Capstone Project

Ekaterina Prokopeva
Machine Learning Engineer Nanodegree

Jan 15, 2018

I. Definition

Project Overview

Object detection has been widely used in everyday life in the last few years. It can be applied to many fields including security and robotics. I have used object detection to solve a children's game. In this project I solved a famous game "Where's Waldo" using machine learning techniques. The algorithm for this task both distinguishes images that contain Waldo and finds the location of Waldo. Using a data set containing waldo images and a data set containing not waldo images, I have trained a machine learning model in order to solve this problem.

Problem Statement

The goal of my project is to locate Waldo in the original image doing the following steps:

1. Do data preprocessing if necessary depending on the algorithm to be used (e.g. resizing the images, converting images into a different format, etc.)
2. Use Waldo and Notwaldo datasets to train the model to distinguish the images that contain Waldo using a binary classifier
3. Use a sliding window to scan the whole original image. Apply the model chosen in step 2 to each window in order to calculate the probability of having Waldo in the window
4. Return the windows with the highest probability of having Waldo

Metrics

I have used two metrics to evaluate the model.
I have used accuracy for measuring the performance of the classifier by running the model on the testing dataset that contains waldo and not waldo images.
Accuracy = (true positives + true negatives) / (true positives + true negatives + false positives + false negatives)
I used this metric since we do not have any preference towards true positive rate or true negative rate. So accuracy as a traditional method works well for this purpose.

I used false positive rate for the evaluation of the object detection part. Since in this problem there can be only one true positive (the window that contains Waldo) and one false negative it is not reasonable to use other metrics such as f1 score for the model evaluation. Instead, I decided to evaluate the algorithm by checking if it managed to identify Waldo (this is the most important task) and checking its false positive rate:
False Positive Rate = false positives / (false positives + true negatives)

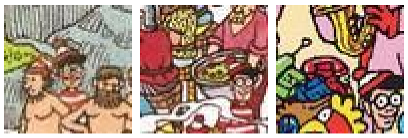
II. Analysis

Data Exploration

The kaggle data set can be obtained at https://www.kaggle.com/residentmario/wheres-waldo. I have used the original images data set and not waldo dataset from kaggle and have created my own waldo dataset.
Original image example:

Notwaldo example:

I did not use waldo data set since the images in this dataset include other characters and items besides Waldo which can create noise during training the model. Instead I have searched the web in order to find Waldo images and resized them to have a consistent image size throughout the whole project. The hat should also be included in the image since this is the characteristic that distinguishes Waldo from other characters in the game. Very often Waldo is not depicted fully (with the body) in the original images so training the classifier on Waldo's face is more efficient. Waldo's face is pretty small therefore the sliding window should be around 32x32 px.
Kaggle waldo image examples (note that they include other characters and items):



My waldo image examples that only includes Waldo's face:

Number of original images: 19

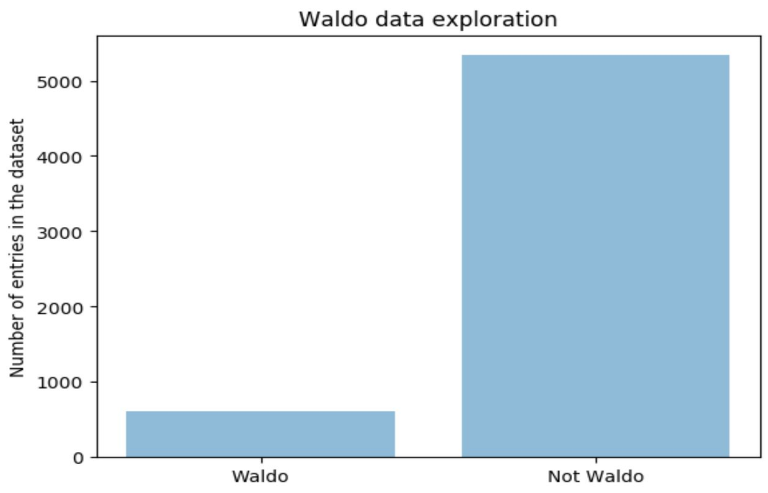
Number of waldo images: 30

Number of notwaldo images: 5337

We can see that the data is very disbalanced so the model's predictions may not be accurate. However, one can tweak this problem by doing data augmentation which adds more images to the waldo dataset. In this scenario data augmentation could prevent the model from underfitting.

