dyojf4kbj

July 14, 2025

1 Sliding Window

Here we will code the **Sliding Window** mechanism to locate Waldo in an entire image.

1.1 1. Loading Dependencies

Apart from defining the modules, we also need to load the weights from the model we previously trained and apply them to a network with the same architecture — MobileNetV2.

[14]:
```
import numpy as np
import torchvision
from torch.utils import data
from torchvision import transforms
from torch.utils.data import random_split
from torch import nn
import torch
import torchvision.transforms as T
from PIL import Image, ImageDraw
```

[15]:
```
from torchvision.models import mobilenet_v2, MobileNet_V2_Weights
import torch
from torch import nn

weights = MobileNet_V2_Weights.DEFAULT
Patch_Detector = mobilenet_v2(weights=weights)

Patch_Detector.classifier[1] = nn.Linear(1280, 1)

Patch_Detector.load_state_dict(torch.load("../model/waldo_detector_64x64_v1.0.
 ↪pth"))
Patch_Detector.eval()
print("-")
```

-

We also need to create the same preprocessor that we used before.

[16]:
```
preprocesser = transforms.Compose([
    transforms.Resize((256,256)),
```

```
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])
```

1.2 2.Sliding Window Detection with Scale Pyramid and Color Filtering

First of all, the detection process uses a sliding window approach that systematically scans the entire image with a fixed **64×64 pixel window**. For each window position, the model generates a prediction, and these predictions are stored along with their corresponding window coordinates—referred to as **boxes**.

To improve detection across objects of varying sizes, the function supports an optional **scale pyramid**. Instead of resizing the image, this technique scales the sliding window itself, applying it at multiple sizes across the original image.

While it is true that the model has been trained with 64×64 patches, applying resizing operations to the entire image alters its visual structure and may confuse the model. Changes in texture, edge sharpness, or object proportions can degrade detection performance, as the input no longer resembles the data distribution the model was trained on.

Since the model was trained on patches extracted from images at a fixed resolution, it is important that the objects in the input image are presented at a similar scale. By resizing only the extracted window (and not the entire image), the model receives inputs that remain visually consistent with its training data.

Because evaluating every patch at multiple scales can be computationally intensive, an additional optimization is implemented: patches that lack sufficient **red color** content—characteristic of Waldo's outfit—are filtered out before model inference. This step reduces unnecessary computations by skipping areas unlikely to contain the target.

Overall, this combination of a sliding window, scale pyramid, and color-based filtering provides a robust and efficient method for detecting Waldo in images of varying sizes and complexity.

1.2.1 2.1. Redness Dominance

The function `redness_dominance` returns a **boolean** indicating if a patch has enough red pixels based on two parameters:

- `red_factor`: how much red must dominate green and blue (e.g., 1.2)
- `red_threshold`: minimum proportion of red-dominant pixels to pass the filter (0 to 1)

It calculates the ratio of pixels where:

R > red_factor * G and R > red_factor * B

and returns `True` if this ratio is above `red_threshold`, else `False`.

By default, the filter is enabled by setting:

```
red_factor = 0.1
red_threshold = 1.5
```

1.3 2.2 Multi-Scale Pyramid

We also provide an **optional scale pyramid mode**, which automatically applies the sliding window over multiple predefined window scales (e.g., `1`, `1.5`, `2`). In this mode, the **window size is reduced by the inverse of the scale factor**—that is, for a base window of `64×64`:

- **Scale = 1** → window size = `64 × (1/1) = 64` pixels
- **Scale = 1.5** → window size = `64 × (1/1.5)` 43 pixels
- **Scale = 2** → window size = `64 × (1/2) = 32` pixels

This approach keeps the image at its original resolution and instead adjusts the sliding window dimensions to detect Waldos of different apparent sizes.

By default, when this mode is enabled, the model:

1. Performs inference at all selected scales
2. Aggregates all detection results
3. Sorts and ranks the combined predictions by their confidence scores

This ensures robust multi-scale detection without resizing the full image.

[17]:
```python
def redness_dominance(pil_img, red_threshold=-1, red_factor=0):
    img = np.array(pil_img).astype(np.float32)
    r = img[:, :, 0]
    g = img[:, :, 1]
    b = img[:, :, 2]
    red_dominant = (r > red_factor * g) & (r > red_factor * b)
    red_ratio = red_dominant.sum() / red_dominant.size
    return red_ratio > red_threshold
```

[18]:
```python
from tqdm.notebook import tqdm
from PIL import Image
import torch

def sliding_window(
    image_path,
    model=Patch_Detector,
    preprocessing_method=preprocesser,
    device="cpu",
    stride=16,
    scales=[1],
    red_threshold=0.1,
    red_factor=1.5
):
    image_original = Image.open(image_path).convert("RGB")
    base_window_size = 64
```

```
    all_results = []

    width, height = image_original.size
    total_windows = 0
    sizes_per_scale = []
    counter = 1

    for scale in scales:
        window_size = int(base_window_size * 1/scale)
        stride_scaled = int(stride * 1/scale)
        rangex = (width - window_size) // stride_scaled + 1
        rangey = (height - window_size) // stride_scaled + 1
        total_windows += rangex * rangey
        sizes_per_scale.append((scale, window_size, stride_scaled, rangex,␣
↪rangey))

    progress_bar = tqdm(total=total_windows, desc="Sliding Window Multi-Scale")

    for scale, window_size, stride_scaled, rangex, rangey in sizes_per_scale:
        pasada = False

        for y in range(rangey):
            for x in range(rangex):
                x1 = x * stride_scaled
                y1 = y * stride_scaled
                x2 = x1 + window_size
                y2 = y1 + window_size

                if x2 > width:
                    x1 = width - window_size
                    x2 = width
                if y2 > height:
                    y1 = height - window_size
                    y2 = height

                crop = image_original.crop((x1, y1, x2, y2))
                crop_resized = crop.resize((base_window_size, base_window_size))

                if not redness_dominance(crop_resized, red_threshold,␣
↪red_factor):
                    counter += 1
                    progress_bar.update(1)
                    continue

                input_tensor = preprocessing_method(crop_resized).unsqueeze(0).
↪to(device)
```

```
                with torch.no_grad():
                    model.eval()
                    output = model(input_tensor)
                    prob = torch.sigmoid(output)[0].item()


                box_original_scale = [x1, y1, x2, y2]
                all_results.append([prob, box_original_scale, scale])

                progress_bar.update(1)

    progress_bar.close()
    all_results.sort(key=lambda x: x[0], reverse=True)
    print(counter)

    return all_results
```

This function will always return a two-dimensional tensor. One dimension will contain the probabilities of belonging to class 1 (probabilities, not logits), and the other will contain the coordinates of the corresponding boxes.

