Eye in the Sky: Live Blackjack Card Counting via Real-Time Video Analysis

Ankur Jai Sood
Stanford University
jaisood@stanford.edu

Cameron Heskett
Stanford University
cheskett@stanford.edu

Mini Rawat
Stanford University
minir07@stanford.edu

Abstract

Card counting in blackjack provides players a statistical advantage but is challenging to perform consistently in real time. In this work, we develop and evaluate two fully automated computer vision pipelines to perform Hi-Lo card counting from video inputs. Our pipelines compare YOLOv5 for multiple object detection and SegFormer for semantic segmentation, with models trained both on 52-card and Hi-Lo subgroupings. We achieve high detection accuracy across both card-specific and Hi-Lo class groupings, with mean average precision of 0.974 and 0.932 respectively. We then estimate the running Hi-Lo count by aggregating detected card classes across video frames, using DeepSORT for object tracking. On a held-out test set of 20 videos of Blackjack hands, both our pipelines produce perfect Hi-Lo counts in 40% of hands with a mean absolute count error of ≈ 1.0*. These results demonstrate the feasibility of our approaches for real-time visual card counting using computer vision architectures and highlight the strengths and weaknesses of our different pipelines.*

1. Introduction

Recent advances in deep learning architectures and object tracking algorithms, combined with high-resolution, high-framerate sensors, have enabled the development of real-time computer vision workflows. Powerful edge hardware and low-latency networking further support live video annotation, which has seen increasing research and industrial adoption in recent years. In particular, the live entertainment and surveillance industries have taken the lead in the commercial deployment of such systems [9]. For this project, we were inspired by the National Basketball Association (NBA) and their partner Second Spectrum [10] to investigate the engineering problem of real-time video annotation in live sports and games.

Real-time video annotation is challenging due to strict latency requirements and the need for robustness in dynamic environments. Whether running on the cloud with network delays or on edge devices with limited compute, the system must deliver consistent performance. Many modern high-performance vision models are difficult to deploy in real time due to their size and inference cost [21]. Our project aims to investigate these challenges by designing a real-time pipeline to accurately count cards in the game of Blackjack.

1.1. Problem Statement

Blackjack is a classic casino game where players compete against the casino (the house). Even when employing perfect basic strategy, players still face a slight inherent disadvantage of approximately 0.5% against the casino [18]. Advanced blackjack players attempt to reverse this disadvantage by using card counting strategies, which maintain a running count of high and low valued cards dealt from the deck. This count indicates when the remaining cards favor the player, signaling when to increase or decrease bets accordingly. When executed perfectly, these strategies can shift the advantage to the player by up to 1.2% [18]. However, card counting requires precise mental tracking, making it challenging for human players to perform reliably in practice; even minor errors in the running count can eliminate the player's advantage.

In this project, we explore the use of computer vision techniques including object detection, segmentation, classification, and tracking, to develop a computer vision pipeline capable of accurately performing **real-time card counting** from input **video data**. This automated solution aims to overcome the practical limitations faced by human players, providing consistent precision and reliability.

2. Related Work

There have been previous works focusing on solving the problem of accurate card counting in Blackjack. Zutis et al. [23] used a stereo camera setup and classical computer vision techniques to detect card counters in a fixed environment. The solution was restricted by requiring a fixed stereo camera configuration and although performed well in fixed lighting and background conditions, was untested in dynamic environments.

A more recent example, DeepGamble [14], utilized a mask R-CNN based method to build a card counting and

bet detection pipeline deployed on the cloud, with video feed streamed from a Raspberry Pi acting as an edge device. DeepGamble achieved impressive results in card detection ($\approx$ 98%) when the cards were not occluded, but performance suffered in occluded scenarios. Additionally, throughput was limited by the sequential nature of the R-CNN pipeline and the latency introduced from the cloud architecture ($\leq$ 10 FPS).

More modern object detection architectures can perform object detection, including both classification and localization, in one shot. YOLO (You Only Look Once) is a popular example of this approach [13]. YOLO reframes object detection as a regression problem and directly predicts bounding boxes and class probabilities from the entire input image. YOLO first divides the input image into a $n \times n$ grid, with each grid square predicting m bounding boxes and corresponding class probabilities. As a single convolution network, YOLO is much faster than multi-stage approaches like R-CNN and is suitable for real-time applications.

YOLOv10 [17] introduces further improvements, notably removing Non-Maximum Suppression (NMS) by adding a one-to-one head optimized concurrently during training with the traditional one-to-many head. During inference, only the one-to-one head is used, enhancing speed by $1.8\times$ compared to RT-DETR-R18 on COCO, with $2.8\times$ fewer parameters.

Recently transformers [16] have been heavily utilized for multiple object detection. DETR [4] integrates CNNs and transformers to predict bounding boxes and object classes directly, without anchoring or region proposal networks, significantly reducing preprocessing and meeting real-time requirements for card counting systems.

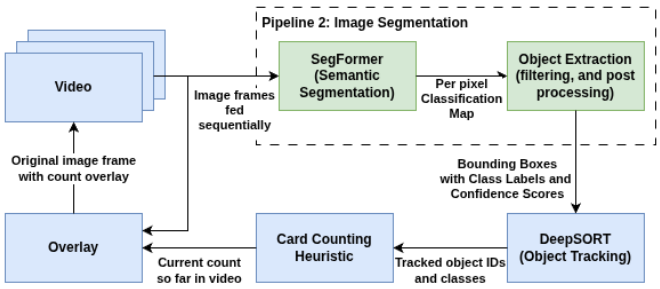
We also investigate semantic segmentation for real-time video annotation. Unlike object detection, semantic segmentation classifies each pixel, enabling detailed visual effects and interactions without extensive manual preprocessing. Early methods leveraging deep learning used large fully convolutional network architectures [7], effective but computationally expensive. Recent advances like SegFormer [20], based on the transformer architecture, improve computational efficiency and performance.

Finally, we examine object tracking using DeepSORT [12], an enhancement to Simple Online and Realtime Tracking (SORT) [19]. DeepSORT incorporates a deep appearance descriptor trained on a large-scale re-identification dataset, combined with motion information via the Mahalanobis distance, significantly reducing identity switches by approximately 45%. Running at roughly 20 Hz on modern GPUs, DeepSORT demonstrates robust, efficient tracking suited for real-time applications.

