Numpy → To handle arrays & matrices.

plt → visualisation tool.

glob → To return all files path that matches a specific pattern.

PIL → Imaging Library.

CV2 → To display an image in window.

tqdm → For creating progress meters.

Similar to other NN, we normalize GAN inputs

1) Improve convergence of nets

2) giving an equal range of value to all features so not to make some features dominate others because of wide range of value.

One more reason is that in this we use tanh function which maps to [-1, 1], thus we are trying to scale our image.

def ~~read_path~~ PATH (~~filepath~~ folder): → folder name i.e. val or train. it returns all files in a particular folder with ~~some~~ .jpg.

class Transform (): → returns a transformed image.

↳ Chained together by Compose.

Train = Dataloader (train_l, bs=16, s=True).

Dataset (train).

files → separated → transformed & returned.

Generator → nn.Module → Pytorch.

we need block

specified in paper.

def gblock(self, input, output, kernel_size = 3 , pool_size = None)

→ 1st → nn.Conv2d (in_c , out_c, ks, padding = (ks-1)//2).

→ 2nd → nn.LeakyRelu (0.2, in_place = True)

→ nn.BatchNorm2d (out_c)

→ nn.RELU ( ).

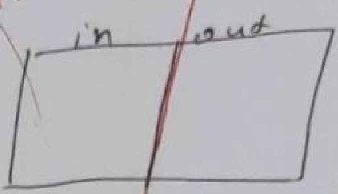
if we are using pool_size, then we have to do average pooling.

nn.AvgPool2d (size)

They accept both PIL & tensor images, thus one need to specify if it is tensor only.

Class Dataset (Object): Object → Base Class Name

① def __init__(self, files) → files were loaded & transformed.
self.files = files ↳ function was created.



② def _seperate (self, img) → ↳ function was created.

return image in img[:, w:, :], img[:, :w, :] → bisected into two.

def __getitem__ (self, idx):
→ image were loaded using ①
→ image was sepanarated using ②
→ input image was transformed using 1
→ output " " " " "
returns in, out.

def __len__ (self):
returns len (self.files) → use to find length of the instance attributes.
↳ total no. of image in folder or glob.

def show_img_sample (— , —):
→ figure was created → 1R & 2C
→ axes were flatten.
→ 1st image was shown using im show → permute(1,2,0)

we are getting torch.size ([3, 5, 2]) → permute [2, 0, 1]
0 1 2

3, 64, 64, thus by doing .permute(1,2,0) = torch size [(2, 3, 5)]

→ 64X64X3 → to display we need w X h X channels.

Batch RGB h w

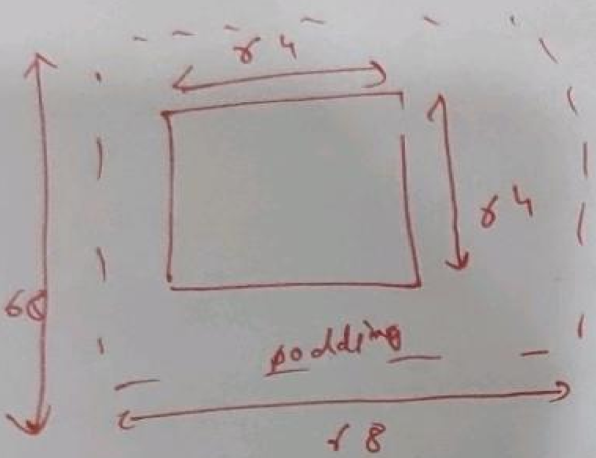
(16, 3, 64, 64)

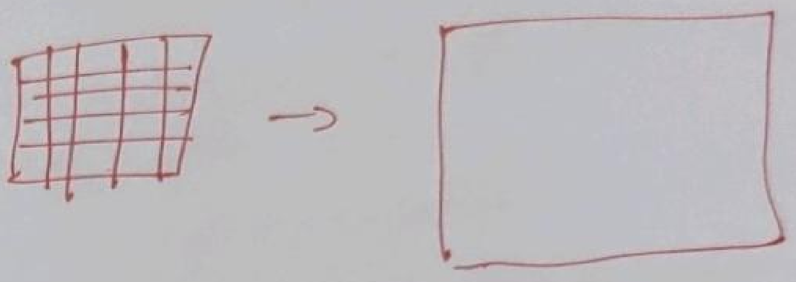
Generator

encoder-1

→ Conv 2d (3, 32, 5, 2]

in_c out_c kernel padding.





$$\frac{64 + 4 - 5}{\text{Stride}=1} + 1$$

(16, 3, 64, 64) → ⇒ 64 i.e.