1.4 3. Non Max Supression (NMS)

The boxes returned by the previous function may overlap. If Waldo in the photo is very clear, this can result in multiple boxes overlapping the same object. In the worst case, if we sort the boxes by their scores in decreasing order, the top boxes might correspond to the same object, preventing us from detecting other relevant objects. To avoid this redundancy, we will use the **Non-Maximum Suppression** algorithm. The intuition behind it comes from set theory and measuring cardinality; however, we will skip the details and use the prebuilt function.

[19]:
```
import torch
from torchvision.ops import nms
from PIL import Image
import matplotlib.pyplot as plt
import matplotlib.patches as patches

def NMS(list_of_results, threshold = 0, n_show = 5):
    scores = torch.tensor([i[0] for i in list_of_results], dtype=torch.float32)
    boxes = torch.tensor([i[1] for i in list_of_results], dtype=torch.float32)

    indices = nms(boxes, scores, threshold)[:n_show]
    final_boxes = boxes[indices]
    final_scores = scores[indices]

    return final_boxes, final_scores
```

1.5 4. Final Visualization of the Results

Now, we will create a function that visualizes the top n_{show} bounding boxes ranked by their scores in descending order. Each selected box will be highlighted with a red square on the full image, accompanied by a zoomed-in view of the region inside the box, along with the corresponding score assigned by the model.

[20]:
```
import torch
from torchvision.ops import nms
from PIL import Image
import matplotlib.pyplot as plt
import matplotlib.patches as patches

def Finding_Waldo(
    image_path,
    model=Patch_Detector,
    preprocessing_method=preprocesser,
    device='cpu',
    n_show=5,
    threshold=0,
    red_threshold=0.1,
    red_factor=1.5,
    stride=16,
    scales=[1]
):
    sliding_results = sliding_window(image_path, model, preprocessing_method,␣
 ↪device, stride, scales, red_threshold, red_factor)
    if len(sliding_results) == 0:
        raise ValueError("No valid patches found during sliding window␣
 ↪inference.")


    boxes = torch.tensor([res[1] for res in sliding_results], dtype=torch.
 ↪float32)
    scores = torch.tensor([res[0] for res in sliding_results])

    keep_indices = nms(boxes, scores, threshold)

    top_indices = keep_indices[:n_show]

    final_boxes = boxes[top_indices]
    final_scores = scores[top_indices]

    image = Image.open(image_path).convert("RGB")

    fig, axs = plt.subplots(n_show, 2, figsize=(10, 4 * n_show))
    if n_show == 1:
        axs = [axs]
```

```
    for i, (box, score) in enumerate(zip(final_boxes, final_scores)):
        x1, y1, x2, y2 = box.tolist()
        crop = image.crop((int(x1), int(y1), int(x2), int(y2)))

        axs[i][0].imshow(image)
        axs[i][0].set_title(f"Detection {i+1} - Confidence: {score:.2f}")
        axs[i][0].axis("off")

        rect = patches.Rectangle((x1, y1), x2 - x1, y2 - y1,
                                 linewidth=2, edgecolor='red', facecolor='none')
        axs[i][0].add_patch(rect)

        axs[i][1].imshow(crop)
        axs[i][1].set_title("Zoom on box")
        axs[i][1].axis("off")

    plt.tight_layout()
    plt.show()

    return final_boxes, final_scores
```

1.6 5. Final Validation: External Evaluation Set

We reserved a set of additional images, separate from the training and test sets, to evaluate our model without any risk of data leakage. This collection includes full pages from *Where's Waldo?* books. By using this diverse set, we aim to better understand the model's generalization and behavior in more realistic scenarios.

1.6.1 Validation Procedure

1. **Image scaling**
   For each image, we run inference at multiple scales to maximize detection accuracy.
   In practice, we typically use `[1.0, 1.5]` or `[1.0, 1.5, 2.0]`, since the model performs best when Waldo occupies a significant portion of the frame.

2. **Success criterion**
   A prediction is considered successful if Waldo appears in the **top 3** bounding boxes returned by the model **at any of the tested scales**.
   While this may seem like a lenient requirement, it is justified due to the **overwhelming number of distractors** deliberately designed to mislead the viewer.
   We do not care about the **confidence score** itself — only about the **ranking order** of the predictions.
   If a non-Waldo object receives a score close to 1 but is ranked **after** a correct Waldo detection, it is not considered a failure.
   If Waldo does not appear among the first 3 predictions at any tested scale, the attempt is counted as a failure.

Additionally, if Waldo is not detected at first but **is successfully found** after adjusting the red

filter parameters (e.g., `red_factor` or `red_threshold`), this also counts as a **successful detection**.

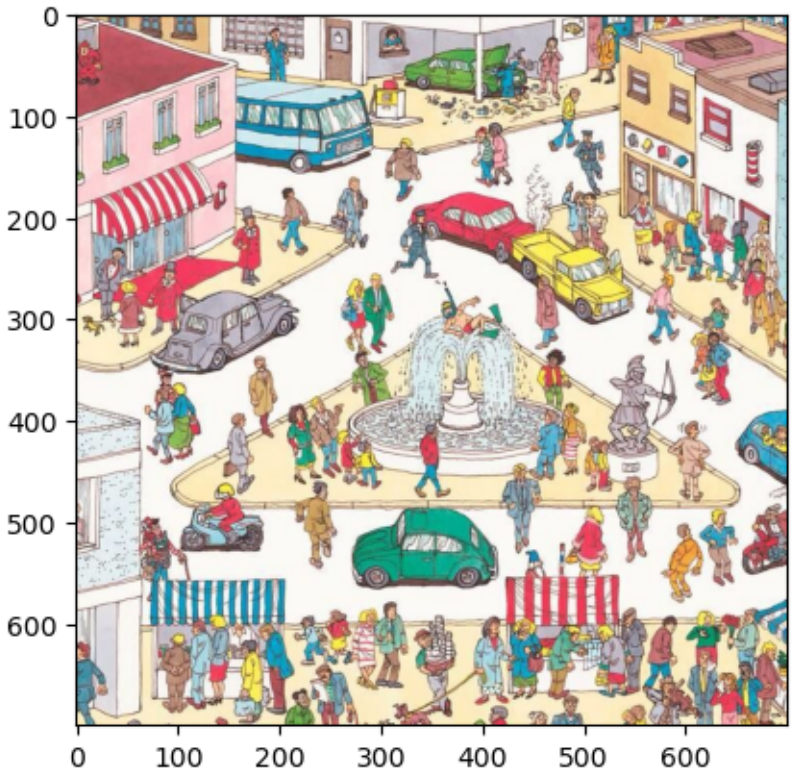
[21]:
```
import matplotlib.pyplot as plt
from PIL import Image

def print_image(image_path):
    plt.imshow(Image.open(image_path).convert("RGB"))
```

1.6.2 5.1. Inference on Full Images

1.6.3 Image 1

[22]:
```
path_1 = "../data/Validation_Set/image1.png"
print_image(path_1)
```



In this image, Waldo is located around the point **(100, 400)**, partially hidden behind two other people.
There are several distractors, such as people wearing red clothes, windows, and structures with red and white stripes — patterns that closely resemble Waldo's iconic shirt.