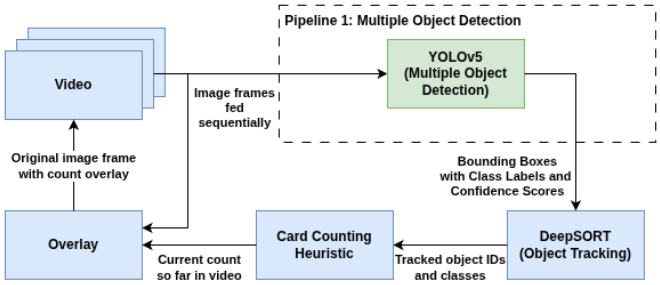
3. Methods

Our approach to automated blackjack card counting uses two complementary architectures for per-frame card localization: SegFormer for semantic segmentation and YOLOv5 for object detection (Figure 1). SegFormer produces pixel-wise masks of visible cards, while YOLOv5 outputs bounding boxes around card regions.

Although both models serve the same high-level goal, SegFormer requires pixel-level annotations, and YOLOv5 requires bounding-box labels, resulting in slightly different training requirements and final model behaviors. To assign consistent frame-to-frame IDs to each detection, we integrate DeepSORT into both pipelines. Detected cards derived from masks or boxes are converted into bounding-box inputs for the tracker, which then assigns a persistent tracking ID to each unique card instance. Using these tracking IDs, we attempt to ensure that every card appearing in the video is counted exactly once despite occlusions or brief tracking drops, and that duplicate detections of the same card within a single hand do not lead to double counting.



(a) Pipeline with multiple object detection & object tracking



(b) Pipeline with semantic segmentation & object tracking

Figure 1: Our two pipeline architectures for performing card counting on input video. Green boxes indicate software components which are different between pipelines.

3.1. Object Detection with YOLO

We built our card detection pipeline using YOLOv5[15], fine-tuning all layers and replacing the final prediction heads for our target classes. Although we initially explored YOLOv10, we chose YOLOv5 for its strong performance,

easier PyTorch integration, and reliable results on our downstream task. We fine-tuned two YOLOv5 variants: one with 52 classes (one per card) and another with three classes designed specifically for directly classifying the Hi-Lo card counting categories (*High*: 10–A, *Low*: 2–6, *None*: 7–9). Both models were trained for 50 epochs on NVIDIA GPUs using YOLOv5's default training hyperparameters along with custom parameters for data augmentation. The full set of training hyperparameters can be found in Appendix A, Table 8.

To train the final versions of our models, we created a custom variant of the Playing Card Detection Dataset (see Section 4.1). In the original dataset, each card is annotated with one or two small, tight bounding boxes, one around each corner pip, resulting in up to two detections per card. Since we only want the model to produce a single detection for each card, we converted those tight-corner boxes into one full-card box. Specifically, whenever a card was represented by two corner boxes, we treated those two boxes as opposite diagonal points of a larger rectangle and replaced them with the enclosing full-card bounding box.

3.2. Semantic Segmentation with SegFormer

For our second pipeline, we investigate semantic segmentation with SegFormer [20]. We use the Python *transformers* library, primarily developed by Hugging Face, to load the *nvidia/segformer-b4-finetuned-ade-512-512* checkpoint and finetune the entire model with our card segmentation task for our card counting task. SegFormer is pretrained on ImageNet1000 [5] with 1000 output classes and so we modify the output of the segmentation head to match the number of classes we are predicting.

During training, each input image is first resized to 512 × 512. We then assemble a batch of these 512×512 images and their corresponding ground-truth masks. The model outputs raw logits for each pixel, which we compare against the integer-valued, 512×512 masks using a pixel-wise cross-entropy loss. For some models we set *ignore_background_pixels* which ignores background pixels for the loss calculation. Backpropagation is performed on this loss to update all model parameters. Unless otherwise stated, we apply the data augmentations and hyperparameters summarized in Appendix A, Table 9 for every experiment.

We experiment with three different target label sets:

- **53-class Segmentation**: We train on the raw unmodified dataset (see Section 4.2) so that each pixel is assigned a class for one of 52 card types or *Background*.
- **4-class Segmentation**: We collapse every card pixel into one of three classes: *High*, *Low*, or *None* and add a fourth *Background* class (see Section 4.4). This parallels our YOLOv5 setup but with *Background* to accommodate a semantic segmentation head.
- **13-class Segmentation**: We combine *High*, *Low*, and *None* with each of the four suits (♣, ♡, ♠, ♢) plus a *Background* class, yielding 13 total classes (see Section 4.4).

3.2.1 Per Pixel Classification Map to Card Objects

Since SegFormer outputs a class label for every pixel in a 512×512 frame, we must convert this dense per-pixel segmentation into discrete card detections (object instances with bounding boxes and confidence scores). At inference time we take the model's raw per-pixel logits and compute a softmax over the class dimension to obtain a $(C \times H \times W)$ tensor of class probabilities, where C is the number of classes. We then collapse that probability volume into a $(H \times W)$ hard mask by applying equation 1.

$$\text{mask}_{\text{hard}}(i,j) \;=\; \arg \max_{c \in \{0,\ldots,C-1\}} p_c(i,j) \tag{1}$$

$$\text{bin}_k(i,j) \;=\; \begin{cases} 1, & \text{if } \text{mask}_{\text{hard}}(i,j) = k, \\ 0, & \text{otherwise} \end{cases} \tag{2}$$

Next, for each non-background class k (e.g. 7♡ or *Low* in the 4 class model), we extract a binary mask (Equation 2) and apply a small morphological closing followed by an opening (using an ellipse shaped kernel of radius $5 - 10$ pixels). This step removes outlier pixels and smooths each contiguous region belonging to class k. We then run a connected component analysis using (OpenCV's `findContours`) on `bin_k` to obtain one or more connected components. For each component, we compute its bounding box (x, y, w, h) and discard any boxes whose area $w \cdot h$ falls below a minimum pixel area threshold or whose average per pixel class-k probability falls below a specified confidence threshold. Each remaining component is then reported as a **card detection** of class k with confidence equal to the mean softmax probability inside that contour. In this way, each SegFormer frame produces a set of $(x, y, w, h, \text{class}, \text{score})$ detections that can be passed to DeepSORT or any other downstream tracker.

