAI have developed concepts [4][3]circumventing this - the main idea being to reshape cross entropy loss such that it down-weights the loss assigned to well-classified examples. The novel focal loss focuses training on a sparse set of hard examples and prevents the vast number of easy negatives from overwhelming the detector during training. Combining this with the concept of pyramidal...., we get the **Focal Loss for Dense Object Detection (RetinaNet)** which currently outperformes all other ....

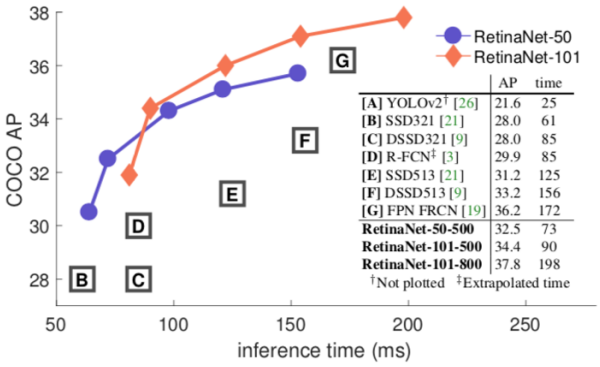


Figure 7: Evaluation of different object detection algorithms on the COCO dataset from the RetinaNet paper. Currently, evaluation performance is dominated by the RetinaNet architecture.

5 Methods

The YOLO approach to object detection

The YOLO model's novel motivation is that it re-frames object detection as a single regression problem, directly from image pixels to bounding box coordinates and class probabilities. This means that the YOLO model only "looks once" at an image for object detection. It works as follows:

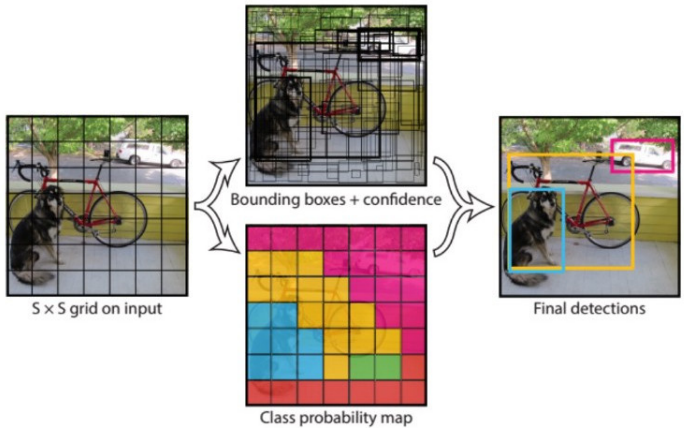


Figure 8: For each of the S^2 grid cells, we make one objectness prediction and four positional predictions for bounding boxes relative to each of the B anchor boxes as well as a global class prediction vector. Combination of these values using NMS grants the final predictions.

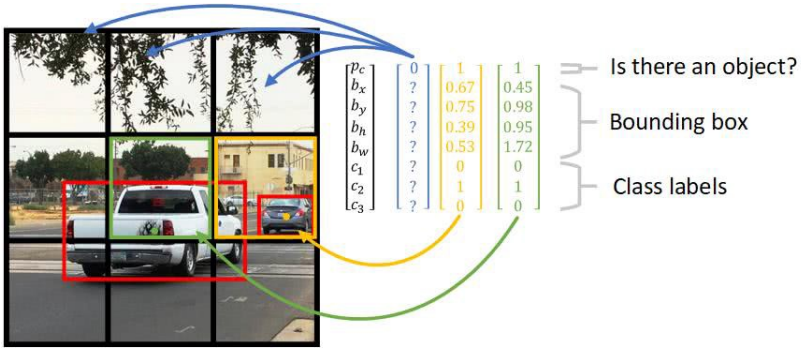


Figure 9: For each grid cell, we predict vectors of multiple bounding boxes (here, only one bounding box is predicted, $B = 1$), but only one class prediction vector of length C (here, C=3).

