Chapter 9. PyTorch in the Wild

Speaker: Brilian
2020/03/10

SOLOMON

Goal

1	2	3	4
Introduce Other Data Augmentation Methods	Introduce Super Resolution (SR) and Generative Adversarial Network (GAN) and Show Their Applications	Brief review of Object Detection and Application	Describe ways to make our object detection to detect the class “incorrectly”, and how to prevent it

Outline

1. Data Augmentation
    a. Definition
    b. Mixup Augmentation
    c. Label Smoothing
2. Super Resolution and GAN
    a. Introduction to SR
    b. Introduction to GAN
    c. Applications
3. Review of Object Detection
    a. Object Detection
    b. Faster RCNN and Mask RCNN detection
4. Adversarial Samples
    a. White-box Attacks
    b. Black-box attacks
    c. Defending Against Adversarial Attacks
5. More Than Meets the Eye: The Transformer Architecture - (for Text)

Outline

1. Data Augmentation
2. Super Resolution and GAN
3. Review of Object Detection
4. Adversarial Samples

Definition

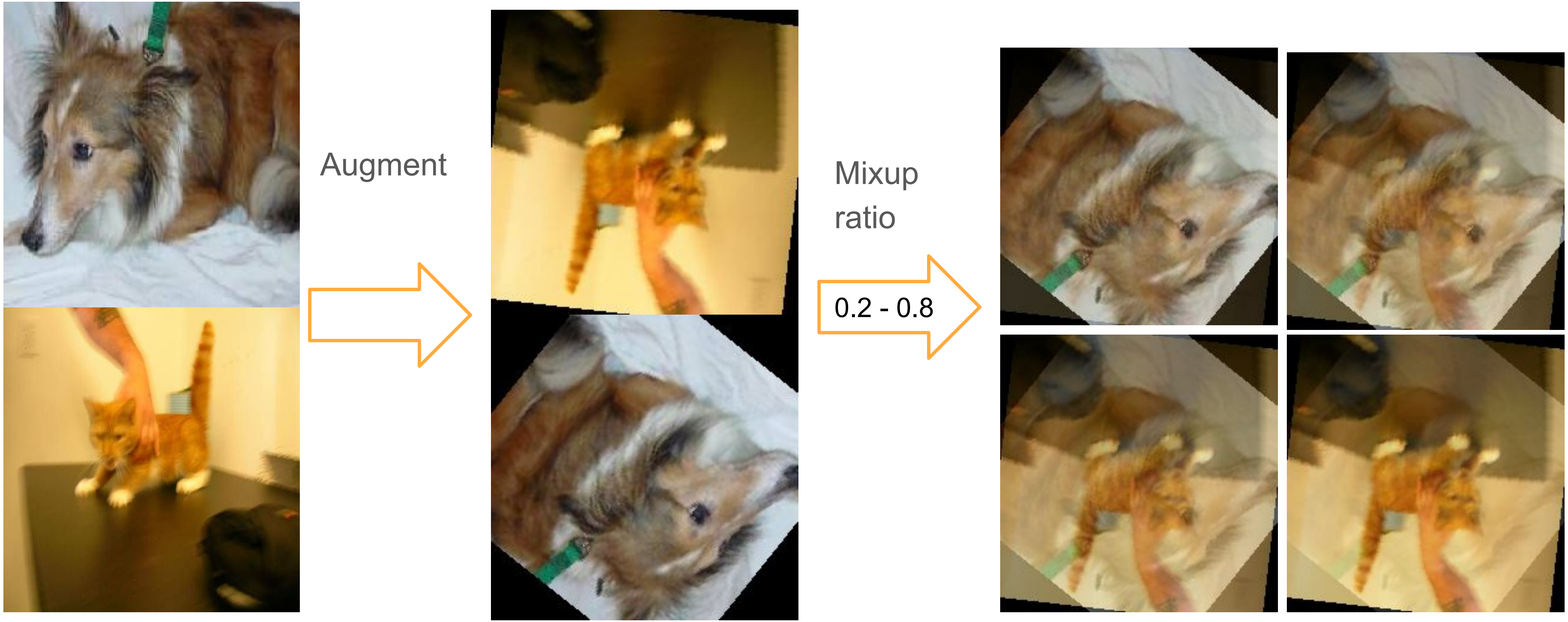
Ref: https://www.techopedia.com/definition/28033/data-augmentation

Data augmentation is a way to add value to base data by adding information derived from internal and external sources.