3.3. Object Tracking with DeepSORT

For both our pipelines we integrate DeepSORT [12] for object tracking. DeepSORT enhances the original SORT [3] tracking algorithm by incorporating appearance information to improve tracking performance, especially in scenarios with occlusions and re-identification challenges. These scenarios are frequently seen in card games as cards are often stacked near or on top of each other, making object detection and tracking significantly more difficult.

SORT employs a Kalman filter for predicting the future position of objects, which helps in associating detections across frames. For data association, SORT uses the Hungarian Algorithm, matching detected objects to existing tracks based on their predicted positions and appearance features. DeepSORT extends the original SORT and uses a learned appearance embedding (such as Resnet, REID, or clip) to extract appearance features from detected objects, which are then used for re-identification purposes across frames. The final output includes not only the detection boxes and class labels but also track IDs that indicate which detections belong to the same object across frames.

The combined system is designed for real-time processing, making it suitable for applications like surveillance, autonomous driving, and robotics. YOLOv5-DeepSort combines two powerful techniques for object detection and tracking: YOLOv5 and DeepSORT. YOLOv5, a family of object detection architectures and models pretrained on the card dataset, are passed to a DeepSORT algorithm which combines motion and appearance information based on Resnet50 in order to tracks the objects.

3.3.1 DeepSORT with Multiple Object Detection

The integration of YOLOv5 and DeepSORT involves using YOLOv5 for detecting objects in each frame of a video and then passing these detections to DeepSORT for tracking. YOLOv5 comes in various sizes (s, m, l). Using a smaller model like YOLOv5s can significantly increase processing speed but at a tradeoff with accuracy and feature representation. Using a large model captures feature vectors more accurately and performs better with detecting cards even with poor image quality. However, if the detection latency is high and cards move around the scene very rapidly missed detections are much more likely.

Applying techniques like model quantization or pruning help improve performance. Model quantization can reduce the model size and increase inference speed. Quantization converts model weights from floating-point to lower precision such as INT8. Model pruning removes less important weights, resulting in a smaller and faster model without significant loss in accuracy.

3.3.2 DeepSORT with Semantic Segmentation

To integrate DeepSORT with SegFormer we needed to perform some additional postprocessing and tuning for class prediction stability. Notably, we perform frame level temporal smoothing where we take a weighted combination of the current frame and the previous frames pixel wise class distributions before extracting objects and bounding boxes.

$$\hat{p}_t(i,j,c) \;=\; \alpha\,\hat{p}_{t-1}(i,j,c) \;+\; (1-\alpha)\,p_t(i,j,c) \qquad (3)$$

In equation 3, $p_t(i,j,c)$ is the softmax probability for class c at pixel (i,j) in frame t, $\hat{p}_t(i,j,c)$ is the smoothed probability, and α is the smoothing factor where $0 \leq \alpha \leq 1$. After smoothing frames we proceed with object extraction as outlined in Section 3.2.1. The objects are then filtered by two thresholds: a minimum pixel area, and a minimum confidence threshold. Finally, the filtered objects are provided to DeepSORT for tracking.

3.3.3 DeepSORT Parameter Tuning

We tune DeepSORT for each of our two pipelines separately. Appendix A, Tables 8 and 9 present our final DeepSORT parameters for each pipeline.

We performed DeepSORT parameter tuning by implementing a grid search for confidence thresholds and IOU thresholds. We used **mAP@0.5** as the objective metric to optimize the thresholds. This approach provided a balance of reducing noise while also increasing the accuracy of our detections. We experimented with various versions of the YOLOv5 model and found optimal results with YOLOv5 tag 7.0. Integrating DeepSORT code with the above YOLOv5 code required some significant code refactoring along with environment updates to produce optimal results. In addition, we also implemented an automated evaluation pipeline to evaluate results for our various experiments. These experiments were run on Amazon AWS with Ubuntu 22.04, PyTorch 2.6.0+cu126, and on a NVidia T4 GPU. [22]

4. Datasets

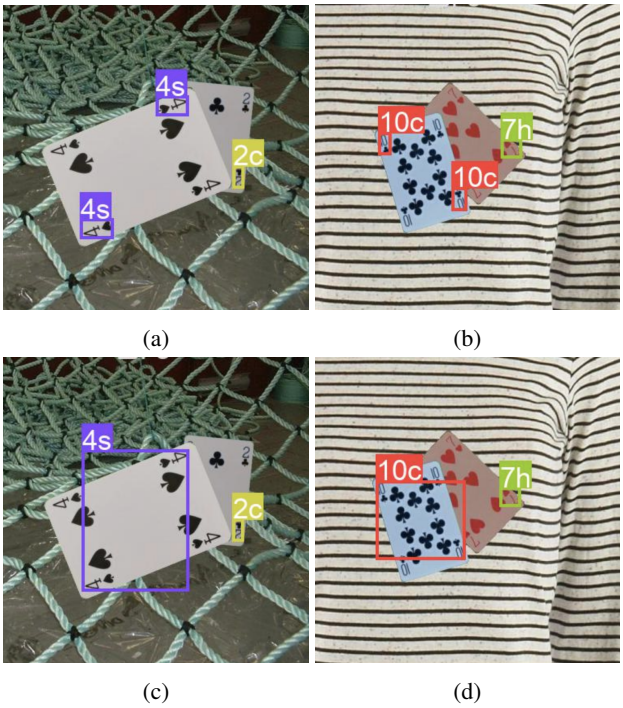
We leverage two datasets for our work in this project and preprocess them to prepare them for training our models for the Blackjack card counting task. We describe these datasets in the sections below. Both datasets were generated by different authors using a well documented playing card dataset generator [6] and were found on Kaggle.

4.1. Playing Card Detection Dataset

The first dataset is the Playing Card Detection dataset [2]. It includes 20000 images of size $416 \times 416 \times 3$ in JPG format. Each image contains multiple playing cards in various orientations and levels of occlusion. Bounding boxes for playing cards are annotated via their corners: i.e there is a bounding box around corners of the playing card withe one of 52 class labels. The dataset is split into train/test/valid $(70/20/10)$. Images are annotated in YOLO v5 PyTorch format. Training examples from this dataset can be seen in Figure 2.

