Real Time Barcode Detection using Deep Learning models and Pyzbar

Santosh Srinivas Ravichandran (sr6411) and Surekha Suresh (ss16055)

Abstract

This project compares real-time barcode detection using Deep Learning (specifically TinyYolo3) and standard Computer Vision (CV) techniques, focusing on their performance with different datasets, including RGB and grayscale. In testing, TinyYolo3 excelled with grayscale images, potentially due to their simplicity aiding model generalization, while Pyzbar showed slightly better alignment with barcodes. The study also highlights challenges in direct comparison due to factors like the complexity of custom loss function development. Overall, the findings indicate comparable performance between the two methods, with specific nuances in different conditions and datasets.

Introduction

Real-time barcode detection serves as a critical component across diverse sectors such as inventory management, supply chain logistics, and retail point-of-sale systems. Its significance lies in swiftly and accurately identifying barcodes within a dynamic environment. Yet, achieving this in real-time poses a persistent challenge. Traditional computer vision techniques, while foundational, grapple with intricate image conditions and diverse barcode variations, often resulting in missed detections or erroneous identifications. This limitation stems from their struggle to adapt to the complexities of barcode types, orientations, and environmental factors.

Deep learning approaches have emerged as a promising avenue in object detection and recognition. They exhibit a capacity to discern and analyze intricate patterns within images, thereby offering a potential solution to the challenges faced by conventional methods. However, their application in real-time barcode detection is encumbered by computational complexities. The trade-off between the computational demands of deep learning models and the necessity for real-time responsiveness remains a pivotal concern in this domain.

In light of these considerations, this project endeavors to perform an extensive comparison between real-time barcode detection methodologies employing deep learning and conventional computer vision techniques. This project aims to conduct a comparison of real-time barcode detection using Deep Learning and standard CV techniques, addressing the advantages and challenges in using both the approaches. Within the Deep Learning approach, we aim to train the model on different datasets such as RGB and grayscale datasets to gather insights on model performance by comparing using several techniques such as training loss curves etc.

For more details, visit our GitHub project: https://github.com/Santoshsrini/Deep-Learning-Barcode/tree/main

Literature Review

Barcode detection technologies have undergone a substantial evolution, transitioning from traditional computer vision methodologies to the realm of advanced deep learning techniques. In the initial stages, methods such as pyzbar relied on algorithms centered around edge detection and thresholding to decipher barcodes [1]. While effective in simpler scenarios, these approaches encountered challenges when confronted with complex environmental conditions. Their limitations became evident in scenarios with varying lighting, orientations, or degraded barcode quality, leading to compromised detection accuracy.

The emergence of deep learning, notably exemplified by the YOLO (You Only Look Once) system, presented a significant leap forward in barcode detection capabilities [2]. YOLO's distinctive strength lies in its ability to achieve heightened accuracy and real-time detection prowess, reshaping standards within the field. Unlike earlier methods, YOLO showcases efficiency in processing diverse barcode orientations and handling complex environmental conditions, setting new benchmarks for detection systems.

Furthermore, recent studies have delved into novel strategies such as training detection models on grayscale images, a departure from the conventional RGB images. This exploration aims to potentially curtail computational demands while upholding detection accuracy [5]. Leveraging grayscale images emphasizes contrast and edge information, attributes pivotal in barcode recognition. This avenue has shown promise in maintaining or even enhancing detection accuracy while potentially mitigating computational overheads.

These advancements signify the dynamic progression within barcode detection technologies. They underscore a shift towards more sophisticated, adaptable, and efficient methodologies. Guided by these innovations, the current

project is focused on delving into and synthesizing these methodologies, aiming to explore their nuances, strengths, and limitations within the context of real-time barcode detection frameworks.

Dataset

We acquired the dataset from a dedicated GitHub repository [6], which encompasses a comprehensive collection of records stored as tensor files. The dataset comprises a total of 595 records.

To facilitate the training of our models, the dataset is partitioned into two subsets: one designated for training, consisting of 535 records, and another for validation, comprising 60 records.

The annotations are provided in a structured CSV file named "annotations.csv". This file contains the scaled bounding box coordinates corresponding to the barcodes present in the tensor files encoded in the TensorFlow Record (TFRecord) format, specifically stored as "all.tf_record".