[23]:
```
Finding_Waldo(path_1, n_show = 3, scales = [1])
```

```
Sliding Window Multi-Scale:   0%|          | 0/1600 [00:00<?, ?it/s]
```

Detection 1 - Confidence: 0.57

Zoom on box

Detection 2 - Confidence: 0.37

Zoom on box

Detection 3 - Confidence: 0.32

Zoom on box

[23]:
```
(tensor([[ 48., 496., 112., 560.],
         [560., 608., 624., 672.],
         [144., 464., 208., 528.]]),
 tensor([0.5674, 0.3711, 0.3229]))
```

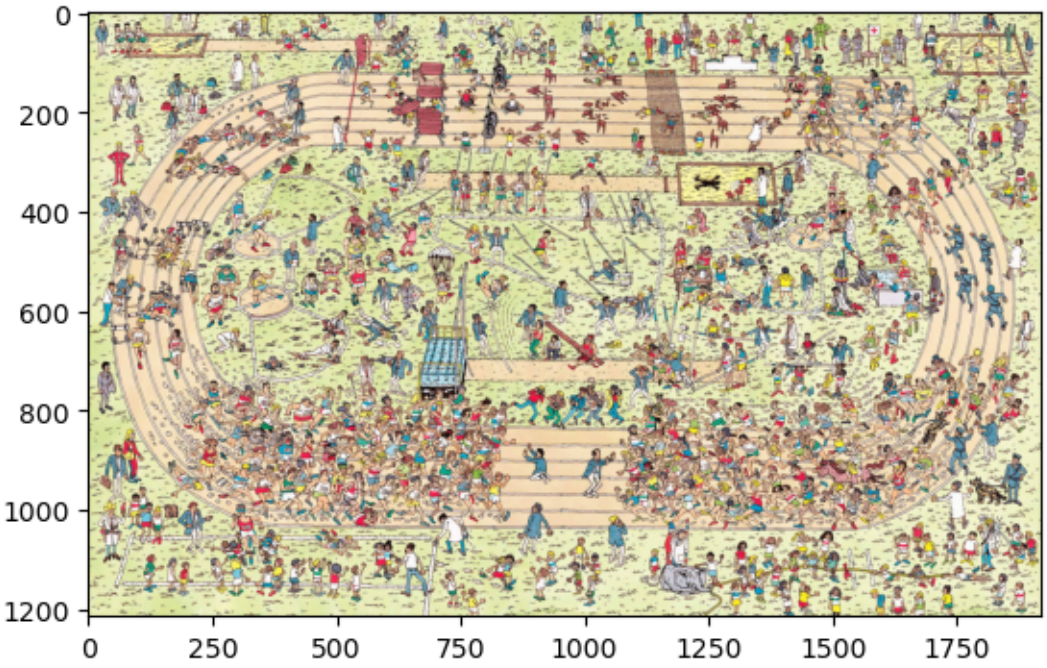
We can observe that the model **correctly detected** Waldo as the top-ranked prediction. There were no **false positives**.

As mentioned earlier, this still counts as a **successful detection**, since Waldo appears in the top 3.

Note that out of 3,844 windows, we skipped 3,002 thanks to the `red_intense` function.
In other words, we only needed to scan 22% of the image.

1.6.4 Image 2

[12]:
```
path_2 = "../data/Validation_Set/image2.png"
print_image(path_2)
```



This image is **large** and densely packed with distractors and various objects, leaving virtually no empty space.
It represents a **challenging test case** for the model due to its visual complexity.

[13]:
```
Finding_Waldo(path_2, n_show = 3, scales = [2])
```

```
Sliding Window Multi-Scale:   0%|          | 0/35076 [00:00<?, ?it/s]
27420
```

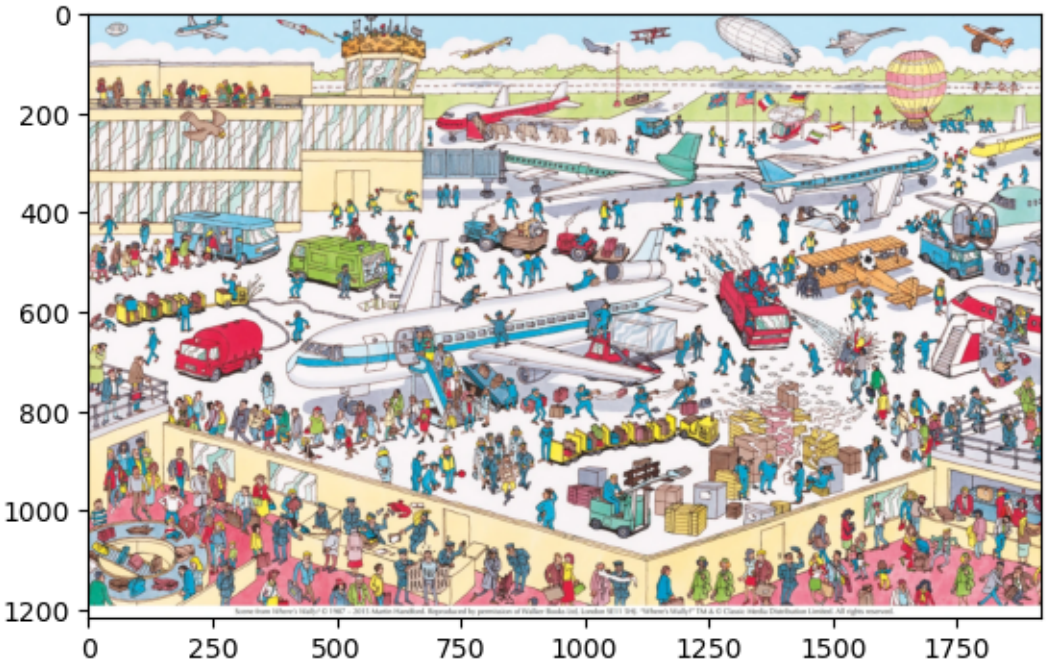
[13]:
```
(tensor([[528., 400., 560., 432.],
         [472., 864., 504., 896.],
         [536.,  80., 568., 112.]]),
 tensor([0.9999, 0.9994, 0.9966]))
```

Once again, the model successfully detected Waldo in the **first position**, which can be considered an **outstanding result** given the complexity of the image. Additionally, it detected Wenda in the second position. The third detection corresponds to a red mummy that is obviously similar to Waldo.

This time we skipped 27443 images from a total of 35076, so we reduced the time computation in a 78%This time, we skipped 27,443 images out of a total of 35,076, reducing computation time by 78%.

1.6.5 Image 3

[23]:
```
path_3 = "../data/Validation_Set/image3.png"
print_image(path_3)
```



We include another **large image** in the evaluation, characterized by a dense arrangement of elements and potential distractors.