Data augmentation that will be discussed in here:

- Mixup augmentation: Mix some between 2 input with its labels to improve generalization
- Label Smoothing: Improve model performance by making the model less sure of its predictions.

Mixup Augmentation




```
inputs_mixed = (mixup * inputs) + (1-mixup * inputs_mix)
```

Mixup Segmentation

A. Prepare input (mix) and target (mix)
B. Prepare beta distribution (the purpose of this is to make the model to work harder to predict the mixed inputs)
C. Mix the input with the input mix with beta distribution
D. Mix the target
E. Do forward propagation (put the mixed input to the model)
F. Calculate loss with mixed input
G. Calculate gradient, then do backward propagation

```
for epoch in range(epochs):
  model.train()
  for batch in zip(train_loader,mix_loader):
```

A

```
    ((inputs, targets),(inputs_mix, targets_mix)) = batch
    optimizer.zero_grad()
    inputs = inputs.to(device)
    targets = targets.to(device)
    inputs_mix = inputs_mix.to(device)
    target_mix = targets_mix.to(device)
```

B

```
    distribution = torch.distributions.beta.Beta(0.5,0.5)
    beta = distribution.expand(torch.zeros(batch_size).shape).sample().to(device)

    # We need to transform the shape of beta
    # to be in the same dimensions as our input tensor
    # [batch_size, channels, height, width]
```

C

```
    mixup = beta[:, None, None, None]

    inputs_mixed = (mixup * inputs) + (1-mixup * inputs_mix)

    # Targets are mixed using beta as they have the same shape
```

D

```
    targets_mixed = (beta * targets) + (1-beta * inputs_mix)
```

E

```
    output_mixed = model(inputs_mixed)
```

```
    # Multiply losses by beta and 1-beta,
    # sum and get average of the two mixed losses
```

F

```
    loss = (loss_fn(output, targets) * beta
           + loss_fn(output, targets_mixed)
           * (1-beta)).mean()
```

```
    # Training method is as normal from herein on
```

G

```
    loss.backward()
    optimizer.step()
    …
```

Label Smoothing

```python
class LabelSmoothingCrossEntropyLoss(nn.Module):
    def __init__(self, epsilon=0.1):
        super(LabelSmoothingCrossEntropyLoss, self).__init__()
        self.epsilon = epsilon

    def forward(self, output, target):
        num_classes = output.size()[-1]
        log_preds = F.log_softmax(output, dim=-1)
        loss = (-log_preds.sum(dim=-1)).mean()
        nll = F.nll_loss(log_preds, target)
        final_loss = self.epsilon * loss / num_classes +
                     (1-self.epsilon) * nll
        return final_loss
```

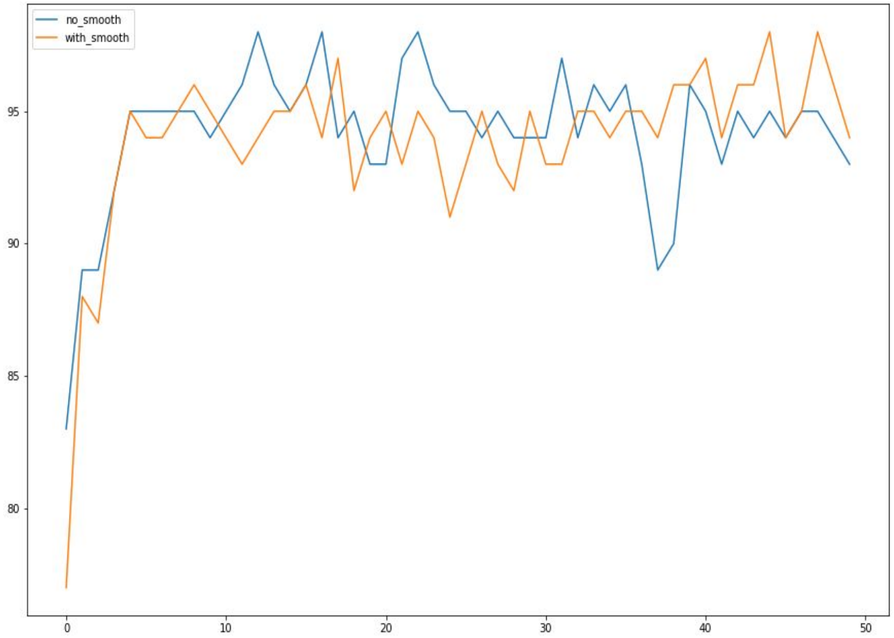
Normal Cross Entropy Loss as for Classification

