How to make a U-Net GAN

1 General stuff

- We're not using BigGAN deep, so the channel multiplier is only 64. That's more than enough (I've heard success stories with a multiplier of 32).
- Independent of what GAN baseline you use, the following should hold true:
  - What used to be the default discriminator is now just your encoder.
  - The decoder should be an exact mirror image of the encoder (strictly speaking it doesn't have to be, but this is the simplest design principle).
  - You may skip the uppermost skip connection, i.e. do not concatenate the direct input to the encoder to the pre-final output of the decoder. It should work either way, but it feels wrong for me to use the input directly.
  - The final output of the decoder always has just one channel (but you can try more if you come up with some other GAN loss. For example, instead of using binary cross-entropy you could use normal cross-entropy, but then you need two output channels)
  - If you think of using attention blocks, only use them in the encoder (or leave it).
- In Fig. 1 you see an overview of the U-Net GAN architecture for resolution 128, based on BigGAN. For resolution 256, the architecture differs slightly. The differences can be seen by comparing the details between 128 and 256 specified in section 2.1.

2 A suggestion for the implementation

Here is a suggestion how to implement the U-Net discriminator architecture using the already existing helper functions in ajbrock's orgininal BigGAN repository.

2.1 Defining the architecture in the discriminator class

1. Clone the BigGAN PyTorch repository `https://github.com/ajbrock/BigGAN-PyTorch`
2. Go to BigGAN.py and change the discriminator class:

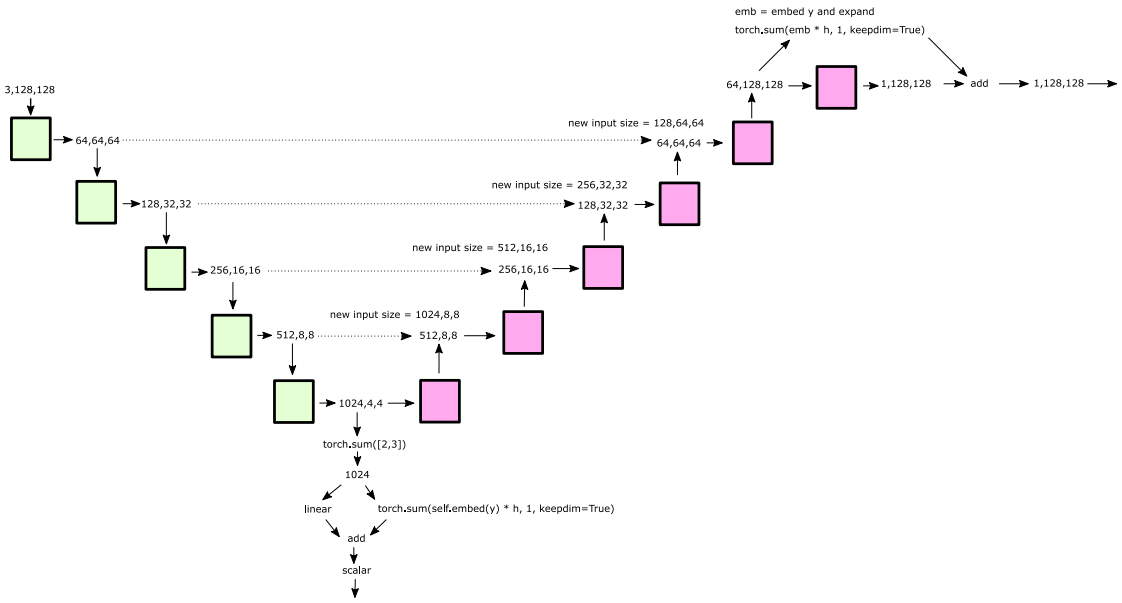

Figure 1: Overview of the 128x128 architecture. The forward pass results in two different outputs (encoder and decoder). To see how the loss is computed from these outputs see section 3.1

(a) First, let's change the architecture definition in `def D_arch`.

(b) `ch=64`, because we use the normal BigGAN.

(c) For resolution 128, the U-Net architecture can be summarized as follows
```
in_channels:  3,ch,2ch,4ch,8ch,16ch,16ch,2*4ch,2*2ch,2*ch,ch
out_channels:  ch, 2ch, 4ch, 8ch, 16ch, 8ch, 4ch, 2ch, ch, ch
downsample :  True*5 + False*5
upsample:  False*5+ True*5
resolution :  64, 32, 16, 8, 4, 8, 16, 32, 64, 128
```
Note that we don't use attention.

(d) For resolution 256 we have
```
in_channels:  3,ch, 2ch, 4ch, 8ch, 8ch, 16ch, 2,*8ch 2*8ch, 2*4ch,
2*2ch, 2*ch , ch
out_channels:  ch, 2ch, 4ch, 8ch, 8ch, 16ch, 8ch, 8ch, 4ch, 2ch, ch,
ch
downsample :  True*6 + False*6
upsample:  False*6+ True*6
resolution :  128, 64, 32, 16, 8, 4, 8, 16, 32, 64, 128, 256
```

3. Next, we go to `class Discriminator`.

4. Here, there is a loop that adds blocks to `self.blocks`. For that, it loops through the `arch` dict defined above. Modify it such that whenever `downsample==True` it adds `layers.DBlock(...)` to the list `self.blocks`, whereas if `upsample==True` it should add `layers.GBlock2(...)`. Double check that none of the blocks use batch normalization. You might want to remove the line that adds attention blocks.

