EPOCH - Evolving Poker Odds Calculations with Hand Analysis

Evaluation and development of a poker playing card classifier

Simon ISLER
simonci

Robin MEIER
robinme

Mathias Birk OLSEN
mathiasbol

IKT450

2023-12-08

Abstract

This study aims at developing a poker playing card classifier, capable of both predicting the number and suit of poker cards found in an input image. To achieve this, the team analysed three different architectures: You Only Look Once (YOLO), Faster Region-based Convolutional Neural Network (Faster R-CNN), and Detection Transformer (DETR).

In order to train the classifier, the team used a publicly available dataset, consisting of 20′000 playing card images, which were recorded under varying light conditions and differing camera angle settings. Since the dataset was labelled with a YOLO-specific format, the dataset has been preprocessed for use with the Faster R-CNN and DETR implementations.

The analysis showed that especially the YOLOv8 model achieved high accuracy, outperforming the Faster-RCNN and DETR models. Faster R-CNN also showed promising results. However, DETR lagged behind due to issues experienced in the implementation phase. The team concludes that while YOLOv8 is currently the most suitable model for this application, further research, especially on DETR, is needed for an extensive evaluation. Future work will include addressing the challenges of DETR and integrating the developed card classifier into a mobile application for real-time analysis of poker hands.

Contents

1 Introduction

The goal of this project is to build a playing card detector as part of the final project for the Deep Neural Networks course at the University of Agder. The model should be able to precisely classify the suit as well as the number of one or more playing cards in a given input image. The main idea is to allow beginners to see what poker hand they have and to check their winning chances. This report however only covers the evaluation and development of a poker card detector. The development of a mobile application is out of scope and is subject to future work. Three different object detection architectures are evaluated in achieving the above mentioned goal: You Only Look Once (YOLO) [1], Faster Region based Convolutional Neural Network (Faster R-CNN) [2] and Detection Transformer (DETR) [3].

2 Background information

Object detection has been a field of study which has seen massive improvements in recent years and multiple different approaches emerging: Non-neural approaches like SIFT or HOG and neural network based approaches, which this project focuses on.

Object detection algorithms can be divided into two main categories: Single-stage and multi-stage detectors. The main difference between the two is the number of steps required to detect objects in an image. Multi-stage detectors use multiple steps to detect objects, while single-stage detectors use only one step. Single-stage detectors are faster, but less accurate, while multi-stage detectors are slower, but more accurate in most cases [4].

Original object detection system were multi-staged. The original R-CNN model e.g. used multiple separately trained models [5]. A CNN to generate the feature maps of the input region proposals, created using mechanisms like selective search, and Support Vector Machines (SVM) trained on each class for classification. This approach leads to slow and inefficient pipelines, since each feature map needs to be run through each SVM. Additionally training is complicated. Later improvements to R-CNN combined the feature extraction and classification into a single convolutional network with sibling dense outputs for softmax classification and bounding box regression [6]. In Faster R-CNN the introduction of the Region Proposal Network (RPN) allowed end-to-end training of the pipeline from generating the region proposals and the resulting classification and bounding box regression [2].

The YOLO system is, since its inception in 2015, designed as a single-stage detector, which defines object detection as a regression problem [7]. YOLO directly predicts bounding box coordinates and class probabilities in one pass through the network. Rather than splitting the image up into multiple sections and then running a neural network multiple times through the sections, YOLO aims to only look once at the image and compute the output on the whole image, keeping the global context. Accordingly, this architecture reduces the complexity of the process and enhances the speed of the classification as well. The increased speed is crucial for real-time applications, for example.

The aim of DETR [3] as designed by Facebook AI research was to apply the at the time new state-of-the-art textual models to the field of machine vision and object detection, while reducing the complexity compared to the era state-of-the-art object detection models. As with YOLO, the DETR model is a single-stage detector. But where YOLO and related models depend on post-processing like NMS, the DETR model outputs the final set of predictions directly [3]. This however moves a lot of the computational complexity back into the model, reducing its capacities in real-time systems [8], in the trade-off for potentially better accuracy.

3 Methodology

This section describes the methodology used to solve the detection problem. First, the dataset is described, including the preprocessing steps. Then, the different architectures that were implemented are described.

3.1 Dataset

A dataset of playing cards was found online [9]. The dataset consists of a total of 20′000 images, each containing one or more playing cards. The author of the Kaggle dataset itself used a script available on GitHub [10] to generate the images. Before executing the script, the author took 20s to 30s videos of all 52 playing cards with variable light conditions and angles. The script then processed the images using open-cv. Moreover, to add more details to the background of the image, the DTD dataset [11] was used to add a random texture to the background. Finally, the images get resized to 416x416 pixels and are split into a train/test/validation set (70/20/10% split).

Figure 1 shows some sample images from the dataset. As can be seen, the images are labelled with bounding boxes around the suit and number of the card. In total, there are 52 different classes, one for each card.

Figure 1: Sample images from the dataset with ground truth labels

3.1.1 Preprocessing

The chosen dataset for the project came pre-annotated in the YOLO format [12]. This meant that the YOLO implementation and training could be underway fast. However, the two other models were created using the Facebook Research Detectron2 Library [13]. The models in the default Detectron model zoo and the DETR models use the COCO format [14]. Because of

this, the dataset had to be converted before it could be used with the Faster R-CNN and DETR models. This conversion was done using a tool developed by Kim TaeYoung [15], with some small modifications.

3.2 Architectures

Three different architectures were implemented to solve the detection problem. The following sections describe the different architectures in detail.

3.2.1 YOLO

The main goal of the YOLO model is to find a good balance between speed and accuracy, while also maintaining a small model size. Moreover, training a YOLO model does not require a high-end GPU, which eases the training process and makes it accessible to a wider audience. YOLO was first presented in 2015 by Joseph Redmon et. al [7] and has since then been greatly improved. In this project, the current YOLOv8 is used, which has been released in 2023 [16].

The way the original YOLO algorithm works is as follows. First, the input image gets divided into a square grid with the dimensions S x S (hyperparameter). If a center of an object falls into a grid cell, this specific cell is then responsible to detect that object. In addition, each cell will predict B bounding boxes and an additional confidence score, which shows how certain the model is that there is an object in this cell. The confidence score gets calculated using the Intersection over Union (IoU) between the predicted and ground truth bounding box. In essence, the IoU measures how much the predicted bounding box overlaps with the ground truth bounding box. During training YOLO assigns multiple bounding boxes to a cell, however it will only keep the bounding box with the highest confidence score. Since the YOLO algorithms is able to classify objects, each grid cell also contains a vector with length C (the number of classes), which includes the class probabilities. Overall, the final output vector can be described as S x S x (B * 5 + C), where 5 is the length of the quintuple describing the bounding box (x, y, bbox_width, bbox_height, confidence_score). Hereby, the x and y coordinates define the center of the bounding box [7]. Figure 2 visualizes the simplified steps from splitting up the image into a grid, to the final detections.