4.2. Playing Card Segmentation Dataset

The second dataset is the Playing Card Segmentation dataset [8]. The dataset is proviced in *.pck* format but we



(a) (b) (c) (d)

Figure 2: (a-b) Original training examples from the Playing Card Detection dataset. (c-d) Our merged dataset examples with calculated full-card bounding boxes.

extract it and process the images masks for prepropcessing. The dataset consists of 3000 images of size $256 \times 256 \times 3$ which we save in *.png* format as well as corresponding image masks for each sample. Every pixel in each mask is labeled with a class label between $0 - 52$ where 0 is the background and each card in a standard deck is assigned a unique class ID. Hence there are 53 classes enumerated. We also save each mask in *.png* format with 8 bits per pixel, preserving the class labels. We split the dataset into train/test/val (80/10/10) with a heavier allocation to train due to the datasets smaller size. A training example from this dataset can be seen in Figure 3.

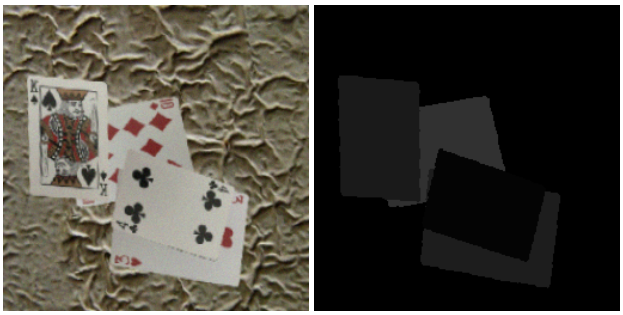


Figure 3: Training example with segmentation mask from the Playing Card Segmentation dataset.

4.3. Blackjack Videos for Count Validation

For validation of our end-to-end card counting pipeline we collect 20 clips of single Blackjack hands from different YouTube videos [11], [1]. We manually annotate the number of *High*, *Low*, and *None* category cards in each clip and we validate our pipelines by comparing the output counts of our pipelines with respect to our ground truth and by counting the errors. We split pipeline errors into two categories: *tracking* errors which correspond to errors related to the object tracking portion of our pipeline such as double counting and *classification* errors which correspond to errors due to the object detection and classification portion of our pipeline such as misclassification.

4.4. Data Preprocessing and Augmentation

To prepare our data for model training we perform the following data preprocessing.

Preprocessing on detection dataset:

1. We preprocess each sample to combine bounding boxes of corners of the same card so that there is only one large bounding box per card (Figure 2).

2. We preprocess each sample in this dataset to create an additional set of labels for each bounding box. The new set relabels each bounding box with $[0 - 2]$ corresponding to $\{$*Low*, *None*, *High*$\}$. We test both sets of labels during out model training.

Preprocessing on segmentation dataset:

1. We preprocess each sample in this dataset from *.pck* format to two *.png* files, one for the image and one for the mask. We resize each image and mask from the original 256×256 to 512×512.

2. We preprocess each sample in this dataset and create two additional set of masks. The first new set relabels each pixel with $[0 - 12]$ where 0 is background and $[1 - 12]$ is $\{$*Low*, *None*, *High*$\}$ for each suit $\{$*Club*, *Spade*, *Heart*, *Diamond*$\}$. The second new set relabels each pixel with $[0-3]$ where 0 is background and $[1-3]$ corresponds to $\{$*Low*, *None*, *High*$\}$.

3. We add 30 additional images of casino tables and casino chips to the dataset where each image corresponds to pure background and the masks are all 0. We do this to solve problems with card hallucinations which we will discuss in the next part of this report.

During training we perform different data augmentation to our samples for each pipeline. For the object detection pipeline using YOLO the augmentations are outlined in Appendix A, Table 8. For the semantic segmentation pipeline using SegFormer the augmentations are outlined in Appendix A, Table 9.

5. Results & Discussion

5.1. Final Scoring Metrics

We categorize each detected card into one of three classes: *Low* ($\{2, 3, 4, 5, 6\}$), *High* ($\{10, J, Q, K, A\}$), or *None* ($\{7, 8, 9\}$), following the standard Hi-Lo card counting scheme [18]. These class labels are either predicted directly or obtained via post-processing for final evaluation. To assess per-class detection quality, we compute:

$$\text{Precision}_c = \frac{\text{TP}_c}{\text{TP}_c + \text{FP}_c}, \text{Recall}_c = \frac{\text{TP}_c}{\text{TP}_c + \text{FN}_c},$$
$$\text{F1}_c = 2\,\frac{\text{Precision}_c \times \text{Recall}_c}{\text{Precision}_c + \text{Recall}_c},$$

where for each $c \in \{\textit{Low}, \textit{High}, \textit{None}\}$:

- TP_c (True Positives): number of correctly detected cards of class c.
- FP_c (False Positives): number of cards incorrectly detected as class c.
- FN_c (False Negatives): number of ground-truth cards of class c that the model failed to detect.

We also report overall precision, recall, and F1-score:

$$\text{Precision}_o = \frac{\sum_c \text{TP}_c}{\sum_c (\text{TP}_c + \text{FP}_c)}, \text{Recall}_o = \frac{\sum_c \text{TP}_c}{\sum_c (\text{TP}_c + \text{FN}_c)},$$
$$\text{F1}_o = 2\,\frac{\text{Overall Precision} \times \text{Overall Recall}}{\text{Overall Precision} + \text{Overall Recall}}.$$

For each video in the test set, we compare the ground-truth Hi-Lo count, computed manually, against the model's predicted score obtained by summing over the detected card classes. According to the Hi-Lo algorithm, *High* cards contribute a score of -1, *Low* cards contribute $+1$, and *None* cards contribute 0. For example, a video containing 4 high cards, 1 low card, and 1 none card would yield a score of:

$$\text{Hi-Lo} = (-1) \times 4 + 0 + (+1) \times 1 = -3.$$

Letting s_{gt} and s_{pred} denote the ground-truth and predicted scores respectively, we evaluate model performance by computing the number of videos for which $s_{\text{gt}} = s_{\text{pred}}$ (i.e., perfect predictions), and analyze the distribution of the signed error $s_{\text{pred}} - s_{\text{gt}}$ for the remaining cases. Final score results for the pipelines discussed in our methods are captured in Table 5.