Instead of trying to force it to predict 1 for the predicted class, we instead alter it to predict 1 minus a small value, i.e. epsilon.

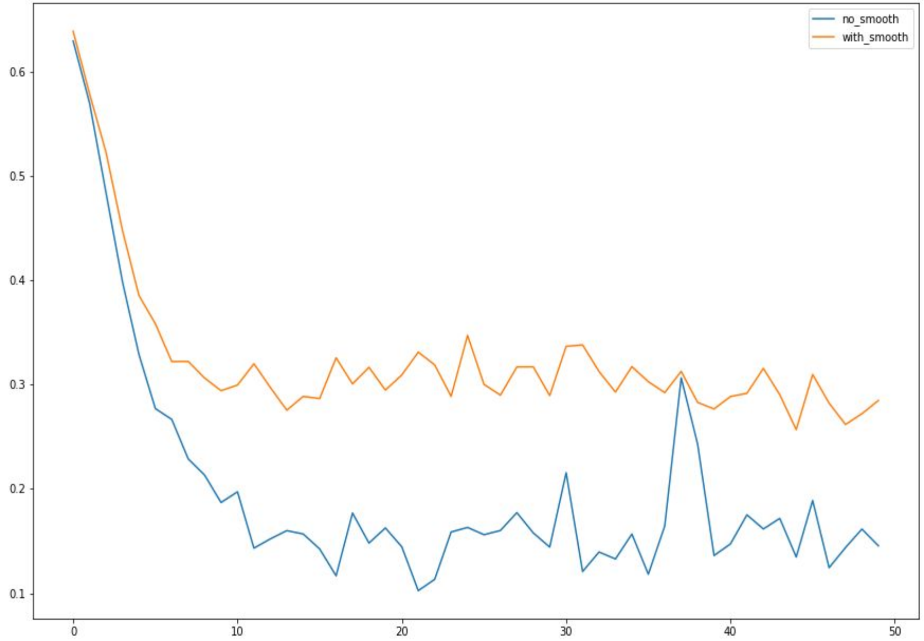
This occurs because we are smoothing not only the label for the predicted class to be 1 minus epsilon, but also the other labels so that they're not being forced to zero, but instead a value between zero and epsilon.

Label Smoothing

The accuracy seems to be better(?)



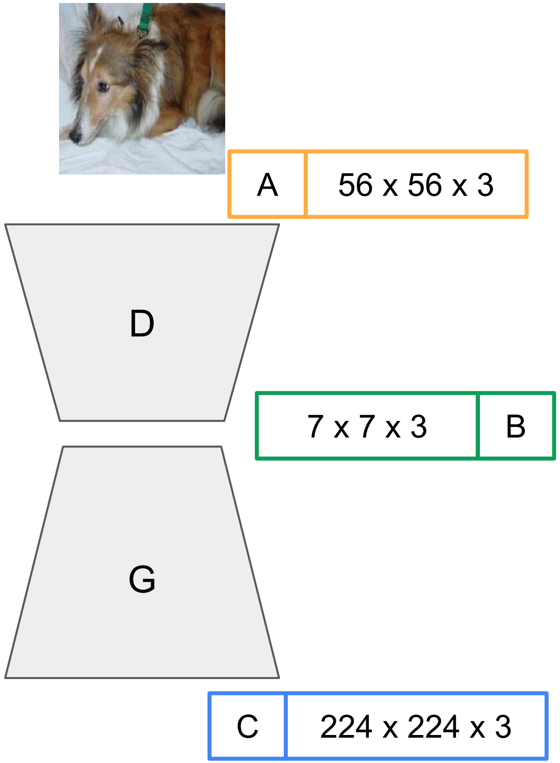
The loss seems to be higher and more stable