Exploratory Visualization

Here is the number of images in the waldo and notwaldo datasets after data augmentation:



We can see that the data is significantly disbalanced in these two classes so one should keep it in mind while developing the model.

Algorithms and Techniques

First, one needs to train the model on waldo and notwaldo datasets using some classifier. I’m going to consider two classifiers for this problem: Histogram of Oriented Gradients (HOG) used with Support Vector Machines (SVM) and Convolutional Neural Networks (CNN).

HOG with SVM is know to be used for face detection which is similar to the problem I’m working on: I need to find Waldo’s face. HOG is one of the traditional ways to represent pixels in the images. The idea behind HOG is the following: create the histograms of gradients that show the direction in which the image gets darker. Using this approach one can find the main features of the image since the histogram shows the edges that shape objects. After converting images into HOG, one can feed this data into an SVM. A good thing behind using SVM is the fact that it doesn’t require as much data as CNN does and the waldo dataset is small. However, the disbalance in the data (there are much more not waldo images than waldo images even after data augmentation) could affect the accuracy of SVM.

Another well-known image classifier is a CNN. I’m going to create a CNN experimenting with the number of convolutional layers and pooling layers. Then a dense layer should be added in the end of the network with a chosen activation function to return the probability of a given image to contain Waldo in it. Since I’m dealing with binary classification I’m first going to try sigmoid activation function for the Dense layer and binary cross entropy loss function for training the model. The model should be trained with the validation set in order to prevent the model from being biased and overfitting.

CNN requires a big amount of data. Luckily, Notwaldo dataset is big enough. However, waldo dataset is very small. It is possible to fix this problem by doing data augmentation and generate more images as described in the section above. After the model is trained on waldo and notwaldo datasets, I’m going to evaluate it using accuracy and tweak the parameters as necessary.

After the training is complete two models can be compared and the model with higher accuracy should be picked for the object detection part. In order to locate Waldo in the original image, a sliding window can be applied to sample the part of the image (window) and use our pretrained model to give the probability of having Waldo in this window. If the probability is higher than some threshold, highlight the window showing that Waldo was detected.

Benchmark

HOG with SVM described above could be used as a benchmark model since I am going to show later that CNNs work better for this particular problem and I would like to focus on a more efficient approach.

III. Methodology

Data Preprocessing

Let's look at data preprocessing for each data set necessary for this problem:

<u>Original Images:</u>

Original images did not require any data preprocessing since they are only used for testing to locate Waldo.

<u>Not waldo images:</u>

I have used the kaggle dataset for this purpose. However, since I'm going to detect Waldo's face, it's better to use 32x32 size images to train the model so I had to resize this dataset using opencv resize method and I used opencv imwrite methods in order to save newly received images.

<u>Waldo images:</u>

I've obtained this dataset on my own by cutting out Waldo's face from different original images in the web. Then I had to resize the images using opencv resize method and save the new images using opencv imwrite. However, I've obtained only 30 images which is not enough to train the model so I used data augmentation technique to increase the size of Waldo data set which leads to more accurate results. I used Keras ImageDataGenerator to generate such images.

The following parameters for data augmentation were used:

- Rotation
- Width shift
- Height shift
- Shear
- Zoom
- Horizontal flip

I particularly had to focus on the zoom parameter because the size of Waldo's face is various in each original image which can strongly deteriorate the accuracy of the model. I set zoom to 0.5 to make certain Image Data Generator generates a big variety of sizes. Here's an example of augmented images. We can clearly see the flipping and the zooming:

Implementation

The implementation can be divided in two parts: classification and object detection.

Classification:

After doing data preprocessing I created two classifiers: SVM and CNN. I'm going to talk about each classifier separately.

<u>SVM:</u>

In order to use SVM one needs to convert each image into HOG before feeding the image into the model.

- I used skimage library in order to calculate the hog. Then I converted all the training files into HOGs.
- Before feeding the data into the classifier I shuffled the data to make the model less biasd.

- I used sklearn SVM class in order to create an instance of a support vector machine with rbf kernel (it is a default value).
- I trained the data on SVM. The model was trained for 15 seconds.

In order to evaluate the model, I used traditional accuracy since it works well for classification problems.
The accuracy of this problem was 0.2 with the following parameters:

- Kernel = ‘rbf’
- Gamma = 1 / number of features

Due to huge disbalance in data the model failed to give accurate results.

CNN:
CNN is known as the most efficient technique for image classification nowadays and CNNs do not require much image preprocessing unlike SVMs since CNNs look at an image as a whole without modifying it. I used Keras library with Tensorflow backend for CNN implementation.
The input to my CNN is a matrix representing an RGB image of the following shape: (32, 32, 3) where 3 represents RGB value and 32 is the height and width of the image (I chose 32x32 px images for detecting Waldo’s face). There are only a few changes one needs to make before feeding the data into Keras CNN with Tensorflow backend:

- Convert images into numpy arrays using numpy img_to_array(img) function;
- Change the shape of arrays from 3D (32, 32, 3) to 4D (numImageSamples, 32, 32, 3). I used path_to_tensor(img_path) and paths_to_tensor(img_paths) functions from the Dog Breed Classification project for this task;
- Normalize the pixels in images by dividing each pixel (rgb representation) by 255;

It is time to create our CNN network by adding convolutional, pooling, and dropout layers and dense layer.

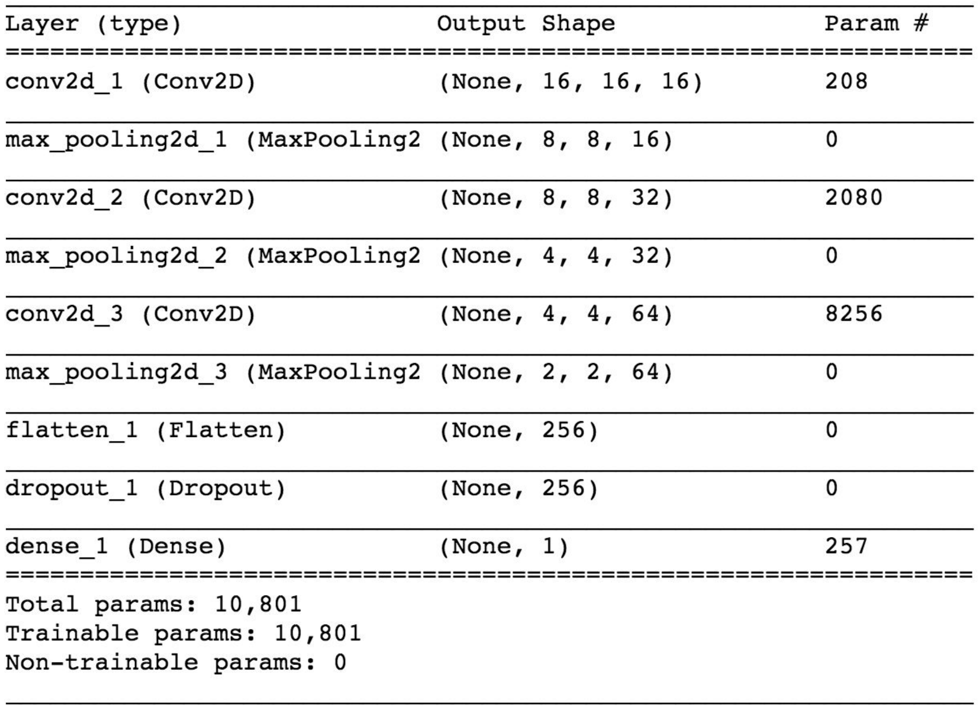
Convolutional layer uses some predefined filters usually of size 2-4 px in order to find the matching between these filters and the image. The filter window slides by some value called a stride along the image. The bigger the stride, the smaller the output image is.
Another parameter to consider is padding: it defines whether one should preserve the original size of the image after filtering by adding some extra zeros to the border of the image. I have

decided to use ‘same’ padding to preserve the information at the borders of the image and improve the performance of the model.

Pooling layer is responsible for reducing the size of the image, reducing computation complexity, and extracting the most extreme features of the image. Before passing the matrix into the pooling layer, it is necessary to add some activation function. The function that has shown to work the best is a very simple ReLU (Rectified Linear Units) function that makes all negative values equal to 0. ReLU makes the computation much faster without significantly damaging the accuracy of the model.