5.2. Object Detection with YOLO

We evaluated object detection performance using YOLOv5 trained on two labeling schemes: (1) a fine-grained 52-class setup, and (2) a coarse-grained 3-class Hi-Lo scheme (*Low*, *High*, *None*). The results are summarized in Tables 1 and 2, respectively.

Metric	Avg.	Min	$\text{Class}_{\text{min}}$	Max	$\text{Class}_{\text{max}}$	Std.
Precision	0.960	0.818	$A_\heartsuit$	1.000	$4_\clubsuit$	0.034
Recall	0.969	0.820	$A_\heartsuit$	1.000	$4_\clubsuit$	0.038
mAP@50	0.974	0.895	$A_\heartsuit$	0.995	$4_\clubsuit$	0.022
mAP@50–95	0.834	0.720	$A_\heartsuit$	0.881	$5_\spadesuit$	0.035

Table 1: YOLOv5 Performance Across 52 Card Classes

Metric	Avg.	Min	$\text{Class}_{\text{min}}$	Max	$\text{Class}_{\text{max}}$	Std.
Precision	0.961	0.945	*None*	0.977	*Low*	0.013
Recall	0.972	0.961	*None*	0.990	*Low*	0.013
mAP@50	0.974	0.967	*None*	0.983	*Low*	0.007
mAP@50–95	0.823	0.808	*None*	0.854	*Low*	0.022

Table 2: YOLOv5 Performance Across 3 Card Classes

On the pure detection task, both the 52-class and 3-class models achieved an average **mAP@50 of 0.974**, indicating that the model is highly effective at detecting playing cards regardless of class granularity. The mean performance across precision and recall was similarly strong, with average recall exceeding 0.96 in both labeling schemes.

Within the 52-class model, the best performing cards were consistently **4s** and **5s**, which achieved perfect or near-perfect scores across all metrics. In contrast, the **Aces** exhibited the lowest precision and recall, pulling down the class-wise minimum for each metric. We hypothesize that this is due to visual ambiguity because this card's design has a lot of white space and a centered icon, resulting in difficulty in extracting distinct features.

In the 3-class Hi-Lo version, *Low* achieved the highest precision and recall, while the *None* class underperformed. We hypothesize that the *None* class struggles due to two key factors: (1) the class represents a smaller subset of training examples, and (2) cards in this class share similar characteristics to some cards in both the *High* class (10s) and *Low* class (6s).

Overall, the results validate YOLOv5 as a strong baseline detector for real-time card recognition and suggest that even coarse-grained 3-class labeling is sufficient to enable robust Hi-Lo count estimation downstream.

5.3. Semantic Segmentation with SegFormer

We evaluated semantic segmentation performance similarly but at the **pixel level**. Below we report the results for our 3 labeling schemes: (1) a 53-class setup, (2) a coarser 13-class scheme of (*Low*, *High*, *None*) for each suit (♣, ♡, ♠, ♢) and *Background*, and a 4-class scheme of (*Low*, *High*, *None*, *Background*).

We experimented with ignoring background pixels as we hypothesized that it would provide a stronger learning sig-

nal for learning card pixels since the data was vastly dominated by background pixels (cards take a small portion of the overall scene). We ultimately found that training with background pixels incorporated into the loss calculation was superior and so the results we present are for those model variants. For min/max metrics we exclude the *Background* class and consider only card specific classes.

We first trained our 53-class model using the parameters outlined in Appendix A, Table 9. The results are summarized in Table 3. We see a significant imbalance between SegFormer performance across classes. The model has a much easier time detecting face cards but struggles much more with None category cards like 7 and 8. This is reflected not only in IoU (min 0.669 for $7\diamondsuit$ vs. max 0.943 for $Q\spadesuit$) but also in precision/recall. For face cards both metrics exceed 0.95, whereas precision for $7\spadesuit/8\clubsuit$ dips below 0.85 and recall falls under 0.80. Mean AP (mAP) follows a similar pattern, worst at 0.644 for $6\spadesuit$ and best at 0.943 for $Q\spadesuit$ which mirrors the IoU rankings.

Metric	Avg.	Min	$\text{Class}_{\text{min}}$	Max	$\text{Class}_{\text{max}}$	Std.
Precision	0.924	0.806	$6\spadesuit$	0.985	$A\clubsuit$	0.050
Recall	0.918	0.752	$7\diamondsuit$	0.982	$Q\diamondsuit$	0.062
m(AP)	0.849	0.644	$6\spadesuit$	0.943	$Q\spadesuit$	0.076
IOU	0.851	0.669	$7\diamondsuit$	0.943	$Q\spadesuit$	0.069

Table 3: SegFormer Performance (Pixel Level) Across 53 Card Classes

These results make sense given our relatively small dataset of 3000 examples. Even with data augmentation applied, there aren't enough training examples for the model to effectively learn all the subtle card variations. In the 53 model confusion matrix (Appendix D) we see that the most common misclassifications are between similar looking classes, for example 7s and 8s, whereas very distinct classes like face cards are easier for the model to distinguish. When we evaluate the 53 class version on our video dataset we see a very high FP and FN rate, flickering misclassifications, and poor performance with the overall count.

Metric	Avg.	Min	$\text{Class}_{\text{min}}$	Max	$\text{Class}_{\text{max}}$	Std.
Precision	0.966	0.939	$Low\spadesuit$	0.986	$None\clubsuit$	0.014
Recall	0.964	0.936	$None\diamondsuit$	0.979	$Low\heartsuit$	0.016
m(AP)	0.932	0.899	$None\diamondsuit$	0.947	$Low\clubsuit$	0.024
IOU	0.927	0.901	$None\diamondsuit$	0.947	$Low\clubsuit$	0.016

Table 4: SegFormer Performance (Pixel Level) Across 13 Card Classes

Next we evaluated the same SegFormer model trained on our collapsed 13-class configuration (12 card categories plus background). Table 4 reports pixel-level Precision, Recall, mAP, and IoU. Relative to the 53-class model, overall Precision and Recall each increase by nearly three percentage points, with similar gains in mAP and IoU. The hardest to segment classes remain the *None* cards (particularly $None\diamondsuit$), reflecting the visual ambiguity of mid-rank cards (7–9). In contrast, *Low* cards achieve the highest scores, benefiting most from consolidation. These results confirm that collapsing suits and ranks into broader *Low*, *None*, and *High* groups significantly boosts segmentation reliability while sacrificing only minimal semantic detail, which is acceptable since exact rank recognition is less critical than grouping accuracy for counting.