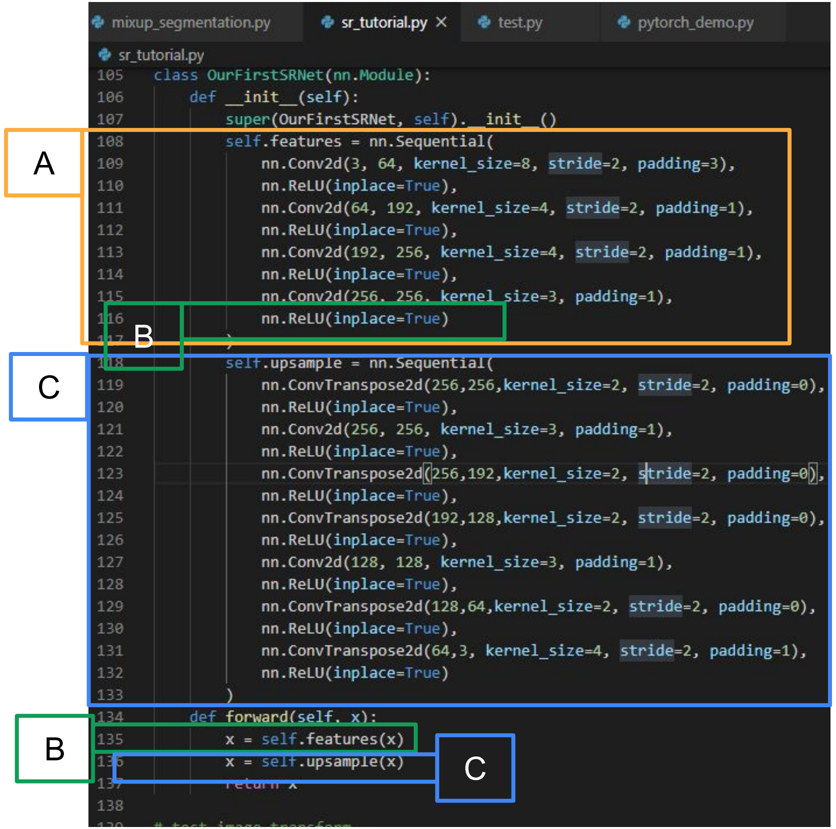


Outline

1. Data Augmentation
2. Super Resolution and GAN
3. Review of Object Detection
4. Adversarial Samples

Super Resolution

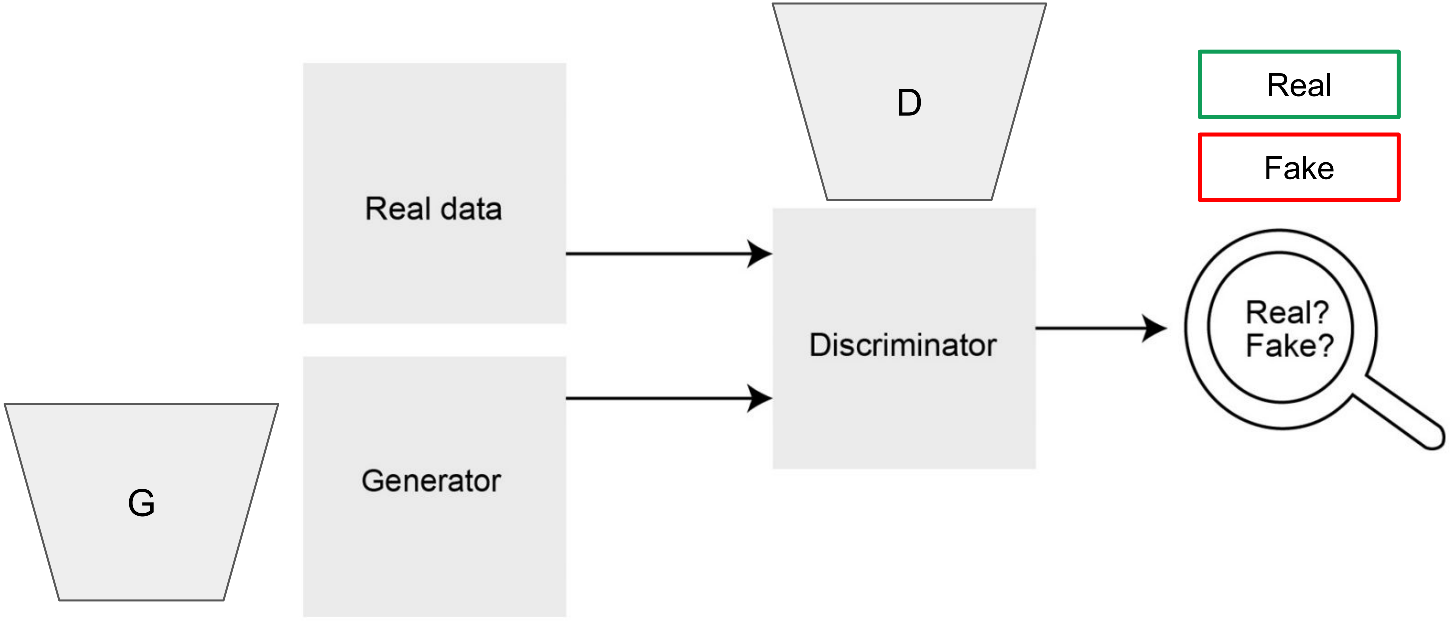





```python
class OurFirstSRNet(nn.Module):
    def __init__(self):
        super(OurFirstSRNet, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=8, stride=2, padding=3),
            nn.ReLU(inplace=True),
            nn.Conv2d(64, 192, kernel_size=4, stride=2, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(192, 256, kernel_size=4, stride=2, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True)
        )
        self.upsample = nn.Sequential(
            nn.ConvTranspose2d(256,256,kernel_size=2, stride=2, padding=0),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.ConvTranspose2d(256,192,kernel_size=2, stride=2, padding=0),
            nn.ReLU(inplace=True),
            nn.ConvTranspose2d(192,128,kernel_size=2, stride=2, padding=0),
            nn.ReLU(inplace=True),
            nn.Conv2d(128, 128, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.ConvTranspose2d(128,64,kernel_size=2, stride=2, padding=0),
            nn.ReLU(inplace=True),
            nn.ConvTranspose2d(64,3, kernel_size=4, stride=2, padding=1),
            nn.ReLU(inplace=True)
        )
    def forward(self, x):
        x = self.features(x)
        x = self.upsample(x)
        return x