Figure 1: Sample Image

4000445116300-01_N95-2592x1944_scaledTo640x480bilinear	0.3671875	0.322916666666667	0.684375	0.629166666666667	0
4000400139672-01_N95-2592x1944_scaledTo640x480bilinear	0.337500	0.325000	0.732812	0.614583	0
4000196015112-01_N95-2592x1944_scaledTo640x480bilinear	0.328125	0.352083	0.696875	0.579167	0
4000400037947-01_N95-2592x1944_scaledTo640x480bilinear	0.329688	0.341667	0.656250	0.579167	0
4000492002496-01_N95-2592x1944_scaledTo640x480bilinear	0.364063	0.310417	0.679688	0.616667	0

Figure 2: Annotation

The dataset is structured with entries of the form image-filename, bounding box coordinates, class label.

To enhance the training process, a grayscale version of the dataset was generated. This grayscale dataset was created by converting each image to its grayscale version and then transforming it back into tensor format. The process of grayscale conversion was applied to both the "all.tf_record", "validation.tf_record" and "training.tf_record" files.

Model Summary

The "yolov3_tiny" model is a streamlined version of the YOLO object detection architecture, optimized for speed and suitable for real-time applications like barcode detection. It processes input images of size 416×416 pixels with three color channels (RGB).

```
Model: "yolov3_tiny"
_________________________________________________________________________________________________
 Layer (type)                Output Shape              Param #   Connected to
=================================================================================================
 input (InputLayer)          [(None, 416, 416, 3)]     0         []

 yolo_darknet (Functional)   ((None, None, None, 256),  6298480   ['input[0][0]']
                              (None, None, None, 1024))

 yolo_conv_0 (Functional)    (None, 13, 13, 256)       263168    ['yolo_darknet[0][1]']

 yolo_conv_1 (Functional)    (None, 26, 26, 384)       33280     ['yolo_conv_0[0][0]',
                                                                  'yolo_darknet[0][0]']

 yolo_output_0 (Functional)  (None, 13, 13, 3, 85)     1312511   ['yolo_conv_0[0][0]']

 yolo_output_1 (Functional)  (None, 26, 26, 3, 85)     951295    ['yolo_conv_1[0][0]']

=================================================================================================
Total params: 8858734 (33.79 MB)
Trainable params: 8852366 (33.77 MB)
Non-trainable params: 6368 (24.88 KB)
_________________________________________________________________________________________________
```

Figure 3: Model

Input Layer: Accepts variable-sized batches of 416×416 RGB images.

yolo_darknet (Functional): Serves as the core feature extractor of the network, producing high-dimensional representations of the input images at two scales for detecting objects at different sizes.

yolo_conv_0 and yolo_conv_1 (Functional): These convolutional layers further process the feature maps from Darknet, refining the features for the subsequent detection layers.

yolo_output_0 and yolo_output_1 (Functional): These are the final detection layers. Each provides a grid (13×13 for larger objects, 26×26 for smaller objects) with three bounding box predictions per grid cell, encapsulating object location, size, and class probabilities.

These layers collectively enable the model to detect and classify objects with high efficiency, making it suitable for real-time applications.

Loss function Description

The loss function for YOLOv3 Tiny (YoloLoss) is a custom function designed to train an object detection model. It computes the loss for a single YOLO layer and is composed of several components:

Coordinate Loss: The loss between the predicted box (*x, y, width, height*) and the ground truth. It uses the squared difference between the predicted and true values for the box's center coordinates (*x, y*) and a logarithmic loss for the width and height, which are scaled by the anchor dimensions.

Objectness Loss: This binary cross-entropy loss measures whether an object is present in the box. It penalizes the model if it is confident about an object's presence when there isn't one, or if it's not confident when an object is present.

Class Loss: Also a binary cross-entropy loss, it calculates the difference between the predicted class probabilities and the one-hot encoded true class.

Box Loss Scale: A scaling factor that gives more weight to smaller boxes. This is important because small boxes are usually harder to detect and therefore need more emphasis during training.

Ignore Mask: It is used to ”ignore” certain box predictions if their IoU with the ground truth is above a certain threshold (ignore_thresh). This helps in preventing the model from being penalized for predictions that are close to the ground truth but not perfect.