- divide an image into an $S \times S$ grid
- predict B bounding boxes and a confidence score for each box representing the probability that an object is contained inside a bounding box (often called *objectness score*) as well as a single C-dimensional vector of class probabilities
- the output vector $S \times S \times (B \times 5 + C)$-dimensional
- predictions are made using a technique called non-maximum suppression in which we remove low probability bounding boxes that are very close to high probability bounding boxes

At test time the conditional class probabilities and the individual box confidence predictions are multiplied, resulting in a feature map of bounding boxes "weighted" by their per-class confidence scores. see figure 5

Anchor boxes

Instead of wildly predicting B bounding boxes per grid cell, the idea of YOLO is to predict offsets to B so-called anchor boxes. Anchor boxes are rectangular boxes that are not learned, but rather pre-defined - the idea being that they should be representative of as many ground-truth bounding boxes as possible.
Usually, one sets these boxes beforehand. However, an even better way to do this in one of the later YOLO research papers, is to use a k-means algorithm, to group together conceivable anchor boxes of objects shapes you tend to get. Confidence is formally defined as $P(Object) \times \text{IOU}_{truth}^{pred}$. So its not something that model learns but something that you have to do up front. Look at the shape of bounding boxes in training data and separate then (maybe using some sort of clustering) and move those boxes to correct anchorbox (part of array) of training data.

Architecture and training hyperparameters

YOLO is implemented as a 32 layer deep convolutional neural network (DNN). The open source implementation released along with the paper is built upon a custom DNN framework written by YOLO's authors, called darknet 1 . This application provides the baseline by which we compare

our implementation of YOLO 2 . Redmon et al. have released several variants of YOLO. For our purposes, we chose the variant "tiny YOLOv3" which is a simplified version of YOLOv3 for restricted environments such as ours.[5]. In places in which the paper lacks details, we refer to the baseline darknet implementation to resolve ambiguities.

Loss function

YOLO's loss function must optimize the network's parameters as to simultaneously solve the object localization and object classification tasks. This function simultaneously penalizes incorrect object detections as well as considers what the best possible classification would be. The YOLO is trained to minimize the sum of squared errors, with scale parameters to control how much we want to increase the loss from bounding box coordinate predictions ($\lambda_{\mathbf{coord}}$) and how much we want to decrease the loss from confidence predictions for boxes that don't contain objects ($\lambda_{\mathbf{no\_obj}}$) We implement the following loss function, composed of five terms: $\mathbb{1}_{ij}^{\text{obj}}$ is 1 for the i-th grid square and j-th bounding box predictor and 0 otherwise $\mathbb{1}_{i}^{\text{obj}}$ is 1 if an object appears in the i-th cell and 0 otherwise. $p_i(c)$ and $\hat{p}_i(c)$ represent the actual and predicted conditional probabilities of whether cell i contains an object of class c.

$$\lambda_{\mathbf{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right] \textbf{ coordinate loss}$$

$$+\lambda_{\mathbf{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[\left(\sqrt{w_i} - \sqrt{\hat{w}_i} \right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i} \right)^2 \right]$$

$$+\lambda_{\mathbf{obj}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left(C_i - \hat{C}_i \right)^2 \textbf{ objectness loss}$$

$$+\lambda_{\mathbf{no\_obj}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{no\_obj}} \left(C_i - \hat{C}_i \right)^2$$

$$+ \sum_{i=0}^{S^2} \mathbb{1}_{i}^{\text{obj}} \sum_{c \in \text{classes}} (p_i(c) - \hat{p}_i(c))^2 \textbf{ classification loss}$$

techniques

Non-max suppression Non max suppression removes the low probability bounding boxes which are very close to a high probability bounding boxes.

tiny YOLO-v3

The particular architecture we used for training on our dataset is **tiny YOLOv3** which essentially is a smaller version of YOLO for constrained environments.

5.1 Webcam deployment

To test the performance of our network and getting into real-life applications, we needed a solution running on a computer to deliver the recognition results. As YOLO is a state-of-the-art real-time object detection system, we managed to deploy tinyYOLO on a webcam using a (rather slow) GPU (GeForce GTX 960M) getting about 35 FPS depending on brightness levels, which is essentially real-time. But in order to achieve this, we had to fix some obstacles. One of them was that the YOLO model was developed for the DarkNet framework, which was written in C and doesn't have any other programming interface. And as we were **noob in C? and wanted to used Pytorch**, we had to mange to integrate it using our skills.