# test image transform
```

Introduction of GAN (Generative Adversarial Network)



GAN Structure in Pytorch

A. Initialize generator and discriminator
B. Initialize gradient descent optimizer
C. Iterate over epochs and batches
D. Only enable gradient on discriminator
E. Call discriminator model, calculate loss for real image, do backpropagation
F. Call generator, call discriminator, calculate loss for fake generated image, do backpropagation
G. Only enable gradient on generator
H. Call generator, call discriminator, calculate loss for forged generator and discriminator, calculate gradient of generator
I. Calculate gradient, then do backpropagation

A
```
generator = Generator()
discriminator = Discriminator()
```

B
```
# Set up separate optimizers for each network
generator_optimizer = ...
discriminator_optimizer = ...
```

```
def gan_train():
```

C
```
  for epoch in num_epochs:
    for batch in real_train_loader:
```

D
```
      discriminator.train()
      generator.eval()
      discriminator.zero_grad()
```

E
```
      preds = discriminator(batch)
      real_loss = criterion(preds, torch.ones_like(preds))
      discriminator.backward()
```

F
```
      fake_batch = generator(torch.rand(batch.shape))
      fake_preds = discriminator(fake_batch)
      fake_loss = criterion(fake_preds, torch.zeros_like(fake_preds))
      discriminator.backward()

      discriminator_optimizer.step()
```