Summing Losses: Finally, the individual losses (coordinate, objectness, and class) are scaled by their respective masks and summed up to give the total loss for each prediction.

The YoloLoss function is used during the training of the model to optimize the weights. It effectively trains the model to improve its predictions by reducing the discrepancies between the predictions and the ground truths across the multiple aspects of object detection: localization (where), confidence (whether), and classification (what).

Training Details

The training process for the TinyYolo model, as implemented in the code, involves several steps and components tailored for an object detection task. The training process drew inspiration from [6], from where we obtained the dataset. Below are the steps:

Pre-trained Weights: The model loads pre-trained weights using model_pretrained.load_weights(pretrained_weights) as it starts with knowledge learned from a previous dataset (often a large and diverse one like COCO).

Weight Transfer: The weights from the yolo_darknet layer of the pre-trained model are transferred to the new model with model.get_layer('yolo_darknet').set_weights(...). The foundational features learned by the Darknet backbone are being reused.

Freezing Layers: The freeze_all function is called on the yolo_darknet layer, which typically freezes the layers to prevent their weights from being updated during the initial training phase. This is a common practice in fine-tuning to maintain the integrity of the learned features while the rest of the model is being adapted to the new task.

Continued Training: After setting up the pre-trained weights and freezing layers, the model is trained on the barcode dataset that contains the target objects. This step adapts the later layers of the model to the specifics of the new task.

Together, these steps constitute fine-tuning, where the model initially retains much of its learned knowledge from a general task and then adapts to a more specific task through additional training.

Output: The method returns the history object containing training statistics, such as loss and accuracy for each epoch.

Training and Validation Loss Curves for RGB Images

After the model is trained for a specified number of epochs using the prepared training and validation datasets, the fit method's history object captures the training progress and metrics, which can be used for analysis or reporting training loss curves.

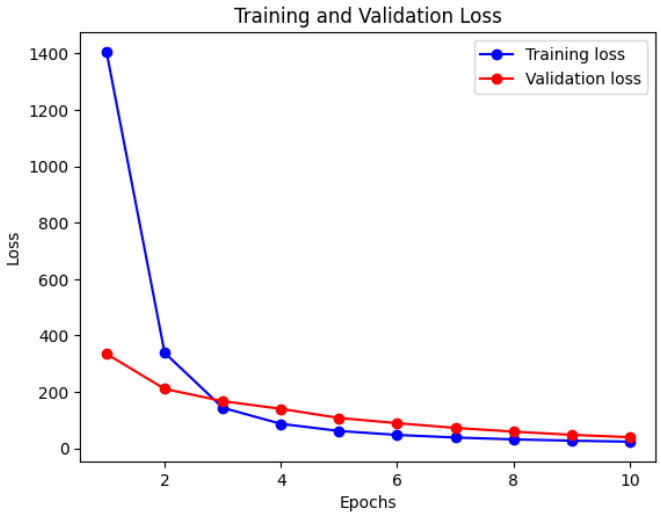

Figure 4: TrainingLoss

During the training process, the TinyYolo model exhibits a decreasing loss trend, indicative of learning. By the 10th epoch, both validation and training losses converge, suggesting that the model is reaching a stable state.

Training and Validation Loss curves for grayscale images

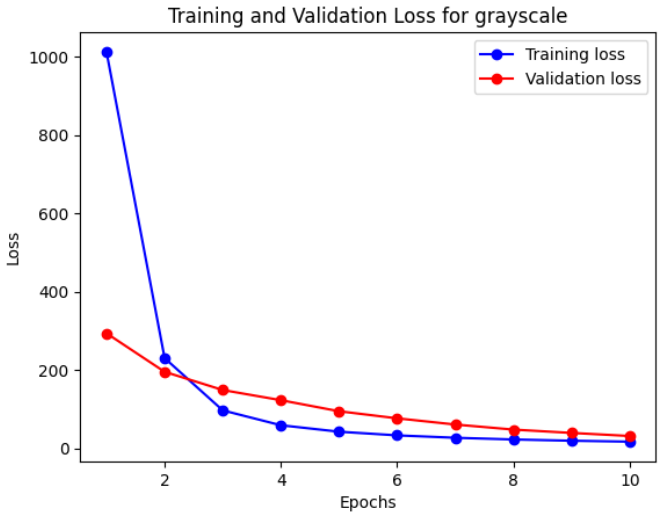

Figure 5: TrainingLossGray

The loss is observed to decrease over epochs during the training process. At the 10th epoch, both validation and training losses exhibit convergence, indicating that the model is learning and generalizing effectively to the given dataset.