[5] https://github.com/pjreddie/darknet/blob/master/cfg/yolov3-tiny.cfg

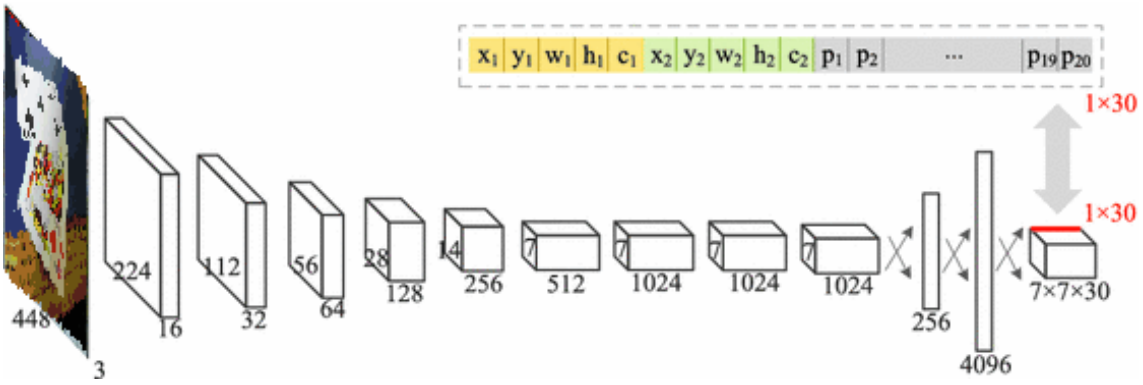


Figure 10: A simple YOLO-v3 feature extractor with a grid size of 7×7, $B = 2$ and 20 classes, making the final prediction vector $(7 \times 7 \times 30)$ - dimensional.

5.2 Evaluation

Each model is judged by its performance over the validation dataset. For each application, it is critical to find a metric that can be used to objectively compare models. In Object detection in particular, it is crucial to measure how well the detection and classification was performed. A widely used approach for evaluating localization and classification is given by the Mean Average Precision (mAP).
The first main idea for this approach, is to judge how well the detection was performed. For this, we measure the intersection of the Bounding Box detection with the ground truth, over the the union of both. This rate (so called IoU), let us designate our detection as True Positive, if the IoU is greater than a fix threshold value, or False Positive otherwise. Having calculated this for each image on the validation set, we can talk about how well the detection was. A quantity for this is given by the so called Precision, which is given by:

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}},$$

where Precision $\in [0, 1]$ has a higher value when the detection was well performed and a low value otherwise. We can compute the Precision for each class and then take the average of this. This bring us to the Average Precision.
Notice that for this computations, we fixed a threshold value for the IoU. In order to get the mAP, we compute the Precision of each class varying at a set of eleven equally spaced levels $[0, 0.1, 0.2, ..., 1]$, and after this computing the mean of them.

6 Results

Our object detection endeavours using a tiny YOLOv3 detector resulted in diverse results depending on the complexity and pecularities of the synthesized dataset in use - the main results are summarize in table 1. In termns of mAP, we generally achieved the best results for the datasets without using background textures and only simple transformations (mAP = 99.1%). However, comparing mAPs across different datasets is not very sensible - our test sets were composed of 5 % random samples from the particular dataset that had been trained on which made it a lot easier for the "earlier" dataset as their test set was easy.

In total, training always converged rather quickly 6 Again, datasets without complex textures led to better (i.e. faster) convergence which intuitively makes sense as the algorithm does not need to spend time ruling out complex dependencies that might be learned from unimportant pixels or regions outside the card.

Exemplary test results and failure cases

Metrics are most helpful in measuring the effectiveness of algorithms, but more often than not, computer vision allows for an additional inspection of results using our human visual perceptive