There are a couple of options for pooling layer such as max pooling and average pooling. I decided to choose max pooling because it extracts the most extreme features whereas average pooling ‘averages’ these features and sometimes does not do good job.

Finally, dropout layer is used to avoid overfitting of the model that randomly (we can set the probability of the dropout layer) eliminates activations.
I’ve decided to use three convolutional layers and three max pooling layers and one dropout layer for my initial layer. My initial model had the following structure:



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d_1 (Conv2D)            (None, 16, 16, 16)        208       
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 8, 8, 16)          0         
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 8, 8, 32)          2080      
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 4, 4, 32)          0         
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 4, 4, 64)          8256      
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 2, 2, 64)          0         
_________________________________________________________________
flatten_1 (Flatten)          (None, 256)               0         
_________________________________________________________________
dropout_1 (Dropout)          (None, 256)               0         
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 257       
=================================================================
Total params: 10,801
Trainable params: 10,801
Non-trainable params: 0
_________________________________________________________________
```

Now we need to compile the model using some loss function. I decided to use binary cross entropy since it is known to work well with binary classification problems.

Finally, the model is ready so it can be fit. For better results and less bias one can use a validation split. Keras fit has such an option where one can choose the size of the validation split. Usually it is set to 0.2 or 0.3.
After the model was trained, I tested it on my test set. The accuracy of the model was 0.8 which is significantly better than the SVM.

Object Detection:

I decided to use CNN for the object detection part since CNN deals better with the disbalance in data and shows much better accuracy. Object detection was implemented using a sliding window and the earlier trained classifier.

For the sliding window, I have implemented a custom function that scans the image and yields a square part of the image (sliding_wibdow function).

For better results, I decided to add a pyramid function to the program that resizes the original image and runs the same algorithm again on images of different scales. This can improve the accuracy of the algorithm since the objects in the images are presented in different sizes/scales. I have used the following opencv tutorial for pyramid implementation: https://www.pyimagesearch.com/2015/03/16/image-pyramids-with-python-and-opencv/