The training and validation loss curves indicate that the TinyYOLOv3 model trained on grayscale images converges slightly faster and as effectively as the model trained on

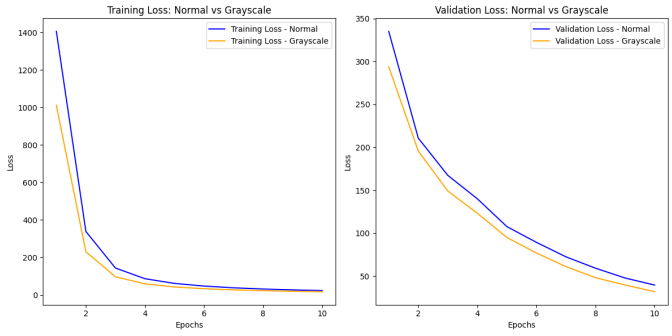

Figure 6: comparisonNormalGray

normal images, suggesting efficient learning without a loss in generalization ability. Both approaches show similar final performance, implying that grayscale images could offer computational benefits with comparable detection accuracy.

The loss curves provided for the TinyYOLOv3 model trained on both normal (RGB) and grayscale images offer some interesting insights:

Convergence Speed: Both the training and validation loss curves for the grayscale images appear to decrease slightly faster than those for the normal images. This suggests that the model might be learning more efficiently or converging faster when trained on grayscale images.

Final Loss Values: Towards the end of the training (around epoch 10), the loss values for both grayscale and normal images converge to similar values. This indicates that by the end of the training, both models achieve comparable performance in terms of loss.

Model Complexity and Data Simplicity: The fact that grayscale images have only one color channel compared to the three channels in normal images could mean a lower computational load and less complex data for the model to learn from, which might explain the faster initial decrease in loss.

Generalization Ability: The validation loss curves are particularly informative as they suggest how well the model generalizes to unseen data. Both models seem to generalize at a similar rate, with the grayscale model having a very slight edge.

Overfitting: There is no evident sign of overfitting in either model, as the validation loss decreases in tandem with the training loss. Overfitting would typically be indicated by a validation loss that starts to increase as the training loss continues to decrease.

TinyYOLO Barcode Detection: After the TinyYolov3 model undergoes the training process, it becomes capable of detecting barcodes within images. This detection process is initiated using the predict method, which applies the trained model to images in order to locate barcodes.

When the predict method is employed, it performs an analysis of the image data through the trained TinyYolov3 model. This analysis results in the computation of two tensors, which essentially represent sets of bounding box coordinates encapsulating the detected barcode regions within the image. These coordinates define the spatial boundaries of the identified barcodes, indicating their positions and sizes within the image.

However, it's essential to note that the process of detecting the bounding box coordinates does not involve the actual decoding of the barcode content itself. Instead, it primarily focuses on localizing and outlining the areas in which the barcodes are present within the image.

The generated bounding box coordinates serve as crucial information for subsequent steps, potentially facilitating the extraction of barcode content or aiding in further analysis or actions related to the detected barcode regions. The separation between the detection of bounding boxes and the decoding of barcode information allows for distinct stages in the barcode detection and recognition pipeline, enabling more efficient and modular processing of barcode data within applications.

Standard Computer Vision Techniques (Pyzbar) Barcode Detection:

Pyzbar is a Python library created with the specific intention of reading and interpreting different kinds of barcodes and QR codes. Within the scope of our project, Pyzbar assumes a central role in the identification and subsequent decoding of barcodes embedded within images. However, prior to utilizing Pyzbar for the detection and decoding process, the images undergo several crucial preliminary preprocessing steps.

These preprocessing steps are pivotal as they serve to optimize the images for enhanced accuracy in barcode detection and decoding. Initially, the images are subjected to resizing, which standardizes their dimensions to a consistent format. Subsequently, the images are transformed into grayscale, a process that accentuates contrasts and emphasizes edge information within the images—essential factors that aid in successful barcode detection.