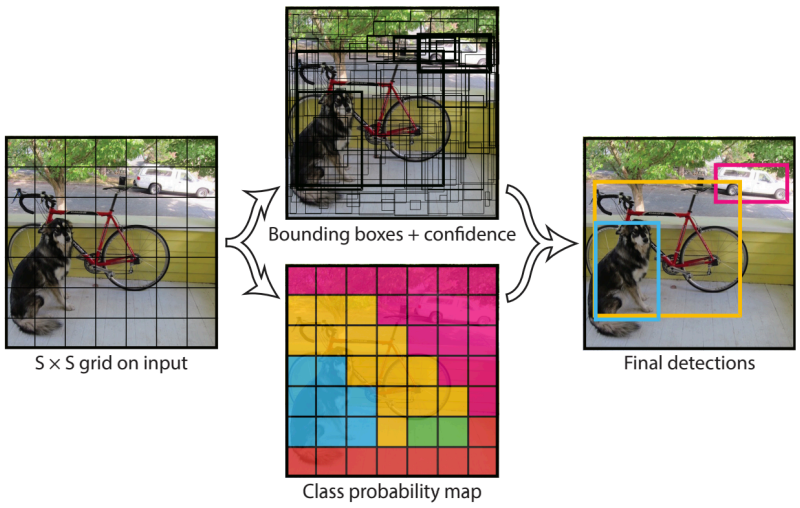

Figure 2: Original YOLO algorithm flow [7]

3.2.1.1 Network

The YOLOv8 architecture was used to train the model on the given dataset. It utilizes several essential components to perform object detection tasks. It features a backbone consisting of a series of convolutional layers, which extract relevant features from the input image. The SPPF module, along with additional convolutional layers, processes these features across various scales, while an upsample layer enhances the feature map resolution. The C2f module merges high-level features with contextual data, boosting recognition accuracy. The final stage, the detection module, utilizes a combination of convolutional and linear layers to transform high-dimensional features into final outputs of bounding boxes and object classes. As seen in the following image, the head is fully decoupled from the backbone. To summarize, the head basically takes the generated feature maps from the backbone and then further processes them to produce the final output (bounding boxes and class probabilities). This design ensures YOLOv8 is both lightweight and efficient, maintaining high accuracy in detection [17].

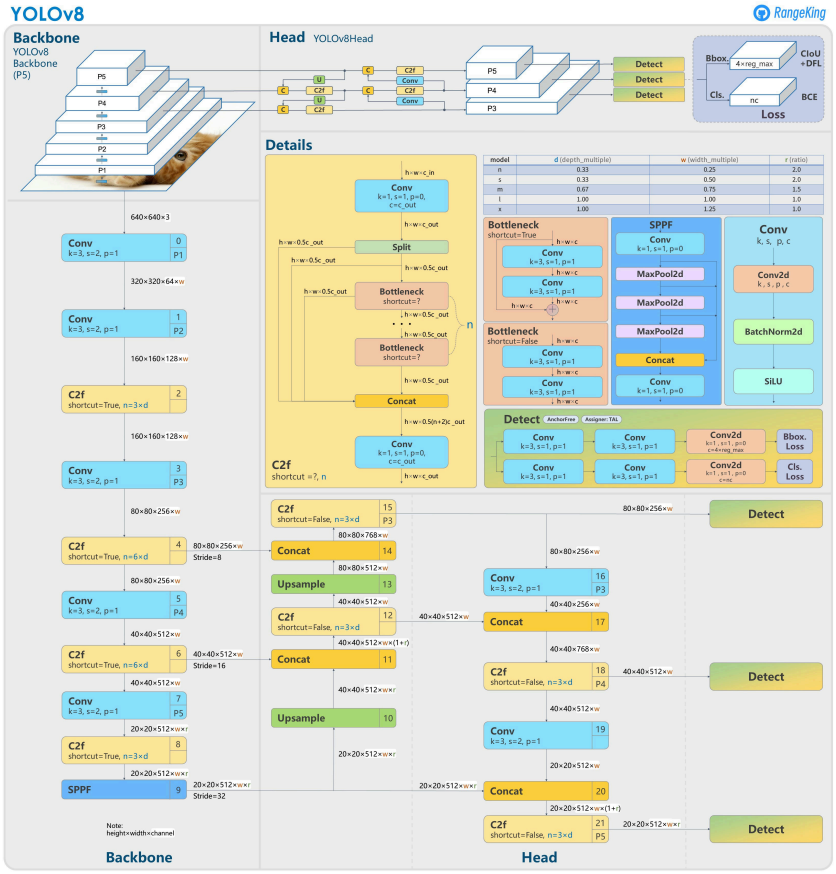

Figure 3: YOLOv8 architecture visualised [17]

3.2.1.2 Training

The ultralytics package [18] was used, in order to load a pre-trained model. The models have been pre-trained on the COCO dataset [19]. The ultralytics package provides different versions of YOLOv8, which mainly differ in the number of trainable parameters, where a higher number usually results in an increased mAP. In this project, the large YOLOv8 model was trained on the dataset mentioned in Section 3.1 for 100 epochs. The model started to train with an initial learning rate of 0.01 and was updated slightly during the training process.

3.2.2 Faster R-CNN

3.2.2.1 Network

The evaluation of the Faster R-CNN architecture was done using the pre-trained Faster R-CNN Model X101-FPN from Metas Detectron2 Model Zoo [20]. The model uses the ResNeXt-101 network in the 32x8d setting [21] as the network backbone with the addition of a Feature Pyramid Network (FPN) [22].

The ResNeXt network extracts feature maps at four different scales using bottleneck blocks, defined as:

```
BottleneckBlock(
 (conv1): Conv2d(
   256, 256, kernel_size=(1, 1), stride=(1, 1), bias=False
   (norm): FrozenBatchNorm2d(num_features=256, eps=1e-05)
 )
 (conv2): Conv2d(
   256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), groups=32, bias=False
   (norm): FrozenBatchNorm2d(num_features=256, eps=1e-05)
 )
 (conv3): Conv2d(
   256, 256, kernel_size=(1, 1), stride=(1, 1), bias=False
   (norm): FrozenBatchNorm2d(num_features=256, eps=1e-05)
 )
```

One bottleneck block calculates multiple convolutions in parallel and aggregates the results. The number of parallel convolutions i.e. the cardinality C used is 32. The four scales in the network use 3, 4, 23, and 3 bottleneck blocks respectively. At each scale the FPN takes an upsampled version of the lower-scale feature map and a lateral output of the corresponding original feature map of the same scale from the ResNeXt network to create a high-value feature map [23]. This should theoretically allow the network to be able to detect small features just as well as big features. The generated feature maps get fed into a Region Proposal Network (RPN), which generates 1000 object region proposals. The object region proposals and feature maps get pooled using ROIAlign pooling [24], flattened and fed into two fully connected layers of size 1024 and two sibling outputs. The first output of size 53 is used to classify each individual card type in addition of a background class. The second output of size 208 (52x4) regresses bounding boxes. Figure 4 shows the architecture of BaseRCNN FPN [25] on which X101-FPN is based on. Note however, that the number of convolutional layers and outputs are not the same as in the architecture used in the project.

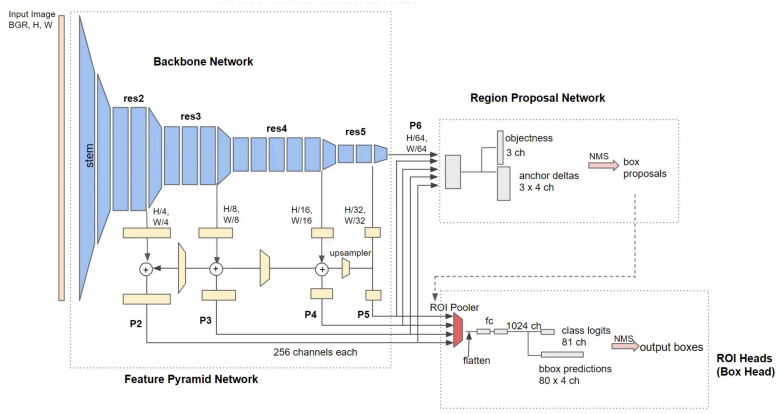

Figure 4: Architecture of BaseRCNN FPN [25]

3.2.2.2 Training

The model was loaded with pre-trained ImageNet weights and fine-tuned on the dataset described in Section 3.1. The training was run over 10000 iterations, a batch-size of 512 and a learning rate of 0.00125.

