Nizhny Novgorod State University
Institute of Information Technologies, Mathematics and Mechanics
Department of Computer Software and Supercomputer Technologies

Educational course «Modern methods and technologies of deep learning in computer vision»

Practice №2
Object detection in images using deep neural networks

Supported by Intel

Vasiliev E.P.

Nizhny Novgorod
2020

Content

1 Introduction 3

2 Guidelines 3

2.1 Goals and tasks 3

2.2 Practice structure 3

2.3 Recommended study sequence 3

3 Object detection using the Intel Distribution of OpenVINO Toolkit 3

3.1 Deep models for object detection included in the Open Model Zoo repository 3

3.2 Extensions of layer implementations for deep models inference using Inference Engine 4

3.3 Executing the OpenVINO sample of object detection 5

4 Developing the object detection application using the OpenVINO Toolkit 6

4.1 File structure 6

4.2 Loading model 6

4.3 Loading and preprocessing image 7

4.4 Inferring model 7

4.5 Processing model output 7

4.6 Implementing sample 8

4.6.1 Parsing command line options 8

4.6.2 Implementing main function 8

5 Executing developed sample 9

6 Additional tasks 10

7 Literature 10

7.1 Books 10

7.2 Further reading 10

7.3 References 10

1 Introduction

In this practice, we solve the problem of object detection and propose its solution using the Intel Distribution of OpenVINO Toolkit [7]. Pre-trained deep models for object detection included in the Open Model Zoo repository are used [8].

2 Guidelines

2.1 Goals and tasks

The goal of this practice is to study deep models for solving the problem of object detection using the Intel Distribution of OpenVINO Toolkit.

To achieve this goal, it is necessary to solve the following tasks:

- Study the lecture “Object detection in images using deep neural networks”.
- Study deep models for object detection included in the Open Model Zoo, and load a model for further solving the task. This tutorial is based on the example of SSD300 [2].
- Develop an application based on the Inference Engine component for object detection using the chosen model. The detection result should be displayed on the original image.
- Execute and verify the developed sample.

2.2 Practice structure

First, a brief description of deep models for object detection that are available in the Open Model Zoo is provided. Further, the application for solving the problem of object detection is developed step-by-step.

2.3 Recommended study sequence

The recommended practice sequence is as follows:

- Study the lecture “Object detection in images using deep neural networks”.
- Setup software environment for the Intel Distribution of OpenVINO Toolkit.
- Study deep models for object detection included in the Open Model Zoo, and load a model for further solving the task.
- Develop an application based on the Inference Engine component for object detection using the chosen model. The detection result should be displayed on the original image.

Note that the environment setup is described in detail in the first practice; so this step is omitted in this tutorial.

3 Object detection using the Intel Distribution of OpenVINO Toolkit

3.1 Deep models for object detection included in the Open Model Zoo repository

Currently, there are several architectures of deep models that are used to solve the problem of object detection.

- Region-based Convolutional Networks (RCNN) [3].
- Region-Based Fully Convolutional Networks (RFCN) [4].
- Single Shot MultiBox Detector (SSD) [2].
- You Only Look Once (YOLO) [5].

More information can be found in the lecture “Object detection in images using deep neural networks”, which is a part of this course. Here we will consider some models for object detection included in the Open Model Zoo.

Open Model Zoo contains a set of models based on the ***Single Shot Detector (SSD)***: ssd300, ssd512, mobilenet-ssd and others. These models have similar inputs and outputs. The input of the SSD-based models which were trained using Caffe is a tensor of the size $[B \times C \times H \times W]$, where B is the number of processed images in a batch, C is the number of image channels, H, W is the height and width of the input images. The output of the SSD-based models is a tensor of the size $[1 \times 1 \times N \times 7]$, in which each row (the last dimension of the tensor) contains the following parameters: [image_number, classid, score, left, bottom, right, top], where 'image_number' is an image number; 'classid' is a class identifier; 'score' is a confidence of the object location in the selected area; 'left, bottom, right, top' are coordinates of the bounding boxes in the range from 0 to 1.