Following the conversion to grayscale, additional enhancements are applied, such as gradient calculations, which further refine the image features. These refined features contribute to the identification and delineation of barcode patterns and structures, facilitating more accurate barcode detection and subsequent decoding by Pyzbar.

These preprocessing steps collectively prepare the images, ensuring they are optimized to facilitate Pyzbar's efficient and accurate detection and decoding of the barcodes contained within them.

The mentioned preprocessing steps are crucial for optimizing the images to enhance the accuracy of barcode detection and decoding. Initially, the images undergo resizing, which adjusts their dimensions to a standardized format. Subsequently, the images are converted to grayscale, a process that emphasizes contrast and edge information within the images, elements vital for successful barcode detection.

Following the grayscale conversion, gradient calculations are applied to further enhance and refine the image features, aiding in the identification of barcode patterns and structures. Once these preprocessing steps are completed, Pyzbar is utilized to detect and subsequently decode the barcodes present within the processed images.

Comparing the functionality between TinyYolo3 and Pyzbar, there are distinct differences in their output and ca-

Figure 7: TinyYolo3 detection 1

Figure 8: TinyYolo3 detection 2

Figure 9: Pyzbar detection 1

Figure 10: Pyzbar detection 2

pabilities. While TinyYolo3 primarily returns the bounding box coordinates outlining the detected barcodes within the image without actually decoding the barcode content itself, Pyzbar, on the other hand, performs both barcode detection and subsequent decoding.

Pyzbar not only identifies the presence and location of barcodes within the images but also goes a step further by decoding the actual information encoded within those barcodes. This includes extracting text, numerical data, or other encoded information from the barcode, providing the decoded output as part of its functionality.

This differentiation in capabilities between TinyYolo3 and Pyzbar highlights their distinct roles within barcode processing pipelines. While TinyYolo3 specializes in identifying the spatial boundaries of barcodes through bounding box coordinates, Pyzbar excels in both detection and decoding, providing access to the deciphered information encoded within the detected barcodes.

Conclusion

In the Deep Learning approach, the TinyYolo3 model seemed to perform better on grayscale datasets both in training and validation. This might be because the simplicity of grayscale images may help the model to better generalize from the training data, as it emphasizes contrast and edges which are critical in barcode recognition, potentially leading to improved detection performance.

Comparing Pyzbar and deep learning, from the testing on images used for the project, pyzbar and Tinyolo3 performed similarly, with pyzbar performing very slightly better, as can be seen from the images where the barcode coincided exactly with the bounding box. However, this can't be generalized, as the model performance depends on the number of training epochs. Additionally, we don't know the perfor-

mance on both barcodes in various lighting conditions. Another difficulty that arose in the project while comparing was that defining a custom loss function for pyzbar to directly compare with TinyYolo proved extremely difficult, but it remains a possibility for the future to gather better insights.

References

1. Atiqul Islam Chowdhury, Mushfika Sharmin Rahman, Nazmus Sakib. "A Study on Multiple Barcode Detection from an Image in Business System, 2019."
2. Daniel Kold Hansen, Kamal Nasrollahi, Christoffer B. Rasmusen, and Thomas B. Moeslund. "Real-Time Barcode Detection and Classification using Deep Learning, 2017."
3. GitHub Repository: BenSouchet/barcode-datasets. [Online] Available at: https://github.com/BenSouchet/barcode-datasets#muenster-barcodedb [Accessed on 2023-11-17].
4. Kaggle Dataset: whoosis/barcode-detection-annotated-dataset. [Online] Available at: https://www.kaggle.com/datasets/whoosis/barcode-detection-annotated-dataset [Accessed on 2023-11-17].
5. Towards Data Science Article: "Transfer Learning on Greyscale Images: How to Fine-Tune Pretrained Models on Black and White." [Online] Available at: https://www.kaggle.com/code/poojag718/transfer-learning-on-greyscale-images?scriptVersionId=95417284.
6. GitHub Repository: dchakour/Barcode-detection. [Online] Available at: https://github.com/dchakour/Barcode-detection/tree/main.