3.2.3 DETR

As stated in Section 2, the aim of the DETR model is to reduce the complexity of object detection models. The basis for this comes from looking at the problem from a fixed set perspective. This means that DETR will try to predict N objects in a picture every time. At the topmost layer, it simplifies the problem, by removing the need for the number of objects as well as their class and bounding boxes. Where N with no class simply is marked as $\emptyset$, and disregarded in the end [3].

Further down, it is the basis for the use of the transformer model on this problem space and allows for loss calculation.

3.2.3.1 Network

Following the goal of reduced complexity, the architecture flow is also quite simple. This can be seen in Figure 5. Where the first parallel to its textual counterparts is visible in the form of the initial encoding of the data and the positional embedding.

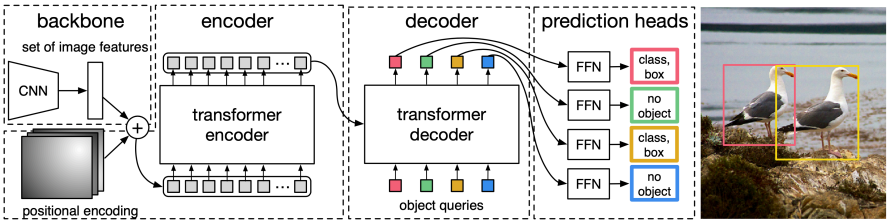

Figure 5: Original DETR algorithm flow [3]

As with the transformers used in language models, the data is first encoded into a latent space [26]. With respect to object detection this is done using a Convolutional Neural Network. This is the backbone of the model, which in the case of the implementation used is a ResNet [13]. Since transformers work on sequences, the feature maps are flattened along the filter axes before they are positionally encoded.

All of this is then fed into a series of transformer encoders, each constructed by a single multi-headed attention layer and a feed forward network, with batch normalisation and residual connections in between and on the output.

Once encoded the output is sent to the transformer decoder. The decoder takes a series of N object queries and applies multi-head attention (MHA) over them. These object queries are learned and will eventually point towards a separate part of the image [3], [27], which makes it the natural query part of the next block of the decoder. Here the MHA layer combines the keys, values and positional encodings from the encoder stage with the queries of the objects, before finally sending the data out to the two FFN networks, that make up the head of the model. Given that the model makes a fixed set prediction, one side of the head predicts probability for the class of the object, and the other the bounding box.

The output $\hat{y}$ then becomes a set of N elements with the type $\left(\hat{c_i}, \hat{b}_i\right)$. Where c_i is a prediction of the probability of classes of this object. b_i is a vector of the bounding box coordinates (x, y, w, h)

Further details around the architecture can be read in [3], especially in Appendix A.3

3.2.3.2 Loss

One of the main challenges in object detection is the calculation of loss. This is again simplified by the fixed set prediction. Where the ground truth for the image y now can be expressed as a set padded with $\emptyset$ elements to the size N. A permutation of $\hat{y}$ can then be found where the predictions will be mapped to the ground truths, allowing the loss to be calculated be only taking the cases where c_i is not $\emptyset$ [3], [27].

$$\mathcal{L}_{\text{match}}\left(y_i, \hat{y}_{\sigma(i)}\right) = -\mathbb{1}_{\{c_i \neq \emptyset\}} \hat{P}_{\sigma(i)}(c_i) + \mathbb{1}_{\{c_i = \emptyset\}} \mathcal{L}_{\text{box}}\left(b_i, \hat{b}_{\sigma(i)}\right)$$

equation 1: DETR Loss calculation [3]

Where $\mathcal{L}_{\text{box}}$ is a similar to YOLOs IoU as described in Section 3.2.1

3.2.3.3 Training

The models' backbone was loaded with ImageNet weights and trained upon the dataset. The model was trained over a series of iterations and two different backbones, with a batch size of 2 and a learning rate of 0.0001.

4 Results

This section contains the results of the implemented models. The results are presented in the form of graphs and tables, in order to evaluate the performance and accuracy of the classifiers. Finally, we compare the implemented architectures against each other to see which one performed best.

4.1 YOLO

As described in Section 3.2.1, the team used the YOLOv8 version to train the model. The results of this model are presented in this chapter.

4.1.1 Training

Training lasted for 18h 44m and 19s. This equals about 11.25 minutes per epoch. The performance of the model can be evaluated in several ways. First, the team analysed the mean average precision (mAP), which calculates the average AP values across all classes. The average precision (AP) itself can be considered as the area under the precision-recall curve [19]. Figure 6 shows both the mAP50 and mAP50-95 scores. These diagrams show the mean average precision for different intersections over union thresholds. While the mAP50 shows the mAP for an IoU threshold of 0.5, the mAP50-95 diagram shows the mAP with differing IoU thresholds between 0.5 and 0.95. The main difference between these two diagrams is, that the mAP50 mainly displays how well the model predicts generally predicts the bounding boxes, whereas the mAP50-95 metric shows how the model performs with a more strict threshold.

The model reached a final mAP50-95 value of 0.9718 after 100 epochs. In addition, the mAP50 metric reached 0.9943, showing that the easier detections were almost always detected.

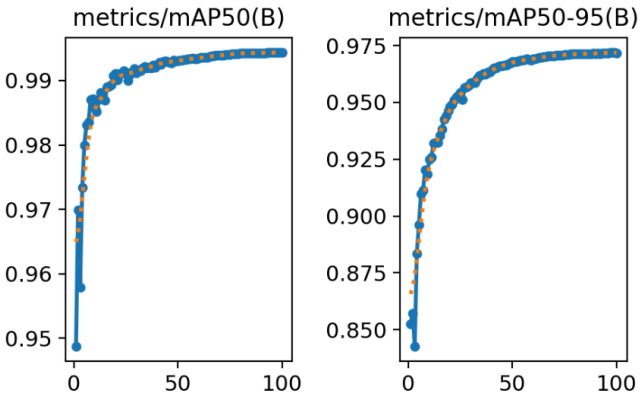

Figure 6: mAP metrics for YOLOv8 model

Figure 7 shows the losses for the bounding box prediction and class detection.

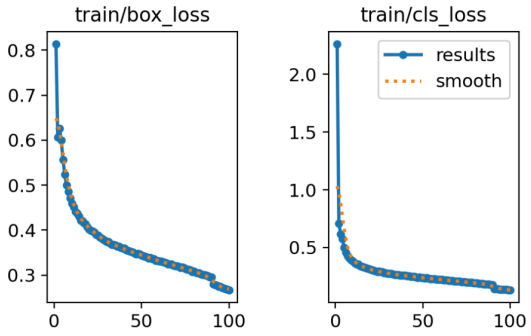

Figure 7: box and class los for YOLOv8 model

Figure 8 shows the confusion matrix for the classifier. Instead of showing the raw counts, the normalized values are displayed in order to have a better understanding of the proportions. Overall, the model was able to detect all classes very well.