[24]:
```
Finding_Waldo(path_3, n_show = 3, scales = [1.5])
```

```
Sliding Window Multi-Scale:   0%|          | 0/22184 [00:00<?, ?it/s]
```

```
18228
```


```
[24]: (tensor([[ 750.,  740.,  792.,  782.],
               [ 510.,  410.,  552.,  452.],
               [1530., 1060., 1572., 1102.]]),
       tensor([0.9994, 0.9969, 0.9967]))
```

Once again, the model detected Waldo, ranking him second. The model's behavior here is consistent with what we observed in the previous example. However, this time, the model detected a **false positive** which corresponds to a woman in the image.

Since now we will display the results of the images remaining in the Validation Set, commenting only on the relevant detections.

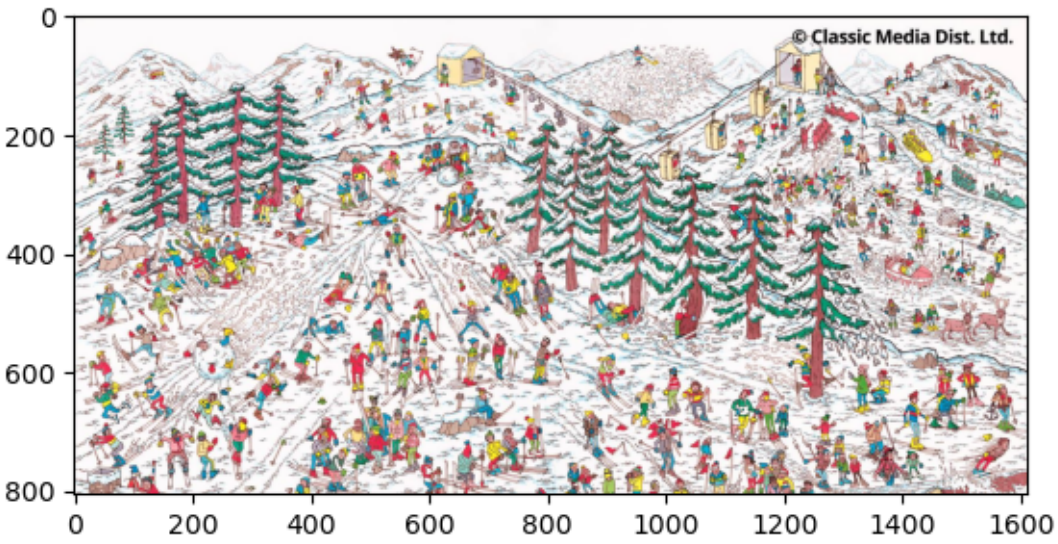
By default, we print the result only at the scale where the model successfully detects Waldo.

However, the most realistic approach involves estimating the size of the people in the photo to infer the best scale to use.

You can try multiple scales, and although this increases computation time, it improves the chance of detecting Waldo at least once among all tested scales.

1.6.6 Image 4

[25]:
```
path_4 = "../data/Validation_Set/image4.png"
print_image(path_4)
```



[26]:
```
Finding_Waldo(path_4, n_show = 3)
```

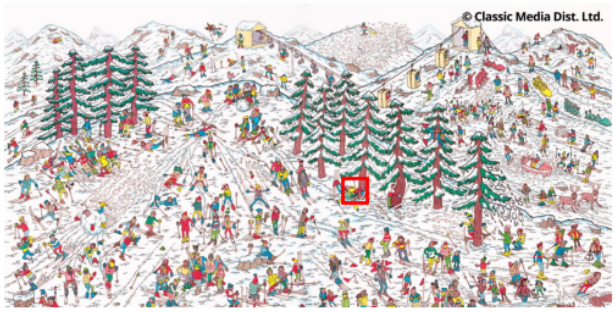
```
Sliding Window Multi-Scale:   0%|          | 0/4559 [00:00<?, ?it/s]
```

```
3472
```

Detection 1 - Confidence: 0.90

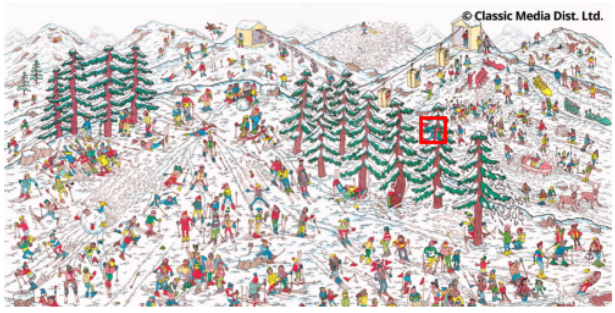
Zoom on box

Detection 2 - Confidence: 0.50

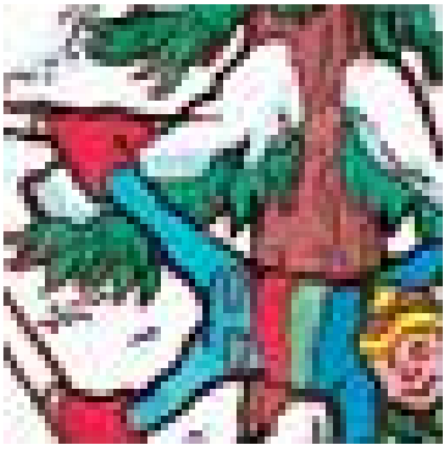


Zoom on box

Detection 3 - Confidence: 0.34



Zoom on box



```
[26]: (tensor([[1344.,  720., 1408.,  784.],
               [ 896.,  464.,  960.,  528.],
               [1104.,  304., 1168.,  368.]]),
       tensor([0.9026, 0.4956, 0.3445]))
```

The model **successfully detected** Waldo, ranking him first.

1.7 Image 5

[28]:
```
path_5 = "../data/Validation_Set/image5.png"
Finding_Waldo(path_5, n_show = 3, red_threshold = 0.1, red_factor = 2)
```

```
Sliding Window Multi-Scale:   0%|          | 0/29648 [00:00<?, ?it/s]
```

```
29395
```


```
[28]: (tensor([[ 144., 1568.,  208., 1632.],
               [1536.,  784., 1600.,  848.],
               [3296., 1616., 3360., 1680.]]),
       tensor([0.9916, 0.7393, 0.5310]))
```

Waldo was detected in the **first-ranked image**.
Note that, due to the large size of the photo and the lack of red in it, we increased the red intensity parameters.

In this detection, we avoided computing 29,395 windows out of a total of 29,648, which represents a significant optimization.
Otherwise, the processing time would have been close to 10 minutes instead of just 5 seconds.