Open Model Zoo also includes the ***RCNN-based models***. There are models faster_rcnn_resnet50_coco, faster_rcnn_inception_v2_coco, mask_rcnn_resnet50_atrous_coco, mask_rcnn_inception_v2_coco and some others. The input of the RCNN-based models trained using the TensorFlow library is different from the SSD-based models. These models converted into intermediate representation have two input tensors: the first tensor of the size $[B \times C \times H \times W]$ corresponds to the batch of processed images, the second tensor of the size $[B \times C]$, where C is a vector consisting of three values in the format $[H, W, S]$, where H is an image height, W is an image width, S is an image scale factor (usually 1). The output of the RCNN-based models, by analogy with the SSD-based models, is a tensor of the size $[1 \times 1 \times N \times 7]$ or $[N \times 7]$, in which each row (the last dimension of the tensor) contains the following parameters: [image_number, classid, score, (x_min, y_min), (x_max, y_max)], where 'image_number' is an image number; 'classid' is a class identifier; 'score' is a confidence of the object location in the selected area; '(x_min, y_min), (x_max, y_max)' are coordinates of the bounding boxes normalized to the image size in the range from 0 to 1.

In addition to the public models, the Open Model Zoo also contains deep models pre-trained by Intel. These models solve the problems of detecting specific objects (vehicles, car plates, people, faces, text). There are the following models: face-detection-retail, person-detection-retail, vehicle-detection-adas, vehicle-license-plate-detection-barrier-0106 and others. These models have an input and output similar to the SSD-based models. A full description of the detected classes, input and output for each model is available in the OpenVINO documentation or directory listed below.

```
<openvino_dir>/deployment_tools/open_model_zoo/models/intel
```

3.2 Extensions of layer implementations for deep models inference using Inference Engine

Deep models consist of a large number of different layers: convolutions, max pooling layers, etc. Usually, the layer implementation is assigned to the developers of a particular framework. When executing deep models on the CPU using the OpenVINO Toolkit, layer implementation from the Deep Neural Network Library (DNNL, previous name MKL-DNN) is used [8]. Over time, new models and new layers are developed. There is a possibility to extend a set of supported layers. The Inference Engine provides API to link dynamic libraries with user layer implementation. Thus, the OpenVINO team supports the models, for which the implementation of layers is not available in DNNL. An example of such layer is the Non-Maximum Suppression Layer which is the last layer of models based on SSD [2]. A dynamic library with special layers can be compiled using SSE or AVX instructions to speed up inference.

In samples and demos to link a dynamic library of the specific layer implementation, as a rule, it is required to specify the **`-l`** argument and the path to this dynamic library.

```
python any_openvino_sample.py -l
"path_to_your_compiled_extension\cpu_extension.dll" <other_arguments...>
```

Developing application using the Inference Engine component after creating an **`IECore`** environment object, you should call the **`add_extension`** method, which takes two parameters: the path to the dynamic library and the computational device type (**`'CPU'`**).

```
ie = IECore()
```

```
ie.add_extension(cpu_extension_path, 'CPU')
```

3.3 Executing the OpenVINO sample of object detection

The OpenVINO Toolkit contains the **`detection_sample.py`** file, which allows you to detect objects in images using deep neural networks. On the official web-site there is a full description of this example and execution guide [10]. Here we give a brief information necessary for a quick start.

To detect objects using the corresponding OpenVINO sample, please, download and convert the SSD300 model from the Open Model Zoo repository, and execute the sample. The sequence of commands is given below, you need to replace the paths in angle brackets with the real paths of your computer.

```
python "C:\Program Files
(x86)\Intel\openvino_2021\deployment_tools\tools\model_downloader\download
er.py" --name ssd300 --output_dir <destination_folder>
python "C:\Program Files
(x86)\Intel\openvino_2021\deployment_tools\tools\model_downloader\converte
r.py" --name ssd300 --download_dir <destination_folder>
python "C:\Program Files
(x86)\Intel\openvino_2021\inference_engine\samples\python\object_detection
_sample_ssd\object_detection_sample_ssd.py" -i <path_to_image> -m
<path_to_model>\ssd300.xml
```