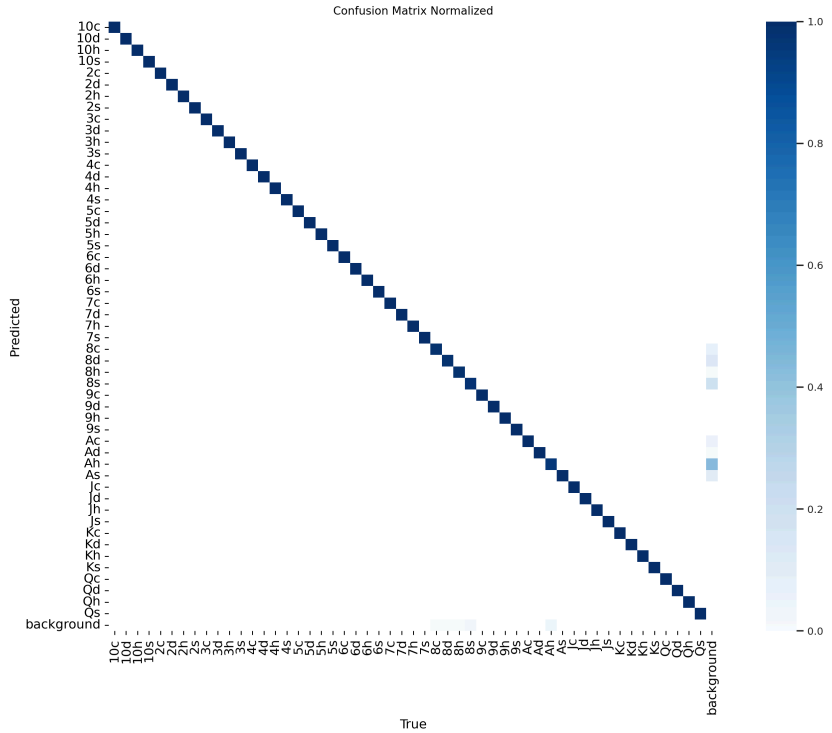

Figure 8: Normalized confusion matrix for YOLOv8 model

4.1.2 Validation

The validation set consisted of 4000 images, which equals 20% of the whole dataset. Figure 9 shows a sample image from the validation set with the predictions as annotations. It can be seen, that the model was able to classify and identify almost all cards correctly and with high confidence.

Figure 9: YOLOv8 validation sample

4.1.3 Testing on a different card set

After training the model with a given dataset, the model was tested on a different set of playing cards. Overall, the model was able to classify the most cards in Figure 10 correctly, however it performed worse than with the testing set of the original dataset. This might be a sign of overfitting, as the model does seem to perform worse with a different set of playing cards.

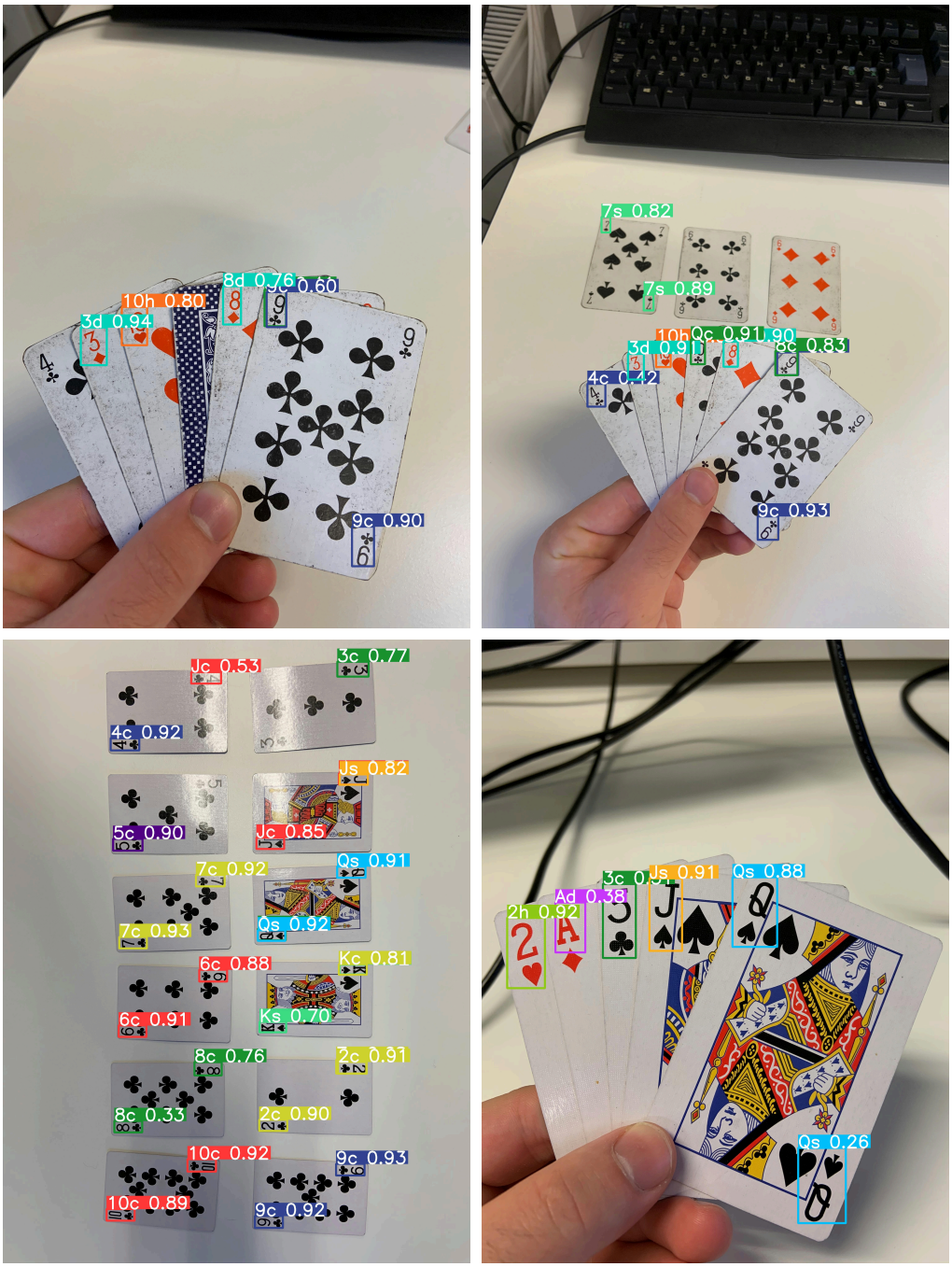

Figure 10: Sample predictions from a dataset with different playing cards

4.2 Faster R-CNN

4.2.1 Training

Training of Faster R-CNN ran over a time of 1 hour 28 minutes and 45 seconds, with an average time of 0.5275 seconds per iteration. The maximum reached class accuracy is 0.9658 after 9499 iterations. Interesting to note is, that the model only started being able to identify cards after around 700 iterations. This is seen in Figure 11.

