YOLO for Real-time Detection

Marcel Dokters and Torben Bietendüvel

Contributing authors: mdokters@uos.de; tbietendueve@uos.de;

Abstract

Real-time object detection has been an important topic in recent years. Detecting objects from live-camera feeds and having a fast inference as well as a high precision has been heavily studied in research. In this project we implemented YOLOv1 with the goal of doing inference on a raspberry pi without an accelerator. We used the standard architecture of YOLOv1, but due to computing resource constraints and the goal of testing inference time on a raspberry pi couldn't be reached. For the convolutional part of the network EfficientNet was used.

1 Introduction

Machine learning has been an active topic of interest for the last years. Image classification has become one of the most relevant task done with neural networks and is a branch of artificial intelligence where CNNs (convolutional neural networks) have produced outstanding results. Object detection as a part of image classification has also undergone a change with the rise of CNNs. While early object detection architectures as Faster R-CNN [1] are slow in inference and training, with YOLOv1 a fast network architecture has been presented, that is capable of real time inference[2]. The advantage of real time object detection is obvious. While the real-time capability of YOLOv1 reduces precision, such networks may be used in autonomous driving and other task, where fast inference is needed. Additionally fast inference also means, that the network could be used in low-energy, limited capability hardware systems, such as an raspberry pi. Our project aims at using the current progress in fast CNNs for mobile devices, such as MobileNet and EfficientNet and use it as a base CNN for the YOLOv1 architecture[3][4]. Research has already tested inference on raspberry pi, which may yield a few fps for object detection. While this is not great, for static contexts, like detection of humans in automated surveillance systems, it is enough.

2 Related Work

Object detection has been an active research topic in computer vision task in the last years and according to [5] has undergone a change in recent years from traditional methods to neural networks . Object detection in general covers different kind of tasks. In [6] different kinds of object detection task are presented and will be shortly covered here. The first differentiation is made between the objects themselves. There are object detection tasks, where specific objects as the Mona Lisa or generic objects as cars and cats need to detected. The latter is the more common and also the one which we tried to predict in our project. The second difference of object detection problems is related to the granularity of object detection. In Figure 1 different detection levels are shown. Object classification is to detect which objects are present in the image, without any further information, where they lie in the image. Bounding Box detection is to draw a rectangular bounding box in the image, where the detected object lies. Segmentation tasks are even more fine-granular, detecting the exact shape of the object. In this project, bounding box detection is the object detection task used. The Methods for object detection can be categorized into two main approaches: Traditional and Deep Learning Based approaches. While the traditional methods were mostly researched before 2014, with the rise of CNNs and deep learning in general, deep learning approaches yield better results and are nowadays the preferred solution for object detection. For bounding box object detection, there are in general two main approaches: one-stage and two-stage detectors. One-stage detection directly generates bounding boxes for the objects in one pass of the image through a network. Two-stage detection on the other hand first detects RoI (Regions of Interest), where potential objects could lie. In a second stage, the objects are then detected, classified and bounding boxes are determined with a regressor. One-stage detectors are faster but often lack accuracy, while two-stage detectors are hard to train and have long inference times. One-stage detectors consist of a CNN and afterwards dense layers for bounding box regression. The CNN and dense layers can be trained in one step as or separately as in YOLOv1. For real-time applications the usage of two-stage detectors is not viable and therefore one-stage detectors have to be used. The first one-stage was the YOLO network architecture, which is re implemented in this project[2]. While the original YOLO architecture was already fast but lacked precision, successor version as YOLOv2 achieved greater precision while still being real-time capable[7]. The single shot detector is another one-stage approach, which still can be viable for real-time detection with suited hardware. The evaluation of object detection and training of neural networks in multiple datasets exists, with COCO and Pascal VOC being one of the most famous.[6]