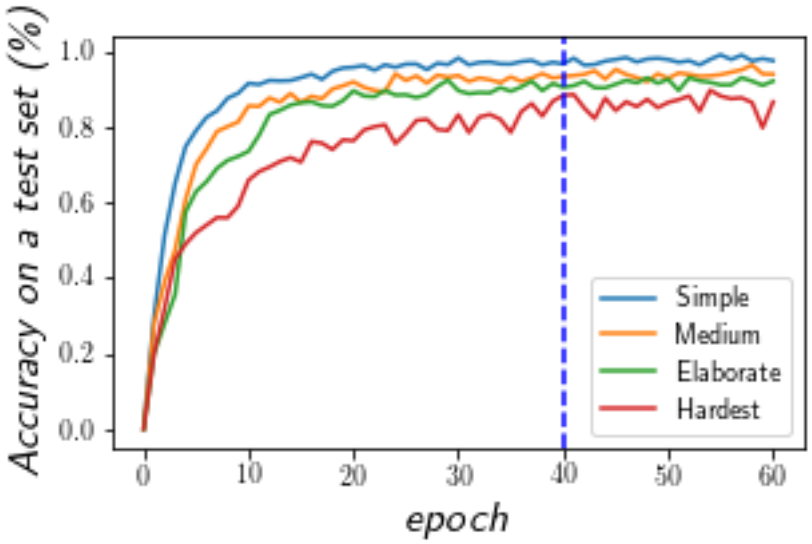


Figure 11: Precision values after each training epochs evaluated on the test set pertaining to the dataset being trained on. As you would expect, the easier the dataset, the faster the training and the better the results.

Dataset situation				
name	description	precision	recall	mAP
1 - Simple	Paste cards on simple canvases *random rotations, brightness, blurring*	0.977	0.983	**0.991**
2 - Medium	Paste randomly scaled cards on simple canvases *random rotations, brightness, blurring*	0.977	0.983	**0.989**
3 - Elaborate	Paste randomly scaled cards on textures *random rotations, brightness, blurring*	0.977	0.983	**0.971**
4 - Hardest	Paste randomly scaled cards on textures *random rotations, brightness, blurring, less zoom*	0.977	0.983	**0.973**

Table 1: Precision and recall values have been calculated using a IOU threshold of 0.5. mAP values are based on averaged precision values over IOU thresholds of $[0.1, 0.2, \ldots 0.8, 0.9]$

system.
Hence, we compiled a couple of results on unseen test images (see figure **??**). When multiple cards were present in an image, it was harder to find a completely correctly classified image than to find one that our model did some mistakes on (middle column) as we trained on images with single cards only. Yet, in images with only one card, our model seems quite robust, at least if there is not too much blurring or other types of cluttering going on. Also, as we trained on a single deck of cards, it seems natural that the model has problems with predicting cards coming from completely different decks.

Extensions and tricks used in training

Transfer learning

For all of the experiments above, we started training using a network pre-trained on the VOC dataset [2] [6]. After having trained our network on our own dataset, we also experimented with using our own fine-tuned weights from previous datasets for training subsequent models. This generally led to distinctly faster convergence which makes sense as the distribution of weights for similar datasets will obviously be similar. However, this was not much of a concern as the training procedure was fast and large speed-ups were not needed.

Webcam deployment

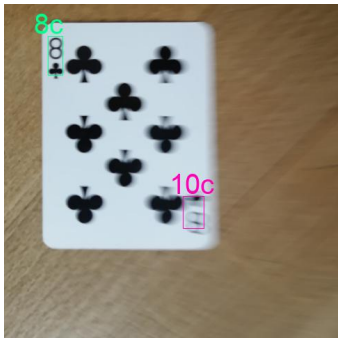
Depending on the scenery, our model running on a laptop webcam resulted in satisfactory to good results.

[6] PyTorch weights are available at `https://pjreddie.com/media/files/yolov3-tiny.weights`.

success: scenario with a highly diverse background

success: multiple cards with trivial background

success: angled shot of sheared card in front of highly diverse background



fail: misdetection of 8c due to motion blur

fail: misdetection of Qc and non-detection of 9h for unknown reason

fail: non-detection of Kd, Kh, Ks due to wildly different distribution

Figure 12: Successful cases of detection (top row) and typical fail cases (bottom row)

7 Discussion and Future Work

7.1 Overview

Using an artificially created annotated dataset of playing cards in situations reflecting realistic scenarios as good as possible, we perform object detection using the YOLOv3 object detection algorithm, achieving a mAP score of 97.30% on a holdout dataset.

Training process

The training procedure went rather swiftly. Apparently, the particular task of detecting ranks/suits of playing cards is easy for our model - it can be thought of as 2D rather than 3D - in the end, playing cards are flat pieces of paper that do not show much variation in a 3D-world unlike other natural objects where algorithms have to learn representations of varying angle and shadow conditions or situations involving occlusion and the likes.