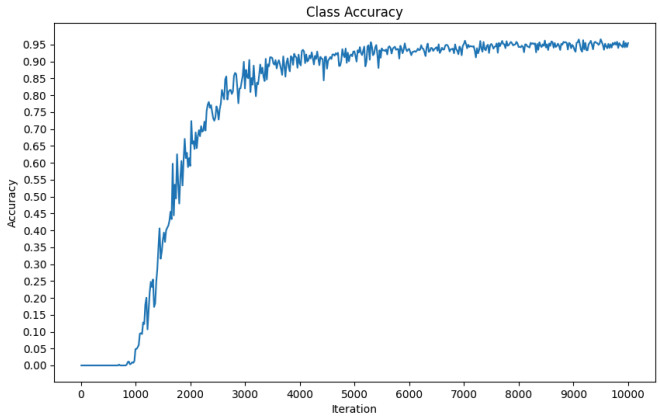

Figure 11: Training accuracy of Faster R-CNN over 10000 iterations

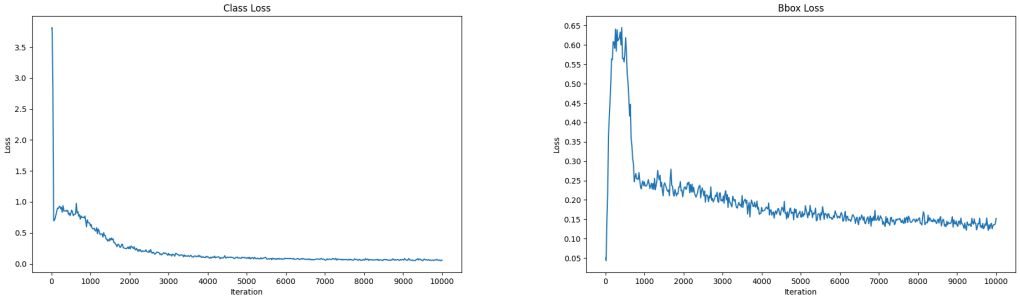

Figure 12: Training loss of Faster R-CNN over 10000 iterations

4.2.2 Testing

The highest Average Precision (AP) on the testing dataset reached is 77.5% for mAP50-95 (See Section A.1 for detailed results). The highest reached per class AP was the 9 of Spades with a score of 85% and the lowest was the Ace of Spades with an AP score of 59%. An interesting observation in Figure 13 is, that the hardest symbols to recognize seem to be the Aces, with the lowest three AP scores being the Ace of Spades, Hearts and Diamonds. Also difficult were the numbers 10 and 8. It seems no suite in particular, was harder or easier to identify.

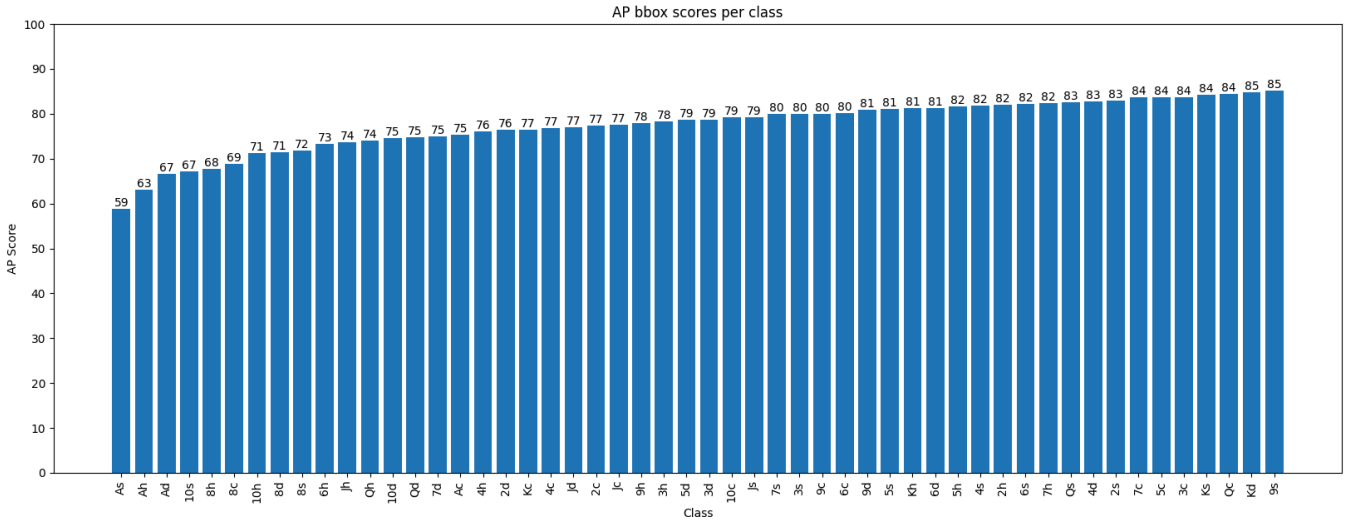

Figure 13: Faster R-CNN Average AP Scores

4.3 DETR

The transformers have been overtaking the machine learning world in the past years and this is why their application on object detection was chosen for comparison with YOLO and Faster R-CNN in this project. However, the results obtained from DETR were rather bad.

Figure 14 shows a loss curve, where it can be seen that the model learns very early on, but quickly stagnates and never picks up learning again. This same pattern was true for longer training times and different backbones.

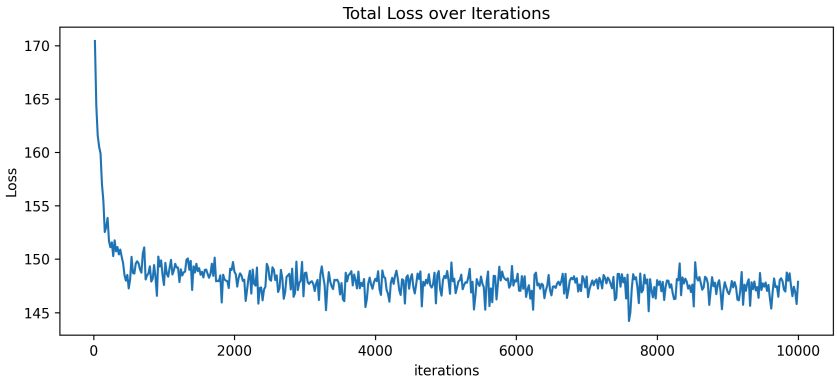

Figure 14: 10k iteration training with DETR with a r50 backbone

The additional results can be found in Section A.2. Running interference on the images follows the loss curve above and gives poor results.

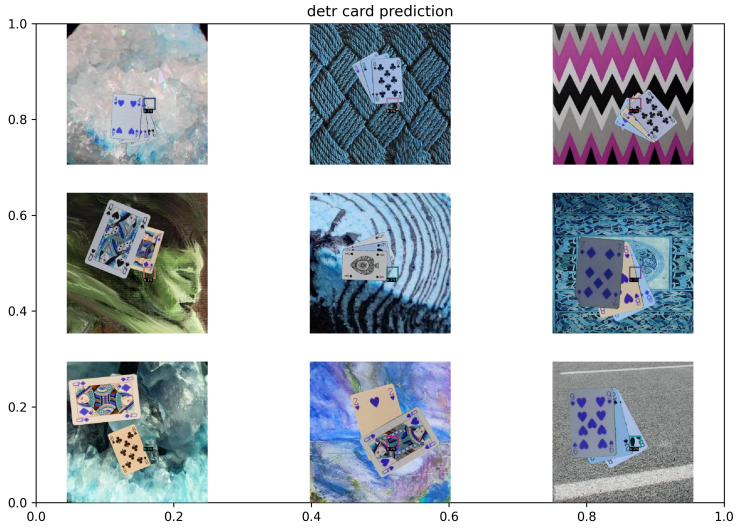

Figure 15: Results from the DETR model

4.3.1 Analysis

The results here are not in line with the results of the experiments conducted with the offical DETR publications [3], [8] and the only conclusion is that there is a fault in the setup and implementations for this project, prohibiting the model from functioning correctly. However, where these faults lie in the implementation are at the time of writing not known. Multiple configurations were tried on both short and long training sessions, though always with the same result.

Multiple hypetheses could be made to explain this result. The most obvious being the mismatch between the data and the model. The data might need to be augmented to fit the specific model architecture better. Another lead would be the backbone, which should come ready with ImageNet weight, but again there might be a mismatch between the backbone and the pixel mean [13] of DETR.