The second part of our project is inference with the YOLO network on a Raspberry Pi without a hardware accelerator, using only the CPU. Multiple research papers have examined to speed up the inference time of CNNs, with the results being smaller and faster CNN as MobileNetV2, which uses a new convolution operator or EfficientNet. These CNN have already been tested on multiple versions of Raspberry Pi with inference time ranging from approximate 100 ms to 300 ms depending on the number of classes.[8]

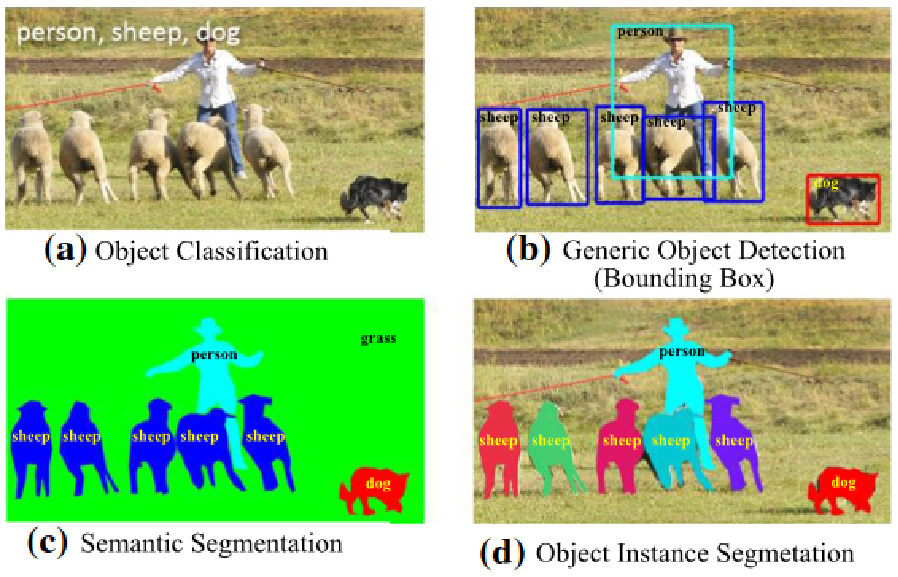

Fig. 1 Different tasks of object detection[6]

3 YOLO v1 Architecture

YOLOv1 is an one-stage deep learning network for real-time object detection. To unify the separate components of object detection into a single neural network for both classification and localization, the YOLOv1 will segmented the image into an S × S grid. For localization, in every grid cell, the network will predict B bounding boxes. Each bounding box contains information about the position and shape of the box (x, y, w, h), as well as a confidence score. The (x, y) represents the center of the box, while the confidence score represents the model's confidence that the box contains an object and how accurately the box predicts that object. The score should be zero when there is no object, and with an object, it should be equivalent to the intersection over union between the box and the box. For classification, every grid cell will have a probability for each possible class C. The described procedure is illustrated in figure 2.[2]

For this task the input image is first sent through a CNN and afterwords sent to two dense layers. The original architecture of the YOLO network is presented in figure 3. The last layer generates the final output, which is a tensor of size $S, S, (B * 5) + C$, where S is the gridsize, B is the number of predicting bounding boxes and C is the number of classes.[2]

4 Implementation

Our implementation of YOLO consists of different parts. We used the pascal VOC and coco dataset, therefore we needed a data preprocessing pipeline to bring the COCO and VOC data into the right format to be processed by our neural network and loss