Finally, we trained a further-collapsed 4-class model (see Section 4.4). Although it outperforms the 13-class version on core metrics (Appendix B, Table 10), it proved unsuitable for card counting due to overlapping cards of the same class being frequently detected as a single object (see Section 7, video results).

5.4. YOLOv5 & DeepSORT Tracking Results

We evaluated our end-to-end pipeline using the best-performing 3-class YOLOv5 model on 20 Blackjack hand video clips (see Section 4.3). As shown in Table 5, the pipeline achieved a perfect Hi-Lo count on 40% of the clips, with a mean absolute count error of 1.15 cards and a worst-case error of 6 cards. Several factors contributed to errors. When cards were visible for only a few consecutive frames, DeepSORT often failed to assign cards tracking IDs, leading to missed detections. Reducing the `n_init` parameter helped in those cases but also introduced false positives due to ID fragmentation, leading to double counting when confidence scores fluctuated across frames. Missed or incorrect detections were especially common when cards appeared at the top of the frame, farthest from the camera. We hypothesize this is because our training data contained few examples of cards at that angle. Adding more examples that more closely resemble the evaluation videos could improve performance in such cases.

Metric	YOLOv5-3	YOLOv5-52	SegFormer-13
Perfect Predictions	8/20 (40%)	7/20 (35%)	8/20 (40%)
Mean Abs. Error	1.15	1.05	0.90
Max Error	6	3	2

Table 5: Hi-Lo count results on 20 blackjack hands for our YOLOv5 and one SegFormer pipelines

5.5. SegFormer & DeepSORT Tracking Results

To evaluate our end to end segmentation pipeline, we train versions of our 13 class and 4 class SegFormer models trained for 60 epochs. We evaluate these models on the same

Class	TP	FP	FN	Precision	Recall	F1
Low	41	16	2	0.719	0.953	0.820
High	40	8	3	0.833	0.930	0.879
None	17	2	5	0.895	0.773	0.829
Overall	98	26	10	0.790	0.907	0.845

Table 6: Classification Accuracy by Hi-Lo Class (YOLOv5 3 class with DeepSORT)

Class	TP	FP	FN	Precision	Recall	F1
Low	40	7	4	0.851	0.909	0.879
High	37	3	6	0.925	0.860	0.892
None	15	4	6	0.789	0.714	0.750
Overall	92	14	16	0.868	0.852	0.860

Table 7: Classification Accuracy by Hi-Lo Class (SegFormer 13 class with DeepSORT)

Figure 4: Histogram of Hi-Lo count errors across 20 test videos using our **YOLOv5** pipeline. Most predictions are within ± 2 of the ground-truth count, with a mean absolute error of 1.15.

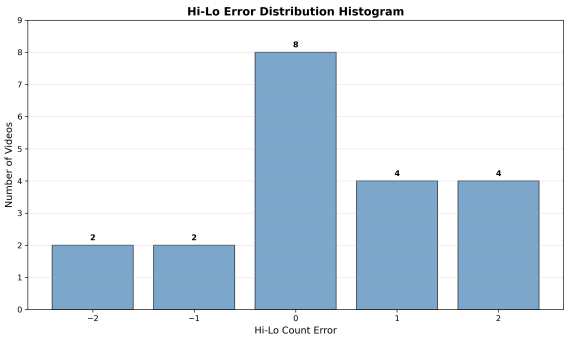


Figure 5: Histogram of Hi-Lo count errors across 20 test videos using our **SegFormer** pipeline. All predictions are within ± 2 of the ground-truth count, with a mean absolute error of 0.90.

20 Blackjack hand video clips (see section 4.3). Our 13 class version outperforms our 4 class version and we summarize its performance in Table 5.

Over all 20 Blackjack hands our pipeline is able to perfectly predict the count across 40% of hands with a mean count error of less than 1 per hand. In its worst performing hand, it is off from the true count by 2. Compared to Pipeline 1, our new pipeline has a lower FP rate (Table 7), hence higher precision, but a higher FN rate, which corresponds to the observed drop in recall. Figure 5 shows our distribution of count errors across each video with respect to the ground truth. We see that the distribution here is tighter than in pipeline 1 but that this pipeline seems to over count when compared to pipeline 1. Qualitatively, we found that double counting often occurs when a card moves quickly and the tracker loses its original identity, causing a new track to be created.

6. Conclusion & Future Work

Our pipelines achieved a perfect count on 40% of the Blackjack hand videos, each of which exhibited distinct strengths and failure modes. YOLOv5 performed well on static detection, but struggled once DeepSORT tracking was introduced, especially when cards appeared on-screen for only a few frames. It also often failed to recognize cards near the top of the frame, possibly because those cards appeared smaller in the video frames than in our training data. SegFormer, in contrast, localized the card boundaries more accurately, but was prone to double counting once tracking was introduced whenever the confidence per frame fluctuated. Although our 13-class and 4-class segmentation models reduced double-counting, they sometimes merged adjacent cards of the same class into a single segment, leading to false negatives. For interesting outputs from our pipelines see Appendix C.

To improve performance in future work, we can consider enhancing our training data by generating more varied synthetic examples, rather than simply merging bounding boxes. Incorporating temporal modeling directly into the pipeline by training end-to-end on video clips rather than individual frames may also be worth exploring to reduce tracking error as compared to DeepSORT. For the segmentation approach, exploring architectures like MaskFormer, which combine semantic and instance segmentation, could enable the model to distinguish touching cards of the same class at the pixel level, thereby reducing false negatives.