1.8 Image 6

[29]:
```
path_6 = "../data/Validation_Set/image6.png"
Finding_Waldo(path_6, n_show = 3)
```

```
Sliding Window Multi-Scale:   0%|          | 0/2014 [00:00<?, ?it/s]
```

```
1208
```


```
[29]: (tensor([[592., 336., 656., 400.],
               [176.,  16., 240.,  80.],
               [448.,  64., 512., 128.]]),
       tensor([0.9497, 0.4909, 0.3906]))
```

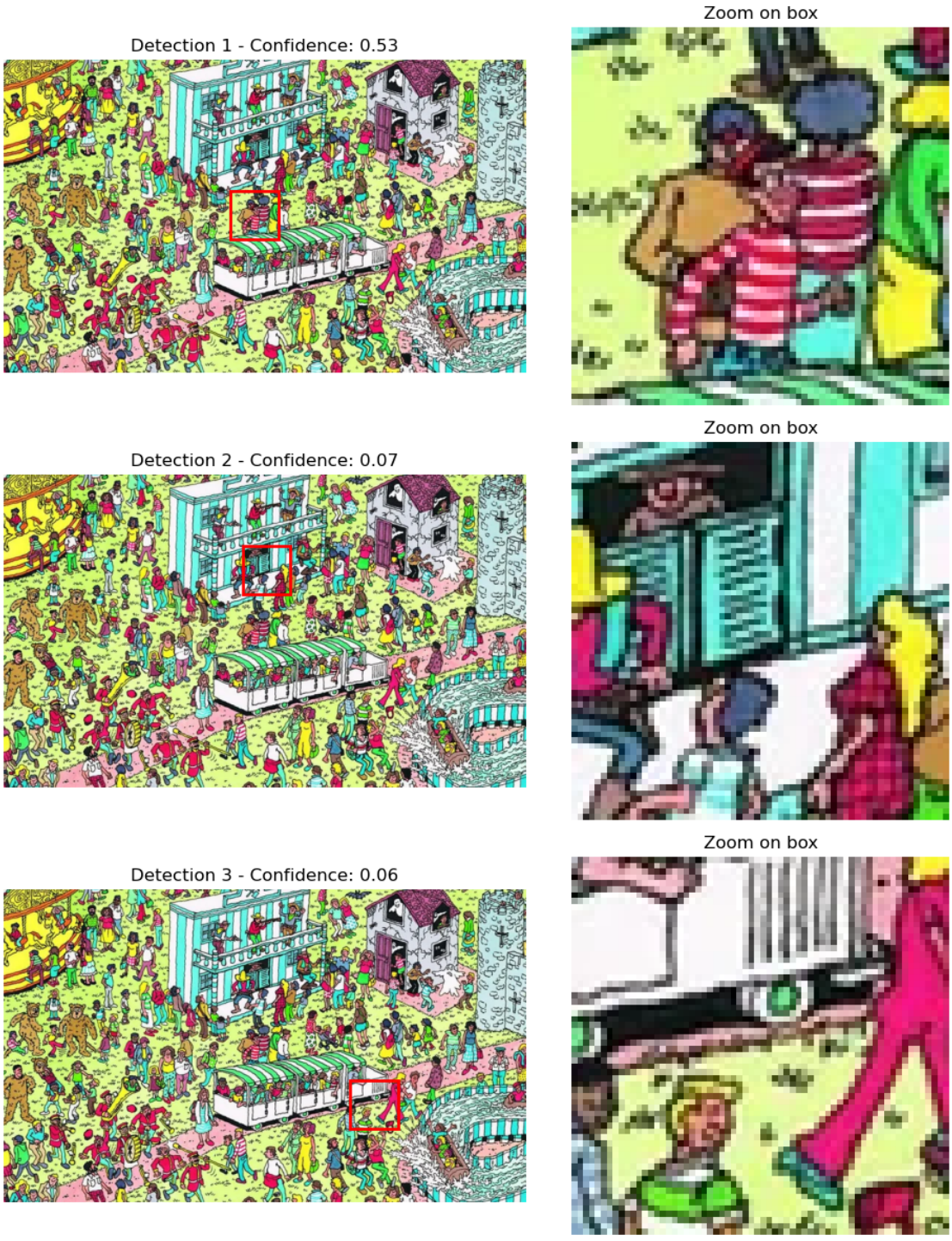
Waldo was detected in the **first-ranked image**.

1.9 Image 7

[30]:
```
path_7 = "../data/Validation_Set/image7.png"

Finding_Waldo(path_7, n_show = 3)
```

```
Sliding Window Multi-Scale:   0%|          | 0/920 [00:00<?, ?it/s]
639
```




```
[30]: (tensor([[304., 176., 368., 240.],
               [320.,  96., 384., 160.],
               [464., 256., 528., 320.]]),
       tensor([0.5327, 0.0684, 0.0601]))
```

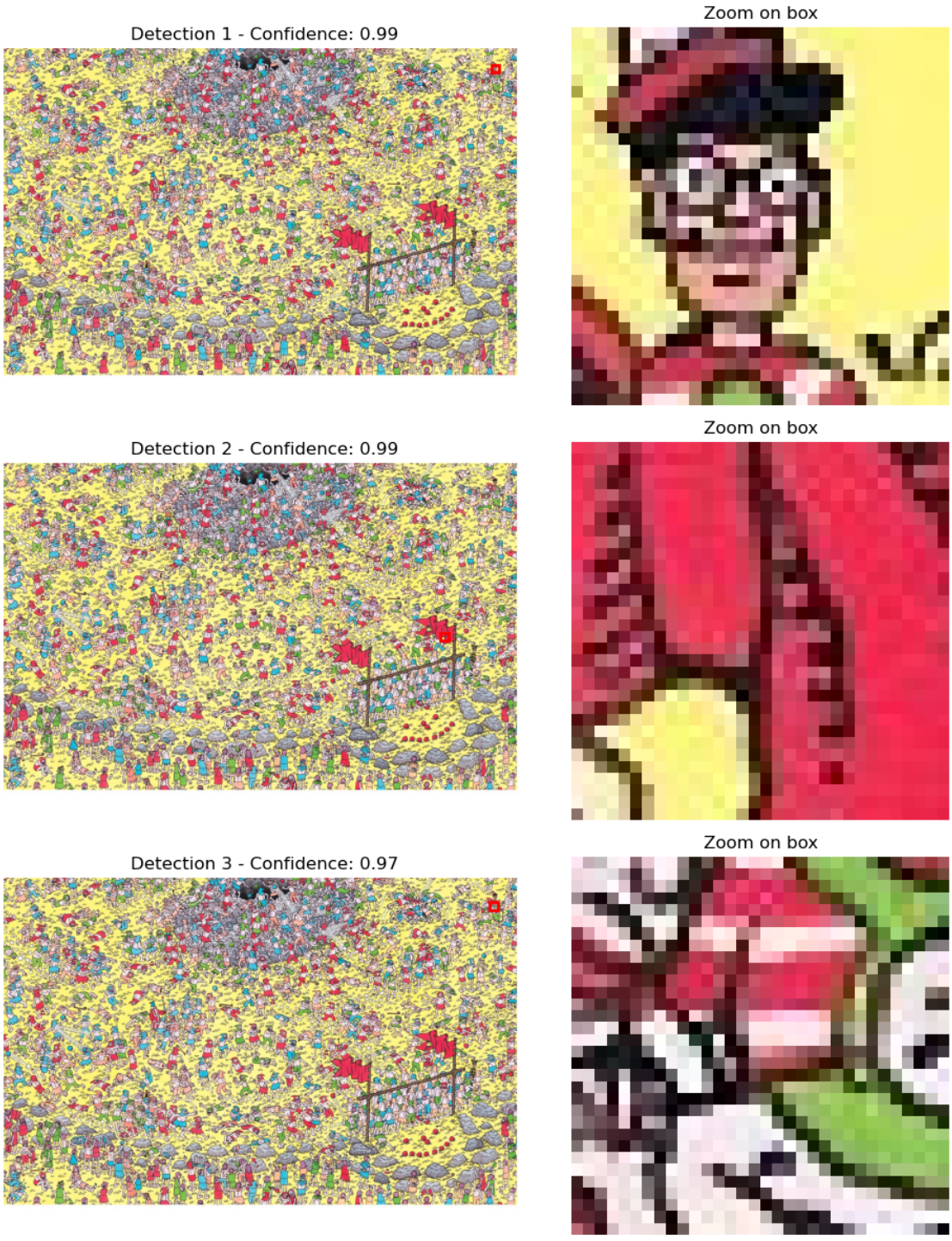
Waldo was detected in the **first-ranked** image.

