problem1

April 23, 2022

1 Problem 1

Problem 1 - SSD, ONNX model, Visualization, Inferencing - 25 points

In this problem we will be inferencing SSD ONNX model using ONNX Runtime Server. You will follow the github repo and ONNX tutorials (links provided below). You will start with a pretrained Pytorch SSD model and retrain it for your target categories. Then you will convert this Pytorch model to ONNX and deploy it on ONNX runtime server for inferencing.

1.1 1

Download pretrained pytorch MobilenetV1 SSD and test it locally using Pascal VOC 2007 dataset.

Show the test accuracy for the 20 classes. (3)

[2]:
```
import torchvision
```

[3]:
```
test_dataset = torchvision.datasets.VOCDetection(root="./cached_datasets/",␣
↪year="2007", image_set="test", download=True)
```

```
Downloading
http://host.robots.ox.ac.uk/pascal/VOC/voc2007/VOCtest_06-Nov-2007.tar to
./cached_datasets/VOCtest_06-Nov-2007.tar

100%|     | 451020800/451020800 [00:23<00:00, 19024674.39it/s]

Extracting ./cached_datasets/VOCtest_06-Nov-2007.tar to ./cached_datasets
```

Average Precision Per-class: aeroplane: 0.6843271224059599 bicycle: 0.791107801540578 bird: 0.6171805882596568 boat: 0.5612237523337076 bottle: 0.3485127722238744 bus: 0.7677950222981742 car: 0.7280909456890072 cat: 0.8369208203985581 chair: 0.5169101226046323 cow: 0.6237776339227519 diningtable: 0.7062934388182485 dog: 0.7872444117558195 horse: 0.819446325939355 motorbike: 0.7918539457195842 person: 0.7023473947442752 pottedplant: 0.39857841799969324 sheep: 0.6066767547850755 sofa: 0.7573708212365823 train: 0.8262441264750008 tvmonitor: 0.6462024812595099

Average Precision Across All Classes:0.6759052350205023

1.2 2

Select any two related categories from Google Open Images dataset and finetune the pretrained SSD model. Examples include, Aircraft and Aeroplane, Handgun and Shotgun. You can use open_images_downloader.py script provided at the github to download the data. For finetuning you can use the same parameters as in the tutorial below. Compute the accuracy of the test data for these categories before and after finetuning. (4+4)

[2]:
```
import torch
```

1.2.1 Before Finetuning

Average Precision Per-class: aeroplane: 0.6843271224059599

1.2.2 After Finetuning

Average Precision Per-class: Aircraft: 0.28223158259175773

Average Precision Across All Classes:0.28223158259175773

1.3 3

Convert the Pytorch model to ONNX format and save it. (2)

[4]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms
from torch.autograd import Variable
import torch.onnx as torch_onnx

# class Model(nn.Module):
#     def __init__(self):
#         super(Model, self).__init__()
#         self.conv = nn.Conv2d(in_channels=3, out_channels=32,␣
 ↪kernel_size=(3,3), stride=1, padding=0, bias=False)

#     def forward(self, inputs):
#         x = self.conv(inputs)
#         #x = x.view(x.size()[0], x.size()[1], -1)
#         return torch.mean(x, dim=2)

# # Use this an input trace to serialize the model
# input_shape = (3, 100, 100)
# model_onnx_path = "torch_model.onnx"
# model = Model()
# model.train(False)

# # Export the model to an ONNX file
# dummy_input = Variable(torch.randn(1, *input_shape))
# output = torch_onnx.export(model,
#                            dummy_input,
#                              model_onnx_path,
#                              verbose=False)
# print("Export of torch_model.onnx complete!")
```

[14]:
```
cd "third_party/pytorch-ssd"
```