7. Demonstration Video & Source Code

Our code is available at:
https://github.com/ankurjaisood/cs231n_eye-in-the-sky
Demonstration video of our card-counting pipelines:
https://www.youtube.com/watch?v=yUcEbM6uDno

8. Contributions & Acknowledgments

- **Ankur Jai Sood**: Initial finetuning of YOLOv10 for the milestone, dataset preprocessing for classifiction dataset and for segmentation dataset, SegFormer training pipeline, DeepSORT integration with SegFormer, SegFormer evaluation pipeline, integrating data augmentations for SegFormer, SegFormer + DeepSORT frame interpolation, tuning DeepSORT for SegFormer, weights and biases integration, multiple rounds of training for final SegFormer and DeepSORT pipelines (53 class / 13 class / 3 class versions), various bug fixes.
- **Cameron Heskett**: Multiple rounds of training and finetuning of final YOLOv5 and DeepSORT YOLOv5 3 Class detection pipeline, creation of scoring metrics and tooling for evaluating all pipelines, custom data augmentations for YOLOv5 training, created blackjack evaluation video dataset with ground truth labels, edited and uploaded YouTube video demonstrating pipeline examples.
- **Mini Rawat**: Finetune YOLO and DeepSORT for YOLOv5 52 Class pipeline; DeepSORT and YOLOv5 latest model integrations; AWS Training machine setup; Evaluating different YOLO/DeepSORT models by running several hours of Deep Learning training on NVidia T4 GPUs in AWS; Grid Search implementation for Conf and IOU thresholds; Tuning model for optimal results balancing accuracy and noise; Identified bugs and corner cases. Generated perfect and interesting case video samples from model runs.

We would also like to thank our TA mentor Zhoujie (Jason) Ding and the entire CS231n teaching team for their help and hardwork during the Spring 2025 quarter.

References

[1] All-Casino-Action. https://www.youtube.com/watch?v=J5hF0umt0jE, May 2025. YouTube video, All Casino Action.

[2] Andy8744. Playing cards object detection dataset. https://www.kaggle.com/datasets/andy8744/playing-cards-object-detection-dataset, 2021. Accessed: 2025-05-16.

[3] A. Bewley, Z. Ge, L. Ott, F. Ramos, and B. Upcroft. Simple online and realtime tracking. In *2016 IEEE International Conference on Image Processing (ICIP)*. IEEE, Sept. 2016.

[4] N. Carion, F. Massa, G. Synnaeve, N. Usunier, A. Kirillov, and S. Zagoruyko. End-to-end object detection with transformers, 2020.

[5] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009.

[6] Geaxgx. Playing Card Detection. https://github.com/geaxgx/playing-card-detection, 2019. GitHub repository.

[7] J. Long, E. Shelhamer, and T. Darrell. Fully convolutional networks for semantic segmentation. *CoRR*, abs/1411.4038, 2014.

[8] luanademi. Playing card ensemble – segmentation masks. https://www.kaggle.com/datasets/luanademi/playing-card-ensemble-segmentation-masks. Accessed: 2025-05-20.

[9] S. Nallola and V. Ayyasamy. Twenty-five years of real-time surveillance video analytics: a bibliometric review. *Multimedia Tools and Applications*, 83:1–34, 01 2024.

[10] National Basketball Association. Nba and genius sports/second spectrum expand partnership to deepen nba league pass innovations with enhanced basketball analytics and develop new next gen platform. https://pr.nba.com/nba-genius-sports-second-spectrum-expanded-partnership/, Mar. 2023. Accessed: 2025-05-10.

[11] Perfect-Pair. https://www.youtube.com/watch?v=al-T7OsAR8g, May 2025. YouTube video, Perfect Pair.

[12] A. Pujara and M. Bhamare. Deepsort: Real time multi-object detection and tracking with yolo and tensorflow. In *2022 International Conference on Augmented Intelligence and Sustainable Systems (ICAISS)*, pages 456–460, 2022.

[13] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection, 2016.

[14] D. Syed, N. Gandhi, A. Arora, and N. Kadam. Deepgamble: Towards unlocking real-time player intelligence using multi-layer instance segmentation and attribute detection. In *2020 19th IEEE International Conference on Machine Learning and Applications (ICMLA)*, page 376–383. IEEE, Dec. 2020.

[15] ultralytics. yolov5. https://github.com/ultralytics/yolov5. Yolov5 GitHub repository.

[16] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[17] A. Wang, H. Chen, L. Liu, K. Chen, Z. Lin, J. Han, and G. Ding. Yolov10: Real-time end-to-end object detection, 2024.

[18] Wikipedia contributors. Card counting. https://en.wikipedia.org/w/index.php?title=Card_counting&oldid=1267976662, 2025. Last edited on 7 January 2025; accessed 25 April 2025.

[19] N. Wojke, A. Bewley, and D. Paulus. Simple online and realtime tracking with a deep association metric, 2017.

[20] E. Xie, W. Wang, Z. Yu, A. Anandkumar, J. M. Álvarez, and P. Luo. Segformer: Simple and efficient design for semantic segmentation with transformers. *CoRR*, abs/2105.15203, 2021.

[21] J. Yang, S. Liu, Z. Li, X. Li, and J. Sun. Streamyolo: Real-time object detection for streaming perception, 2022.

[22] ZQPei. yolov5. https://github.com/ZQPei/deep_sort_pytorch. Deep Sort Pytorch GitHub repository.

[23] K. Zutis and J. Hoey. Who's counting? real-time blackjack monitoring for card counting detection. pages 354–363, 10 2009.

A. Training & Augmentation Hyperparameters

YOLOv5 Optimizer & Loss	
lr0	0.01
lrf	0.1
momentum	0.937
weight_decay	0.0005
warmup_epochs	3.0
warmup_momentum	0.8
warmup_bias_lr	0.1
box	0.05
cls	0.3
cls_pw	1.0
obj	0.7
obj_pw	1.0
iou_t	0.20
anchor_t	4.0
fl_gamma	0.0
Dataset Augmentation Parameters	
hsv_h	0.015
hsv_s	0.7
hsv_v	0.4
degrees	15.0
translate	0.2
scale	0.9
shear	10.0
perspective	0.0005
flipud	0.0
fliplr	0.5
mosaic	1.0
mixup	0.15
copy_paste	0.3
DeepSORT Paramaters	
max_cosine_distance	0.2
min_confidence	0.5
nms_max_overlap	0.5
max_iou_distance	0.7
max_age	70 frames
n_init	3 frames
embedder	REID