1.10 Image 8

[32]:
```
path_8 = "../data/Validation_Set/image8.png"
Finding_Waldo(path_8, n_show = 3, scales = [2])
```

```
Sliding Window Multi-Scale:   0%|          | 0/40733 [00:00<?, ?it/s]
```

```
30733
```



[32]:
```
(tensor([[1912.,   64., 1944.,   96.],
         [1712.,  664., 1744.,  696.],
         [1904.,   96., 1936.,  128.]]),
 tensor([0.9949, 0.9904, 0.9683]))
```

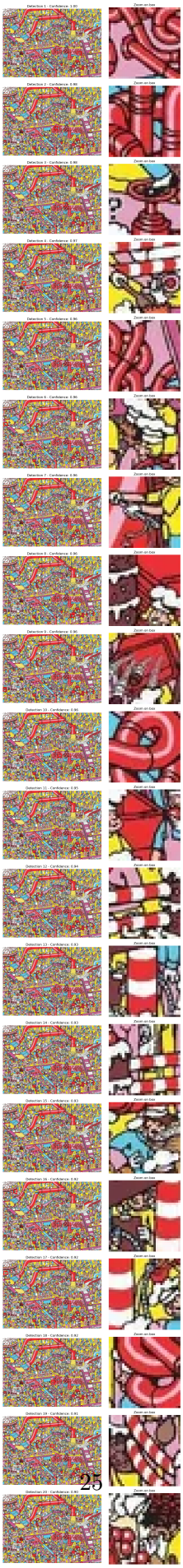
Waldo was detected in the **first-ranked image**.

1.11 Image 9

[25]:
```
path_9 = "../data/Validation_Set/image9.png"
Finding_Waldo(path_9, n_show = 20, scales = [2])
```

```
Sliding Window Multi-Scale:   0%|          | 0/8175 [00:00<?, ?it/s]
```

```
2968
```

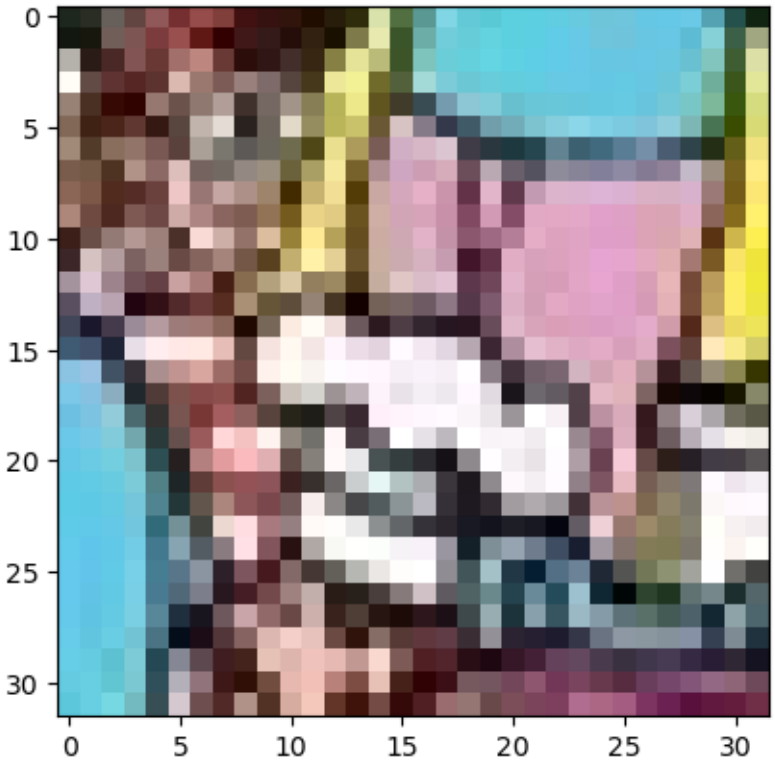

Detection 1 - Confidence: 1.00
Zoom on box
Detection 2 - Confidence: 0.98
Zoom on box
Detection 3 - Confidence: 0.98
Zoom on box
Detection 4 - Confidence: 0.97
Zoom on box
Detection 5 - Confidence: 0.96
Zoom on box
Detection 6 - Confidence: 0.96
Zoom on box
Detection 7 - Confidence: 0.96
Zoom on box
Detection 8 - Confidence: 0.96
Zoom on box
Detection 9 - Confidence: 0.96
Zoom on box
Detection 10 - Confidence: 0.96
Zoom on box
Detection 11 - Confidence: 0.95
Zoom on box
Detection 12 - Confidence: 0.94
Zoom on box
Detection 13 - Confidence: 0.93
Zoom on box
Detection 14 - Confidence: 0.93
Zoom on box
Detection 15 - Confidence: 0.93
Zoom on box
Detection 16 - Confidence: 0.92
Zoom on box
Detection 17 - Confidence: 0.92
Zoom on box
Detection 18 - Confidence: 0.92
Zoom on box
Detection 19 - Confidence: 0.91
Zoom on box
Detection 20 - Confidence: 0.90
Zoom on box

[25]:
```
(tensor([[680., 392., 712., 424.],
         [464., 360., 496., 392.],
         [552., 344., 584., 376.],
         [176., 352., 208., 384.],
         [712., 384., 744., 416.],
         [  8., 208.,  40., 240.],
         [176.,  80., 208., 112.],
         [616., 528., 648., 560.],
         [  0., 488.,  32., 520.],
         [504., 376., 536., 408.],
         [752., 520., 784., 552.],
         [584., 520., 616., 552.],
         [ 64.,  88.,  96., 120.],
         [368., 312., 400., 344.],
         [344., 104., 376., 136.],
         [ 96.,  72., 128., 104.],
         [  8., 104.,  40., 136.],
         [440., 392., 472., 424.],
         [432., 256., 464., 288.],
         [136., 408., 168., 440.]]),
 tensor([0.9983, 0.9837, 0.9768, 0.9720, 0.9647, 0.9612, 0.9600, 0.9583, 0.9566,
         0.9561, 0.9547, 0.9386, 0.9324, 0.9291, 0.9275, 0.9200, 0.9197, 0.9163,
         0.9096, 0.9010]))
```

Waldo was not detected. Let's take a look at Waldo and see how he appears in the image.