5. Now you have all blocks of the discriminator neatly lined up in `self.blocks`, which the code automatically puts in a ModuleList so that all paramters are registered. You can now go to the forward function and do whatever you want in there.

2.2 The forward function

The original forward function converts an input image into a scalar that indicates real or fake. Make sure not to change that flow, but keep the intermediate feature outputs. You could conveniantly loop over `self.blocks` and save inermediate outputs in a list. Add the decoder part and concatenate the intermediate encoder features to the inputs of each decoder layer. You best follow the architecture in Fig. 1. If you are doing class conditional image generation, make sure to also to compute a class projection and add it to the output. For that you have to create an additional embedding function next to the existing one that is used for the encoder output `self.embed2 = self.which_embedding(self.n_classes,64)`. On the other side, if your GAN is unconditional, leave out the embedding pathway in Fig. 1.

3 Training

3.1 Loss function

How to implement the loss function? An example for the "real loss" in the discriminator step:
`L_real = binary_cross_entropy_with_logits(out, 1) +`
`binary_cross_entropy_with_logits(out2.view(-1), 1)`. Here, out is the encoder output and out2 is the decoder output of size [1,128,128]. When you do .view(-1) the BCE loss is computed for each "pixel" in that discriminator output and then averaged (That's just one way of implementing it and assumes that all pixel losses should have the same weight). In the D-step the target is 1 for real and 0 for fake. In the G-step the target is 1.

3.2 CutMix

(a) Take the function randbbox from here:
`https://github.com/clovaai/CutMix-PyTorch/blob/master/train.py`

(b) A CutMix mask can then be generated as outlined in the following pseudo code which uses the randbbox function from the previous bullet point:

```
def CutMix(image_size=128):
    alpha = random number from beta(1,1)
    bbx1, bby1, bbx2, bby2 = randbbox(image_size, alpha)
    # if you want to continue using alpha, adjust it to exactly match pixel ratio
    alpha = 1 - ((bbx2 - bbx1) * (bby2 - bby1) / (image_size * image_size))
    map = a bunch of ones of dimension (image_size, image_size)
    map[bbx1:bbx2,bby1:bby2]=0
    if some_random_numer > 0.5:
        map = 1 - map
    return map
```

(c) To get a batch of cutmix masks call the CutMix function via list comprehension for range(batch_size) and torch.stack them. Let's say you call this tensor "target_map".

(d) I suggest getting mixed images by `mixed = target_map*x+(1-target_map)*G_z`, where x are a tensor of real images and G_z your generated ones.

3.2.1 CutMix augmentation

Say you have used "mixed" as input to the discriminator. You get D_mixed as output. Now you got to calculate a loss, but what is your target? Is it 0, is it 1? Your target is the target_map:
`L_decoder = bce_with_logits(D_mixed.view(-1), target_map.view(-1))`.
Do you see it? For each pixel, we might have a 0 or 1 as the target, depending on how it was mixed because the CutMix target map consists of zeros and ones.
What about the encoder loss? Well, you could use either soft targets like in the original CutMix paper, or you set the target to 0 = fake. This is the save option because cut and mixed images are obviously fake. They don't occur in nature. If you use soft targets there is a chance your generator starts producing artificial CutMix images.

3.2.2 CutMix consistency regularization

CutMix consistency regularization is different from CutMix augmentation and is what really brings the most improvement. You could use CutMix consistency regularization without the CutMix augmentation. Combined with the augmentation, you get an edge in performance. To add the consistency loss, compute `mse(D_mixed, mixed_D)`, where mixed_D is `D_real*target_map + D_fake*(1-target_map)` and mse is the mean-squared-error loss.

4 Random tips and sonme things that might not be too clear from the paper

1. The consistency loss minimizes the distance between the logits of the decoder (Meaning, the bare output of D^U. There is no nonlinearity/sigmoid on the output of the discriminator forward function. Instead, it is applied in the binary cross entropy with logits function)

2. The very last decoder block `(N,64,128,128) to (N,1,128,128)` is not a resnet block but just a conv2d with kernel size 1.

3. If your implementation is unstable, leave out the CutMix augmentation, and just use the CutMix consistency loss. The fact that the mixed image has the ground truth "fake" can be confusing for the encoder loss. This was only observed for FFHQ, where the FID often jumps up around 100k steps and then goes down again.

4. When you only use the consistency loss, you can compute it on top of the normal batch, without warmup and without switching between mixed batches and uniformn batches. The strategy where you switch between mixed-only and normal batches was adopted for GPU memory reasons and because it is the default in a "CutMix for GANs" implementation found on github.

5. As mentioned in the point above, the CutMix augmentation loss can also be added for *every* batch. Here, just multiply the CutMix augmentation loss with the factor `min(1,current_epoch/warmup_epochs)`. Use the number of warmup epochs mentioned in the paper. It seems that this method also alleviates instabilities.

6. The method is more stable on some datasets than others. On CelebA it always works, for CoCo animals it works most of the time, for FFHQ it works 50 percent of the time. Note that the original BigGAN is not better in this regard either ;)

5 Additional questions

Are some things are not clear from the description? Do you think you might have found a typo? Do you have some additional questions regarding the implementation? Are you trying to apply U-Net GAN to a new problem and could use some tips? Then reach out to `edgarschoenfeld@life.de` or `edgar.schoenfeld@bosch.com`.