Table 8: YOLOv5 Training and Augmentation Hyperparameters

Segformer Optimizer & Loss		
lr		0.00005
weight_decay		0.01
epochs		20.0
batch_size		16.0
ignore_background_pixels		0
Dataset Augmentation Parameters		
Type	Value	Probability
random_rescale	$\pm 20\%$	100%
image_interpolation	LINEAR	if rescaled
mask_interpolation	NEAREST	if rescaled
padding	0 (masks/images)	if rescaled down
random_crop	512×512	if rescaled up
horizontal_flip	-	50%
vertical_flip	-	20%
random_rotation	± 15 deg	50%
random_brightness	$\pm 20\%$	50%
random_contrast	$\pm 20\%$	50%
DeepSORT Paramaters		
max_cosine_distance		0.2
min_confidence		0.6
nms_max_overlap		0.5
max_iou_distance		0.7
max_age		20 frames
n_init		5 frames
embedder		clip_RN50x4
min_pixel_area		1500 pixels
min_bounding_box_area		250 pixels

Table 9: SegFormer Training and Augmentation Hyperparameters

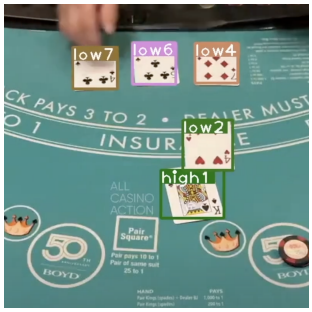
B. Additional Results & Tables

Metric	Avg.	Min	$\text{Class}_{\text{min}}$	Max	$\text{Class}_{\text{max}}$	Std.
Precision	0.976	0.958	*None*	0.976	*Low*	0.014
Recall	0.977	0.965	*None*	0.974	*Low*	0.013
m(AP)	0.957	0.926	*None*	0.954	*Low*	0.012
IOU	0.942	0.925	*None*	0.952	*Low*	0.012

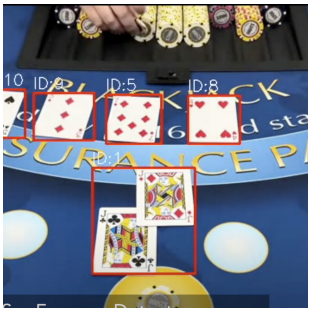
Table 10: SegFormer Performance (Pixel Level) Across 4 Card Classes

C. Interesting Examples from Pipeline Outputs

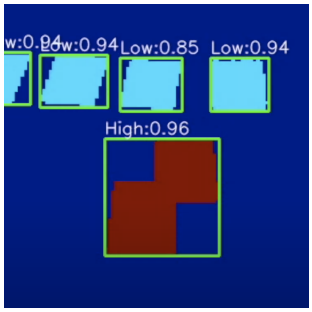
(a) 52 Class YOLOv5 + DeepSORT Pipeline



(b) 3 Class YOLOv5 + DeepSORT Pipeline

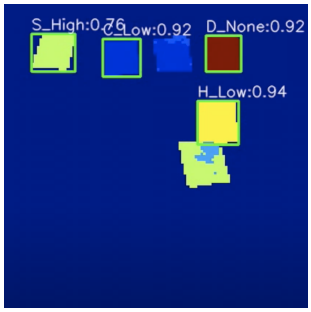


(c) SegFormers-4 pipeline showing bounding boxes tracked by DeepSORT



(d) SegFormers-4 segmentation mask showing overlapping class detection in 4 class model

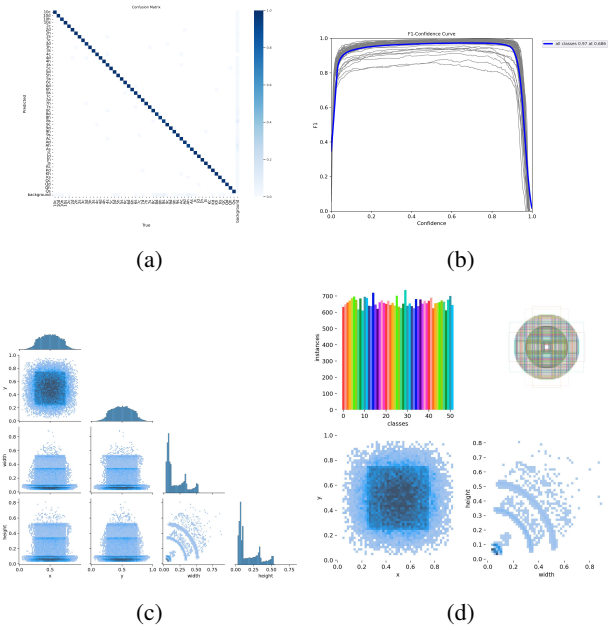
(e) SegFormers-13 pipeline showing bounding boxes tracked by DeepSORT



(f) SegFormers-4 segmentation mask showing segmentation mask class detections

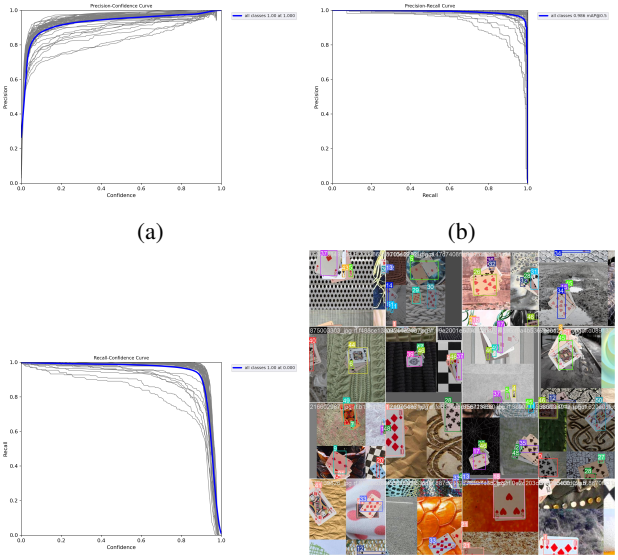
Figure 6: Six interesting outputs from our image pipelines

D. YOLOv5 Training Graphs (52-class Model)



(a) (b) (c) (d)

Figure 7: (a) Confusion Matrix (b) F1 Curve (c) Labels Correlogram (d) Labels



(a) (b) (c) (d)

Figure 8: (a) P Curve (b) PR Curve (c) R Curve (d) Training Batch 0