4.4 Overall comparison of the models

Due to the shortcomings of the DETR implementation, it does not make sense to compare its results with the YOLO and Faster R-CNN models. But between the two successful models, two distinct results can be observed. The YOLO implementation reaches very high precision, seen in both the mAP50 and mAP50-95 metrics, where the Faster R-CNN model does not perform as well. The Faster R-CNN architecture is generally good at predicting as the mAP50 shows, however when we look at a more restricted set, we see that it loses its accuracy when a higher threshold is used. The opposite is true for the YOLO model, which keeps its high precision across the classes as higher thresholds are added in. The second observation that can be made, is that the YOLO model after very few epochs shot past the best result of the Faster R-CNN. So even though the training of the Faster R-CNN was significantly shorter, the YOLO model was able to get better results at the same time or faster. The remaining 17 hours of training simply added extra points to the model.

Both these results are completely inline with the literature research and expectations, since YOLO is a much more modern model. It would however be interesting to see the results of Faster R-CNN with the same training times as the YOLO model.

Metrics	**Faster R-CNN**	**YOLO**
Training Time	1h 28m 45s	18h 44m 19s
mAP50	0.975	0.9943
mAP50-95	0.775	0.9718

Table 1: Overall comparison of the implemented models

Reflecting on the results, it is clear who the winning model is for the use case at hand. However, in a more in-depth analysis, it would have been beneficial to set specific benchmarks for comparison. An example for this could be an inference speed metric, like seen in [8], since this is an important factor of usability of the model in final product.

5 Conclusion

As seen in Section 4.4, the winning model for the use case of card predictions is YOLOv8. This is however, not a fully evaluated result, given the subpar results the DETR model produced. The conclusion is, that the YOLOv8 model with additional work and a more extensive dataset would be a suitable candidate.

The comparison between Faster R-CNN and YOLO was as expected, as it was never considered to be a real competitor, but mostly served as a baseline and historical perspective.

5.1 Future

In the future of this project, the DETR implementation could be analysed more in depth, so that it can be added to the comparison of the YOLO and Faster R-CNN models and may serve as an alternative solution. As stated in Section 4.4 additional metrics could also be added to the comparison in general. As an extension of this a new more modern version of the DETR implementation was published in the summer of 2023, namely Real Time Detection Transformer or RT-DETR [8]. This model claims to be the new state-of-the-art in both speed and accuracy, and should therefore be a consideration for this project as well.

In the grander scope of the project, the card prediction system can be applied to its original intent, poker hand analysis. This would take the form of a mobile app for the users' phone, so they can take a picture of their current poker hand and identify how good their winning chances are.

List of Figures

List of Tables

Bibliography

[1] C.-Y. Wang, A. Bochkovskiy, and H.-Y. M. Liao, "YOLOv7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors". Accessed: Oct. 25, 2023. [Online]. Available: http://arxiv.org/abs/2207.02696

[2] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks". Accessed: Oct. 28, 2023. [Online]. Available: http://arxiv.org/abs/1506.01497

[3] N. Carion, F. Massa, G. Synnaeve, N. Usunier, A. Kirillov, and S. Zagoruyko, "End-to-End Object Detection with Transformers". Accessed: Oct. 25, 2023. [Online]. Available: http://arxiv.org/abs/2005.12872

[4] T. Diwan, G. Anirudh, and J. V. Tembhurne, "Object detection using YOLO: challenges, architectural successors, datasets and applications", *Multimedia Tools and Applications*, vol. 82, no. 6, pp. 9243–9275, Mar. 2023, doi: 10.1007/s11042-022-13644-y.

[5] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation". Accessed: Oct. 25, 2023. [Online]. Available: http://arxiv.org/abs/1311.2524

[6] R. Girshick, "Fast R-CNN". Accessed: Oct. 25, 2023. [Online]. Available: http://arxiv.org/abs/1504.08083

[7] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection", in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2016, pp. 779–788. doi: 10.1109/CVPR.2016.91.

[8] W. Lv *et al.*, "DETRs Beat YOLOs on Real-time Object Detection". Accessed: Nov. 23, 2023. [Online]. Available: http://arxiv.org/abs/2304.08069

[9] "Playing Cards Object Detection Dataset". Accessed: Nov. 23, 2023. [Online]. Available: https://www.kaggle.com/datasets/andy8744/playing-cards-object-detection-dataset

[10] geaxgx and geaxgx, "Creating a playing cards dataset". Accessed: Nov. 23, 2023. [Online]. Available: https://github.com/geaxgx/playing-card-detection/blob/master/creating_playing_cards_dataset.ipynb

[11] "Describable Textures Dataset". Accessed: Nov. 23, 2023. [Online]. Available: https://www.robots.ox.ac.uk/~vgg/data/dtd/

[12] "YOLO - How to export and import data in YOLO format". Accessed: Nov. 22, 2023. [Online]. Available: https://opencv.github.io/docs/manual/advanced/formats/format-yolo/

[13] "DETR implemetation". [Online]. Available: https://github.com/facebookresearch/detr

[14] "COCO - Common Objects in Context". Accessed: Nov. 22, 2023. [Online]. Available: https://cocodataset.org/#format-data

[15] T. Y. Kim, "Yolo-to-COCO-format-converter". [Online]. Available: https://github.com/Taeyoung96/Yolo-to-COCO-format-converter

[16] M. Hussain, "YOLO-v1 to YOLO-v8, the Rise of YOLO and Its Complementary Nature toward Digital Manufacturing and Industrial Defect Detection", *Machines*, vol. 11, no. 7, p. 677, Jul. 2023, doi: 10.3390/machines11070677.

[17] RangeKing, “Brief summary of YOLOv8 model structure · Issue #189 · ultralytics/ultralytics”. Accessed: Nov. 21, 2023. [Online]. Available: https://github.com/ultralytics/ultralytics/issues/189

[18] Ultralytics, “Ultralytics”. Accessed: Nov. 23, 2023. [Online]. Available: https://github.com/ultralytics/ultralytics

[19] Ultralytics, “YOLO Performance Metrics”. Accessed: Nov. 23, 2023. [Online]. Available: https://docs.ultralytics.com/guides/yolo-performance-metrics

[20] Yuxin Wu and Alexander Kirillov and Francisco Massa and Wan-Yen Lo and Ross Girshick, “Detectron2”. [Online]. Available: https://github.com/facebookresearch/detectron2

[21] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, “Aggregated Residual Transformations for Deep Neural Networks”. Accessed: Nov. 16, 2023. [Online]. Available: http://arxiv.org/abs/1611.05431

[22] T.-Y. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan, and S. Belongie, “Feature Pyramid Networks for Object Detection”. Accessed: Oct. 29, 2023. [Online]. Available: http://arxiv.org/abs/1612.03144

[23] “detectron2.modeling — detectron2 0.4.1 documentation”. Accessed: Nov. 16, 2023. [Online]. Available: https://detectron2.readthedocs.io/en/v0.4.1/modules/modeling.html#detectron2.modeling.FPN

[24] K. He, G. Gkioxari, P. Dollár, and R. Girshick, “Mask R-CNN”. Accessed: Oct. 28, 2023. [Online]. Available: http://arxiv.org/abs/1703.06870

[25] H. Honda, “Digging into Detectron 2”. Accessed: Nov. 16, 2023. [Online]. Available: https://medium.com/@hirotoschwert/digging-into-detectron-2-47b2e794fabd

[26] A. Vaswani *et al.*, “Attention Is All You Need”. Accessed: Oct. 25, 2023. [Online]. Available: http://arxiv.org/abs/1706.03762