After starting the sample, the output of the sample should be written in the console, and the **`out.bmp`** image with bounding boxes around detections will appear in the current directory (if the directory is open for writing).

```
[ INFO ] Loading Inference Engine
[ INFO ] Loading network files:
ssd300.xml
ssd300.bin
[ INFO ] Device info:
CPU
MKLDNNPlugin version ......... 2.1
Build ........... 2021.1.0-1237-bece22ac675-releases/2021/1
inputs number: 1
input shape: [1, 3, 300, 300]
input key: data
[ INFO ] File was added:
[ INFO ] dog.jpg
[ WARNING ] Image dog.jpg is resized from (1200, 2274) to (300, 300)
[ INFO ] Preparing input blobs
[ INFO ] Batch size is 1
[ INFO ] Preparing output blobs
[ INFO ] Loading model to the device
[ INFO ] Creating infer request and starting inference
[ INFO ] Processing output blobs
[0,3] element, prob = 0.015166 (1551,32)-(1695,129) batch id : 0
[1,3] element, prob = 0.00843134 (1507,22)-(1727,174) batch id : 0
[2,3] element, prob = 0.00616033 (1589,58)-(1711,146) batch id : 0
...
[197,18] element, prob = 0.00396589 (-132,-63)-(532,257) batch id : 0
[198,18] element, prob = 0.00327471 (-256,571)-(692,828) batch id : 0
[199,18] element, prob = 0.00327383 (-265,698)-(696,968) batch id : 0
[ INFO ] Image out.bmp created!
[ INFO ] Execution successful
[ INFO ] This sample is an API example, for any performance measurements
please use the dedicated benchmark_app tool
```

The source code of this sample and other demos can be used to study the OpenVINO interface. The next section provides the step-by-step tutorial for developing your own application for object detection.

4 Developing the object detection application using the OpenVINO Toolkit

4.1 File structure

For this practice, please, create two files: **`ie_detector.py`** is a file containing the **`InferenceEngineDetector`** class, and **`detection_sample.py`** is a file containing the testing code for the **`InferenceEngineDetector`** class.

Methods of the **`InferenceEngineDetector`** class:

- **`_init_`** is a constructor, it initializes the Inference Engine and loads the model from file.
- **`_prepare_image`** is a method to convert the image into the deep model input array.
- **`detect`** is a method to detect objects in images using the deep model.
- **`draw_detection`** is a method to draw detected objects on the image.

```
class InferenceEngineDetector:
    def __init__(self, configPath = None, weightsPath = None,
            extension=None, classesPath = None):
        pass

    def _prepare_image(self, image, h, w):
        pass

    def detect(self, image):
        pass

    def draw_detection(self, detections, image, confidence=0.5,
                       draw_text=True):
        pass
```

In the next subsections, we implement the above methods.

4.2 Loading a deep model from file

In order to load the model, we need to implement the constructor of the **`InferenceEngineDetector`** class placed in the **`ie_detector.py`** file. The constructor receives the following required and optional parameters:

- **`configPath`** is a path to the .xml file of the model description.
- **`weightsPath`** is a path to the .bin file of the model weights.
- **`classesPath`** is a path to the file containing class names for the given detection model.
- **`extension`** is a path to the file of the specific layer implementation.

The constructor performs the following actions:

- Creating an object of the **`IECore`** class.
- Creating an object of the **`IENetwork`** class with parameters corresponding to the model paths.
- Loading the created object of the **`IENetwork`** class into the **`IECore`** object, this means loading the model into the plugin.
- Loading the class names from the file located at the **`classesPath`** path.

Objects of the **`IECore`**, **`IENetwork`**, **`ExecutableNetwork`** classes, should be stored as the **`InferenceEngineDetector`** class fields.

4.3 Loading and preprocessing image

The next step is to implement the **_prepare_image** method. Deep models require images in a per-channel format, and not pixel-by-pixel format, input images have to be converted from the format RGBRGBRG... to the format RRRGGGBBB... The **transpose** function can be used for such conversion.

```
image = image.transpose((2, 0, 1))
```

It is also required to resize the image to the size of the network input.

```
image = cv2.resize(image, (w, h))
```

In common, a 4-dimensional tensor should be set to the model input, for example, tensor [1,3,300,300], where the first coordinate is the number of images in a batch; 3 is a number of color channels of the image; 300, 300 are width and height of the image. However, if a 3-dimensional tensor [3,300,300] is set as the network input, then the OpenVINO Toolkit will automatically add the fourth dimension.