```
/home/xiangpan/Labs/NYU_DL_Sys/HW5/third_party/pytorch-ssd
```

[15]:
```
pwd
```

[15]:
```
'/home/xiangpan/Labs/NYU_DL_Sys/HW5/third_party/pytorch-ssd'
```

[16]:
```
from vision.ssd.mobilenetv1_ssd import create_mobilenetv1_ssd
```

[19]:
```
model = create_mobilenetv1_ssd(num_classes=21)
```

[20]:
```
state_dict_path = "/home/xiangpan/Labs/NYU_DL_Sys/HW5/cached_models/
 ↪mobilenet-v1-ssd-mp-0_675.pth"
state_dict = torch.load(state_dict_path)
```

[21]:
```
model.load_state_dict(state_dict)
```

[21]:
```
<All keys matched successfully>
```

[22]:
```
model
```

[22]:
```
SSD(
  (base_net): Sequential(
    (0): Sequential(
      (0): Conv2d(3, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1),
bias=False)
      (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): ReLU(inplace=True)
    )
    (1): Sequential(
      (0): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
groups=32, bias=False)
      (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): ReLU(inplace=True)
      (3): Conv2d(32, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (5): ReLU(inplace=True)
    )
    (2): Sequential(
      (0): Conv2d(64, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1),
groups=64, bias=False)
      (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): ReLU(inplace=True)
      (3): Conv2d(64, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (4): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (5): ReLU(inplace=True)
    )
    (3): Sequential(
      (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
groups=128, bias=False)
      (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): ReLU(inplace=True)
      (3): Conv2d(128, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (4): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (5): ReLU(inplace=True)
    )
    (4): Sequential(
      (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1),
groups=128, bias=False)
      (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): ReLU(inplace=True)
      (3): Conv2d(128, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (5): ReLU(inplace=True)
    )
    (5): Sequential(
      (0): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
groups=256, bias=False)
      (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): ReLU(inplace=True)
      (3): Conv2d(256, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (5): ReLU(inplace=True)
    )
    (6): Sequential(
      (0): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1),
groups=256, bias=False)
      (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): ReLU(inplace=True)
      (3): Conv2d(256, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (4): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (5): ReLU(inplace=True)
    )
    (7): Sequential(
      (0): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
groups=512, bias=False)
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): ReLU(inplace=True)
      (3): Conv2d(512, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (4): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (5): ReLU(inplace=True)
    )
    (8): Sequential(
      (0): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
groups=512, bias=False)
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): ReLU(inplace=True)
      (3): Conv2d(512, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (4): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (5): ReLU(inplace=True)
    )
    (9): Sequential(
      (0): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
groups=512, bias=False)
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): ReLU(inplace=True)
      (3): Conv2d(512, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (4): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (5): ReLU(inplace=True)
    )
    (10): Sequential(
      (0): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
groups=512, bias=False)
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): ReLU(inplace=True)
      (3): Conv2d(512, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (4): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (5): ReLU(inplace=True)
    )
    (11): Sequential(
      (0): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
groups=512, bias=False)
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): ReLU(inplace=True)
      (3): Conv2d(512, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (4): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (5): ReLU(inplace=True)
    )
    (12): Sequential(
      (0): Conv2d(512, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1),
groups=512, bias=False)
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): ReLU(inplace=True)
      (3): Conv2d(512, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (4): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (5): ReLU(inplace=True)
    )
    (13): Sequential(
      (0): Conv2d(1024, 1024, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
groups=1024, bias=False)
      (1): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): ReLU(inplace=True)
      (3): Conv2d(1024, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (4): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (5): ReLU(inplace=True)
    )
  )
  (extras): ModuleList(
    (0): Sequential(
      (0): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1))
      (1): ReLU()
      (2): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
      (3): ReLU()
    )
    (1): Sequential(
      (0): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1))
      (1): ReLU()
      (2): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
      (3): ReLU()
    )
    (2): Sequential(
      (0): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1))
      (1): ReLU()
      (2): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
      (3): ReLU()
    )
    (3): Sequential(
      (0): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1))
      (1): ReLU()
      (2): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
      (3): ReLU()
    )
  )
  (classification_headers): ModuleList(
    (0): Conv2d(512, 126, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): Conv2d(1024, 126, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (2): Conv2d(512, 126, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): Conv2d(256, 126, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): Conv2d(256, 126, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (5): Conv2d(256, 126, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  )
  (regression_headers): ModuleList(
    (0): Conv2d(512, 24, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): Conv2d(1024, 24, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (2): Conv2d(512, 24, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): Conv2d(256, 24, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): Conv2d(256, 24, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (5): Conv2d(256, 24, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  )
  (source_layer_add_ons): ModuleList()
)
```

[24]:
```
model.train(False)

model_onnx_path = "/home/xiangpan/Labs/NYU_DL_Sys/HW5/cached_models/
 ↪mobilenet-v1-ssd-mp-0_675.onnx"
# Export the model to an ONNX file
input_shape = (3, 300, 300)
dummy_input = Variable(torch.randn(1, *input_shape))
output = torch_onnx.export(model,
                          dummy_input,
                          model_onnx_path,
                          verbose=False)
```

```
print("Export of torch_model.onnx complete!")
```

```
Export of torch_model.onnx complete!
```

1.4 4

Visualize the model using net drawer tool. Compile the model using embed_docstring flag and show the visualization output. Also show doc string (stack trace for PyTorch) for different types of nodes. (4)

1.4.1 visualize_model

Without embed_docstring

With embed_docstring

1.4.2 compile

Compile the model using embed_docstring flag and show the visualization output. Also show doc string (stack trace for PyTorch) for different types of nodes.

TODO!!!

doc string

[35]:
```
cd /home/xiangpan/Labs/NYU_DL_Sys/HW5
```

```
/home/xiangpan/Labs/NYU_DL_Sys/HW5
```

[36]:
```
torch_model = model

from torch.autograd import Variable
batch_size = 1    # just a random number

# Input to the model
x = Variable(torch.randn(batch_size, 3, 224, 224), requires_grad=True)

# Export the model
torch_out = torch.onnx._export(torch_model,             # model being run
                               x,                       # model input (or a␣
 ↪tuple for multiple inputs)
                               "./cached_models/mobilenet-v1-ssd-mp-0_675.
 ↪onnx",       # where to save the model (can be a file or file-like object)
                               export_params=True)      # store the trained␣
 ↪parameter weights inside the model file
```

1.5 5

Deploy the ONNX model on ONNX runtime (ORT) server. You need to set up the environment following steps listed in the tutorial. Then you need make HTTP request to the ORT server. Test the inferencing set-up using 1 image from each of the two selected categories. (4)

1.6 6

Parse the response message from the ORT server and annotate the two images. Show inferencing output (bounding boxes with labels) for the two images. (4)