G
```
      discriminator.eval()
      generator.train()
      generator.zero_grad()
```

H
```
      forged_batch = generator(torch.rand(batch.shape))
      forged_preds = discriminator(forged_batch)
      forged_loss = criterion(forged_preds, torch.ones_like(forged_preds))
```

I
```
      generator.backward()
      generator_optimizer.step()
```

Common Issue with GAN

Issue: Mode Collapse. If the discriminator may decide that anything that looks like the first type is actually fake, even the real example itself, and the generator then starts to generate something that looks like other types.

Solution: Add similarity score to the generated data, so that potential collapse can be detected and averted, keeping a replay buffer of generated images around so that the discriminator doesn’t overfit onto just the most current batch of generated images

ESRGAN

baboon_rlt.png

comic_rlt.png

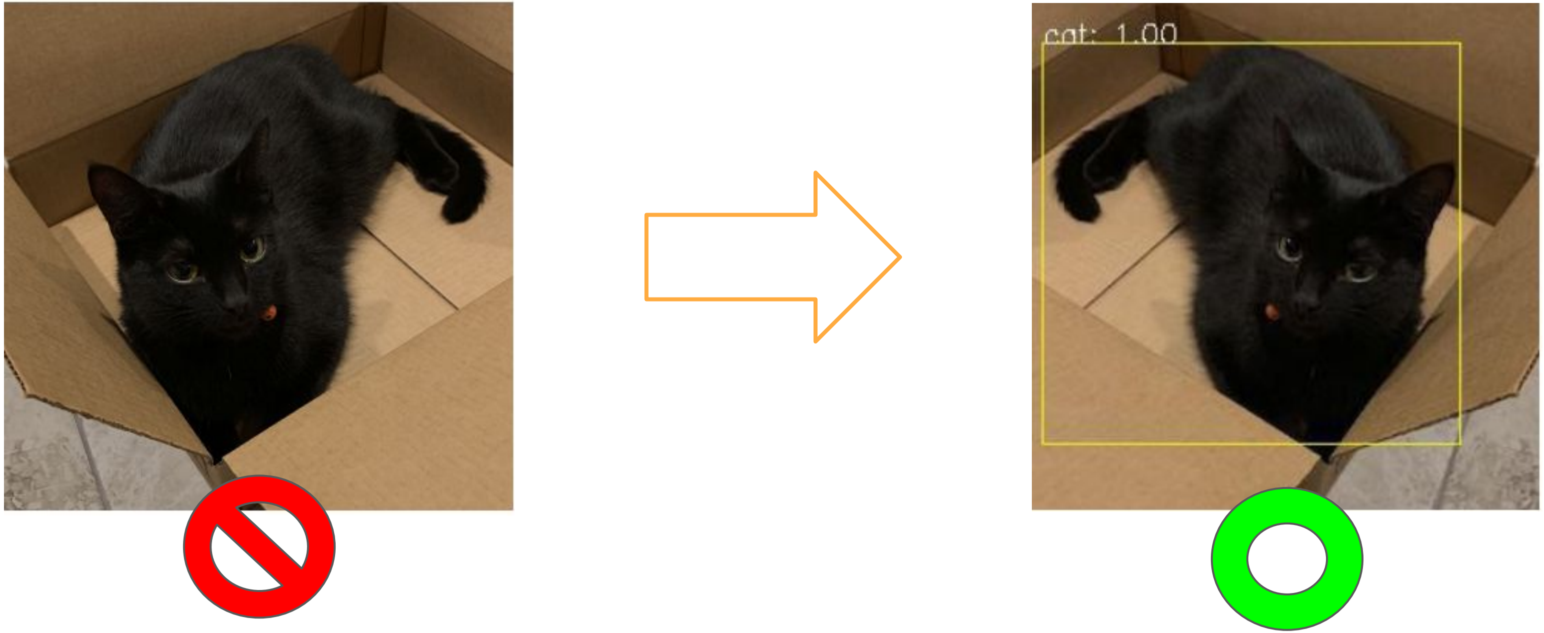
group_rlt.png

Outline

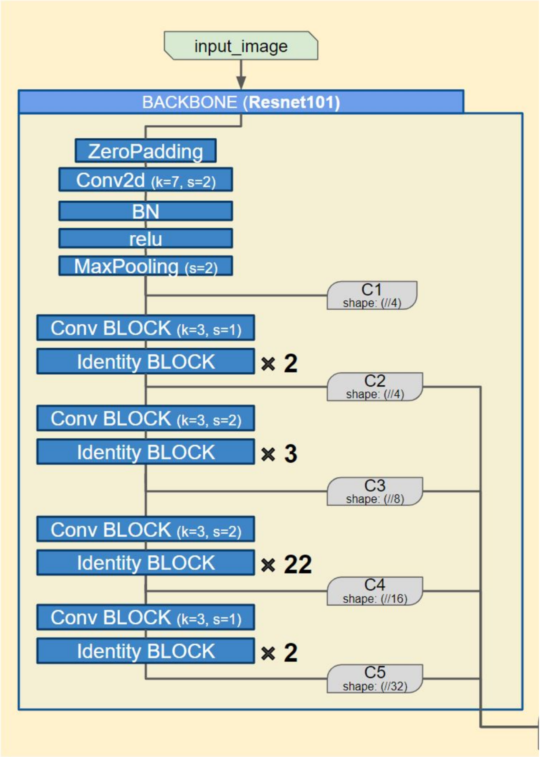
1. Data Augmentation
2. Super Resolution and GAN
3. Review of Object Detection
4. Adversarial Samples

Object Detection (OD)

Goal



OD - Faster RCNN and Mask-RCNN