[27] J. Briones, “Object Detection with Transformers”. Accessed: Nov. 23, 2023. [Online]. Available: https://medium.com/swlh/object-detection-with-transformers-437217a3d62e

A Appendix

A.1 Faster R-CNN Testing Results

Metric	IoU	Area	MaxDets	Value
Average Precision	0.50:0.95	all	100	0.775
Average Precision	0.50	all	100	0.975
Average Precision	0.75	all	100	0.966
Average Precision	0.50:0.95	small	100	0.769
Average Precision	0.50:0.95	medium	100	0.811
Average Precision	0.50:0.95	large	100	−1.000
Average Recall	0.50:0.95	all	1	0.524
Average Recall	0.50:0.95	all	10	0.831
Average Recall	0.50:0.95	all	100	0.831
Average Recall	0.50:0.95	small	100	0.825
Average Recall	0.50:0.95	medium	100	0.850
Average Recall	0.50:0.95	large	100	−1.000

Table 2: Faster R-CNN Average AP and AR Scores over different IoU

Category	AP	Category	AP	Category	AP
10c	79.214	10d	74.506	10h	71.159
10s	67.214	2c	77.298	2d	76.479
2h	82.058	2s	82.993	3c	83.734
3d	78.617	3h	78.325	3s	79.948
4c	76.841	4d	82.728	4h	76.056
4s	81.742	5c	83.685	5d	78.577
5h	81.589	5s	81.038	6c	80.070
6d	81.352	6h	73.282	6s	82.232
7c	83.683	7d	75.036	7h	82.283
7s	79.880	8c	68.776	8d	71.493
8h	67.800	8s	71.871	9c	79.963
9d	80.954	9h	77.982	9s	85.166
Ac	75.283	Ad	66.527	Ah	63.066
As	58.743	Jc	77.495	Jd	77.059
Jh	73.737	Js	79.245	Kc	76.518
Kd	84.831	Kh	81.249	Ks	84.276
Qc	84.376	Qd	74.764	Qh	74.105
Qs	82.532				