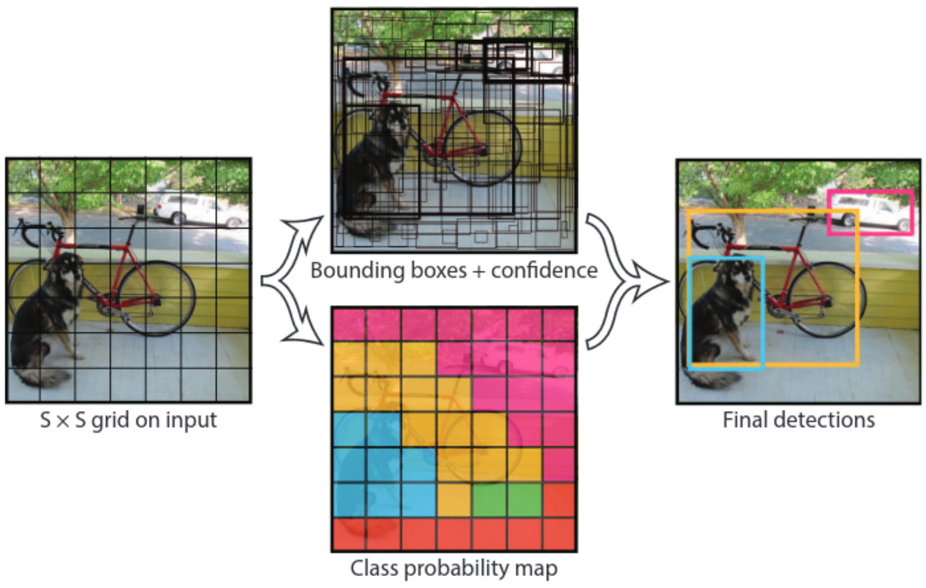

Fig. 2 Bounding Box and Class propabilty visualization[2]

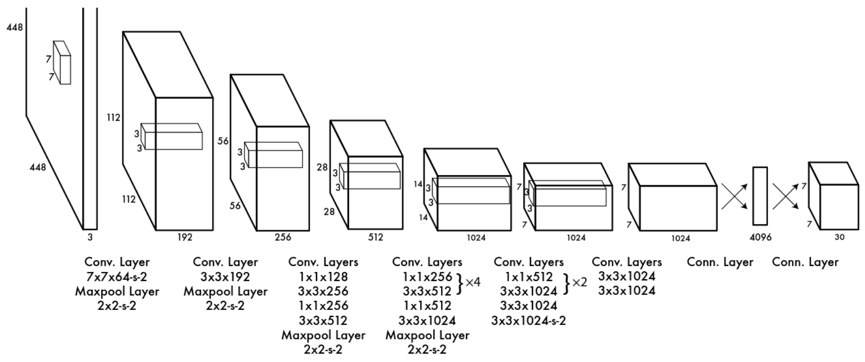

Fig. 3 The original architecture of the YOLO network[2]

function. We needed to write a custom loss function to implement the YOLO loss function as it is an unique loss function not implemented in keras and depended on multiple parameters, such as the number of classes and bounding boxes. Also we build a custom model.

Preprocessing

Preprocessing the input data from VOC or COCO format into the designated format for our YOLO model was done with a tf graph function. To pre-process the original images, they were first rescaled to the same size, which is $224X224$. This is due to our usage of the EfficientNetB0 model as a pretrained convoluational neural network. The next step was reshaping the VOC labels and bounding boxes into the same format as our YOLO network output, so we could calculate the loss easily. First an empty tensor of shape $SxSx(B*5)+C$ was created. This tensor was then updated through the tf.tensorupdatescatternd function. The updates were done for each grid cell containing an object. For this looping through all existing objects for the according voc image has been done. The grid, in which the object is centered was calculated and bounding boxes in the format for YOLO have been calculated, with the confidence score set to 1 and the probability of the class also to 1.

Model

The model is a python class and was a subclass from tf.keras.model. The model takes the parameters number of classes, gridsize and number of boxes, as these parameters are needed to determine output size of the last layer. The model itself is pretty simple, consisting of pretrained CNN, followed up by a dense layer, a leaky ReLu layer and another dense layer. The output size of the first dense layer is 4096, which is taken from cite YOLO. The second dense layer has an output size of $SxSx(B*5)+C$. After the second dense layer the classes are splitted away with tf.split and a softmax is applied on the classes, then the classes and bounding boxes are concated again and returned as the result tensor.