Resnet101 ref: yaoying

https://github.com/facebookresearch/maskrcnn-benchmark

import

config

predict

```python
import os
try:
    print("...Check your packages: pip install apex yacs cython tqdm...")
    os.system('pip install apex yacs cython tqdm')
except:
    print("[ERROR LOG] There is some error in installing your packages")
import sys
sys.path.insert(0, './')
import matplotlib.pyplot as plt
from PIL import Image
import numpy as np
import sys
from maskrcnn_benchmark.config import cfg
from predictor import COCODemo
from time import time
config_file = "configs/caffe2/e2e_faster_rcnn_R_101_FPN_1x_caffe2.yaml"
cfg.merge_from_file(config_file)
cfg.merge_from_list(["MODEL.DEVICE", "cpu"])

coco_demo = COCODemo(
    cfg,
    min_image_size=500,
    confidence_threshold=0.7,
)
pil_image = Image.open(sys.argv[1])
image = np.array(pil_image)[:, :, [2, 1, 0]]

start_time = time()
predictions = coco_demo.run_on_opencv_image(image)
print('...runtime: {:.2f}s'.format(time()-start_time))
predictions = predictions[:,:,::-1]

save_img_path = sys.argv[1].split('.')
save_img_path[0] + '_out.'
save_img_path = ''.join(save_img_path)
plt.imsave(save_img_path, predictions)
```