Eventually, I created the for loop that iterates the images generated by the pyramid function and checks every part (window) of the image to calculate the probability of having waldo in it using the CNN written earlier. If the probability is higher than a chosen threshold (I initially set it to 0.5), the window is going to draw a blue square around the window it considers to contain Waldo.

I also added the false positive and true negative counts to this for loop to calculate the false positive rate for the model evaluation.

Refinement

In order to improve the accuracy of my CNN I've changed the following parameters:

- Number of epochs
- Number of convolutional layers
- Number of pooling layers
- The value of the dropout layer
- The size of the filter window (kernel)

- The size of the stride
- Threshold

Let's go through each parameter and show how the accuracy changed depending on the value of the parameter.

<u>Number of epochs:</u>

- 3 Accuracy: 0.8
- 3 Accuracy: 0.9

<u>Number of convolutional and pooling layers:</u>

I've tried 2 and 3 convolutional/pooling layer which did not significantly affect the accuracy.

<u>The value of the dropout layer:</u>

0.2

<u>The kernel size:</u>

- 2 Accuracy: 0.6
- 3 Accuracy: 0.85
- 4 Accuracy: 0.9

<u>The size of stride:</u>

- 1 Accuracy: 0.9
- 2 Accuracy: 0.85

<u>Threshold</u>

- The model generally showed better results with the threshold set to 0.7
-

Thus, for the final model I picked three convolutional and pooling layers, kernel equal to 4, stride equal to 1, 4 epochs, the dropout layer equal to 0.2, and threshold equal to 0.7. With dropout layer set to 0.2, the model became less prone to overfitting.

After finding the optimal parameters for the CNN I used <u>hard negative mining</u> technique to improve the accuracy of the model. Hard negative mining is creating useful negative image examples from the original image and retraining the model on such images. The biggest challenge of this project is to reduce the false positive rate so hard negative mining can be a great refinement. While scanning the original image, I used opencv imwrite function in order to save the false positive examples. Then I retrained the original model on such images. In order to access the previously trained model, I save it in the best.hdf5 file using Keras ModelCheckpoint class. Then I used load_model method from Keras in order to load the model and trained it again using fit method.

Hard negative mining only slightly improved the results since my notwaldo dataset was already big enough so this technique is not necessary to use.