4.4 Inferring model

The following step is to implement the **detect** method, which launches the inference of a deep model on the device specified in the constructor. The sequence of operations for the **detect** method is as follows:

1. Get information about model input and output.

```
input_blob = next(iter(self.net.inputs))
out_blob = next(iter(self.net.outputs))
```

2. From the model input, obtain the input dimension required by the model for the image.

```
n, c, h, w = self.net.inputs[input_blob].shape
```

3. Preprocess image using the **_prepare_image** method.

```
blob = self._prepare_image(image, h, w)
```

4. Infer the model in synchronous mode.

```
output = self.exec_net.infer(inputs = {input_blob: blob})
```

5. Extract the tensor with the detection result from the model output.

```
output = output[out_blob]
```

4.5 Processing model output

The output of the most SSD-based models is a tensor of the size [1,1,N,7], in which each row (the last dimension of the tensor) contains the following parameters: [image_number, classid, score, left, bottom, right, top], where ‘image_number’ is a number of images; ‘classid’ is a class identifier; ‘score’ is a confidence of the object location in the selected area; ‘left, bottom, right, top’ are coordinates of the bounding boxes in the range from 0 to 1.

To process the output, you should implement the **draw_detection** method, which draws the constructed bounding boxes in the image. The sequence of the actions is as follows. In a loop through the output rows:

1. Extract the current row of the matrix.
2. Extract the confidence of the detected object (third parameter in the row).
3. If the confidence is greater than a threshold value (0.5 is recommended), then get the class identifier and the coordinates of the bounding box. The class identifier can be used to get the class name. To obtain the coordinates of the bounding boxes in the coordinate system associated with the image, it is necessary to multiply the normalized values obtained from the output tensor by the height and width of the input image.

4. Draw a rectangle on the image using OpenCV. To draw rectangle, use the **`cv2.rectangle`** function. The parameters description and example of using this function are given below.

```
cv2.rectangle(img, point1, point2, color, line_width)
```

- **`img`** is an image to draw detections.
- **`point1 = (x,y)`** is a tuple of two integers corresponding to the coordinates of the top left corner of the bounding box.
- **`point2 = (x,y)`** is a tuple of two integers corresponding to the coordinates of the bottom right corner of the bounding box.
- **`color = (B,G,R)`** is a tuple of three integers from 0 to 255, which determines the color of the line.
- **`line_width = 1`** is a floating-point number that determines the thickness of the line.

To display an object class name on the image, use the **`cv2.puttext`** function. The parameters description and example of using the function are given below.

- **`img`** is an image to draw detections.
- **`text`** is a text for label.
- **`point`** are coordinates of the start text position.
- **`color`** is a tuple of three integers from 0 to 255, which determines the color of the text.
- **`text_size = 0.45`** is a floating-point number that determines the size of the text.

4.6 Implementing sample

4.6.1 Parsing command line options

In this practice, the following command line arguments will be required:

- Path to the input image (required).
- Path to the model weights file (required).
- Path to the model configuration file (required).
- Path to the dynamic library with custom layers (it is required to infer the SSD-based models on CPU) (optional).
- Path to the file containing class names (optional).

The implementation of the command line parser using the **`argparse`** package is represented below.

```
def build_argparser():
    parser = argparse.ArgumentParser()
    parser.add_argument('-m', '--model', help = 'Path to an .xml \
        file with a trained model.', required = True, type = str)
    parser.add_argument('-w', '--weights', help = 'Path to an .bin file \
        with a trained weights.', required = True, type = str)
    parser.add_argument('-i', '--input', help = 'Path to \
        image file', required = True, type = str)
    parser.add_argument('-l', '--cpu_extension', help='MKLDNN \
        (CPU)-targeted custom layers. Absolute path to a shared library \
        with the kernels implementation', type=str, default=None)
    parser.add_argument('-c', '--classes', help = 'File containing \
        classnames', type = str, default = None)
    return parser
```

4.6.2 Implementing main function

In the file **`detection_sample.py`** create a function **`main`** that implements the following steps:

- Parsing command line arguments.

- Creating an object of the **`InferenceEngineDetector`** class with the necessary parameters.
- Reading the image.
- Detecting objects in the image.
- Drawing detected bounding boxes in the image.
- Displaying the detection results on the screen.

```
def main():
    args = build_argparser().parse_args()

    ie_detector = InferenceEngineDetector(configPath=args.model,
        weightsPath=args.weights, device=args.device,
        extension=args.cpu_extension, classesPath=args.classes)

    img = cv2.imread(args.input)

    detections = ie_detector.detect(img)

    image_detected = ie_detector.draw_detection(detections, img)

    cv2.imshow('Image with detections', image_detected)
    cv2.waitKey(0)
    cv2.destroyAllWindows()

    return
```

5 Executing developed sample

The easiest way to execute your sample is the command line represented below.

```
python ie_detection_sample.py -i image.jpg -m ssd300.xml -w ssd300.bin \
    -c voc_labels.txt
```

The **`-i`** argument specifies the path to the image, the **`-m`** argument specifies the model configuration path, the **`-w`** argument specifies the model weight path, the **`-c`** argument specifies the path to the file containing object class names.