Outline

1. Data Augmentation
2. Super Resolution and GAN
3. Review of Object Detection
4. Adversarial Samples

Adversarial Samples

- White-box Attack
- Black-box attack
- Defending Against Adversarial Attacks

Adversarial Samples

Goal

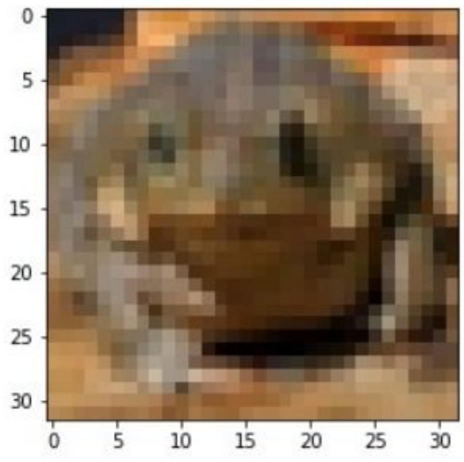


Figure 9-9. Our frog example

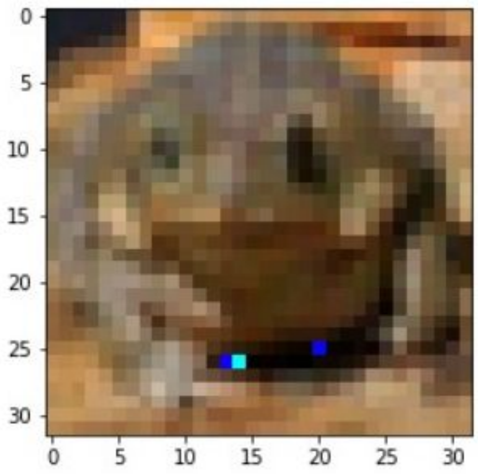


Figure 9-10. Our adversarial frog

```
model_to_break(adversarial_image.unsqueeze(-1))
# look up in labels via argmax()
>> 'cat'
```

White-Box Attack

Steps:

- Do forward pass
- Calculate loss
- Calculate gradient
- Maximize loss with torch.sign()
- Multiple it with Epsilon (ex. 1e-2 on 5 classes and 7e-1 on 2 classes)
- Do addition math with the original image
- Do forward pass, and predict output

This approach is called as Fast Gradient Sign Method (FGSM)

White-box Attack

With two classes, it's hard to fool the model

```
PS D:\pytorch_tutorial> python mixup_segmentation.py detecting adversarial
...You pick mode detecting...
Try to predict input label:  cat 19
(without adversarial) Predict acc : [0.57569957 0.4243004 ], Highest accuracy class: cat
...You pick mode detecting adversarial...
Mean fgsm:  -3.919536e-05
Mean perturbed_image:  0.30877587
(with adversarial) Predict acc: [0.4290714 0.5709286], Highest accuracy class: dog
```

With more classes, it's easier to fool one

```
PS D:\pytorch_tutorial> python mixup_segmentation.py detecting adversarial
...You pick mode detecting...
Try to predict input label:  cat 19
(without adversarial) Predict acc : [0.5620013  0.34930083 0.03456338 0.01864267 0.03549182], Highest accuracy class: cat
...You pick mode detecting adversarial...
Mean fgsm:  0.0001269531
Mean perturbed_image:  0.3016626
(with adversarial) Predict acc: [0.03858734 0.03947267 0.82507455 0.0178773  0.07898819], Highest accuracy class: fish
```

White-box Attacks

Minor const: It becomes harder to fool the model with fewer classes

Main const: Need to know a lot about the structure of the model to know what's going on and exploit the model

Then, what about if don't have any info about the model, loss etc?

Black-box attack

Well, if there is a will, there is a way...

Steps:

- Input
- Output (from the model without knowing the structure), set as Labels
- Train a new model with these two informations
- Do FGSM as in the white-box attack

Defending Against Adversarial Attacks

Distilling a model by using it to train another model seems to help.

Using label smoothing with the new model, as outlined earlier in this chapter, also seems to help.

Making the model less sure of its decisions appears to smooth out the gradients somewhat (mixup augmentation).

Have a filter that allows in only images that pass some filtering tests. You could in theory make a neural net to do that too, because then the attackers have to try to break two different models with the same image!

Implementation

Nothing...

Ooopss

Just kidding...

How to Run...

How to use it (mixup_segmentation.py):
- Go to the main folder (pytorch_tutorial folder)
- open cmd in that folder (or you could win+R -> cmd -> go to this folder)
- Type python mixup_segmentation.py {training or mixup_augmentation} #without bracket {} -> for mixup segmentation
- Type python mixup_segmentation.py training (label_smooth) -> for label smoothing
- Type python mixup_segmentation.py plot_training -> for plotting training output (with or without label_smooth)
- Type python mixup_segmentation.py detecting (adversarial) -> for using adversarial

How to use it (mixup_segmentation.py):
- Go to the main folder (pytorch_tutorial folder)
- open cmd in that folder (or you could win+R -> cmd -> go to this folder)
- Type python sr_tutorial.py training/detecting

Terima Kasih
Thank you
谢谢
cảm ơn bạn
Gracias