IV. Results

Model Evaluation and Validation

As mentioned in the refinement section the following parameters were used to create the final CNN:

- Three convolutional and pooling layers
- Kernel size = 4
- Stride = 1
- Number of epochs = 2
- Threshold = 0.7

These parameters showed the best accuracy using the testing dataset.

I also boosted the accuracy by adding more zoom data augmentation to the training waldo dataset. Before doing that, the model could not detect Waldo's face while scanning the original image since it was too small compared to the images in the testing dataset (only 8 out of 19 images were detected correctly). After changing the training dataset this number increased to 13.

After testing the final algorithm on all 19 images I got the following results:

Image Name	**Was Waldo successfully detected?**	**False Positive Rate (%)**
1.jpg	yes	2

2.jpg	yes	5
3.jpg	yes	4.9
4.jpg	yes	3.3
5.jpg	no	1
6.jpg	yes	4.5
7.jpg	no	1.3
8.jpg	no	0.5
9.jpg	yes	2.9
10.jpg	yes	3.4
11.jpg	no	1.1
12.jpg	yes	1.8
13.jpg	no	0.01
14.jpg	yes	2
15.jpg	yes	2.4
16.jpg	yes	2
17.jpg	yes	2.8
18.jpg	yes	5.3
19.jpg	no	0.9

Summary:

Number of times Waldo was detected successfully: 13 /19

Average false positive rate: 47.11 / 19 = 2.48

Average false positive rate when Waldo is successfully found: 43.2 / 19 = 2.27

Average false positive rate when Waldo is not found: 3.91 / 19 = 0.2

From the results above it can be seen that images with larger figures have a higher false positive rate and successfully detect Waldo. However, images with very small faces have a false positive rate close to 0 and do not detect Waldo. It implies that the Waldo and not waldo dataset should have a bigger variety of images of different sizes and that the original images should be scaled.
The model was tested on 19 different images so it can be trusted but for better assurance one should use more testing examples.

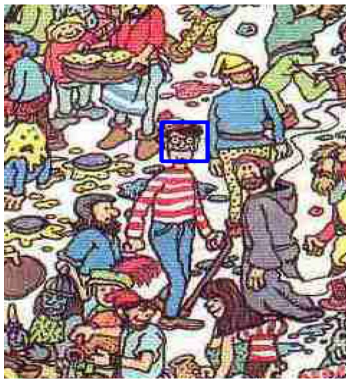
Justification

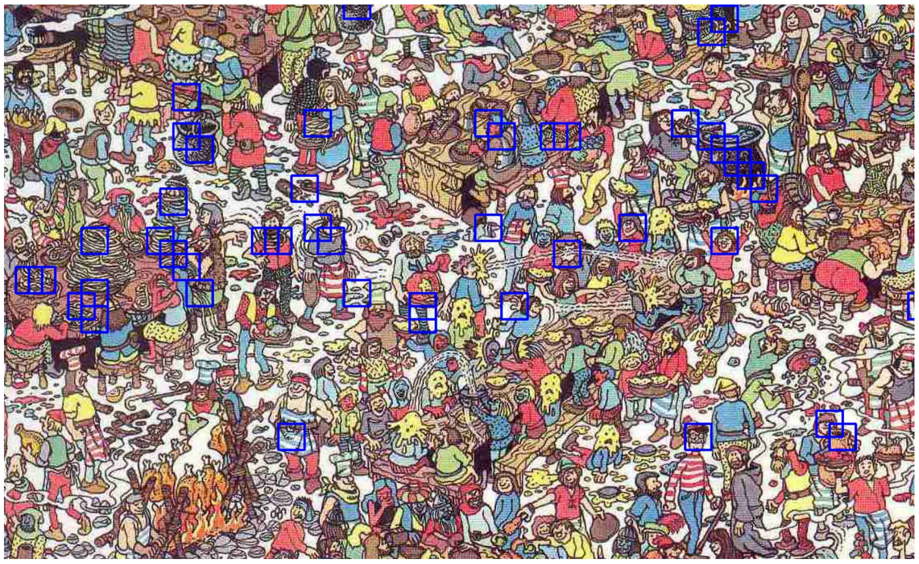
CNN has generally shown much better results compared to the benchmark model (HOG with SVM). CNN deals much better with the disbalance in data which was the main challenge of this problem. Also, HOG represents the change in brightness of a certain image and does not consider color (it works with grey scaled images). RGB values, however, are important for recognizing Waldo since Waldo’s most conspicuous characteristic is red and white striped hat. CNN considers the colors which increases the accuracy of the model.

V. Conclusion

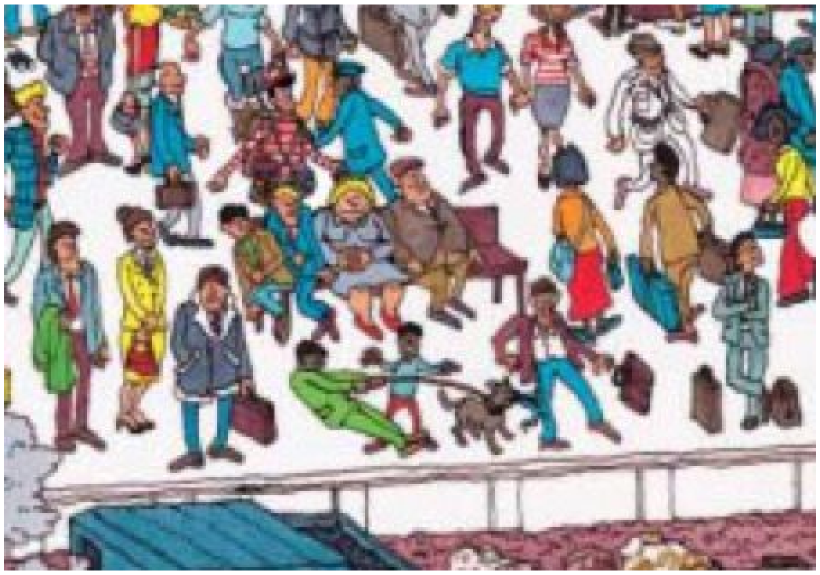
Free-Form Visualization

Here’s the result of running the algorithm on a couple of original images. You can see the false positive squares present in the image but an important thing is that the false positive rate is pretty low and Waldo was successfully found in most of the original images:
Example of successfully detecting Waldo with false positives:

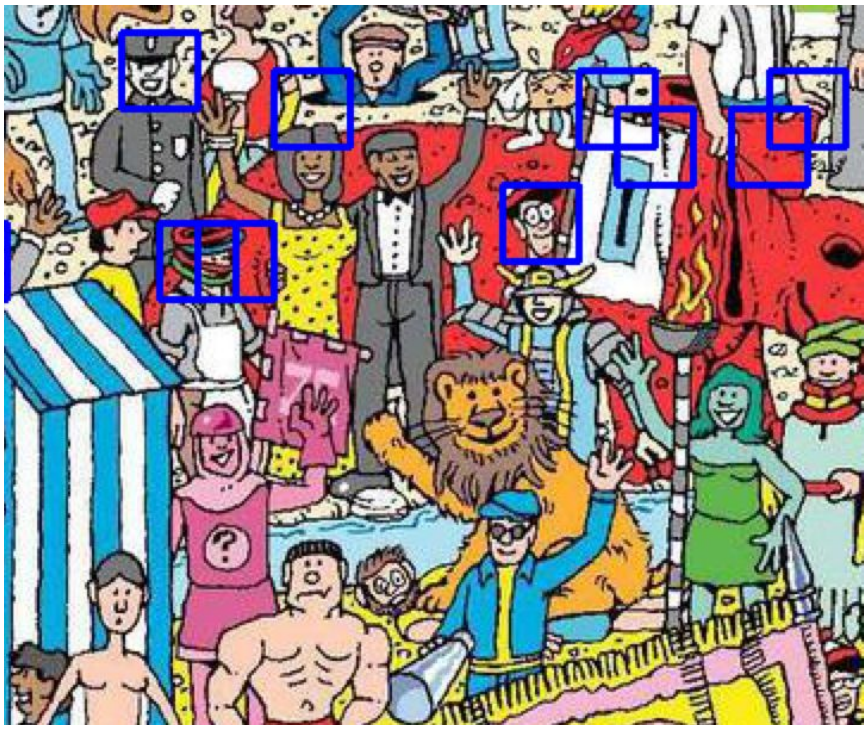


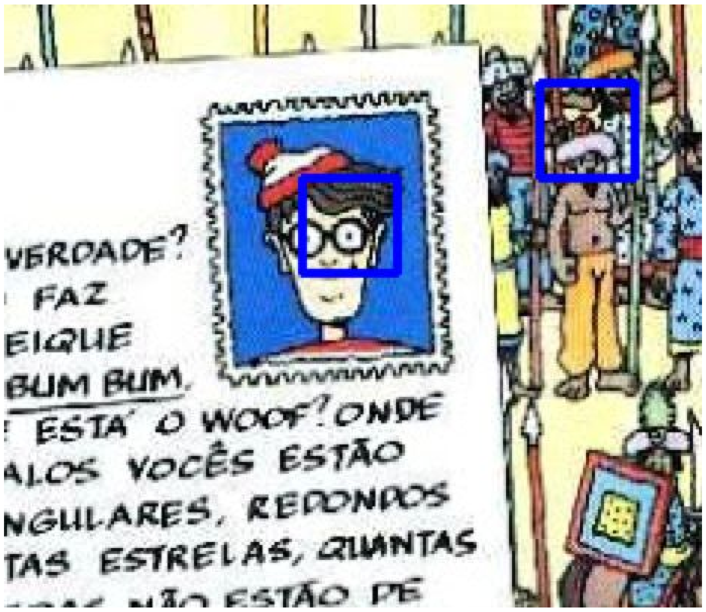


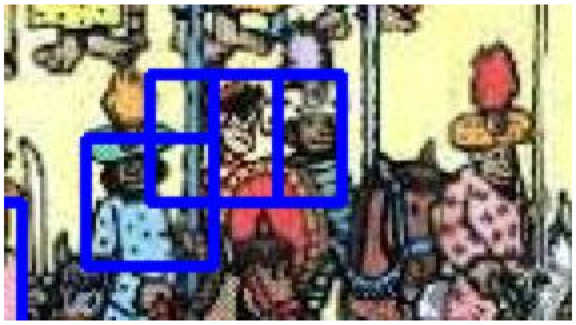
An example of an image with very small figures so the model could not identify any positives:



Successful example with some false positives:







Reflection

In order to find Waldo, I had to first collect appropriate data. Then I implemented data augmentation in order to generate more Waldo images and images of different scales since Waldo can be of any size in the real game. Then I compared two models: HOG with SVM and CNN. CNNs turned out to deal better with the data disbalance and suited this problem much better.

Then I used sliding window in order to scan original images and detect Waldo. In order to reduce the number of false positives I did hard negative mining on the dataset which only slightly improved the accuracy of the model.

The most challenging part of this project was the lack of data images and disbalance in data. I had to use some additional techniques in order to solve this issues.

Overall, I am satisfied with the results I have got considering the data I had to work with. Waldo was detected successfully in 70% of cases with some false positives.

Improvement

First, I would increase the size of my Waldo dataset. Unfortunately, I was not able to find enough Waldo images to efficiently train the CNN. Even if I had 100 Waldo images, I could use data augmentation to increase this dataset to a size of a couple of thousands images which would be enough to train the model.

Here're two algorithms that I also considered for this problem: R-CNN and YOLO.

R-CNN looks at an image through a series of windows of random sizes and placed at a random spot. Then it groups the generated rectangles by similar texture, color, or intensity to identify objects. Thus, it creates region proposals. Then we run these images using the pretrained CNN. This algorithm is proved to be the most effective approach for object detection. However, in Where's Waldo problem there's an object/character in every centimeter of the image, and every object has a similar texture and color. I believe that this algorithm would not work well in this case and scanning of the original image is required.

There is another algorithm YOLO (you look only once) that divides an image in squares of equal sizes. Then it tries to find the rectangles that enclose some object. It returns a confidence score of containing a certain object in it and a class. I believe that YOLO could be an algorithm that I would like to implement next in order to solve Where's Waldo game.