RTML Lab 4: YOLO (v3 and v7)

NAME = "Todsavad Tangtortan" ID = "123012"

New Homework (Will release on 3 Feb 2023)

Independent exercise: YOLOR

Part I: Inference (due next week)

In the lab, we saw how the Darknet configuration file for YOLOv3 could be read in Python and mapped to PyTorch modules.

For your independent work do the same thing for YOLOv4. Download the `yolov4.cfg` file from the YOLOv4 GitHub repository and modify your `MyDarknet` class and utility code (`darknet.py` , `util.py`) as necessary to map the structures to PyTorch.

The changes you'll have to make:

1. Implement the mish activation function
2. Add an option for a maxpool layer in the `create_modules` function and in your model's `forward()` method.
3. Enable a `[route]` module to concatenate more than two previous layers
4. Load the pre-trained weights provided by the authors
5. Scale inputs to 608×608 and make sure you're passing input channels in RGB order, not OpenCV's BGR order.

Modify `darknet.py`

- Implement the mish activation function.

```
class Mish(nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, x):
        return x * torch.tanh(F.softplus(x))

if activation == "mish":
    activn = Mish()
    module.add_module("mish_{0}".format(index), activn)
```

- Add an option for a maxpool layer in the `create_modules` function.

```
elif x["type"] == "maxpool":
    stride = int(x["stride"])
    size = int(x["size"])
    assert size % 2
    maxpool = nn.MaxPool2d(kernel_size=size, stride=stride, padding=size // 2)
    module.add_module("maxpool_{0}".format(index), maxpool)
```

- Add condition in your model's `forward()` method.

```
 if module_type == "convolutional" or module_type == "upsample" or module_type == "maxpool":
     x = self.module_list[i](x)
```

- Enable a `[route]` module to concatenate more than two previous layers

```
elif module_type == "route":

    # concat layers
    layers = module["layers"]
    layers = [int(a) for a in layers]

    if (layers[0]) > 0:
        layers[0] = layers[0] - i

    if len(layers) == 1:    # 1 item in layer
        x = outputs[i + (layers[0])]

    else:   # more than 1 item in layer
        if len(layers) == 4:       # 4 items in layer
            if (layers[1]) > 0:
                layers[1] = layers[1] - i

            if (layers[2]) > 0:
                layers[2] = layers[2] - i

            if (layers[3]) > 0:
                layers[3] = layers[3] - i

            map1 = outputs[i + layers[0]]
            map2 = outputs[i + layers[1]]
            map3 = outputs[i + layers[2]]
            map4 = outputs[i + layers[3]]
            x = torch.cat((map1, map2, map3, map4), 1)

        else:           # 2 items in layer
            if (layers[1]) > 0:
                layers[1] = layers[1] - i

            map1 = outputs[i + layers[0]]
            map2 = outputs[i + layers[1]]
            x = torch.cat((map1, map2), 1)
```

- Load the pre-trained weights provided by the authors

```
model = Darknet("cfg/yolov4.cfg")
model.load_weights("yolov4.weights")
```

- Scale inputs to 608×608 and make sure you're passing input channels in RGB order, not OpenCV's BGR order.

Modify `util.py`

```
def prep_image(img, inp_dim):
    """
    Prepare image for inputting to the neural network.

    Returns a Variable
    """
    # pylint: disable=no-member
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    img = letterbox_image(img, (inp_dim, inp_dim))
    img = img[:,:,::-1].transpose((2,0,1)).copy()
    img =  torch.from_numpy(img).float().div(255.0).unsqueeze(0)
```

In [1]:
```
from darknet import *
import torch
import cv2
from util import *

def load_classes(namesfile):
    fp = open(namesfile, "r")
    names = fp.read().split("\n")[:-1]
    return names

model = Darknet("cfg/yolov4.cfg")
model.module_list[114].conv_114 = nn.Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
input = get_test_input()
print(input.shape)
prediction = model(input, False)
# print(prediction)
print(prediction.shape)

model.load_weights("yolov4.weights")
input = get_test_input()
prediction = model(input, False)
write_results(prediction.detach(), 0.5, 80, nms_conf = 0.4)
# print(prediction)
print(prediction.shape)
# num_classes = 80
# classes = load_classes("../data/coco.names")

# print(classes)
```

```
/usr/local/lib/python3.8/dist-packages/tqdm/auto.py:22: TqdmWarning: IProgress not found. Please update jupyter and i
pywidgets. See https://ipywidgets.readthedocs.io/en/stable/user_install.html
  from .autonotebook import tqdm as notebook_tqdm
```

```
torch.Size([1, 3, 608, 608])
torch.Size([1, 22743, 85])
torch.Size([1, 22743, 85])
```

Part II: Training (due in two weeks)

Train the YOLOv4 model on the COCO dataset (or another dataset if you have one available). Here the purpose is not to get the best possible model (that would require implementing all of the "bag of freebies" training tricks described in the paper), but just some of them, to get a feel for their importance.

1. Get a set of ImageNet pretrained weights for CSPDarknet53 from the Darknet GitHub repository
2. Add a method to load the pretrained weights into the backbone portion of your PyTorch YOLOv4 model.
3. Implement a basic `train_yolo` function similar to the `train_model` function you developed in previous labs for classifiers that preprocesses the input with basic augmentation transformations, converts the anchor-relative outputs to bounding box coordinates, computes MSE loss for the bounding box coordinates, backpropagates the loss, and takes a step for the optimizer. Use the recommended IoU thresholds to determine which predicted bounding boxes to include in the loss. You will find many examples of how to do this online.
4. Train your model on COCO. Training on the full dataset to completion would take several days, so you can stop early after verifying the model is learning in the first few epochs.
5. Compute mAP for your model on the COCO validation set.
6. Implement the CIoU loss function and observe its effect on mAP.
7. (Optional) Train on COCO to completion and see how close you can get to the mAP reported in the paper.

There is some useful information on working with the COCO dataset as a Torchvision Dataset in this blog. For your work on this lab, the instructor will place the entire COCO training and validation datasets on a shared network drive for you to access so that we don't use resources for multiple copies of the dataset. Once you have access to the dataset you can use the dataset easily: