How to Use Masked Noise Encoder

December 10, 2022

1 General notes

In this repository, we only provide the core components of our masked noise encoder and the training pipeline, *without* the generator part. To complete the whole pipeline, please carefully go through the instruction in section 2, where StyleGAN2 generator is used as an example. We emphasize that our masked noise encoder is independent of the generator implementation.

Besides, here are several points might be helpful for the reproduction/development:

- Ablation studies of the masked noise encoder design is based on scene-centric datasets, e.g., Cityscapes. For object-centric datasets, the design choice, e.g., resolution, number of channels of the noise map could be different.
- We found that including GAN synthesized fake images with ground-truth w latents are quite helpful for speeding up the encoder training. And we only use them at the beginning of training.
- The noise map prediction plays a more important role for improving the reconstruction quality, compared to the discriminator etc.

2 Implementation suggestion for generator integration

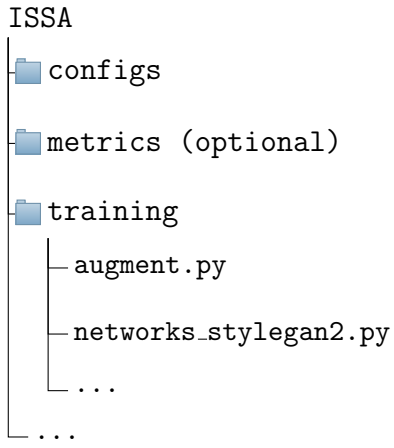
In this work, we propose the masked noise encoder for GAN inversion, where the GAN generator is frozen during the encoder training. Therefore, the generator can be chosen freely and not limited to a certain implementation. For example, several repositories of GAN implementation can be used, e.g., StyleGAN, lucidrains, rosinality.

In what follows, a walkthrough of how to apply the masked noise encoder for inverting a GAN generator is described in detail, where StyleGAN is considered as an example.

2.1 Preparation

The official StyleGAN repo includes many useful functions and classes, e.g., for logging, evaluation. Please check out their repository for a more compact training and evaluation pipeline.

To prepare the integration, please make sure the folders are structured as follow:



2.2 Generator modification

2.2.1 Add layer noise map as an additional input

1. In **SynthesisNetwork**

```
class SynthesisBlock(torch.nn.Module):
    ...
    def forward(self, ws, input_noise=None, layer_noise=None,
                noise_predict_from=None,
                noise_predict_until=None,
                **block_kwargs):
        # input_noise is optional, see the explaination below
        ...
        x = img = None
        i = 0
        if layer_noise is not None:
            layer_noise_id = -1

        for res, cur_ws in zip(self.block_resolutions, block_ws):
            block = getattr(self, f'b{res}')
            if i == 0:
                x, img = block(x, img, cur_ws,
                               input_noise=input_noise,
                               **block_kwargs)
            else:
                # ! Attention here !
                # pass layer_noise to the synthesis block
                if layer_noise is not None and \
                    res >= noise_predict_from and \
                    res <= noise_predict_until:
                    layer_noise_id += 1
                    x, img = block(x, img, cur_ws,layer_noise=layer_noise[
                                   layer_noise_id * 2: layer_noise_id * 2 + 2],
                                   **block_kwargs)
                else:
                    x, img = block(x, img, cur_ws, **block_kwargs)
            i += 1
        return img
```

2. In **SynthesisBlock**

```
class SynthesisBlock(torch.nn.Module):
    ...
    def forward(self, x, img, ws, input_noise=None, layer_noise=None,
        force_fp32=False, fused_modconv=None, update_emas=False, **layer_kwargs):
        ...
        # Main layers.
        if self.in_channels == 0:
            ...
        else:
            if layer_noise is None:
                x = self.conv0(x, next(w_iter), fused_modconv=fused_modconv,
                               **layer_kwargs)
                x = self.conv1(x, next(w_iter), fused_modconv=fused_modconv,
                               **layer_kwargs)
            else:
                # ! New !
                x = self.conv0(x, next(w_iter), layer_noise=layer_noise[0],
                               fused_modconv=fused_modconv, **layer_kwargs)
                x = self.conv1(x, next(w_iter), layer_noise=layer_noise[1],
                               fused_modconv=fused_modconv, **layer_kwargs)
```

3. In **SynthesisLayer**

```
class SynthesisLayer(torch.nn.Module):
    ...
    def forward(self, x, w, layer_noise=None,
                noise_mode='random', fused_modconv=True, gain=1):
        ...
        noise = None
        if layer_noise is not None:
            noise = layer_noise
        else:
            if self.use_noise and noise_mode == 'random':
                ...
            if self.use_noise and noise_mode == 'const':
                ...
        flip_weight = (self.up == 1)
```

2.2.2 Add image aspect ratio as an argument (optional)

Original StyleGAN2 implementation only supports training with square aspect ratio. Optionally, we can add a new argument *img_aspect_ratio* for the generator training. For instance, we can add modify the initialization function of the Generator class:

```
class Generator(torch.nn.Module):
    def __init__(self,
        z_dim,
        c_dim,
        w_dim,
        img_resolution,
        img_channels,
        img_aspect_ratio = 1.0, # Image aspect ratio, for non-square image.
        ...
        **synthesis_kwargs,
    ):
```

It should be noted that the discriminator and the shape assertion in all classes, e.g., in SynthesisBlock, should be changed accordingly as well.

2.2.3 Add an option to use random noise as input of the generator (optional)

We tried to replace the constant input of the generator with random noises. However, we do not observe obvious benefit of such adjustment. I would recommend *not* to change this at the beginning.

```
from torch import nn
from training.custom_layers import PixelNorm,EqualConvTranspose2d
...
class SynthesisBlock(torch.nn.Module):
    def __init__(self,
            ...
            input_mode= 'const', # Choose between 'const' and 'random'
            **layer_kwargs,
        ):
        self.input_mode = input_mode
        if in_channels == 0:
            if input_mode == 'const':
                self.const = torch.nn.Parameter(torch.randn(
                [out_channels, int(resolution/img_aspect_ratio), resolution]))
            elif input_mode == 'random':
                self.out_channels = out_channels
                self.input_mapping = nn.Sequential(EqualConvTranspose2d(out_channels,
    out_channels,kernel_size=(int(4/img_aspect_ratio),4),stride=1, padding=0),
                                        PixelNorm(),
                                        nn.LeakyReLU(0.2))
            else:
                raise ValueError('Input mode is invalid!')
        ...
    def forward(self, x, img, ws, input_noise=None, layer_noise=None,
        force_fp32=False, fused_modconv=None, update_emas=False, **layer_kwargs):
        ...
        # Input.
        if self.in_channels == 0:
            if self.input_mode == 'const':
                x = self.const.to(dtype=dtype, memory_format=memory_format)
                x = x.unsqueeze(0).repeat([ws.shape[0], 1, 1, 1])
            else:
                # ! New !
                if input_noise  is not None:
                    x = input_noise
                else:
                    x = torch.randn(ws.shape[0], self.out_channels).to(ws.device)
                x = self.input_mapping(x.view(ws.shape[0], self.out_channels,1,1))
        else:
            ...
```

3 Anything unclear?

If you have any further questions regarding the implementation or cannot reproduce satisfying results, don't hesitate to open a Github issue or contact me directly.

Feel free to reach out *liyumeng07@outlook.com* or *yumeng.li@de.bosch.com*.