Wensong Hu
EECS
University of Michigan

April 4th, 2024

HW5 — Generative Models

1 Section 1: Pix2Pix

1.1 Task 1: Dataloading

Code submitted to Canvas. Full notebook see appendix.

```
class Edges2Image(Dataset):
  def __init__(self, root_dir, split='train', transform=None):
    """
    Args:
        root_dir: the directory of the dataset
        split: "train" or "val"
        transform: pytorch transformations.
    """

    self.transform = transform

    self.files = glob.glob(os.path.join(root_dir, split, '*.jpg'))

  def __len__(self):
    return len(self.files)

  def __getitem__(self, idx):
    img = Image.open(self.files[idx])
    img = np.asarray(img)
    if self.transform:
        img = self.transform(img)
    return img

transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize(mean=(0.5, 0.5, 0.5), std=(0.5, 0.5, 0.5))
])

#
   ############################################################################

# TODO: Construct the dataloader
   #
# For the train_loader, please use a batch size of 4 and set shuffle True
   #
```

```
# For the val_loader, please use a batch size of 5 and set shuffle False    #
# Hint: You'll need to create instances of the class above, name them as    #
# tr_dt and te_dt. The dataloaders should be named as train_loader and      #
# test_loader. You also need to include transform in your class             #
#instances                                                                  #
#                                                                           ##############################################################################

tr_dt = Edges2Image(root_dir="mini-edges2shoes", split='train', transform=transform)
te_dt = Edges2Image(root_dir="mini-edges2shoes", split='val', transform=transform)

train_loader = DataLoader(tr_dt, batch_size=4, shuffle=True)
test_loader = DataLoader(te_dt, batch_size=5, shuffle=True)

#                                                                           ##############################################################################
#                                END OF YOUR CODE                           #
#                                                                           ##############################################################################

# Make sure that you have 1,000 training images and 100 testing images before moving on
print('Number of training images {}, number of testing images {}'.format(len(tr_dt), len(te_dt)))
```

1.2 Task 2: Training Pix2Pix

1.2.1 Codes

Code submitted to Canvas. Full notebook see appendix.

```
# Hint: you could use following loss to complete following function
BCE_loss = nn.BCELoss().cuda()
L1_loss = nn.L1Loss().cuda()

def train(G, D, num_epochs = 20):
  hist_D_losses = []
  hist_G_losses = []
  hist_G_L1_losses = []
  #                                                                         ##############################################################################
```

```
  # TODO: Add Adam optimizer to generator and discriminator
     #
  # You will use lr=0.0002, beta=0.5, beta2=0.999
     #
  #
   ############################################################################


  G_optimizer = optim.Adam(G.parameters(), lr=2e-4, betas=(0.5, 0.999))
  D_optimizer = optim.Adam(D.parameters(), lr=2e-4, betas=(0.5, 0.999))

  #
   ############################################################################

  #                                END OF YOUR CODE
     #
  #
   ############################################################################


  print('training start!')
  start_time = time.time()
  for epoch in range(num_epochs):
    print('Start training epoch %d' % (epoch + 1))
    D_losses = []
    G_losses = []
    epoch_start_time = time.time()
    num_iter = 0
    for x_ in train_loader:
      y_ = x_[:, :, :, img_size:]
      x_ = x_[:, :, :, 0:img_size]

      x_, y_ = x_.cuda(), y_.cuda()
      #
   ############################################################################

      # TODO: Implement training code for the discriminator.
         #
      # Recall that the loss is the mean of the loss for real images and
   fake    #
      # images, and made by some calculations with zeros and ones
         #
      # We have defined the BCE_loss, which you might would like to use.
         #
      #
         #
      # NOTE: While training the Discriminator, the output of the
   generator      #
      # must be detached from the computational graph. Refer to the method
         #
      # torch.Tensor.detach()
         #
      #
   ############################################################################
```

```
44
45      N = x_.shape[0]
46      # Generate data
47      fake_data = G.forward(x_).detach()
48
49      #1. Train the discriminator
50      # D real data BCE loss
51      D_real_preds = D.forward(torch.cat((x_, y_), dim=1))
52      D_y_real = torch.ones_like(D_real_preds)
53      # D_real_loss = torch.sum(torch.log(D_real_preds))
54      D_real_loss = BCE_loss(D_real_preds, D_y_real)
55
56      # D fake data BCE loss
57      D_fake_preds = D.forward(torch.cat((x_, fake_data), dim=1))
58      D_y_fake = torch.zeros_like(D_fake_preds)
59      # D_fake_loss = torch.sum(torch.log(1 - D_fake_preds))
60      D_fake_loss = BCE_loss(D_fake_preds, D_y_fake)
61
62      # D loss
63      loss_D = D_real_loss + D_fake_loss
64
65      # Train D
66      D_optimizer.zero_grad()
67      loss_D.backward()
68      D_optimizer.step()
69
70      #
    ##########################################################################

71      #                                 END OF YOUR CODE
           #
72      #
    ##########################################################################

73
74      #
    ##########################################################################

75      # TODO: Implement training code for the Generator.
           #
76      #
    ##########################################################################

77
78      # 1. Train the generator
79      # 2. Append the losses to the lists 'hist_G_L1_losses' and '
    hist_G_losses'
80      #  (Only append the data to the list, not the complete tensor, refer
81      #  torch.Tensor.item()).
82
83      # Generate data
84      fake_data = G.forward(x_)
85
```

```
      # 1. Train the generator
      # G BCE loss
      G_fake_preds = D.forward(torch.cat((x_, fake_data), dim=1))
      G_y_fake = torch.zeros_like(G_fake_preds)
      G_bce_loss = BCE_loss(G_fake_preds, G_y_fake)

      # G l1 loss
      G_l1_loss = L1_loss(fake_data, y_)

      # G loss
      lamb = 100
      loss_G = G_bce_loss +  lamb * G_l1_loss

      # Train G
      G_optimizer.zero_grad()
      loss_G.backward()
      G_optimizer.step()

      # 2. Append the losses to the lists 'hist_G_L1_losses' and '
    hist_D_losses'
      #  (Only append the data to the list, not the complete tensor, refer
      #  torch.Tensor.item()).
      hist_G_losses.append(G_bce_loss.detach().item())
      hist_G_L1_losses.append(G_l1_loss.detach().item())
      #
    ##########################################################################

      #                                 END OF YOUR CODE
         #
      #
    ##########################################################################


      D_losses.append(loss_D.detach().item())
      hist_D_losses.append(loss_D.detach().item())
      G_losses.append(loss_G)
      num_iter += 1



    epoch_end_time = time.time()
    per_epoch_ptime = epoch_end_time - epoch_start_time

    print('[%d/%d] - using time: %.2f seconds' % ((epoch + 1), num_epochs,
    per_epoch_ptime))
    print('loss of discriminator D: %.3f' % (torch.mean(torch.FloatTensor(
    D_losses))))
    print('loss of generator G: %.3f' % (torch.mean(torch.FloatTensor(
    G_losses))))
    if epoch == 0 or (epoch + 1) % 5 == 0:
      with torch.no_grad():
        show_result(G, fixed_x_, fixed_y_, (epoch+1))

  end_time = time.time()
```

```
131   total_ptime = end_time - start_time
132
133   return hist_D_losses, hist_G_losses, hist_G_L1_losses
```

1.2.2 Report Results

Figure 1: Result Visualization after 20 training epoches

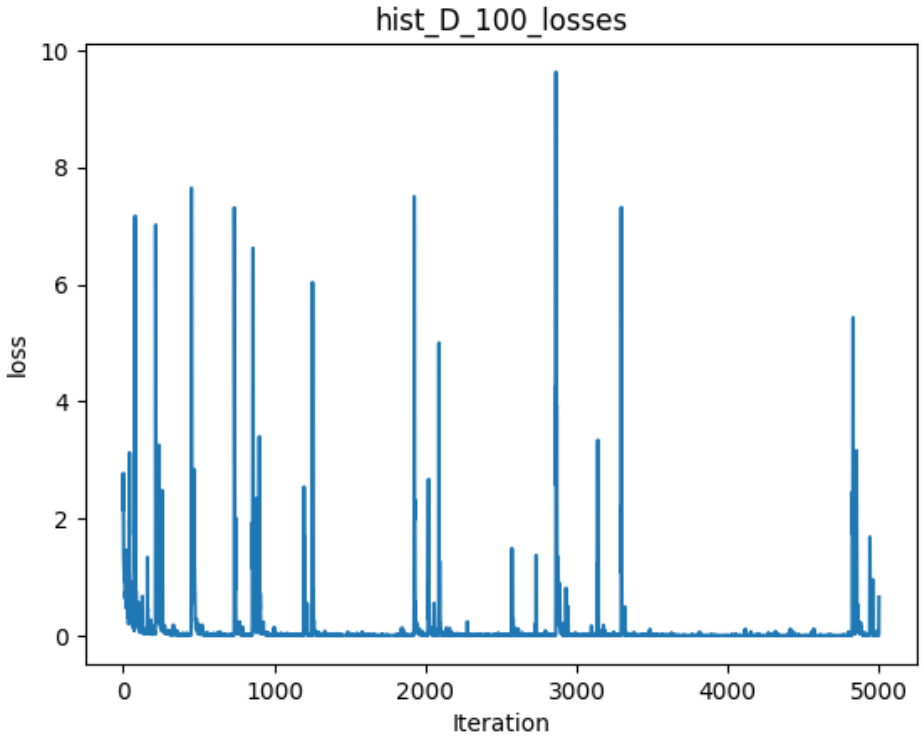


Figure 2: Discriminator BCE loss

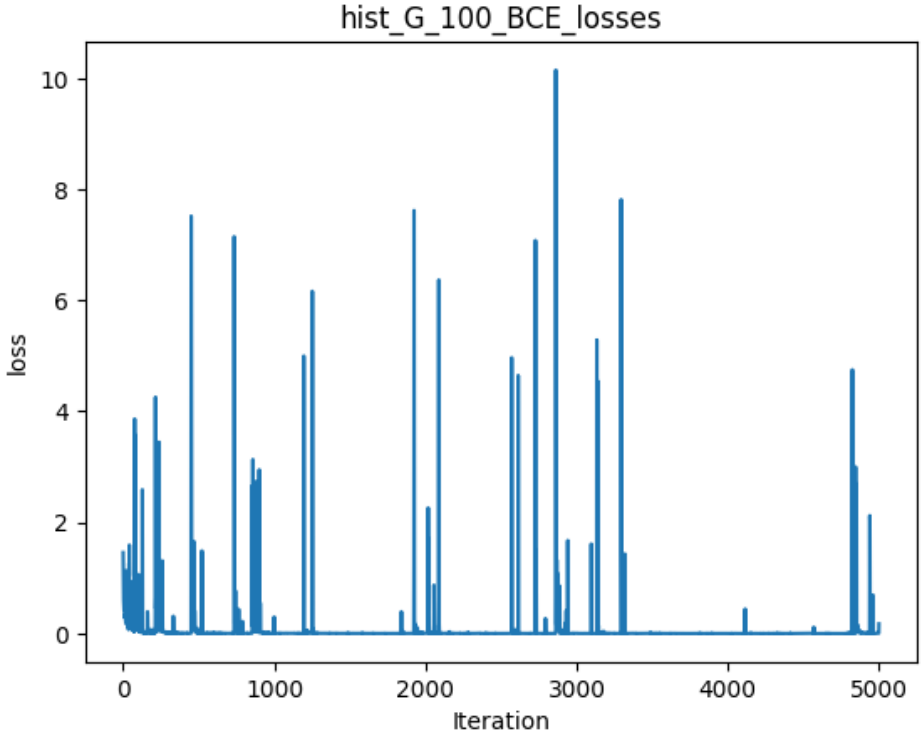


Figure 3: Generator BCE loss

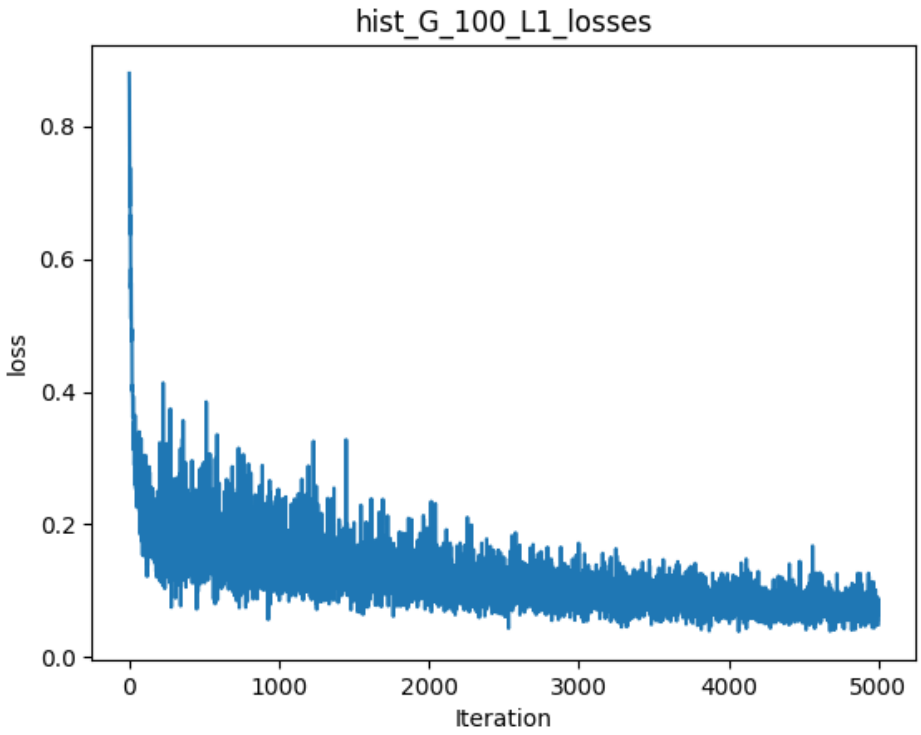


Figure 4: Generator L1 loss

2 Section 2: Diffusion Models

2.1 Task 3: Unconditional Sampling using DDPM: get_name_beta_schedule

Code submitted to Canvas.

```
def get_named_beta_schedule(schedule_name, num_diffusion_timesteps,
    beta_min=0.0001, beta_max=0.02):
    """
    Get a pre-defined beta schedule for the given name.

    Args:
        schedule_name: str, name of the variance schedule, 'linear' or '
    cosine'
        num_diffusion_timesteps: int, number of the entire diffusion
    timesteps
        beta_min: float, minimum value of beta
        beta_max: float, maximum value of beta

    Returns:
        betas: np.ndarray, a 1-d array of size num_diffusion_timesteps,
    contains all the beta for each timestep

    """
    betas = None
    if schedule_name == "linear":
        ########## START TODO ##########
```

```
        # Implement the linear schedule
        # Uniformly divide the [beta_min, beta_max) to
    num_diffusion_timesteps values.
        betas = np.linspace(beta_min, beta_max, num_diffusion_timesteps)
        ########## END TODO ##########


    elif schedule_name == "cosine":
        ########## START TODO ##########
        # Implement the cosine schedule
        # Assume s = 0.008 and beta_clip=0.999
        s = 0.008
        beta_clip = 0.999

        betas = np.zeros(num_diffusion_timesteps)
        f_0 = 0.0
        alpha_bar_tminus1 = 1.0
        for t in range(num_diffusion_timesteps):
            if t == 0:
                f_0 = (np.cos((0/num_diffusion_timesteps + s) / (1 + s) *
    np.pi / 2)) ** 2
                # alpha_bar_tminus1 = 1

            else:
                f_t = (np.cos((t/num_diffusion_timesteps + s) / (1 + s) *
    np.pi / 2)) ** 2
                alpha_bar_t = f_t / f_0
                # import pdb; pdb.set_trace()
                betas[t-1] = np.clip((1 - alpha_bar_t / alpha_bar_tminus1)
    , a_min=0, a_max=beta_clip)
                alpha_bar_tminus1 = alpha_bar_t
                # import pdb; pdb.set_trace()
        betas[-1] = beta_clip

        ########## End TODO ##########
    else:
        raise NotImplementedError(f"unknown beta schedule: {schedule_name}
    ")

    return betas
```

2.2 Task 4: Unconditional Sampling using DDPM: DDPM

2.2.1 Code

Code submitted to Canvas:

```
@register_sampler("ddpm")
class DDPMDiffusion:
    def __init__(self,
                 betas,
                 dynamic_threshold,
                 clip_denoised,
                 rescale_timesteps,
```

```
                **kwargs
                ):

        # use float64 for accuracy.
        betas = np.array(betas, dtype=np.float64)
        self.betas = betas
        assert self.betas.ndim == 1, "betas must be 1-D"
        assert (0 < self.betas).all() and (self.betas <=1).all(), "betas
    must be in (0..1]"

        self.num_timesteps = int(self.betas.shape[0])
        self.rescale_timesteps = rescale_timesteps

        ########### START TODO  ###########
        # Calculate the values of alpha
        # Also we will need the cumulated product of alpha.
        # And during sampling we need the value of cumulated product of
    alpha from
        # previous or next timestep.
        self.alphas = 1 - betas
        self.alphas_cumprod = np.cumprod(self.alphas)  # cumpulated
    product of alphas
        self.alphas_cumprod_prev = np.concatenate((np.array([1]), self.
    alphas_cumprod[:-1])) # first T-1 elements of alphas_cumprod, append
    1.0 at the begining, to make its length T
        self.alphas_cumprod_next = np.concatenate((self.alphas_cumprod
    [1:], np.array([0]))) # last T-1 elements of alphas_cumprod, append 0.0
     at the end, to make its length T

        ########### END TODO  ###########
        assert self.alphas_cumprod_prev.shape == (self.num_timesteps,)

        self.mean_processor = EpsilonXMeanProcessor(betas=betas,
                                                    dynamic_threshold=
    dynamic_threshold,
                                                    clip_denoised=
    clip_denoised)

        self.var_processor = LearnedRangeVarianceProcessor(betas=betas)

    def p_sample_loop(self,
                      model,
                      x_start,
                      record,
                      save_root,
                      measurement=None,
                      measurement_cond_fn=None,
                      uncond=False):
        """
        The function used for sampling from noise.

        Args:
            model: nn.Module, the pretrained model that is used to predict
     the score and variance
```

```
52            x_start: torch.Tensor, random noise input
53            measurement: torch.Tensor, our corrupted observation
54            measurement_cond_fn: conditional function used to perform
  conditional sampling, is None for unconditional sampling
55            record: Bool, save intermediate results if True
56            save_root: str, root of the directoy to save the results
57            uncond: Bool, perform unconditional sampling if True, else
  perform conditional sampling
58        """
59        if not uncond:
60            assert measurement is not None and measurement_cond_fn is not
  None, \
61                "measurement and measurement conditional function is
  required for conditional sampling"
62
63        img = x_start   # start from random noise
64        device = x_start.device
65
66        ############ Start TODO #############
67        # Implement the sample loop
68        # Call p_sample for every iteration
69        # It requires only one line of code implementation here
70
71        pbar = tqdm(list(range(self.num_timesteps))[::-1])
72        for idx in pbar:
73            time = torch.tensor([idx] * img.shape[0], device=device)
74
75            img = self.p_sample(model=model, x=img, t=time)['x_t_minus_1']
76
77            img = img.detach_()
78
79            if record:
80                if idx % 10 == 0:
81                    file_path = os.path.join(save_root, f"progress/x_{str(
  idx).zfill(4)}.png")
82                    plt.imsave(file_path, clear_color(img))
83
84        return img
85
86    def p_sample(self, model, x, t):
87        """
88        Posterior sampling process, when given the model, x_t and timestep
   t, it returns predicted
89        x_0 and x_t_minus_1
90
91        We have already provide you with the function to get the log of
  the variance.
92        Use self.var_processor.get_variance(var_values, t), where
  var_values is
93        the 3:6 channels of the direct output of the model.
94        example usage: log_variance = self.var_processor.get_variance(
  var_values, t)
95
```

```
        You can also use the helper function extract_and_expand() to extract the value
        corresponding to timestep and expand it to the save size as the target for broadcast.
        example usage: coef1 = extract_and_expand(self.posterior_mean_coef1, t, x_start)

        Args:
            model: nn.Module, the UNet model, you can call model(x, t) to get the output tensor with size (B, 6, H, W)
            x: torch.Tensor, shape (1, 3, H, W), x_t
            t: torch.Tenosr, shape (1,), timestep

        Returns:
            output_dict: dict, contains predicted x_t_minus_1 and x_0
        """
        #####Start TODO#####
        ##### Get the predicted score and variance of the pretrained model #####
        model_output = model.forward(x, t)
        pred_noise = model_output[:, :3, :, :]
        var_values = model_output[:, 3:, :, :]
        ##### End TODO #####

        log_variance = self.var_processor.get_variance(var_values, t)   # get the log  of variance

        #####  Start TODO   #####
        ##### get predicted x_0 and x_t_minus_1  #####
        ##### don't forget to add noise for all the steps, except for the last one  #####
        if t > 1:
            z = torch.randn(x.shape, dtype=x.dtype, device=x.device)
        else:
            z = torch.zeros_like(x, device=x.device)
        # import pdb; pdb.set_trace()
        alpha = extract_and_expand(self.alphas, t, x)
        alpha_bar = extract_and_expand(self.alphas_cumprod, t, x)
        x_t_minus_1 = (1 / torch.sqrt(alpha)) * (x - ((1 - alpha) / torch.sqrt(1 - alpha_bar)) * pred_noise) + torch.sqrt(torch.exp(log_variance)) * z

        #####  End TODO   #####

        assert x_t_minus_1.shape == log_variance.shape == x.shape

        output_dict = {'x_t_minus_1': x_t_minus_1}
        return output_dict
```

2.2.2 Result

Figure 5: DDPM Sampling Result

2.3 Task 5: Unconditional Sampling using DDIM

2.4 Code

Code submitted to Canvas.

```
@register_sampler("ddim")
class DDIMDiffusion(DDPMDiffusion):

    def __init__(self, use_timesteps, **kwargs):

        self.timestep_map = []
        self.original_num_steps = len(kwargs["betas"])

        base_alphas_cumprod = DDPMDiffusion(**kwargs).alphas_cumprod  # pylint: disable=missing-kwoa
        last_alpha_cumprod = 1.0
        new_betas = []
        self.use_timesteps = set(use_timesteps)

        for i, alpha_cumprod in enumerate(base_alphas_cumprod):
            if i in self.use_timesteps:
                new_betas.append(1 - alpha_cumprod / last_alpha_cumprod)
                last_alpha_cumprod = alpha_cumprod
                self.timestep_map.append(i)
        kwargs["betas"] = np.array(new_betas)
        super().__init__(**kwargs)


```

```
    def _scale_timesteps(self, t):
        if self.rescale_timesteps:
            return t.float() * (self.original_num_steps / self.
num_timesteps)
        return t

    def p_sample(self, model, x, t, eta=0.0):
        ##############################################
        #####   TODO    #####
        ##### Get the predicted score and variance of the pretrained model
     #####
        ##### Don't forget to use _scale_timesteps to scale the timestep
for calling the model prediction.
        ##### You don't need to scale the timestep for further
computations of x_t_minus_1.
        ##### NOTE: Since this version of the model learns the variance
along with the score function,
        ##### the output of the model would have double the number of
channels as that of the input.
        ##### So assign the predicted score and variance values to the
variables below. Refer to
        ##### torch.split method.
        ##############################################
        #####  Start TODO   #####
        model_output = model.forward(x, self._scale_timesteps(t))
        # import pdb; pdb.set_trace()
        pred_noise, var_values = torch.split(model_output, 3, dim=1)
        ##### End TODO #####

        model_mean, pred_xstart = self.mean_processor.get_mean_and_xstart(
x, t, pred_noise)
        log_variance = self.var_processor.get_variance(var_values, t)   #
get the log  of variance # This is not useful for DDIM, use equation
provided

        #####   TODO    #####
        ##### Step 1: Implement the variance parameter 'sigma' for DDIM
sampling.   #####
        ##### Step 2: Imeplemnt x_t_minus_1 using the pred_xstart. Don't
forget     #####
        #####  to add noise for all the steps, except for the t=0.
             #####
        #####
             #####
        ##### You may use the function 'extract_and_expand' to expand the
timestep  #####
        ##### variable 't' to the input's shape.
             #####
        ##### Assign them to the variables x_t_minus_1.
             #####

        #####  Start TODO   #####
        if t > 1:
            z = torch.randn(x.shape, device=x.device)
```

```
        else:
            z = torch.zeros_like(x)

        # alpha = extract_and_expand(self.alphas, t, x)
        alpha_bar = extract_and_expand(self.alphas_cumprod, t, x)
        alpha_bar_prev = extract_and_expand(self.alphas_cumprod_prev, t, x)

        eta = 1
        sigma = eta * torch.sqrt((1 - alpha_bar_prev) / (1 - alpha_bar)) * torch.sqrt(1 - (alpha_bar) / (alpha_bar_prev))

        x_t_minus_1 = torch.sqrt(alpha_bar_prev) * pred_xstart + torch.sqrt(1 - alpha_bar_prev - sigma ** 2) * pred_noise + sigma * z
        ##### End TODO #####

        return {"x_t_minus_1": x_t_minus_1, "pred_xstart": pred_xstart}
        ################################################


    def predict_eps_from_x_start(self, x_t, t, pred_xstart):
        coef1 = extract_and_expand(self.sqrt_recip_alphas_cumprod, t, x_t)
        coef2 = extract_and_expand(self.sqrt_recipm1_alphas_cumprod, t, x_t)
        return (coef1 * x_t - pred_xstart) / coef2
```

2.4.1 Result

Figure 6: DDIM Sampling Result

2.5 Task 6: Image Inpainting using RePaint

2.5.1 Code

Code submitted to Canvas

```
@register_sampler(name='repaint')
class Repaint(DDIMDiffusion):

    def undo(self, image_before_step, img_after_model, est_x_0, t, debug=False):
        return self._undo(img_after_model, t)


    def _undo(self, img_out, t):

        beta = extract_and_expand(self.betas, t, img_out)

        img_in_est = torch.sqrt(1 - beta) * img_out + \
            torch.sqrt(beta) * torch.randn_like(img_out)

        return img_in_est


    def p_sample(
        self,
        model,
        x_t_minus_one_unknown,
        t,
        clip_denoised=True,
        denoised_fn=None,
        model_kwargs=None,
        conf=None,
        pred_xstart=None,
    ):
        """
        Sample x_{t-1} from the model at the given timestep.

        :param model: the model to sample from.
        :param x_t_minus_one_unknown: the unknown tensor at x_{t-1} (model's predicted sample in the previous timestep).
        :param t: the value of t, starting at 0 for the first diffusion step.
        :param clip_denoised: if True, clip the x_start prediction to [-1, 1].
        :param denoised_fn: if not None, a function which applies to the
            x_start prediction before it is used to sample.
        :param model_kwargs: if not None, a dict of extra keyword arguments to
            pass to the model. This can be used for conditioning.
        :return: a dict containing the following keys:
                 - 'sample': a random sample from the model.
                 - 'pred_xstart': a prediction of x_0.
        """
        noise = torch.randn_like(x_t_minus_one_unknown)
```

```
45
46          ################################################
47          #####   TODO    #####
48          ##### Here updated sample x_t_minus_one refers to the noisy image,
     where the known region is    #####
49          ##### obtained by adding noise to GT, and the unknown region is
    obtained from                  #####
50          ##### x_t_minus_one_unknown (which is the predicted sample from
    previous timestep) and the      #####
51          #####  known and unknown region are combined using the ground
    truth mask (gt_keep_mask).        #####
52          ##### Compelete the implementation to compute the updated sample (
    x_t_minus_one) for the        #####
53          ##### timestep t. Make use of the variables gt_keep_mask and gt,
    to access the                   #####
54          ##### ground-truth image. and the ground-truth mask.
                                   #####
55          ################################################
56
57          if conf["inpa_inj_sched_prev"]:
58
59              if pred_xstart is not None:
60
61                  gt_keep_mask = model_kwargs['gt_keep_mask']
62                  if gt_keep_mask is None:
63                      gt_keep_mask = conf.get_inpa_mask(
    x_t_minus_one_unknown)
64
65                  gt = model_kwargs['gt']
66
67                  # Get x_t_minus_one_known
68                  if t > 1:
69                      epsilon = torch.randn(x_t_minus_one_unknown.shape,
    device=x_t_minus_one_unknown.device)
70                  else:
71                      epsilon = torch.zeros_like(x_t_minus_one_unknown,
    device=x_t_minus_one_unknown.device)
72                  alpha_bar = extract_and_expand(self.alphas_cumprod, t,
    x_t_minus_one_unknown)
73                  x_t_minus_one_known = torch.sqrt(alpha_bar) * gt + torch.
    sqrt(1 - alpha_bar) * epsilon
74
75                  # Get x_t_minus_one
76                  x_t_minus_one = gt_keep_mask * x_t_minus_one_known + (1 -
    gt_keep_mask) * x_t_minus_one_unknown
77
78
79              else:
80                  x_t_minus_one = x_t_minus_one_unknown
81
82          # TODO #####
83          # One-step denoising using the model: Perform a forward pass on
    the model.
```

```
84          # Remember to scale the timestep 't' using '_scale_timesteps'
    method.
85          # NOTE: Since this version of the model learns the variance along
    with the score function,
86          # the output of the model would have double the number of channels
     as that of the input.
87          # So assign the predicted score and variance values to the
    variables below. Refer to
88          # torch.split method.
89
90          model_output = model.forward(x_t_minus_one, self._scale_timesteps(
    t))
91          pred_score, var_values = torch.split(model_output, 3, dim=1)
92
93          model_mean, pred_xstart = self.mean_processor.get_mean_and_xstart(
    x_t_minus_one, t, pred_score)
94          log_variance = self.var_processor.get_variance(var_values, t)   #
    get the log  of variance
95
96          ################################################
97          ##### TODO
                ######
98          ##### Compute the noisy sample for timestep 't'
                ######
99          ##### You should use the 'log_variance' to calculate the variance
    of noise  #####
100         ##### to be added.
                ######
101         ##### Assign the sample to the variable 'sample'
                ######
102         ################################################
103
104         if t > 1:
105             z = torch.randn(x_t_minus_one_unknown.shape, device=
    x_t_minus_one_unknown.device)
106         else:
107             z = torch.zeros_like(x_t_minus_one_unknown, device=
    x_t_minus_one_unknown.device)
108
109         sample = model_mean + torch.sqrt(torch.exp(log_variance)) * z
110
111
112
113         result = {"sample": sample,
114                   "pred_xstart": pred_xstart, 'gt': model_kwargs.get('gt')
    }
115
116         return result
117
118
119     def p_sample_loop(
120         self,
121         model,
122         shape,
```

```
        noise=None,
        clip_denoised = True,
        denoised_fn=None,
        model_kwargs=None,
        device=None,
        progress=True,
        return_all=False,
        conf=None
    ):
        """
        Generate samples from the model.

        :param model: the model module.
        :param shape: the shape of the samples, (N, C, H, W).
        :param noise: if specified, the noise from the encoder to sample.
                      Should be of the same shape as `shape`.
        :param clip_denoised: if True, clip x_start predictions to [-1,
    1].
        :param denoised_fn: if not None, a function which applies to the
            x_start prediction before it is used to sample.
        :param cond_fn: if not None, this is a gradient function that acts
                        similarly to the model.
        :param model_kwargs: if not None, a dict of extra keyword
    arguments to
            pass to the model. This can be used for conditioning.
        :param device: if specified, the device to create the samples on.
                       If not specified, use a model parameter's device.
        :param progress: if True, show a tqdm progress bar.
        :return: a non-differentiable batch of samples.
        """
        final = None
        for sample in self.p_sample_loop_progressive(
            model,
            shape,
            noise=noise,
            clip_denoised=clip_denoised,
            denoised_fn=denoised_fn,
            model_kwargs=model_kwargs,
            device=device,
            progress=progress,
            conf=conf
        ):
            final = sample

        if return_all:
            return final
        else:
            return final["sample"]

    def p_sample_loop_progressive(
        self,
        model,
        shape,
        noise=None,
```

```
        clip_denoised=True,
        denoised_fn=None,
        model_kwargs=None,
        device=None,
        progress=False,
        conf=None
    ):
        """
        Generate samples from the model and yield intermediate samples from
        each timestep of diffusion.

        Arguments are the same as p_sample_loop().
        Returns a generator over dicts, where each dict is the return value of
        p_sample().
        """
        if device is None:
            device = next(model.parameters()).device
        assert isinstance(shape, (tuple, list))
        if noise is not None:
            image_after_step = noise
        else:
            image_after_step = torch.randn(*shape, device=device)

        self.gt_noises = None  # reset for next image


        pred_xstart = None

        idx_wall = -1
        sample_idxs = defaultdict(lambda: 0)

        if conf["schedule_jump_params"]:
            times = get_schedule_jump(**conf["schedule_jump_params"])
            time_pairs = list(zip(times[:-1], times[1:]))

            if progress:
                from tqdm.auto import tqdm
                time_pairs = tqdm(time_pairs)

            for t_last, t_cur in time_pairs:
                idx_wall += 1
                t_last_t = torch.tensor([t_last] * shape[0],
                                        device=device)

                if t_cur < t_last:  # reverse
                    with torch.no_grad():
                        image_before_step = image_after_step.clone()
                        out = self.p_sample(
                            model,
                            image_after_step,
                            t_last_t,
                            clip_denoised=clip_denoised,
```

```
                            denoised_fn=denoised_fn,
                            model_kwargs=model_kwargs,
                            conf=conf,
                            pred_xstart=pred_xstart
                        )
                        image_after_step = out["sample"]
                        pred_xstart = out["pred_xstart"]

                        sample_idxs[t_cur] += 1

                        yield out

                else:
                    t_shift = conf.get('inpa_inj_time_shift', 1)

                    image_before_step = image_after_step.clone()
                    image_after_step = self.undo(
                        image_before_step, image_after_step,
                        est_x_0=out['pred_xstart'], t=t_last_t+t_shift,
    debug=False)
                    pred_xstart = out["pred_xstart"]
```

2.5.2 Result

Figure 7: RePaint Inpainting Result

2.6 Task 7: Image Inpainting using Diffusion Posterior Sampling

2.6.1 Code

Code submitted to Canvas.

```python
@register_conditioning_method(name='ps')
class PosteriorSampling(ConditioningMethod):
    def __init__(self, operator, noiser, **kwargs):
        super().__init__(operator, noiser)
        self.scale = kwargs.get('scale', 1.0)

    def conditioning(self, x_i, x_t_minus_one, x_0_hat, measurement, **kwargs):
        """
        The conditioning function as shown in line 7

        Args:
            x_i: torch.Tensor, x_i
            x_t_minus_one, torch.Tensor, x_t_minus_1 prime
            x_0_hat: torch.Tensor, predicted x_0
            measurement: torch.Tensor, y, the corrputed image
        """
        # norm_grad, norm = self.grad_and_value(x_prev=x_prev, x_0_hat=x_0_hat, measurement=measurement, **kwargs)
        ############    Start TODO  ###########
        ###### Implentment the conditional sampling in line 7 #######
        ###### A(x_0_hat) is already provided to you as A ########
        ###### Also torch.autograd.grad() is provided to you to calculate the gredient of the
        ###### norm term with respect to x_i, you can check https://pytorch.org/docs/stable/generated/torch.autograd.grad.html#torch.autograd.grad
        ###### for its detailed usage. You only need to specify the outputs and inputs here.
        A = self.operator.forward(x_0_hat, **kwargs)
        new_x_t_minus_one = None
        difference = None
        norm = None
        diff_output = None  # outputs of the differentiated function
        diff_input = None   # Inputs w.r.t. which the gradient will be returned

        # My code
        difference = measurement - A
        norm = torch.norm(difference)
        diff_output = norm
        diff_input = x_i

        ## TODO: Don't delete this line, you will use this
        norm_grad = torch.autograd.grad(outputs=diff_output, inputs=diff_input)[0]

        new_x_t_minus_one = x_t_minus_one - self.scale * norm_grad


        ############    END TODO  ###########
        return new_x_t_minus_one
```

2.6.2 Result

Figure 8: DPS Inpainting Result

3 Appendix

Full Notebook pdf given in next page

EECS 442 PS5: Generative Adversarial Models

Please provide the following information (e.g. Jinfan Zhou, zjf): Wensong Hu, umhws

Starting

Run the following code to import the modules you'll need. After you finish the assignment, remember to run all cells and convert the notebook to a .pdf file for Gradescope submission.

In [1]:
```
!pip install torchsummary
import pickle
import numpy as np
import matplotlib.pyplot as plt
import os
import time
import itertools
from matplotlib import image
import glob as glob
from PIL import Image

import torch
import torchvision
from torchvision import datasets, models, transforms
import torch.nn as nn
import torch.optim as optim
from torch.autograd import Variable
import torch.nn.functional as F
from torch.utils.data import DataLoader, Dataset
from torchsummary import summary

print("PyTorch Version: ",torch.__version__)
print("Torchvision Version: ",torchvision.__version__)
# Detect if we have a GPU available
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
if torch.cuda.is_available():
  print("Using the GPU!")
else:
  print("WARNING: Could not find GPU! Using CPU only. If you want to enable
```

```
Requirement already satisfied: torchsummary in /home/umhws/anaconda3/envs/ee
cs442/lib/python3.10/site-packages (1.5.1)
PyTorch Version:  2.2.1+cu118
Torchvision Version:  0.17.1+cu118
Using the GPU!
```

Problem 6.1 pix2pix

You will build pix2pix for image translation.

In this question, you will need to:

1. Contruct dataloaders for train/test datasets
2. Build Generator and Discriminator
3. Train pix2pix and visualize the results during training
4. Plot the loss of generator/discriminator v.s. iteration
5. Design your own shoes (optional)

Step 0: Downloading the dataset.

We first download the `mini-edges2shoes` dataset sampled from the original `edges2shoes` dataset. The `mini-edges2shoes` dataset contains 1,000 training image pairs, and 100 testing image pairs.

There's nothing you need to implement for this part.

In [ ]:
```
# Download the mini-edges2shoes dataset
!rm -r mini-edges2shoes.zip
!rm -r mini-edges2shoes
!wget http://www.eecs.umich.edu/courses/eecs442-ahowens/mini-edges2shoes.zip
!unzip -q mini-edges2shoes.zip

!nvidia-smi
```

Step 1: Build dataloaders for train and test

We will first build dataloaders with PyTorch built-in classes. Here we build a Custom Dataset rather than a built-in dataset.

In [3]:
```
class Edges2Image(Dataset):
  def __init__(self, root_dir, split='train', transform=None):
    """
    Args:
        root_dir: the directory of the dataset
        split: "train" or "val"
        transform: pytorch transformations.
    """

    self.transform = transform
```

```
    self.files = glob.glob(os.path.join(root_dir, split, '*.jpg'))

  def __len__(self):
    return len(self.files)

  def __getitem__(self, idx):
    img = Image.open(self.files[idx])
    img = np.asarray(img)
    if self.transform:
        img = self.transform(img)
    return img

transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize(mean=(0.5, 0.5, 0.5), std=(0.5, 0.5, 0.5))
])

################################################################################
# TODO: Construct the dataloader                                               #
# For the train_loader, please use a batch size of 4 and set shuffle True #
# For the val_loader, please use a batch size of 5 and set shuffle False  #
# Hint: You'll need to create instances of the class above, name them as  #
# tr_dt and te_dt. The dataloaders should be named as train_loader and    #
# test_loader. You also need to include transform in your class           #
#instances                                                                     #
################################################################################

tr_dt = Edges2Image(root_dir="mini-edges2shoes", split='train', transform=tr
te_dt = Edges2Image(root_dir="mini-edges2shoes", split='val', transform=trar

train_loader = DataLoader(tr_dt, batch_size=4, shuffle=True)
test_loader = DataLoader(te_dt, batch_size=5, shuffle=True)

################################################################################
#                                END OF YOUR CODE                              #
################################################################################

# Make sure that you have 1,000 training images and 100 testing images befor
print('Number of training images {}, number of testing images {}'.format(ler
```

```
Number of training images 1000, number of testing images 100
```

In [4]:
```
#Sample Output used for visualization
test = test_loader.__iter__().__next__()
img_size = 256
fixed_y_ = test[:, :, :, img_size:].cuda()
fixed_x_ = test[:, :, :, 0:img_size].cuda()
print(len(train_loader))
print(len(test_loader))
print(fixed_y_.shape)

# plot sample image
fig, axes = plt.subplots(2, 2)
axes = np.reshape(axes, (4, ))
for i in range(4):
  example = train_loader.__iter__().__next__()[i].numpy().transpose((1, 2, 0
```

```
  mean = np.array([0.5, 0.5, 0.5])
  std = np.array([0.5, 0.5, 0.5])
  example = std * example + mean
  axes[i].imshow(example)
  axes[i].axis('off')
plt.show()
```

```
/home/umhws/anaconda3/envs/eecs442/lib/python3.10/site-packages/torchvision/
transforms/functional.py:153: UserWarning: The given NumPy array is not writ
able, and PyTorch does not support non-writable tensors. This means writing
to this tensor will result in undefined behavior. You may want to copy the a
rray to protect its data or make it writable before converting it to a tenso
r. This type of warning will be suppressed for the rest of this program. (Tr
iggered internally at ../torch/csrc/utils/tensor_numpy.cpp:206.)
  img = torch.from_numpy(pic.transpose((2, 0, 1))).contiguous()
250
20
torch.Size([5, 3, 256, 256])
```

Step 2: Build Generator and Discriminator

Based on the paper, the architectures of network are as following:

Generator architectures:

U-net encoder:

C64-C128-C256-C512-C512-C512-C512-C512

U-net decoder:

C512-C512-C512-C512-C256-C128-C64-C3

After the last layer in the decoder, a convolution is applied to map to the number of output channels, followed by a Tanh function. As an exception to the above notation, BatchNorm is not applied to the first C64 layer in the encoder. All ReLUs in the encoder are leaky, with slope 0.2, while ReLUs in the decoder are not leaky.

Discriminator architectures

The discriminator architecture is:

C64-C128-C256-C512

After the last layer, a convolution is applied to map to a 1-dimensional output, followed by a Sigmoid function. As an exception to the above notation, BatchNorm is not applied to the first C64 layer. All ReLUs are leaky, with slope 0.2.

We have already implemented the U-net architecture below.

In [5]:
```
def normal_init(m, mean, std):
  """
  Helper function. Initialize model parameter with given mean and std.
  """
  if isinstance(m, nn.ConvTranspose2d) or isinstance(m, nn.Conv2d):
    # delete start
    m.weight.data.normal_(mean, std)
    m.bias.data.zero_()
    # delete end
```

In [6]:
```
class generator(nn.Module):
  # initializers
  def __init__(self):
    super(generator, self).__init__()

    # Unet generator encoder
    self.conv1 = nn.Conv2d(3, 64, kernel_size=4, stride=2, padding=1)
    self.conv2 = nn.Conv2d(64, 128, kernel_size=4, stride=2, padding=1)
    self.bn2 = nn.BatchNorm2d(128)
    self.conv3 = nn.Conv2d(128, 256, kernel_size=4, stride=2, padding=1)
    self.bn3 = nn.BatchNorm2d(256)
    self.conv4 = nn.Conv2d(256, 512, kernel_size=4, stride=2, padding=1)
    self.bn4 = nn.BatchNorm2d(512)
    self.conv5 = nn.Conv2d(512, 512, kernel_size=4, stride=2, padding=1)
    self.bn5 = nn.BatchNorm2d(512)
    self.conv6 = nn.Conv2d(512, 512, kernel_size=4, stride=2, padding=1)
    self.bn6 = nn.BatchNorm2d(512)
    self.conv7 = nn.Conv2d(512, 512, kernel_size=4, stride=2, padding=1)
    self.bn7 = nn.BatchNorm2d(512)
    self.conv8 = nn.Conv2d(512, 512, kernel_size=4, stride=2, padding=1)

    # Unet generator decoder

    self.deconv1 = nn.ConvTranspose2d(512, 512, kernel_size=4, stride=2, pad
    self.dbn1 = nn.BatchNorm2d(512)
    self.deconv2 = nn.ConvTranspose2d(1024, 512, kernel_size=4, stride=2, pa
```

```
    self.dbn2 = nn.BatchNorm2d(512)
    self.deconv3 = nn.ConvTranspose2d(1024, 512, kernel_size=4, stride=2, pa
    self.dbn3 = nn.BatchNorm2d(512)
    self.deconv4 = nn.ConvTranspose2d(1024, 512, kernel_size=4, stride=2, pa
    self.dbn4 = nn.BatchNorm2d(512)
    self.deconv5 = nn.ConvTranspose2d(1024, 256, kernel_size=4, stride=2, pa
    self.dbn5 = nn.BatchNorm2d(256)
    self.deconv6 = nn.ConvTranspose2d(512, 128, kernel_size=4, stride=2, pad
    self.dbn6 = nn.BatchNorm2d(128)
    self.deconv7 = nn.ConvTranspose2d(256, 64, kernel_size=4, stride=2, padd
    self.dbn7 = nn.BatchNorm2d(64)
    self.deconv8 = nn.ConvTranspose2d(128, 3, kernel_size=4, stride=2, paddi

  # weight_init
  def weight_init(self, mean, std):
    for m in self._modules:
      normal_init(self._modules[m], mean, std)

  # forward method
  def forward(self, input):

    # encoding

    e1 = F.leaky_relu(self.conv1(input), negative_slope=0.2)
    e2 = F.leaky_relu(self.bn2(self.conv2(e1)), negative_slope=0.2)
    e3 = F.leaky_relu(self.bn3(self.conv3(e2)), negative_slope=0.2)
    e4 = F.leaky_relu(self.bn4(self.conv4(e3)), negative_slope=0.2)
    e5 = F.leaky_relu(self.bn5(self.conv5(e4)), negative_slope=0.2)
    e6 = F.leaky_relu(self.bn6(self.conv6(e5)), negative_slope=0.2)
    e7 = F.leaky_relu(self.bn7(self.conv7(e6)), negative_slope=0.2)
    e8 = F.leaky_relu(self.conv8(e7), negative_slope=0.2)

    # decoding

    d1 = F.relu(self.dbn1(self.deconv1(e8)))
    d1 = torch.cat([d1, e7], 1)
    d2 = F.relu(self.dbn2(self.deconv2(d1)))
    d2 = torch.cat([d2, e6], 1)
    d3 = F.relu(self.dbn3(self.deconv3(d2)))
    d3 = torch.cat([d3, e5], 1)
    d4 = F.relu(self.dbn4(self.deconv4(d3)))
    d4 = torch.cat([d4, e4], 1)
    d5 = F.relu(self.dbn5(self.deconv5(d4)))
    d5 = torch.cat([d5, e3], 1)
    d6 = F.relu(self.dbn6(self.deconv6(d5)))
    d6 = torch.cat([d6, e2], 1)
    d7 = F.relu(self.dbn7(self.deconv7(d6)))
    d7 = torch.cat([d7, e1], 1)
    d8 = F.tanh(self.deconv8(d7))

    output = d8

    return output

class discriminator(nn.Module):
  # initializers
```

```
  def __init__(self):
    super(discriminator, self).__init__()

    self.conv1 = nn.Conv2d(6, 64, kernel_size=4, stride=2, padding=1)
    self.conv2 = nn.Conv2d(64, 128, kernel_size=4, stride=2, padding=1)
    self.bn2 = nn.BatchNorm2d(128)
    self.conv3 = nn.Conv2d(128, 256, kernel_size=4, stride=2, padding=1)
    self.bn3 = nn.BatchNorm2d(256)
    self.conv4 = nn.Conv2d(256, 512, kernel_size=4, stride=1, padding=1)
    self.bn4 = nn.BatchNorm2d(512)
    self.conv5 = nn.Conv2d(512, 1, kernel_size=4, stride=1, padding=1)

  # weight_init
  def weight_init(self, mean, std):
    for m in self._modules:
      normal_init(self._modules[m], mean, std)

  # forward method
  def forward(self, input):

    x1 = F.leaky_relu(self.conv1(input))
    x2 = F.leaky_relu(self.bn2(self.conv2(x1)), negative_slope=0.2)
    x3 = F.leaky_relu(self.bn3(self.conv3(x2)), negative_slope=0.2)
    x4 = F.leaky_relu(self.bn4(self.conv4(x3)), negative_slope=0.2)
    x5 = F.sigmoid(self.conv5(x4))

    x = x5

    return x
```

In [7]:
```
# print out the model summary
G = generator().cuda()
D = discriminator().cuda()
summary(G, (3, 256, 256))
summary(D, (6, 256, 256))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 64, 128, 128]           3,136
            Conv2d-2          [-1, 128, 64, 64]         131,200
       BatchNorm2d-3          [-1, 128, 64, 64]             256
            Conv2d-4          [-1, 256, 32, 32]         524,544
       BatchNorm2d-5          [-1, 256, 32, 32]             512
            Conv2d-6          [-1, 512, 16, 16]       2,097,664
       BatchNorm2d-7          [-1, 512, 16, 16]           1,024
            Conv2d-8            [-1, 512, 8, 8]       4,194,816
       BatchNorm2d-9            [-1, 512, 8, 8]           1,024
           Conv2d-10            [-1, 512, 4, 4]       4,194,816
      BatchNorm2d-11            [-1, 512, 4, 4]           1,024
           Conv2d-12            [-1, 512, 2, 2]       4,194,816
      BatchNorm2d-13            [-1, 512, 2, 2]           1,024
           Conv2d-14            [-1, 512, 1, 1]       4,194,816
  ConvTranspose2d-15            [-1, 512, 2, 2]       4,194,816
      BatchNorm2d-16            [-1, 512, 2, 2]           1,024
  ConvTranspose2d-17            [-1, 512, 4, 4]       8,389,120
      BatchNorm2d-18            [-1, 512, 4, 4]           1,024
  ConvTranspose2d-19            [-1, 512, 8, 8]       8,389,120
      BatchNorm2d-20            [-1, 512, 8, 8]           1,024
  ConvTranspose2d-21          [-1, 512, 16, 16]       8,389,120
      BatchNorm2d-22          [-1, 512, 16, 16]           1,024
  ConvTranspose2d-23          [-1, 256, 32, 32]       4,194,560
      BatchNorm2d-24          [-1, 256, 32, 32]             512
  ConvTranspose2d-25          [-1, 128, 64, 64]       1,048,704
      BatchNorm2d-26          [-1, 128, 64, 64]             256
  ConvTranspose2d-27         [-1, 64, 128, 128]         262,208
      BatchNorm2d-28         [-1, 64, 128, 128]             128
  ConvTranspose2d-29           [-1, 3, 256, 256]           6,147
================================================================
Total params: 54,419,459
Trainable params: 54,419,459
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.75
Forward/backward pass size (MB): 54.82
Params size (MB): 207.59
Estimated Total Size (MB): 263.16
----------------------------------------------------------------
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 64, 128, 128]           6,208
            Conv2d-2          [-1, 128, 64, 64]         131,200
       BatchNorm2d-3          [-1, 128, 64, 64]             256
            Conv2d-4          [-1, 256, 32, 32]         524,544
       BatchNorm2d-5          [-1, 256, 32, 32]             512
            Conv2d-6          [-1, 512, 31, 31]       2,097,664
       BatchNorm2d-7          [-1, 512, 31, 31]           1,024
            Conv2d-8            [-1, 1, 30, 30]           8,193
================================================================
Total params: 2,769,601
Trainable params: 2,769,601
```

```
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 1.50
Forward/backward pass size (MB): 27.51
Params size (MB): 10.57
Estimated Total Size (MB): 39.58
----------------------------------------------------------------
```

In [8]:
```
D
```

Out[8]:
```
discriminator(
  (conv1): Conv2d(6, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
  (conv2): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1,
1))
  (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
  (conv3): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1,
1))
  (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
  (conv4): Conv2d(256, 512, kernel_size=(4, 4), stride=(1, 1), padding=(1,
1))
  (bn4): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
  (conv5): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), padding=(1,
1))
)
```

In [9]:
```
G
```

Out[9]:
```
generator(
  (conv1): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
  (conv2): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1,
1))
  (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
  (conv3): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1,
1))
  (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
  (conv4): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1,
1))
  (bn4): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
  (conv5): Conv2d(512, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1,
1))
  (bn5): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
  (conv6): Conv2d(512, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1,
1))
  (bn6): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
  (conv7): Conv2d(512, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1,
1))
  (bn7): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
  (conv8): Conv2d(512, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1,
1))
  (deconv1): ConvTranspose2d(512, 512, kernel_size=(4, 4), stride=(2, 2), p
adding=(1, 1))
  (dbn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_runn
ing_stats=True)
  (deconv2): ConvTranspose2d(1024, 512, kernel_size=(4, 4), stride=(2, 2),
padding=(1, 1))
  (dbn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_runn
ing_stats=True)
  (deconv3): ConvTranspose2d(1024, 512, kernel_size=(4, 4), stride=(2, 2),
padding=(1, 1))
  (dbn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_runn
ing_stats=True)
  (deconv4): ConvTranspose2d(1024, 512, kernel_size=(4, 4), stride=(2, 2),
padding=(1, 1))
  (dbn4): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_runn
ing_stats=True)
  (deconv5): ConvTranspose2d(1024, 256, kernel_size=(4, 4), stride=(2, 2),
padding=(1, 1))
  (dbn5): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runn
ing_stats=True)
  (deconv6): ConvTranspose2d(512, 128, kernel_size=(4, 4), stride=(2, 2), p
adding=(1, 1))
  (dbn6): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_runn
ing_stats=True)
  (deconv7): ConvTranspose2d(256, 64, kernel_size=(4, 4), stride=(2, 2), pa
dding=(1, 1))
  (dbn7): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
```

```
  (deconv8): ConvTranspose2d(128, 3, kernel_size=(4, 4), stride=(2, 2), pad
ding=(1, 1))
)
```

Step 3: Train

In this section, We will complete the training function. Then train our model. Below are some helper function that can be used. We will guide you through the implementation.

In [10]:
```
# Helper function for showing result.
def process_image(img):
  return (img.cpu().data.numpy().transpose(1, 2, 0) + 1) / 2

def show_result(G, x_, y_, num_epoch):
  predict_images = G(x_)

  fig, ax = plt.subplots(x_.size()[0], 3, figsize=(6,10))

  for i in range(x_.size()[0]):
    ax[i, 0].get_xaxis().set_visible(False)
    ax[i, 0].get_yaxis().set_visible(False)
    ax[i, 1].get_xaxis().set_visible(False)
    ax[i, 1].get_yaxis().set_visible(False)
    ax[i, 2].get_xaxis().set_visible(False)
    ax[i, 2].get_yaxis().set_visible(False)
    ax[i, 0].cla()
    ax[i, 0].imshow(process_image(x_[i]))
    ax[i, 1].cla()
    ax[i, 1].imshow(process_image(predict_images[i]))
    ax[i, 2].cla()
    ax[i, 2].imshow(process_image(y_[i]))

  plt.tight_layout()
  label_epoch = 'Epoch {0}'.format(num_epoch)
  fig.text(0.5, 0, label_epoch, ha='center')
  label_input = 'Input'
  fig.text(0.18, 1, label_input, ha='center')
  label_output = 'Output'
  fig.text(0.5, 1, label_output, ha='center')
  label_truth = 'Ground truth'
  fig.text(0.81, 1, label_truth, ha='center')

  plt.show()

# Helper function for counting number of trainable parameters.
def count_params(model):
  '''
  Counts the number of trainable parameters in PyTorch.
  Args:
      model: PyTorch model.
  Returns:
      num_params: int, number of trainable parameters.
  '''
```

```
  num_params = sum([item.numel() for item in model.parameters() if item.requ
  return num_params
```

Objective Functions: Conditional Pix2Pix GAN

Given a generator G and a discriminiator D, the loss function / objective functions to be minimized are given by

$ \mathcal{L}*{cGAN}(G, D) = \frac{1}{N} \left(: \sum*{i=1}^{N} log D(x_i, y_i)

- \sum_{i=1}^{N} log (1 - D:(:G:(x_i),: y_i) :) \right) $

where (x_i, y_i) refers to the pair to the ground-truth input-output pair and $G(x_i)$ refers to the image translated by the Generator.

$\mathcal{L}_{L1}(G, D) = \frac{1}{N} \sum_{i=1}^{N} \| y - G(x_i) \|_1$

The final objective is just a combination of these objectives.

$\mathcal{L}_{final}(G, D) = \mathcal{L}_{cGAN}(G, D) + \lambda \mathcal{L}_{L1}(G, D)$

$G^* = \underset{G}{\operatorname{argmin}} \underset{D}{\max} \mathcal{L}_{final}(G, D)$

You would be implementing these objectives using the `nn.BCELoss` and `nn.L1Loss` as given below.

In [11]:
```
# Hint: you could use following loss to complete following function
BCE_loss = nn.BCELoss().cuda()
L1_loss = nn.L1Loss().cuda()

def train(G, D, num_epochs = 20):
  hist_D_losses = []
  hist_G_losses = []
  hist_G_L1_losses = []
  ###########################################################################
  # TODO: Add Adam optimizer to generator and discriminator
  # You will use lr=0.0002, beta=0.5, beta2=0.999
  ###########################################################################

  G_optimizer = optim.Adam(G.parameters(), lr=2e-4, betas=(0.5, 0.999))
  D_optimizer = optim.Adam(D.parameters(), lr=2e-4, betas=(0.5, 0.999))

  ###########################################################################
  #                               END OF YOUR CODE
  ###########################################################################

  print('training start!')
  start_time = time.time()
  for epoch in range(num_epochs):
    print('Start training epoch %d' % (epoch + 1))
    D_losses = []
```

```
    G_losses = []
    epoch_start_time = time.time()
    num_iter = 0
    for x_ in train_loader:
      y_ = x_[:, :, :, img_size:]
      x_ = x_[:, :, :, 0:img_size]

      x_, y_ = x_.cuda(), y_.cuda()
      ########################################################################
      # TODO: Implement training code for the discriminator.
      # Recall that the loss is the mean of the loss for real images and fak
      # images, and made by some calculations with zeros and ones
      # We have defined the BCE_loss, which you might would like to use.
      #
      # NOTE: While training the Discriminator, the output of the generator
      # must be detached from the computational graph. Refer to the method
      # torch.Tensor.detach()
      ########################################################################

      N = x_.shape[0]
      # Generate data
      fake_data = G.forward(x_).detach()

      #1. Train the discriminator
      # D real data BCE loss
      D_real_preds = D.forward(torch.cat((x_, y_), dim=1))
      D_y_real = torch.ones_like(D_real_preds)
      # D_real_loss = torch.sum(torch.log(D_real_preds))
      D_real_loss = BCE_loss(D_real_preds, D_y_real)

      # D fake data BCE loss
      D_fake_preds = D.forward(torch.cat((x_, fake_data), dim=1))
      D_y_fake = torch.zeros_like(D_fake_preds)
      # D_fake_loss = torch.sum(torch.log(1 - D_fake_preds))
      D_fake_loss = BCE_loss(D_fake_preds, D_y_fake)

      # D loss
      loss_D = D_real_loss + D_fake_loss

      # Train D
      D_optimizer.zero_grad()
      loss_D.backward()
      D_optimizer.step()

      ########################################################################
      #                                END OF YOUR CODE
      ########################################################################

      ########################################################################
      # TODO: Implement training code for the Generator.
      ########################################################################

      # 1. Train the generator
      # 2. Append the losses to the lists 'hist_G_L1_losses' and 'hist_G_los
      #  (Only append the data to the list, not the complete tensor, refer
      #  torch.Tensor.item()).
```

```
      # Generate data
      fake_data = G.forward(x_)

      # 1. Train the generator
      # G BCE loss
      G_fake_preds = D.forward(torch.cat((x_, fake_data), dim=1))
      G_y_fake = torch.zeros_like(G_fake_preds)
      G_bce_loss = BCE_loss(G_fake_preds, G_y_fake)

      # G l1 loss
      G_l1_loss = L1_loss(fake_data, y_)

      # G loss
      lamb = 100
      loss_G = G_bce_loss +  lamb * G_l1_loss

      # Train G
      G_optimizer.zero_grad()
      loss_G.backward()
      G_optimizer.step()

      # 2. Append the losses to the lists 'hist_G_L1_losses' and 'hist_D_los
      #  (Only append the data to the list, not the complete tensor, refer
      #  torch.Tensor.item()).
      hist_G_losses.append(G_bce_loss.detach().item())
      hist_G_L1_losses.append(G_l1_loss.detach().item())
      ##########################################################################
      #                                 END OF YOUR CODE
      ##########################################################################

      D_losses.append(loss_D.detach().item())
      hist_D_losses.append(loss_D.detach().item())
      G_losses.append(loss_G)
      num_iter += 1



    epoch_end_time = time.time()
    per_epoch_ptime = epoch_end_time - epoch_start_time

    print('[%d/%d] - using time: %.2f seconds' % ((epoch + 1), num_epochs, p
    print('loss of discriminator D: %.3f' % (torch.mean(torch.FloatTensor(D_
    print('loss of generator G: %.3f' % (torch.mean(torch.FloatTensor(G_loss
    if epoch == 0 or (epoch + 1) % 5 == 0:
      with torch.no_grad():
        show_result(G, fixed_x_, fixed_y_, (epoch+1))

  end_time = time.time()
  total_ptime = end_time - start_time

  return hist_D_losses, hist_G_losses, hist_G_L1_losses
```