[26]:
```
image = Image.open(path_9).convert("RGB")
image = image.crop((440, 120,472,152))
image = image.resize((32,32))
plt.imshow(image)
image = preprocesser(image)
print(f'The model detects Waldo in this patch with a probability of {torch.
  ↪sigmoid(Patch_Detector(image.unsqueeze(0)))}')
```

```
The model detects Waldo in this patch with a probability of tensor([[0.9903]],
grad_fn=<SigmoidBackward0>)
```



Waldo appears in low resolution due to his small size and the surrounding objects.
The model actually detected him with a probability of 99.03% in the patch.

So, the problem is not the model's ability to recognize Waldo, but rather the **false positives**. Because these distracting elements appear in a wide variety of random configurations, some of them end up looking similar to Waldo from the model's perspective.

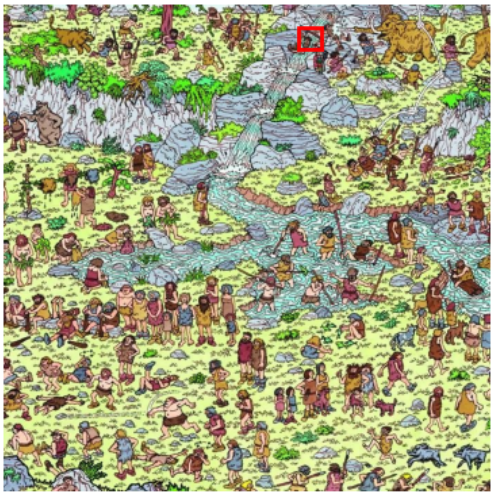
1.12 Image 10

[36]:
```
path_10 = "../data/Validation_Set/image10.png"
Finding_Waldo(path_10, n_show = 3, scales = [2])
```

```
Sliding Window Multi-Scale:   0%|          | 0/7056 [00:00<?, ?it/s]
```

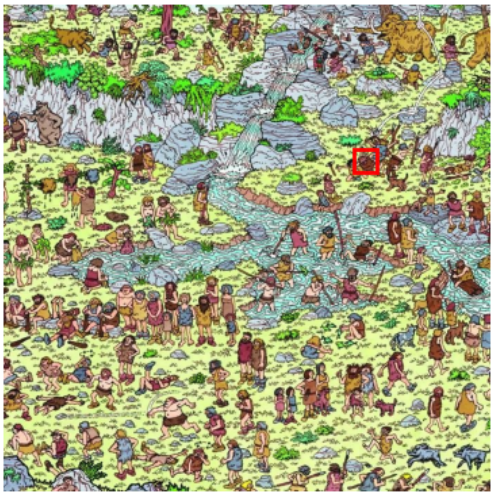
```
5207
```

Detection 1 - Confidence: 1.00



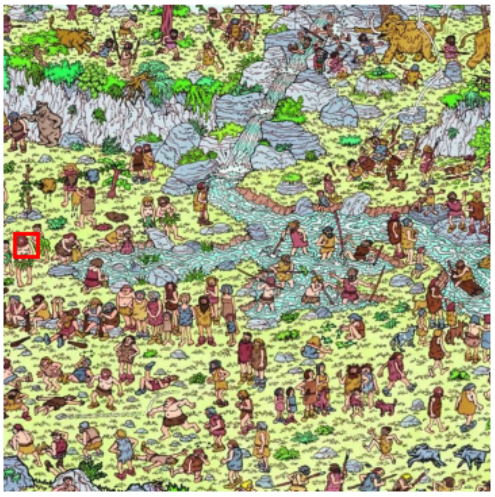
Zoom on box

Detection 2 - Confidence: 1.00



Zoom on box

Detection 3 - Confidence: 0.99



Zoom on box

[36]:
```
(tensor([[424.,  32., 456.,  64.],
         [504., 208., 536., 240.],
         [ 16., 328.,  48., 360.]]),
 tensor([0.9964, 0.9959, 0.9942]))
```

This time, Waldo was detected in the **first-ranked** bounding box. We can see part of his hat in the bottom-right corner of the image.

1.13 Image 11

[37]:
```
path_11 = "../data/Validation_Set/image11.png"
Finding_Waldo(path_11, n_show = 3, scales = [1])
```

```
Sliding Window Multi-Scale:   0%|          | 0/2700 [00:00<?, ?it/s]
```

```
217
```

[37]: (tensor([[656., 224., 720., 288.],
[416., 384., 480., 448.],
[736., 112., 800., 176.]]),

```
tensor([0.9913, 0.9849, 0.8825]))
```

Waldo was detected in the **second-ranked image**.

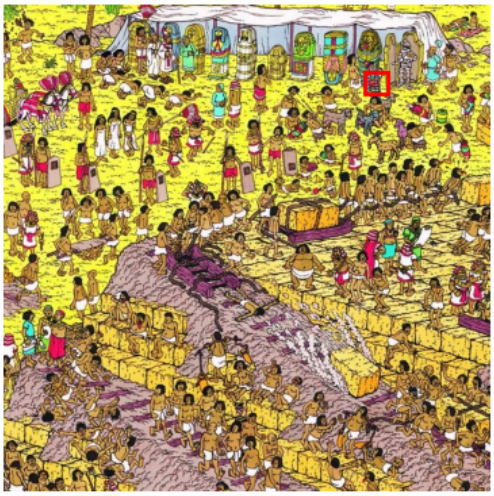
1.14 Image 12

[169]:
```
path_12 = "../data/Validation_Set/image12.png"
Finding_Waldo(path_12, n_show = 3, scales = [1/2])
```

```
Sliding Window Multi-Scale:   0%|          | 0/7056 [00:00<?, ?it/s]
```

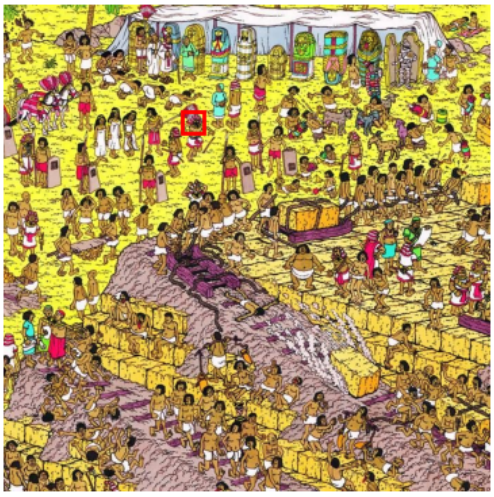
```
3234
```

Detection 1 - Confidence: 1.00



Zoom on box

Detection 2 - Confidence: 0.99



Zoom on box

Detection 3 - Confidence: 0.97

Zoom on box

[169]:
```
(tensor([[520.,  96., 552., 128.],
         [256., 152., 288., 184.],
         [ 80., 520., 112., 552.]]),
 tensor([0.9953, 0.9871, 0.9655]))
```

Waldo's was detected in the **first-ranked** image.