(16, 32, 64, 64)

→ Leaky ReLU

→ x $x \geq 0$

→ negative slope × x $x < 0$

To overcome zero-gradient problem in backpropagation

→ Batch Norm

- for faster training
- → Allows use of higher learning rate
- → Makes weights easier to initialize
- → Makes more activation functions viable
- → Simplifies the creation of deeper networks
- → Provides a bit of regularization

Gradient requires small lr, with deep networks gradients get smaller in back propagation, thus force use of higher lr.

We need to be careful while using ReLU, as they die out, using BN regulates the values going into each activation functions, & Non-linearities.

Batch Norm → bringing data to common scale without distorting its shape.

input

(16, 32, 64, 64)
bs c h w.

Encoder - 2

~~avgPool (32, 64, 3, 1)~~ avgPool(4 , 4)
KS , stride

it calculates the average value from patches of a feature map, thus creating a down shaped feature map.

Purpose :- To add small amount of translation invariance.

Max pool → Extract more pronounced features such as edges.

where as avg pool extracts feature more smoothly.

(16, 32, 64, 64) → 16, 32, 16, 16.

$$h_{out} = \frac{64 - 4}{4} + 1 = 16$$

Conv 2D (32, 64, kernel-size = 3, padding = 1)

$$h_{out} = \frac{16 - 3 + 2 \times 1}{1} + 1 = 16$$

↳ (16, 64, 16, 16).

→ LeakyReLU
→ Batch Norm
→ ReLU

Encoder - 3

$\swarrow (16, 64, 16, 16)$

- Avg pool (KS = 2, stride = 2, padding = 0)

$$\frac{16-2}{2} + 1 = \underline{8} \hookrightarrow (16, 64, 8, 8)$$

→ Conv2d (64, 128, KS = 3, stride = 1, padding = 1)

$$\rightarrow \frac{8-3+2}{1} + 1 = 8$$

→ Leaky ReLU
→ Batch Norm
→ ReLU.

$\hookrightarrow (16, 128, 8, 8)$

Encoder → 4

Avg Pool. (KS = 2, stride = 2, padding = 0).

$$\frac{8-2}{2} + 1 = 4 \qquad \hookrightarrow \underline{(16, 128, 4, 4)}$$

Conv2d (128, 256, KS = 3, stride = 1, padding = 1)

$$\frac{4-3+2}{1} + 1 = 4$$

Leaky ReLU
Batch Norm
ReLU

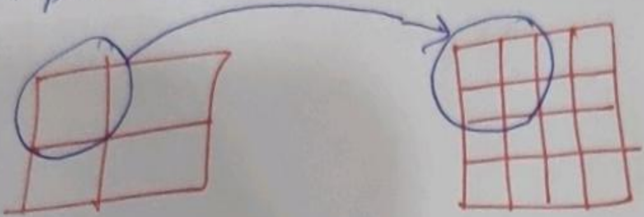
$\rightarrow \underline{(16, 256, 4, 4)}$

Decoder - 1

$(16, 256, 4, 4)$

Upsampling Nearest 2d (factor = 2)

Learned parameters will be copy-pasted. i.e. it will transfer coarse salient feature to a more dense & detailed output.



↳ $(16, 256, 8, 8)$

Conv2d (256, 128, ks = 3, stride = 1, padding = 1)

$$\frac{8-3+2}{1} + 1 = 8$$

↳ $(16, 128, 8, 8)$

BatchNorm + ReLU → $(16, 128, 8, 8)$

Decoder-2

→ $\underbrace{(16,128,8,8)}_{\text{Dec-1}} + \underbrace{(16,128,8,8)}_{\text{encoder-3}} \longrightarrow 16,256,8,8$

→ Upsampling (scaling = 2)

$(16, 256, \underbrace{8, 8}) \rightarrow (16, 256, 16, 16)$

→ Conv2d (256, 64, KS = 3, st = 1, p = 1)

$(16, 256, 16, 16) \rightarrow (16, 64, 16, 16)$

→ BatchNorm 2d.

→ ReLU.

Decoder 3

$\underbrace{(16, 64, 16, 16)}_{\text{Dc-2}} + \underbrace{(16, 64, 16, 16)}_{\text{enc-2}} \rightarrow (16, 128, 16, 16)$

→ ups caling (scaling = 4) → $(16, 128, \underbrace{64, 64})$ ↳ we reached 64, same need of upscaling.