In this part, train your model with $\lambda = 100$ with at least 20 epochs.

In [12]:
```
# Define network
G_100 = generator()
D_100 = discriminator()
G_100.weight_init(mean=0.0, std=0.02)
D_100.weight_init(mean=0.0, std=0.02)
G_100.cuda()
D_100.cuda()
G_100.train()
D_100.train()
```

Out[12]:
```
discriminator(
  (conv1): Conv2d(6, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
  (conv2): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1,
1))
  (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
  (conv3): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1,
1))
  (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
  (conv4): Conv2d(256, 512, kernel_size=(4, 4), stride=(1, 1), padding=(1,
1))
  (bn4): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_runni
ng_stats=True)
  (conv5): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), padding=(1,
1))
)
```

In [13]:
```
#training, you will be expecting 1-2 minutes per epoch.
# TODO: change_num_epochs if you want

hist_D_100_losses, hist_G_100_BCE_losses, hist_G_100_L1_losses = train(G=G_1
```

```
training start!
Start training epoch 1
[1/20] - using time: 33.23 seconds
loss of discriminator D: 0.521
loss of generator G: 25.637
```


Input
Output
Ground truth

Epoch 1

```
Start training epoch 2
[2/20] - using time: 30.38 seconds
loss of discriminator D: 0.233
loss of generator G: 18.436
Start training epoch 3
[3/20] - using time: 30.51 seconds
loss of discriminator D: 0.117
loss of generator G: 17.228
Start training epoch 4
[4/20] - using time: 30.43 seconds
loss of discriminator D: 0.188
loss of generator G: 16.127
Start training epoch 5
[5/20] - using time: 30.52 seconds
loss of discriminator D: 0.041
loss of generator G: 15.267
```


Input
Output
Ground truth

Epoch 5

```
Start training epoch 6
[6/20] - using time: 30.44 seconds
loss of discriminator D: 0.046
loss of generator G: 13.856
Start training epoch 7
[7/20] - using time: 30.49 seconds
loss of discriminator D: 0.005
loss of generator G: 13.028
Start training epoch 8
[8/20] - using time: 30.37 seconds
loss of discriminator D: 0.097
loss of generator G: 12.411
Start training epoch 9
[9/20] - using time: 30.57 seconds
loss of discriminator D: 0.086
loss of generator G: 11.618
Start training epoch 10
[10/20] - using time: 30.50 seconds
loss of discriminator D: 0.006
loss of generator G: 11.079
```


Input
Output
Ground truth

Epoch 10

```
Start training epoch 11
[11/20] - using time: 30.36 seconds
loss of discriminator D: 0.033
loss of generator G: 10.787
Start training epoch 12
[12/20] - using time: 30.41 seconds
loss of discriminator D: 0.197
loss of generator G: 10.169
Start training epoch 13
[13/20] - using time: 30.40 seconds
loss of discriminator D: 0.060
loss of generator G: 9.624
Start training epoch 14
[14/20] - using time: 30.40 seconds
loss of discriminator D: 0.066
loss of generator G: 9.171
Start training epoch 15
[15/20] - using time: 30.53 seconds
loss of discriminator D: 0.004
loss of generator G: 8.806
```


Input
Output
Ground truth

Epoch 15

```
Start training epoch 16
[16/20] - using time: 30.52 seconds
loss of discriminator D: 0.003
loss of generator G: 8.398
Start training epoch 17
[17/20] - using time: 30.42 seconds
loss of discriminator D: 0.005
loss of generator G: 8.094
Start training epoch 18
[18/20] - using time: 30.77 seconds
loss of discriminator D: 0.004
loss of generator G: 7.857
Start training epoch 19
[19/20] - using time: 30.47 seconds
loss of discriminator D: 0.003
loss of generator G: 7.606
Start training epoch 20
[20/20] - using time: 30.52 seconds
loss of discriminator D: 0.216
loss of generator G: 7.544
```


Input
Output
Ground truth

Epoch 20

In [14]:
```
!mkdir models
torch.save(G_100.state_dict(), './models/G_100.pth')
torch.save(D_100.state_dict(), './models/D_100.pth')
```

```
mkdir: cannot create directory ‘models’: File exists
```

The following cell saves the trained model parameters to your Google Drive so you could reuse those parameters later without retraining.

In [ ]:
```
from google.colab import drive
drive.mount('/content/drive')
!cp "./models/" -r "/content/drive/My Drive/"
```