YOLO LOSS Function

We implemented a custom loss function as a subclass of tf.keras.losses.loss as our YOLO loss function. The function takes the number of classes, number of bounding boxes, gridsize and the two hyperparamters lambda coor and lambda noob as parameters for creating an instance of the loss function. For the calculation of the loss at first the the class parts of the tensor are cut of and the rest of the input tensor is reshaped into the following tensor shape. Between the predicted and true bounding boxes, the IoU (Intersection over Union) is then calculated and the predicted box with the highest IoU is used for for further calculation. This is done with argmax and tf.gather. Then the losses are calculated with functions from Keras Backend according the formula from the paper[2].

5 Results

As experiments we did training with validation data from the pascal VOC dataset. We used EfficientNetB0 from keras with image weights as ImageNet and global average as well as max pooling. We trained the network for 10 epochs on the max pooling and 20 epochs on average pooling. Our self implemented custom YOLO loss function was used as loss. As an optimizer the standard Adam optimizer from Keras was used with default settings. For the number of bounding boxes to be predicted and the grid size we used the standard parameters from the original paper[2]. Our results for the average pooling are shown in figures 4 and 5. While the training loss decreases with training, the validation loss is not sinking. The validation loss even increases over epochs. The

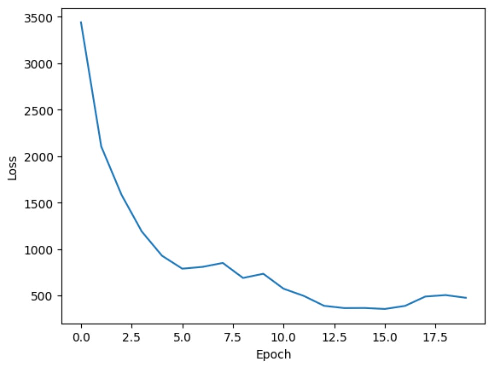

Fig. 4 Training loss for average pooling

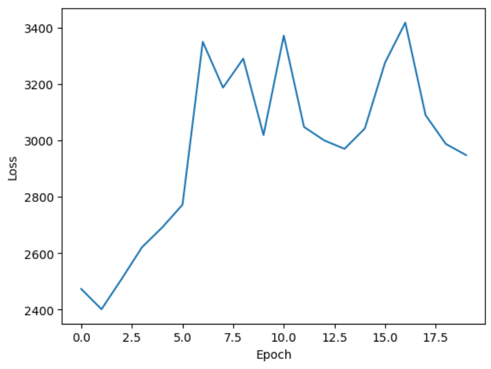

Fig. 5 Validation loss for average pooling

results from our second experiment are shown in figure 6 and 7. The results from the maximum pooling experiments show basically the same trend as the results from average pooling. While the training loss is decreasing over epochs, the validation loss is increasing or at least not decreasing. The reasons behind our network not learning are generalization and work on unseen data. We are discussing that in the next section.

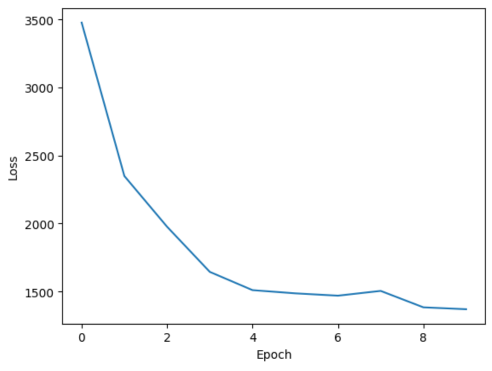

Fig. 6 Training loss for maximum pooling

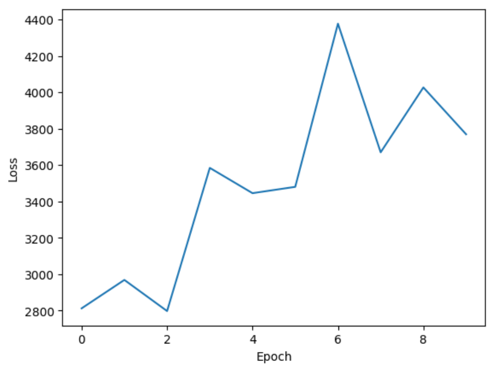

Fig. 7 Validation loss for maximum pooling

6 Discussion

Our results presented in the last section indicate, that our network did not learn a good generalization but instead learned a representation of the training data. There are several reasons why this could have happened:

- Erroneous implementations of data preprocessing and loss function. Our loss function and data preprocessing were self implemented and caused many problems in the project. While we tried to fix all errors and got working functions, we didn't have time for testing them fully and they may include errors. These errors may prevent the network from learning useful representations.
- Training errors in general. Due to limited resources we could only train our network for short periods while the original YOLOv1 network have been trained much longer. We also used the EfficientNet as base model and did not polish the EfficientNet, which may impact our results. Also we used a standard optimizer with fixed learning rate, while the original paper implemented a custom learning rate scheduler.

While these errors may be responsible for our bad results, we also didn't get to test more, which was originally planned. We didn't measure the mean average precision over training and validation data, which is also another metric to see if the network learned a representation.

At the end of this project, we have discovered that our initial plans underestimated the complexity and work needed for this project. The implementation of the preprocessing and loss function took the most work and may still be flawed. However, after training the created model, we can say that while the loss function and preprocessing may be erroneous, the model itself appears to be accurate. Additionally, we discovered that the YOLO network requires a significant amount of time and resources to train and test. As a result, we were unable to fully train the network in the given timeframe and could not test for overfitting. As mentioned above, we had planned to test our model with different networks as the basic part of our YOLO model, but due to our limited resources and overly ambitious goals, we were not able to do so. Overall, while we faced several challenges in this project, we have gained valuable insights that can inform future work in object detection and deep learning. These insights can be used to improve the efficiency of the YOLO network and other object detection systems, as well as to guide the development of more effective preprocessing and loss functions.

For future work, our mostly erroneous loss and preprocessing function should be tested and afterwards the neural network should be trained with a custom learning rate scheduler and enough time. Afterwards the models weights should be saved and then we could continue with testing the inference time on small, low-energy platforms like the raspberry pi. We also didn't have time to test our box drawing from the output.

7 Conclusion

Overall our project was not successfully. This was mostly due to the overestimation of what work can be done in the time frame and our limited knowledge in tensorflow. As a lesson learned, we would say, that choosing a project with less preprocessing and an already implemented loss function in tensorflow would have been better. Having nearly no results and also being incapable of doing the results we were eager to try, a project without these difficulties could have more experiments. Nonetheless we learned about tensorflow and the project improved our skills in machine learning.

References

[1] Ren, S., He, K., Girshick, R., Sun, J.: Faster r-cnn: Towards real-time object detection with region proposal networks. Advances in neural information processing systems **28** (2015)

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

[3] Tan, M., Le, Q.: Efficientnet: Rethinking model scaling for convolutional neural networks. In: International Conference on Machine Learning, pp. 6105–6114 (2019). PMLR

[4] Howard, A.G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., Adam, H.: Mobilenets: Efficient convolutional neural networks for mobile vision applications. arXiv preprint arXiv:1704.04861 (2017)

[5] Zou, Z., Shi, Z., Guo, Y., Ye, J.: Object detection in 20 years: A survey. arXiv preprint arXiv:1905.05055 (2019)

[6] Liu, L., Ouyang, W., Wang, X., Fieguth, P., Chen, J., Liu, X., Pietikäinen, M.: Deep learning for generic object detection: A survey. International journal of computer vision **128**, 261–318 (2020)

[7] Redmon, J., Farhadi, A.: Yolo9000: better, faster, stronger. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 7263–7271 (2017)

[8] Ju, R.-Y., Lin, T.-Y., Jian, J.-H., Chiang, J.-S.: Efficient convolutional neural networks on raspberry pi for image classification. Journal of Real-Time Image Processing **20**(2), 1–9 (2023)