→ Conv2d (128, 32, KS = 3, s = 1, p = 1)

↳ (16, 32, 64, 64)

Decoder-4 $\underbrace{16,32,64,64}_{\text{dec-3}} + \underbrace{16,32,64,64}_{\text{enc-1}} \rightarrow 16,64,64,64$

Conv 2d (64, 3, KS = 5, sh = 1, padding = 2)

↳ $\frac{64 - 5 + 4}{1} + 1 = 64$ → (16, 3, 64, 64)

tanh → to map as we trans based, so better mapping

Discriminator

it needs two images → for Generator propagation [fake & input]
for Discriminator propagation (~~fake~~ real & input)

input_img [ ] → segmented image [: :w, :]
real_img → " " [:, w: , :].
fake → generated by G on input

→ Two images are combined.
(16, 3, 64, 64) + (16, 3, 64, 64) → 16, 6, 64, 64.

layer-1

Conv2d (6, 16, ks = 5, s = 1, p = 2). → (16, 16, 64, 64)
+ Batch Norm
+ Leaky Relu

↳ $\frac{64-5+4}{1} + 1 = 64$

layer-2

Avg Pool. (Ks = 4, s = 4, p = 0) → (16, 16, 16, 16)

$\frac{64-4}{4} + 1 = 16$

Conv2d (16, 32, ks = 3, s = 1, p = 1) → (16, 32, 16, 16)
+ Batch Norm (32)
+ Leaky Relu.
+ Conv 2d (32, 32, ks = 3, s = 1, p = 1)
+ Batch Norm
+ Leaky Relu

↳ 16, 32, 16, 16

Layer-3

$\swarrow$ 16, 32, 16, 16

AvgPool (ks=2, s=2, p=0) (16, 32, 8, 8)

$\frac{16-2}{2}+1$

Conv (32, 64, Ks=3, s=1, p=1)

BatchNorm2d

LR

Conv2d (64, 64, ks=3, s=1, p=1)

BN

LR

$\longrightarrow$ (16, 64, 8, 8)

$\swarrow$

Layer-4

AvgPool2d (ks=2, s=2, p=0) $\rightarrow$ (16, 64, 4, 4)

$\frac{8-2}{2}+1 = 4$

Conv (64, 128) + BN + LR + Conv(128, 128) + BN + LR

$\hookrightarrow$ (16, 128, 4, 4)

$\swarrow$

Layer-5

AvgPool (ks=2, s=2, p=0) $\rightarrow$ (16, 128, 2, 2)

$\frac{4-2}{2}+1 = 2$

Conv (128, 256) + BN + LR + Conv (256, 256) + BN + LR

$\hookrightarrow$ (16, 256, 2, 2)

Layer-6

Conv (256, 1, Ks=1, S=1) $\rightarrow$ (16, 1, 2, 2)

to compare we need this only.

Generator training

$$\min_G \max_D V(G,D) = E_{x \sim p_{data}}[\ln(D(x))] + E_{z \sim p_z}[\ln(1 - D(G(z)))]$$

$P(Y/X) = P(X,Y)/P(X) \rightarrow$ GAN

weights θ_g

$\boxed{G} \rightarrow G(z)$

noise. Generated image.

Generator. $\leftarrow$ $\begin{cases} p_{data} \rightarrow \text{original data} \\ p_z \rightarrow \text{noise.} \\ p_g \rightarrow \text{generator.} \end{cases}$

Above is similar to Binary Cross Entropy.

$$\mathcal{L} = -\sum y \ln \hat{y} + (1-y) \ln(1-\hat{y})$$

$y \rightarrow$ truth
$\hat{y} \rightarrow$ predicted

when $y = 1$ as input $\Rightarrow \hat{y} = D(x)$

$$\mathcal{L} = \ln D(x)$$

when $y = 0$ as input, $\hat{y} = D(G(z))$

$$\mathcal{L} = \ln[1 - D(G(z))]$$

$E \rightarrow$ expectation.
$\hookrightarrow$ avg value

$$E(\mathcal{L}) = \int p_{data}(x) \ln[D(x)]\, dx + \int p_z(z) \ln(1 - D(G(z)))\, dz$$

Training loop

fix learning of G.

Inner loop for D:

- m data samples from original & m from fake.
- update θ_d by gradient descent.

$$\frac{\partial}{\partial \theta_d} \frac{1}{m} \left[\ln[D(x)] + \ln[1 - D(G(z))] \right]$$

fix learning rate of D.

inner exit loop

take m fake datasample

update θ_g by grad. descent.

$$\frac{\partial}{\partial \theta_g} \frac{1}{m} \left[\ln(1 - D(G(z))) \right]$$

; no $\ln D(x)$ term because $\frac{\partial \ln(D(x))}{\partial \theta_g} = 0$

Thus for every k update in D, we are getting one update in G.

In pix to pix

final argument was

$$G^* = \arg\min_G \max_D \mathcal{L}_{cGAN}(G, D) + \lambda \mathcal{L}_{L1}(G)$$

L1 encourages less blurring.