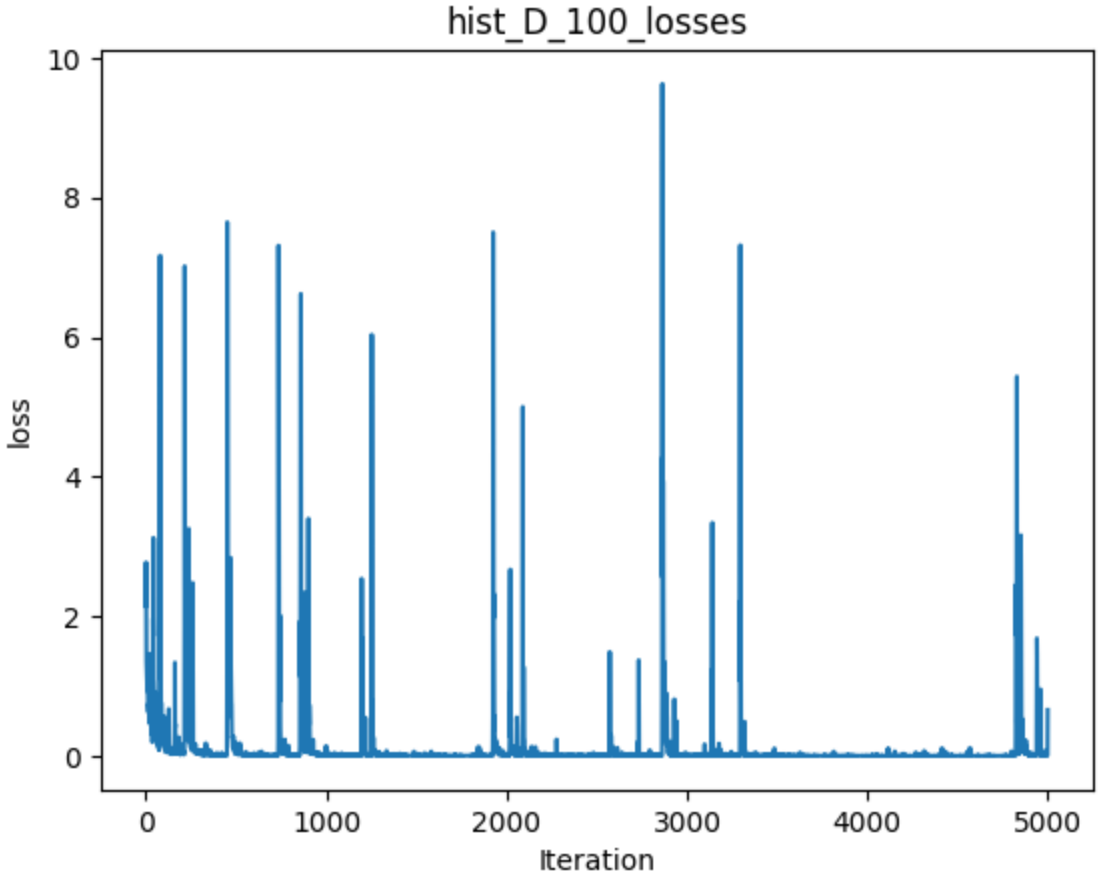
```
Mounted at /content/drive
```

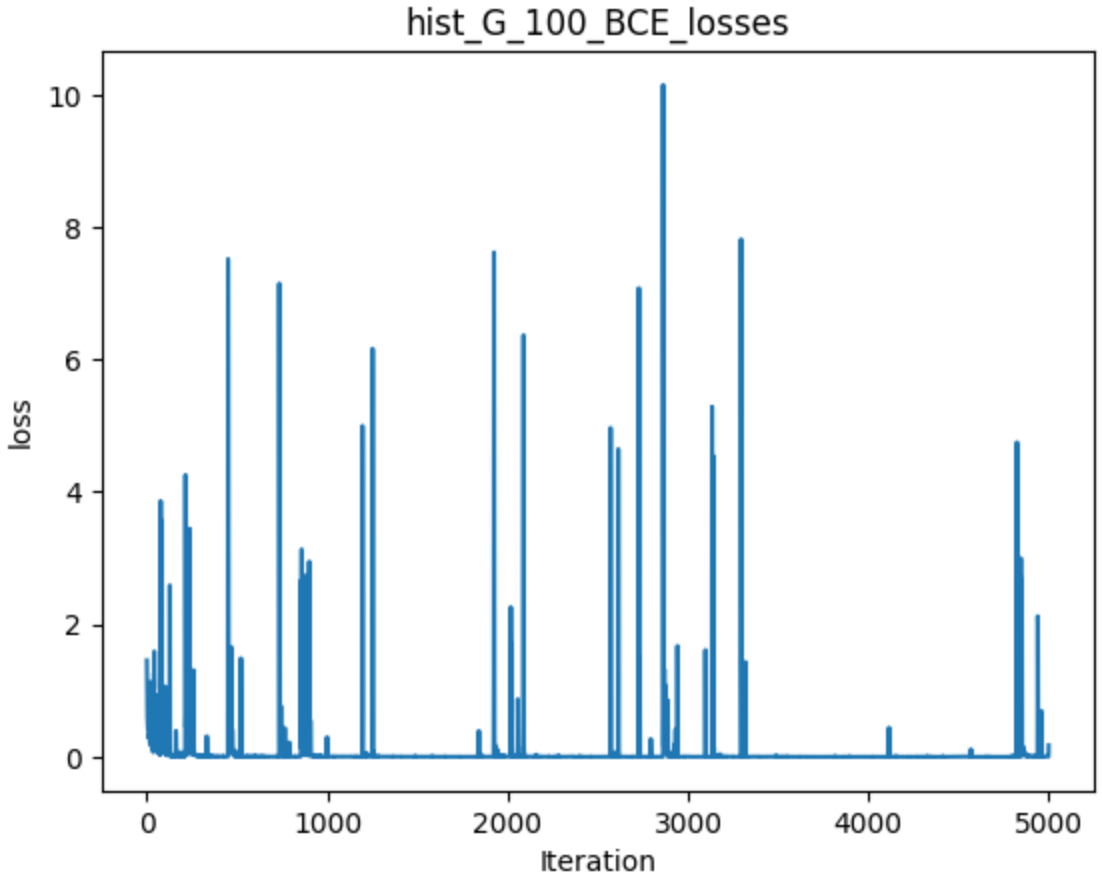
Step 4: Visualization

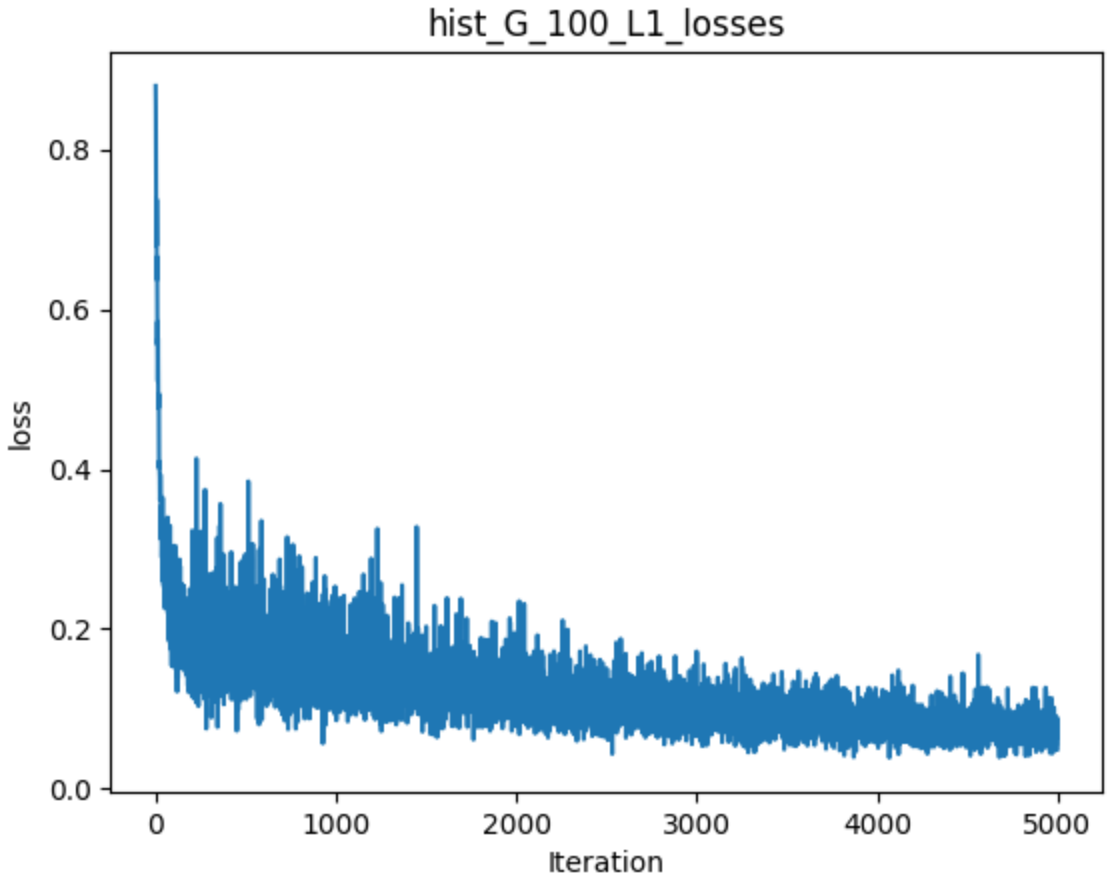
Please plot the generator BCE and L1 losses, as well as the discriminator loss. For this, please use $\lambda = 100$, and use 3 separate plots. We have provided the code for you and you only need to run the code below.

In [15]:
```
# Plot the G/D loss history (y axis) vs. Iteration (x axis)         #
# You will have three plots, with hist_D_100_losses,                #
# hist_G_100_BCE_losses, hist_G_100_L1_losses respectively.         #

# hist_D_100_losses
plt.figure()
plt.plot(range(len(hist_D_100_losses)), torch.tensor(hist_D_100_losses, devi
# plt.legend('hist_D_100_losses')
plt.xlabel('Iteration')
plt.ylabel('loss')
plt.title('hist_D_100_losses')
plt.show()


# hist_G_100_BCE_losses
plt.figure()
plt.plot(range(len(hist_G_100_BCE_losses)), torch.tensor(hist_G_100_BCE_loss
# plt.legend('hist_G_100_BCE_losses')
plt.xlabel('Iteration')
plt.ylabel('loss')
plt.title('hist_G_100_BCE_losses')
plt.show()

# hist_G_100_L1_losses
plt.figure()
plt.plot(range(len(hist_G_100_L1_losses)), torch.tensor(hist_G_100_L1_losses
# plt.legend('hist_G_100_L1_losses')
```

```
plt.xlabel('Iteration')
plt.ylabel('loss')
plt.title('hist_G_100_L1_losses')
plt.show()
```




hist_G_100_BCE_losses
loss
Iteration
0
2
4
6
8
10
0
1000
2000
3000
4000
5000


hist_G_100_L1_losses
loss
Iteration
0.0
0.2
0.4
0.6
0.8
0
1000
2000
3000
4000
5000

Convert Notebook to PDF

Alternative if the cell below doesn't work.

In [ ]:
```
import os
from google.colab import drive
from google.colab import files

drive_mount_point = '/content/drive/'
drive.mount(drive_mount_point)
```

```
Drive already mounted at /content/drive/; to attempt to forcibly remount, ca
ll drive.mount("/content/drive/", force_remount=True).
```

In [ ]:
```
# generate pdf
# Please provide the full path of the notebook file below
# Important: make sure that your file name does not contain spaces!

# Ex: notebookpath = '/content/drive/My Drive/Colab Notebooks/EECS_442_PS4_F
notebookpath = '/content/drive/My Drive/Colab Notebooks/EECS_442-504_PS6_FA_

file_name = notebookpath.split('/')[-1]
get_ipython().system("apt update && apt install texlive-xetex texlive-fonts-
get_ipython().system("jupyter nbconvert --to PDF {}".format(notebookpath.rep
files.download(notebookpath.split('.')[0]+'.pdf')
```

Homework 5: Diffusion Models

Run the following code to setup the necessary requirements

In [ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

In [ ]:
```
# TODO: Fill in the Google Drive path where you uploaded the assignment
# Example: If you create a EECS442 folder and put all the files under HW5 fo
GOOGLE_DRIVE_PATH_AFTER_MYDRIVE = 'EECS442/HW5'
```

In [1]:
```
%load_ext autoreload
%autoreload 2
```

In [ ]:
```
import os
import sys

GOOGLE_DRIVE_PATH = os.path.join('drive', 'MyDrive', GOOGLE_DRIVE_PATH_AFTER
sys.path.append(GOOGLE_DRIVE_PATH)
```

In [ ]:
```
print(GOOGLE_DRIVE_PATH)
```

You need to change your working directory.

In [ ]:
```
%cd /content/drive/MyDrive/EECS442/HW5
```

In [ ]:
```
!pip install certifi>=2022.9.14
!pip install charset-normalizer>=2.1.1
!pip install contourpy>=1.0.5
!pip install cycler>=0.11.0
!pip install fonttools>=4.37.2
!pip install idna>=3.4
!pip install kiwisolver>=1.4.4
!pip install matplotlib>=3.6.0
!pip install numpy>=1.23.3
!pip install packaging>=21.3
!pip install Pillow>=9.2.0
!pip install pyparsing>=3.0.9
!pip install python-dateutil>=2.8.2
!pip install PyYAML>=6.0
!pip install requests>=2.28.1
!pip install scipy>=1.9.1
!pip install six>=1.16.0
!pip install tqdm>=4.64.1
!pip install typing-extensions>=4.3.0
!pip install urllib3>=1.26.12
```

In [ ]:
```
!nvidia-smi
```

Task 1: Unconditional Sampling with DDPM

Setup

In [2]:
```
from functools import partial
import os
import argparse
import yaml

import torch
import torchvision.transforms as transforms
import matplotlib.pyplot as plt

from util.logger import get_logger
# GPUs are preferred
logger = get_logger()
device_str = f"cuda:0" if torch.cuda.is_available() else 'cpu'
logger.info(f"Device set to {device_str}.")
device = torch.device(device_str)

# set output directory
save_dir = 'results'
ddpm_out_path = os.path.join(save_dir, 'uncond_ddpm')
os.makedirs(ddpm_out_path, exist_ok=True)
for img_dir in ['input', 'output', 'progress']:
    os.makedirs(os.path.join(ddpm_out_path, img_dir), exist_ok=True)
```

```
2024-04-09 14:31:59,649 [DPS] >> Device set to cuda:0.
```

In this task, you will implement the sampling Algorithm proposed in Denoising Diffusion Probabilistic Models(DDPM) paper as shown below:

(1) Now let's implement the variance schedule. As you can see in the DDPM sampling algorithm, We will need α_t for each timestep t. α_t is a notaion for $1 - \beta_t$, where β_t is the true variance that increases from $t = 1$ to $t = T$. There are many different variance shedules such as linear schedule and cosine schedule. Follow the instruction in `guided_diffusion/simple_diffusion.py` to implement `get_named_beta_schedule()`.

Cosine schedule is proposed by iDDPM. You can find the detailed motivation in the paper. The calulation of β depends on α, the cumulated product of α is defined as

$$\bar{\alpha}_t = \frac{f(t)}{f(0)}$$

, where

$$f(t) = \cos\left(\frac{t/T + s}{1+s} \cdot \frac{\pi}{2}\right)^2$$

We use small $s = 0.008$ such that $\sqrt{\beta_0}$ was slightly smaller than the pixel bin size 1/127.5. According to the definition of α_t, we can then get β_t as

$$\beta_t = 1 - \frac{\bar{\alpha}_t}{\bar{\alpha}_{t-1}}$$

Also, clip β_t to be no larger than 0.999 to prevent singularities at the end of the diffusion process.

Run the following code to see your output.

In [3]:
```
from guided_diffusion.simple_diffusion import get_named_beta_schedule
import numpy as np

num_steps = 1000
schedule_name = 'cosine'

print('Cosine Error: ', np.sum(get_named_beta_schedule(schedule_name, num_st
```

```
/home/umhws/anaconda3/envs/eecs442/lib/python3.10/site-packages/tqdm/auto.p
y:21: TqdmWarning: IProgress not found. Please update jupyter and ipywidget
s. See https://ipywidgets.readthedocs.io/en/stable/user_install.html
  from .autonotebook import tqdm as notebook_tqdm
Cosine Error:  -6.661338147750939e-16
```

(2) Now you have implemented your variance schedule $\{\beta_1, \beta_2, \ldots, \beta_T\}$. In practice, we use α_t and accumulated product $\bar{\alpha}_t = \prod_{i=1}^{t} \alpha_i$. **Follow the instruction to complete `__init__()` of `DDPMDiffusion`** to compute the needed values that will be used during the sampling process. They only need to be calculated once during initialzation and we can directly access them later during sampling.

Hint: You can use `np.cumprod()` to calculate the cumulated product.

Let's now create the model.

Before you run the code below, make sure that you have downloaded the pretrained model ffhq_10M.pt and put it under `model` directory.