The result of the application execution is as follows. A message about the start of the application is displayed, then a window is opened in which the image with the detected bounding boxes are drawn (Fig. 1).

```
[ INFO ] Start IE detection sample
```

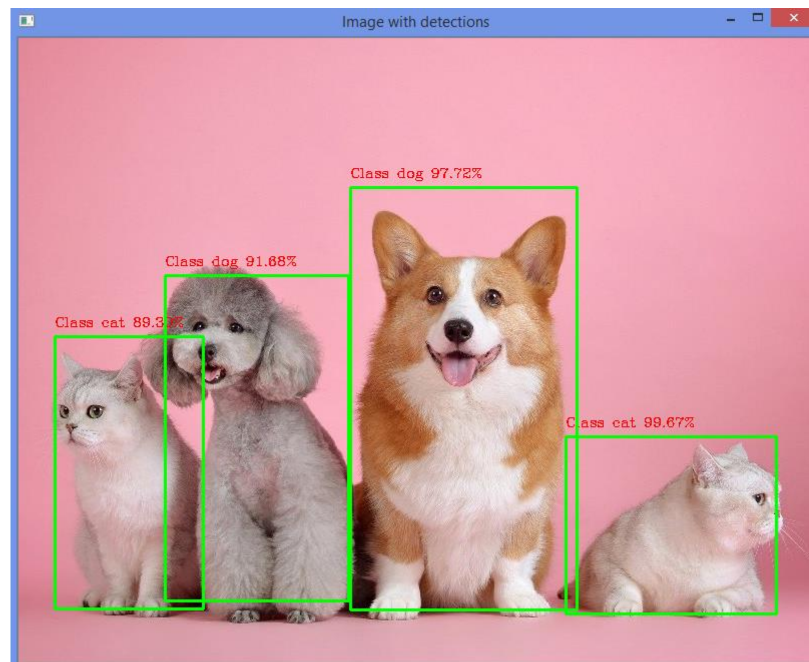

Fig. 1. Example of detected objects

6 Additional tasks

The developed detection sample contains the minimum required functionality. As additional tasks, it is proposed to provide support for the following features:

1. Loading a video and detecting objects in each frame, drawing detected bounding boxes on each frame.
2. Supporting other deep models for object detection included in the Open Model Zoo [8].
3. Measuring the time required image processing.

It is proposed to solve these tasks independently using the documentation and examples included in the OpenVINO Toolkit.

7 Literature

7.1 Books

1. Chollet F. Deep Learning with Python. – Manning Publications Co, NY, USA, – 2017.
2. Liu W., Anguelov D., Erhan D., Szegedy C., Reed S., Fu C.Y., Berg A.C. SSD: Single Shot MultiBox Detector. – 2016.
3. Girshick R., Donahue J., Darrell T., Malik J. Rich feature hierarchies for accurate object detection and semantic segmentation. – 2014.
4. Dai J., Li Y., He K., Sun J. R-FCN: Object detection via region-based fully convolutional networks. – 2016.
5. Redmon J., Divvala S., Girshick R., Farhadi A. You only look once: Unified, real-time object detection. – 2015.

7.2 Further reading

6. Ramalho L. Fluent Python: Clear, Concise, and Effective Programming. – O'Reilly Media, Inc., CA, USA, 2015.

7.3 References

7. OpenVINO detection sample [https://docs.openvinotoolkit.org/latest/openvino_inference_engine_ie_bridges_python_sample_object_detection_sample_ssd_README.html].
8. Open Model Zoo repository of deep models [https://github.com/openvinotoolkit/open_model_zoo].

9. Deep Neural Network Library repository [https://github.com/oneapi-src/oneDNN].
10. OpenVINO detection sample [https://docs.openvinotoolkit.org/latest/openvino_inference_engine_ie_bridges_python_sample_object_detection_sample_ssd_README.html].