Deployment on a webcam

Testing our model on a webcam resulted in satisfactory, results. In many scenarios, especially with very bright backgrounds, the "black" suits *clubs* and *spades* dominated the scenery and were generally easier to be detected. We hypothesize two possible explanations for this: First, our dataset and corresonding transformations might not be general enough to cover the kinds of images captured by a stock laptop webcam. Second,

7.2 Future Work

Our work works pretty well in the limited domain described above. However, if we were to develop a more reliable system, there are several extensions to be made.

more general dataset The probably largest room for improval lies in the generation of a more general dataset - we trained on a single deck of cards, using one set of photographs. A more reliable approach would be to train on several decks of cards, include more than one card per image, use more actual photos of more realistic lighting situations, use realistic images as backgrounds instead of textures etc. Also, we did not include the realistic scenario of shadows in the dataset at all, which also compromises generalisability.

more thorough hyperparameter search There are several hyperparameters of YOLO. We unsystematically experimented with changing the anchor box sizes, learning rate and grid size without experiencing greatly improved results. A more thorough hyperparameter (grid) search might warrant better results.

explore other architectures Quite possibly, the inclusion of a more general dataset would cause tiny YOLOv3 to be overwhelmed. While small architectures are beneficial to training and inference times as well as decreasing the risk of overfitting, they also encounter difficulties when relationships to be learned become overly complex. Especially including images with a very large range of scaling/zoom introduces a large number of more subtelities to be learned.

cross-validation We tested our model on a fixed set of test images. While there is nothing wrong with this approach, it also involves the risk of overfitting hyperparameters to this test set. A more thorough, yet more expensive approach would be repeatedly training models on different h..... We made sure not to make parameters of our final model dependent on evaluation metrics of the test set, so this was not a big deal.

extending the cheating pipeline A system to recognize playing cards is only one part of the big task of effective cheating. The ways in which a system like ours may be used are manifold: for one, one might deploy it into a smart contact lense or smart glasses in order to augment a user's reality with information based on what cards have been seen or, much more evil, might use fixed surveillance cameras in the room to detect cards of opponents and feed this information back to the cheating player.

8 Conclusion

Using the YOLOv3 object detection algorithm, we managed to detect the suits and ranks of playing cards in a general setting. As our project was conceived to be a proof-of-concept - in order to build a model that is actually able to help people cheat effectively, several extensions would need to be made to our approach, which include...
Another issue to solving this is the lack of commodity contact lenses equipped with video cameras and GPUs in close proximity - until then, card game players will be protected from this kind of fraud.
For further development and usage of our data we create a GitHub repository:
https://github.com/DanielGonzalezAlv/PlaycDC.git.

References

[1] Cimpoi, M., Maji, S., Kokkinos, I., Mohamed, S., , and Vedaldi, A. Describing textures in the wild. In *Proceedings of the IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)* (2014).

[2] Everingham, M., Eslami, S. M. A., Van Gool, L., Williams, C. K. I., Winn, J., and Zisserman, A. The pascal visual object classes challenge: A retrospective. *International Journal of Computer Vision 111*, 1 (Jan. 2015), 98–136.

[3] Lin, T., Dollár, P., Girshick, R. B., He, K., Hariharan, B., and Belongie, S. J. Feature pyramid networks for object detection. *CoRR abs/1612.03144* (2016).

[4] Lin, T., Goyal, P., Girshick, R. B., He, K., and Dollár, P. Focal loss for dense object detection. *CoRR abs/1708.02002* (2017).

[5] Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S. E., Fu, C., and Berg, A. C. SSD: single shot multibox detector. *CoRR abs/1512.02325* (2015).

[6] Redmon, J., Divvala, S. K., Girshick, R. B., and Farhadi, A. You only look once: Unified, real-time object detection. *CoRR abs/1506.02640* (2015).

[7] Redmon, J., and Farhadi, A. YOLO9000: better, faster, stronger. *CoRR abs/1612.08242* (2016).

[8] Redmon, J., and Farhadi, A. Yolov3: An incremental improvement. *CoRR abs/1804.02767* (2018).

[9] Ren, S., He, K., Girshick, R. B., and Sun, J. Faster R-CNN: towards real-time object detection with region proposal networks. *CoRR abs/1506.01497* (2015).