Table 3: Faster R-CNN Per-category bbox AP

A.2 Additional DETR results

A.2.1 DETR with r50 backbone 10′000 iterations

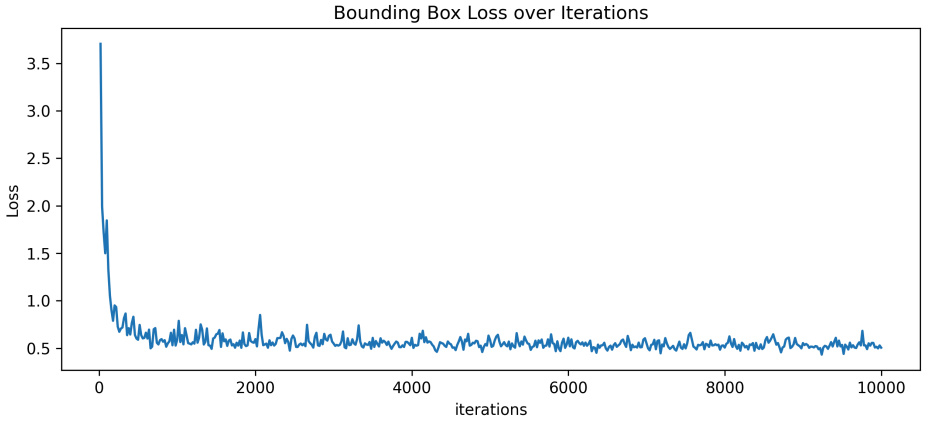

Figure 16: 10k iteration training with DETR with a r50 backbone

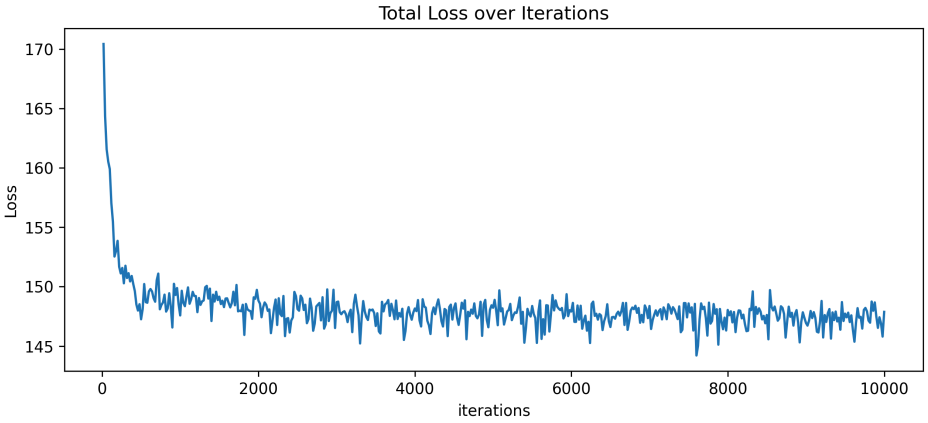

Figure 17: 10k iteration training with DETR with a r50 backbone

Figure 18: 10k iteration training with DETR with a r50 backbone

A.2.2 DETR with r50 backbone 30´000 iterations

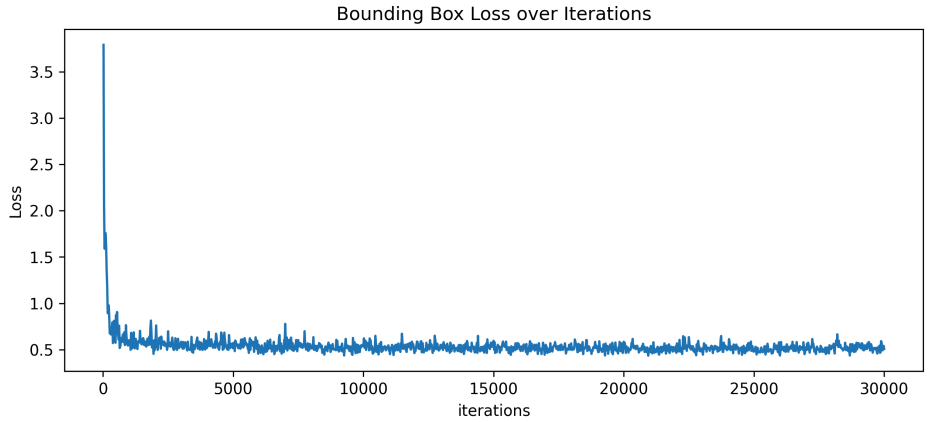

Figure 19: 30k iteration training with DETR with a r50 backbone

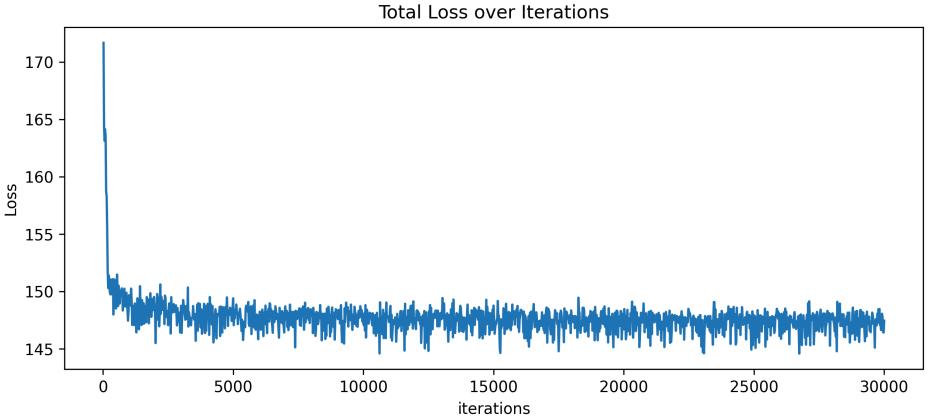

Figure 20: 30k iteration training with DETR with a r50 backbone

Figure 21: 30k iteration training with DETR with a r50 backbone

A.2.3 DETR with r50 backbone 100′000 iterations

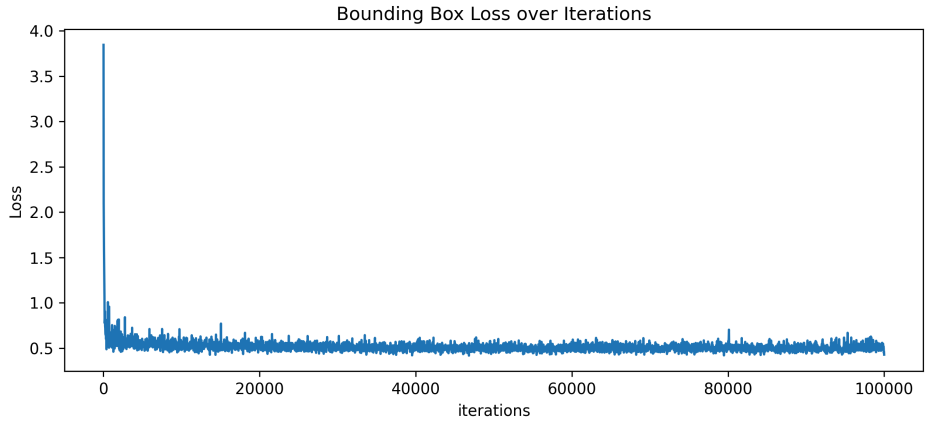

Figure 22: 100k iteration training with DETR with a r50 backbone

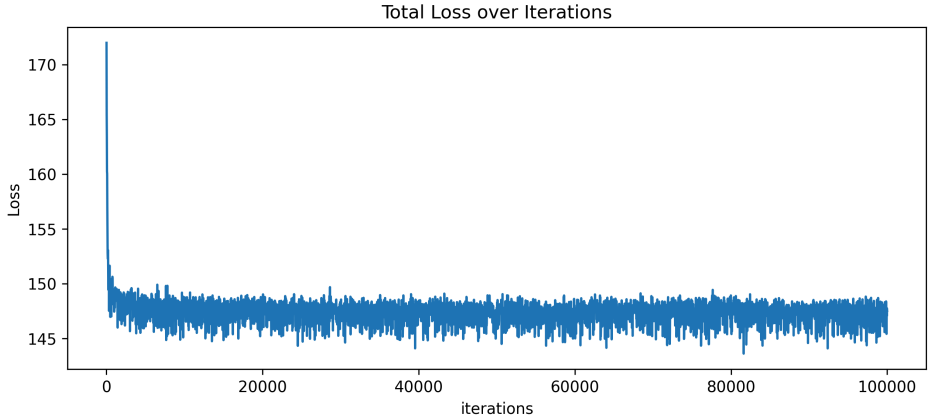

Figure 23: 100k iteration training with DETR with a r50 backbone

Figure 24: 100k iteration training with DETR with a r50 backbone

A.2.4 DETR with r101 backbone 40′000 iterations

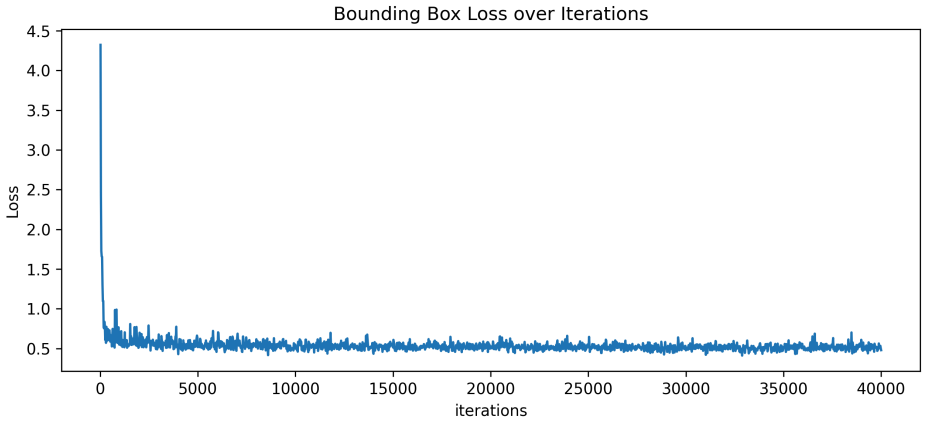

Figure 25: 40k iteration training with DETR with a r101 backbone

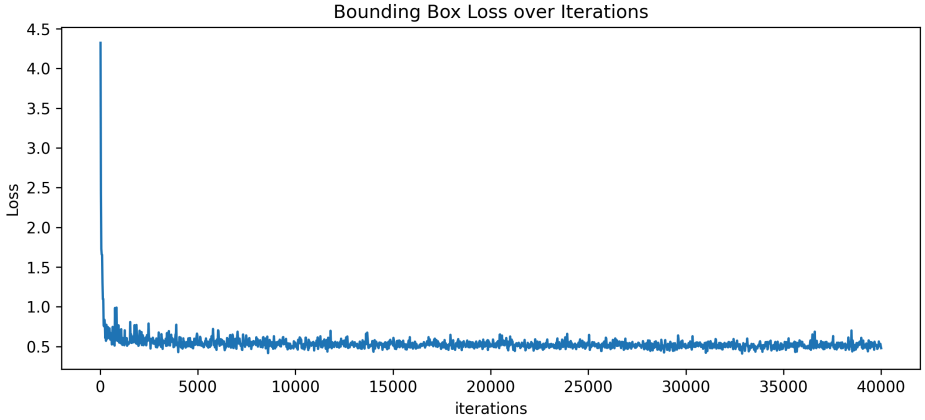

Figure 26: 40k iteration training with DETR with a r101 backbone

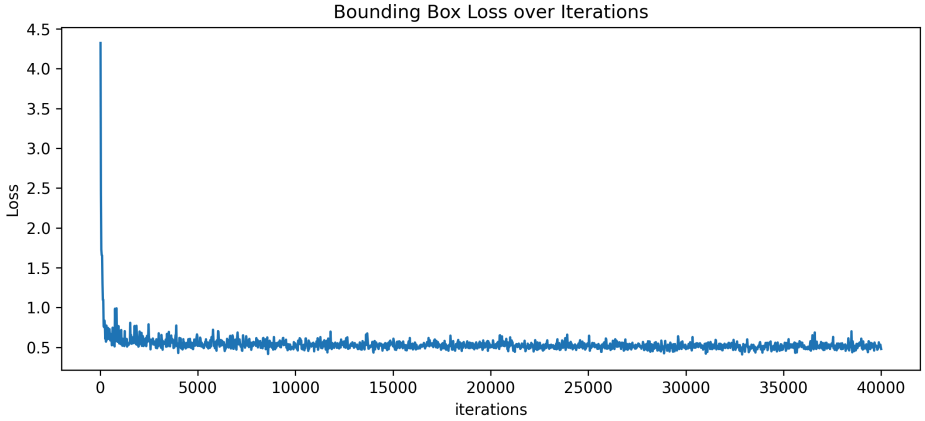

Figure 27: 40k iteration training with DETR with a r101 backbone