1.15 Image 13

[39]:
```
path_13 = "../data/Validation_Set/image13.png"
Finding_Waldo(path_13, n_show = 3, scales = [1])
```

```
Sliding Window Multi-Scale:   0%|          | 0/999 [00:00<?, ?it/s]
```

```
394
```

[39]: (tensor([[352., 16., 416., 80.],
[576., 64., 640., 128.],
[208., 256., 272., 320.]]),

```
tensor([0.8087, 0.7740, 0.5783]))
```

Waldo's was detected in the **first-ranked** image.

1.16 Image 14

[40]:
```
path_14 = "../data/Validation_Set/image14.png"
Finding_Waldo(path_14, n_show = 3, scales = [1])
```

```
Sliding Window Multi-Scale:   0%|          | 0/2516 [00:00<?, ?it/s]
```

```
758
```

Detection 1 - Confidence: 1.00

Zoom on box

Detection 2 - Confidence: 0.91

Zoom on box

Detection 3 - Confidence: 0.77

Zoom on box

[40]:
```
(tensor([[ 144.,  880.,  208.,  944.],
         [ 112.,  480.,  176.,  544.],
         [ 400.,  992.,  464., 1056.]]),
 tensor([0.9998, 0.9065, 0.7689]))
```

Waldo's was detected in the **first-ranked** image.

1.17 Image 15

[42]:
```
path_15 = "../data/Validation_Set/image15.png"
Finding_Waldo(path_15, n_show = 15, scales = [2])
```

```
Sliding Window Multi-Scale:   0%|          | 0/9744 [00:00<?, ?it/s]
```

```
3729
```


Detection 1 - Confidence: 0.98
Zoom on box
Detection 2 - Confidence: 0.98
Zoom on box
Detection 3 - Confidence: 0.97
Zoom on box
Detection 4 - Confidence: 0.97
Zoom on box
Detection 5 - Confidence: 0.97
Zoom on box
Detection 6 - Confidence: 0.97
Zoom on box
Detection 7 - Confidence: 0.96
Zoom on box
Detection 8 - Confidence: 0.96
Zoom on box
Detection 9 - Confidence: 0.96
Zoom on box
Detection 10 - Confidence: 0.96
Zoom on box
Detection 11 - Confidence: 0.95
Zoom on box
Detection 12 - Confidence: 0.94
Zoom on box
Detection 13 - Confidence: 0.94
Zoom on box
Detection 14 - Confidence: 0.93
Zoom on box
Detection 15 - Confidence: 0.93
Zoom on box

```
[42]: (tensor([[824., 104., 856., 136.],
               [  0., 400.,  32., 432.],
               [496., 656., 528., 688.],
               [136., 232., 168., 264.],
               [856., 432., 888., 464.],
               [ 64., 352.,  96., 384.],
               [856., 168., 888., 200.],
               [104., 560., 136., 592.],
               [784., 440., 816., 472.],
               [680., 152., 712., 184.],
               [312., 512., 344., 544.],
               [280., 456., 312., 488.],
               [704.,  24., 736.,  56.],
               [136., 120., 168., 152.],
               [880., 336., 912., 368.]]),
       tensor([0.9838, 0.9765, 0.9732, 0.9707, 0.9692, 0.9661, 0.9637, 0.9602, 0.9590,
               0.9588, 0.9480, 0.9426, 0.9364, 0.9339, 0.9319]))
```

Waldo was detected in the **fifteenth** position of the ranking, which is a failiure.

1.17.1 5.2. Summary of Validation Results

1.17.2 5.2. Validation Results Summary

Image ID	Success	Notes
image1		—
image2		—
image3		—
image4		—
image5		—
image6		—
image7		—
image8		—
image9		Not detected
image10		—
image11		—
image12		—
image13		—
image14		—
image15		Detected in the fifteenth position of the ranking

Summary
- **Total successful detections: 13 / 15 images**

- **Failed detections**: images 9 and 15
- **Images that required red_intensse adjustment**: images 10 and 12

1.18 6. Evaluation Results and Discussion

Waldo was successfully detected in **86.7%** of the test cases.

This result highlights not only the effectiveness of the sliding window approach combined with tuned parameters, but also the **robustness and consistency** of the model across different images.

However, it is important to note that the validation set is relatively small, which limits our ability to confidently generalize this success rate to a broader, more diverse dataset.

The only two images where Waldo was not detected (`image9` and `image15`) are visually complex and contain numerous distracting patterns that closely resemble Waldo's outfit. This makes detection especially challenging—even for a human observer—and reveals the limitations of the current **patch-based detection** strategy.

1.18.1 Missed Cases

- `image9`: Waldo was not detected, likely due to one or more of the following reasons:
  1. The NMS algorithm prioritized overlapping predictions with slightly lower confidence, suppressing the true positive.
  2. The stride may have been too large to properly cover the small region where Waldo's head appears.
- `image15`: Although the model **correctly detected Waldo in its patch**, the final ranking placed **14 false positives** above it. These false positives scored higher, leading to a missed detection in the final output.

1.18.2 Future Directions

To improve detection accuracy and efficiency, a promising direction would be to integrate an **object detection model** to first propose candidate regions before applying the classifier. This could drastically reduce the number of sliding window evaluations needed.

That said, *Where's Waldo?* illustrations contain a **high density of small, visually similar elements**, which may cause traditional region proposal networks to underperform. Significant adaptation would be required to make such models effective in this highly specialized visual domain.

Another promising line of improvement would be to explore **attention-based mechanisms**, which could allow the model to focus more on **where** Waldo might be located in the image, rather than strictly on **how** Waldo looks in isolated patches.

Unlike sliding window approaches that exhaustively scan the image with fixed-size windows, attention algorithms can dynamically prioritize **regions of interest** based on contextual cues, visual saliency, or learned spatial patterns. This paradigm shift could enable the model to process the image more holistically, considering both local features and global layout when estimating the most likely locations of Waldo.

By incorporating transformer-based architectures or visual attention modules, the system could learn to highlight regions that are more likely to contain human figures or clothing patterns similar to Waldo's, without needing to evaluate every possible patch. This would not only reduce the

computational cost but also help the model **disambiguate complex backgrounds** filled with distractors that resemble Waldo.

Such techniques could complement or even replace the current exhaustive patch classification, offering a more **intelligent and efficient** approach to detection, particularly in high-resolution images with dense visual information like *Where's Waldo?* scenes.