# 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 (CITE) for object detection (without additional segmentation) in order to tell apart playing cards of a standard 52-part deck.

## 3 Dataset

DANIEL THAT'S YOUR PLAYGROUND, PLEASE PUT A COUPLE OF PHOTOS :))) As there was no suitable dataset of cards to use, we decided to create it by our own. We did this in basically three main steps to obtain at the end **(Nr. images)**images, as well as the necessary Bounding

Box information required for training our YOLO NN. First we had to create the data doing photos and arrange it in such a way that we can easily use it around our workflow. The next, we had to detect the convex hulls of the card numbers and suits and the last perform rotations/re scaling/... on the images to generate new data.

### 3.1 dataset synthesis pipeline

#### Data preparation

We decided to work with a deck of 52 cards, where each card contain two times the card logo. We took for each card 2 different photos **(TODO: check this there were not really 52)** and after this we extract the cards of the photos and re-scaled them to 600x900 pixels. We choose this resolution arbitrary, as we though that it was a nice size to work with. For this part, we used the selection/rotation/cropping/re-scale -tools provided by GIMP[1]. Concluding this, we proceed by detecting the convex hulls of the cards numbers and suits. We detect the convex hulls using SciPy libraries[2]in a semi-automatic way, verifying for each card manually that we got the desired result **[SEE FIGURE]**. At the last step, we saved the convex hulls as an NumPy array **(cite??)**

#### Data generation

The goal in this step was to generate a big amount of new data for each card destined for training or NN. We wanted to perform image transformations, like translations, rotations and re scaling, as well as blurring and sharping to generate new data. We decided to use the imgaug python library, because it provided a nice way to keep track of the convex hulls after doing transformations.

## 4 Related work - The object detection landscape

The task of object detection in images encompasses both the "simple" form of localizing and subsequently classifying objects as well as the more "advanced" form of additionally segmenting objects pixel-wise. For these task, different deep learning architectures have emerged, of which we will present the most important ones in the following.

In 2015, **Faster Region-based Convolutional Network (Faster R-CNN)** (CITE) has 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.

**You Only Look Once (YOLO) Single-Shot Detector (SSD)**

**Focal Loss for Dense Object Detection (RetinaNet) Mask Region-based Convolutional Network (Mask R-CNN)** The Mask-RCNN consitutes the current state-of-the-art in semantic object segmentation. The basic idea is to..... As this extends the scope of our project, we will not dive deeper and rather refer interested readers to the orginal research paper (CITE) or ...

## 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 by dividing an image into $S \times S$ grid. Each grid cell predicts $B$ bounding boxes and a confidence score for each box. The confidence scores show how confident the model is that there is an object in that bounding box. This is formally defined as:

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

[1]GIMP 2.8.22 - GNU Image Manipulation Program
[2]SciPy: Open Source Scientific Tools for Python

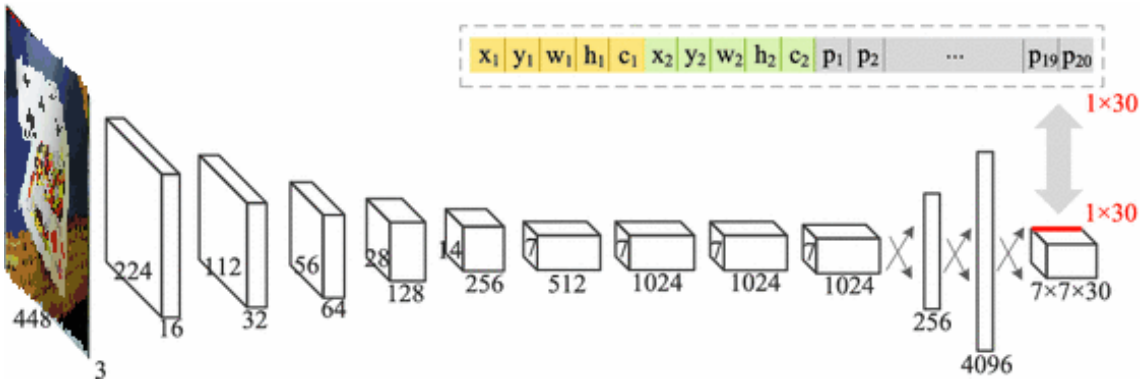



Figure 2: tiny YOLO-v3 feature extractor

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.[3]. 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 simultaneously solve the object detection and object classification tasks. This function simultaneously penalizes incorrect object detections as well as considers what the best possible classification would be. We employ the stochastic gradient descent optimization method offered by TensorFlow[10] with the Adam optimizer [7] to minimize the cost function. We implement the following loss function, composed of five terms:

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

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

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

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

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

### 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.

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

### 5.1 Webcam deployment

DANIEL

### 5.2 Evaluation

DANIEL - WRITE SOMETHING ABOUT MAP, YOU CAN PUT FORMULAS IN HERE LIKE A BOSS, BUILD UPON WHAT I WROTEIN THE POSTER MAYBE

## 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 REF TABLE 1 and REF FIGURE X. We generally achieved the best results for the datasets without textures and .... In total, training always converged rather quickly. 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.

| Dataset situation | | | | |
|---|---|---|---|---|
| name | description | precision | recall | mAP |
| **1 - Simple** | Paste cards on simple canvases<br>*random rotations, brightness, blurring* | 0.977 | 0.983 | **0.991** |
| **2 - Medium** | Paste randomly scaled cards on simple canvases<br>*random rotations, brightness, blurring* | 0.977<br>13 | 0.983 | **0.991** |
| **3 - Elaborate** | Paste randomly scaled cards on textures<br>*random rotations, brightness, blurring* | 0.977<br>13 | 0.983 | **0.991** |
| **4 - Hardest** | Paste randomly scaled cards on textures<br>*random rotations, brightness, blurring, more zoom* | 0.977<br>13 | 0.983 | **0.991** |

Table 1: Your caption here

### Exemplary test results and failure cases

Metrics are helpful in measuring the effectiveness of algorithms, but computer vision often allows for an additional sensoric inspection of results using our human visual perceptive system.
Hence, we compiled a couple of results on unseen test images (REF FIGURE3). As was to be expected, it was harder to find a rightfully classified image with multiple cards 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 ongoing. 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

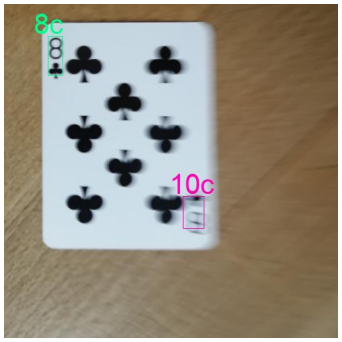
#### Transfer learning

For all experiments above, we started training using a network pre-trained on the VOC dataset (CITE). However, we also experimented with using our own fine-tuned weights from previous datasets for training. 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 further speed-ups were not needed.


**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 3: Successful cases of detection (top row) and typical fail cases (bottom row)

**BLABLABLA**

# 7 Discussion

FRANK and DANIEL We did not include shadow and stuff

# 8 Conclusion

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. The training procedure went rather swiftly. We hypothesize that this has to do with this particular task of detecting ranks/suits of playing cards can be thought of as 2D-ish rather than 3D-ish - 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 angles and shadow conditions.
Further implementations are inclined to.... see githubs