In [4]:
```
from guided_diffusion.unet import create_model
from data.dataloader import get_dataset, get_dataloader

# Here is the model configuration of the Pretrained UNet model that we will
# This configuration should be consistent with the pretrained model, so you
# You can find the detailed definition of the UNet in guided_diffusion/unet.

model_config = {
    'image_size': 256,
    'num_channels': 128,
```

```
    'num_res_blocks': 1,
    'channel_mult': "",
    'learn_sigma': True,
    'class_cond': False,
    'use_checkpoint': False,
    'attention_resolutions': 16,
    'num_heads': 4,
    'num_head_channels': 64,
    'num_heads_upsample': -1,
    'use_scale_shift_norm': True,
    'dropout': 0.0,
    'resblock_updown': True,
    'use_fp16': False,
    'use_new_attention_order': False,
    'model_path': 'models/ffhq_10m.pt'
}

# Load model
ddpm_beta = get_named_beta_schedule('linear', 1000)

model = create_model(betas=ddpm_beta, **model_config)
model = model.to(device)
model.eval()  # Set the model to the evaluation mode as we don't need to tra
```

```
pretrained model loaded!
```

Out[4]:
```
UNetModel(
  (time_embed): Sequential(
    (0): Linear(in_features=128, out_features=512, bias=True)
    (1): SiLU()
    (2): Linear(in_features=512, out_features=512, bias=True)
  )
  (input_blocks): ModuleList(
    (0): TimestepEmbedSequential(
      (0): Conv2d(3, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    )
    (1): TimestepEmbedSequential(
      (0): ResBlock(
        (in_layers): Sequential(
          (0): GroupNorm32(32, 128, eps=1e-05, affine=True)
          (1): SiLU()
          (2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
        )
        (h_upd): Identity()
        (x_upd): Identity()
        (emb_layers): Sequential(
          (0): SiLU()
          (1): Linear(in_features=512, out_features=256, bias=True)
        )
        (out_layers): Sequential(
          (0): GroupNorm32(32, 128, eps=1e-05, affine=True)
          (1): SiLU()
          (2): Dropout(p=0.0, inplace=False)
          (3): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
        )
        (skip_connection): Identity()
      )
    )
    (2): TimestepEmbedSequential(
      (0): ResBlock(
        (in_layers): Sequential(
          (0): GroupNorm32(32, 128, eps=1e-05, affine=True)
          (1): SiLU()
          (2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
        )
        (h_upd): Downsample(
          (op): AvgPool2d(kernel_size=2, stride=2, padding=0)
        )
        (x_upd): Downsample(
          (op): AvgPool2d(kernel_size=2, stride=2, padding=0)
        )
        (emb_layers): Sequential(
          (0): SiLU()
          (1): Linear(in_features=512, out_features=256, bias=True)
        )
        (out_layers): Sequential(
          (0): GroupNorm32(32, 128, eps=1e-05, affine=True)
          (1): SiLU()
```

```
          (2): Dropout(p=0.0, inplace=False)
          (3): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
        )
        (skip_connection): Identity()
      )
    )
    (3): TimestepEmbedSequential(
      (0): ResBlock(
        (in_layers): Sequential(
          (0): GroupNorm32(32, 128, eps=1e-05, affine=True)
          (1): SiLU()
          (2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
        )
        (h_upd): Identity()
        (x_upd): Identity()
        (emb_layers): Sequential(
          (0): SiLU()
          (1): Linear(in_features=512, out_features=256, bias=True)
        )
        (out_layers): Sequential(
          (0): GroupNorm32(32, 128, eps=1e-05, affine=True)
          (1): SiLU()
          (2): Dropout(p=0.0, inplace=False)
          (3): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
        )
        (skip_connection): Identity()
      )
    )
    (4): TimestepEmbedSequential(
      (0): ResBlock(
        (in_layers): Sequential(
          (0): GroupNorm32(32, 128, eps=1e-05, affine=True)
          (1): SiLU()
          (2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
        )
        (h_upd): Downsample(
          (op): AvgPool2d(kernel_size=2, stride=2, padding=0)
        )
        (x_upd): Downsample(
          (op): AvgPool2d(kernel_size=2, stride=2, padding=0)
        )
        (emb_layers): Sequential(
          (0): SiLU()
          (1): Linear(in_features=512, out_features=256, bias=True)
        )
        (out_layers): Sequential(
          (0): GroupNorm32(32, 128, eps=1e-05, affine=True)
          (1): SiLU()
          (2): Dropout(p=0.0, inplace=False)
          (3): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
        )
```

```
        (skip_connection): Identity()
      )
    )
    (5): TimestepEmbedSequential(
      (0): ResBlock(
        (in_layers): Sequential(
          (0): GroupNorm32(32, 128, eps=1e-05, affine=True)
          (1): SiLU()
          (2): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
        )
        (h_upd): Identity()
        (x_upd): Identity()
        (emb_layers): Sequential(
          (0): SiLU()
          (1): Linear(in_features=512, out_features=512, bias=True)
        )
        (out_layers): Sequential(
          (0): GroupNorm32(32, 256, eps=1e-05, affine=True)
          (1): SiLU()
          (2): Dropout(p=0.0, inplace=False)
          (3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
        )
        (skip_connection): Conv2d(128, 256, kernel_size=(1, 1), stride=(1,
1))
      )
    )
    (6): TimestepEmbedSequential(
      (0): ResBlock(
        (in_layers): Sequential(
          (0): GroupNorm32(32, 256, eps=1e-05, affine=True)
          (1): SiLU()
          (2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
        )
        (h_upd): Downsample(
          (op): AvgPool2d(kernel_size=2, stride=2, padding=0)
        )
        (x_upd): Downsample(
          (op): AvgPool2d(kernel_size=2, stride=2, padding=0)
        )
        (emb_layers): Sequential(
          (0): SiLU()
          (1): Linear(in_features=512, out_features=512, bias=True)
        )
        (out_layers): Sequential(
          (0): GroupNorm32(32, 256, eps=1e-05, affine=True)
          (1): SiLU()
          (2): Dropout(p=0.0, inplace=False)
          (3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
        )
        (skip_connection): Identity()
      )
    )
```

```
    (7): TimestepEmbedSequential(
      (0): ResBlock(
        (in_layers): Sequential(
          (0): GroupNorm32(32, 256, eps=1e-05, affine=True)
          (1): SiLU()
          (2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
        )
        (h_upd): Identity()
        (x_upd): Identity()
        (emb_layers): Sequential(
          (0): SiLU()
          (1): Linear(in_features=512, out_features=512, bias=True)
        )
        (out_layers): Sequential(
          (0): GroupNorm32(32, 256, eps=1e-05, affine=True)
          (1): SiLU()
          (2): Dropout(p=0.0, inplace=False)
          (3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
        )
        (skip_connection): Identity()
      )
    )
    (8): TimestepEmbedSequential(
      (0): ResBlock(
        (in_layers): Sequential(
          (0): GroupNorm32(32, 256, eps=1e-05, affine=True)
          (1): SiLU()
          (2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
        )
        (h_upd): Downsample(
          (op): AvgPool2d(kernel_size=2, stride=2, padding=0)
        )
        (x_upd): Downsample(
          (op): AvgPool2d(kernel_size=2, stride=2, padding=0)
        )
        (emb_layers): Sequential(
          (0): SiLU()
          (1): Linear(in_features=512, out_features=512, bias=True)
        )
        (out_layers): Sequential(
          (0): GroupNorm32(32, 256, eps=1e-05, affine=True)
          (1): SiLU()
          (2): Dropout(p=0.0, inplace=False)
          (3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
        )
        (skip_connection): Identity()
      )
    )
    (9): TimestepEmbedSequential(
      (0): ResBlock(
        (in_layers): Sequential(
          (0): GroupNorm32(32, 256, eps=1e-05, affine=True)
```

```
          (1): SiLU()
          (2): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
        )
        (h_upd): Identity()
        (x_upd): Identity()
        (emb_layers): Sequential(
          (0): SiLU()
          (1): Linear(in_features=512, out_features=1024, bias=True)
        )
        (out_layers): Sequential(
          (0): GroupNorm32(32, 512, eps=1e-05, affine=True)
          (1): SiLU()
          (2): Dropout(p=0.0, inplace=False)
          (3): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
        )
        (skip_connection): Conv2d(256, 512, kernel_size=(1, 1), stride=(1,
1))
      )
      (1): AttentionBlock(
        (norm): GroupNorm32(32, 512, eps=1e-05, affine=True)
        (qkv): Conv1d(512, 1536, kernel_size=(1,), stride=(1,))
        (attention): QKVAttentionLegacy()
        (proj_out): Conv1d(512, 512, kernel_size=(1,), stride=(1,))
      )
    )
    (10): TimestepEmbedSequential(
      (0): ResBlock(
        (in_layers): Sequential(
          (0): GroupNorm32(32, 512, eps=1e-05, affine=True)
          (1): SiLU()
          (2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
        )
        (h_upd): Downsample(
          (op): AvgPool2d(kernel_size=2, stride=2, padding=0)
        )
        (x_upd): Downsample(
          (op): AvgPool2d(kernel_size=2, stride=2, padding=0)
        )
        (emb_layers): Sequential(
          (0): SiLU()
          (1): Linear(in_features=512, out_features=1024, bias=True)
        )
        (out_layers): Sequential(
          (0): GroupNorm32(32, 512, eps=1e-05, affine=True)
          (1): SiLU()
          (2): Dropout(p=0.0, inplace=False)
          (3): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
        )
        (skip_connection): Identity()
      )
    )
    (11): TimestepEmbedSequential(
```

```
      (0): ResBlock(
        (in_layers): Sequential(
          (0): GroupNorm32(32, 512, eps=1e-05, affine=True)
          (1): SiLU()
          (2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
        )
        (h_upd): Identity()
        (x_upd): Identity()
        (emb_layers): Sequential(
          (0): SiLU()
          (1): Linear(in_features=512, out_features=1024, bias=True)
        )
        (out_layers): Sequential(
          (0): GroupNorm32(32, 512, eps=1e-05, affine=True)
          (1): SiLU()
          (2): Dropout(p=0.0, inplace=False)
          (3): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
        )
        (skip_connection): Identity()
      )
    )
  )
  (middle_block): TimestepEmbedSequential(
    (0): ResBlock(
      (in_layers): Sequential(
        (0): GroupNorm32(32, 512, eps=1e-05, affine=True)
        (1): SiLU()
        (2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
      )
      (h_upd): Identity()
      (x_upd): Identity()
      (emb_layers): Sequential(
        (0): SiLU()
        (1): Linear(in_features=512, out_features=1024, bias=True)
      )
      (out_layers): Sequential(
        (0): GroupNorm32(32, 512, eps=1e-05, affine=True)
        (1): SiLU()
        (2): Dropout(p=0.0, inplace=False)
        (3): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
      )
      (skip_connection): Identity()
    )
    (1): AttentionBlock(
      (norm): GroupNorm32(32, 512, eps=1e-05, affine=True)
      (qkv): Conv1d(512, 1536, kernel_size=(1,), stride=(1,))
      (attention): QKVAttentionLegacy()
      (proj_out): Conv1d(512, 512, kernel_size=(1,), stride=(1,))
    )
    (2): ResBlock(
      (in_layers): Sequential(
        (0): GroupNorm32(32, 512, eps=1e-05, affine=True)
```

```
        (1): SiLU()
        (2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
      )
      (h_upd): Identity()
      (x_upd): Identity()
      (emb_layers): Sequential(
        (0): SiLU()
        (1): Linear(in_features=512, out_features=1024, bias=True)
      )
      (out_layers): Sequential(
        (0): GroupNorm32(32, 512, eps=1e-05, affine=True)
        (1): SiLU()
        (2): Dropout(p=0.0, inplace=False)
        (3): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
      )
      (skip_connection): Identity()
    )
  )
  (output_blocks): ModuleList(
    (0): TimestepEmbedSequential(
      (0): ResBlock(
        (in_layers): Sequential(
          (0): GroupNorm32(32, 1024, eps=1e-05, affine=True)
          (1): SiLU()
          (2): Conv2d(1024, 512, kernel_size=(3, 3), stride=(1, 1), padding
=(1, 1))
        )
        (h_upd): Identity()
        (x_upd): Identity()
        (emb_layers): Sequential(
          (0): SiLU()
          (1): Linear(in_features=512, out_features=1024, bias=True)
        )
        (out_layers): Sequential(
          (0): GroupNorm32(32, 512, eps=1e-05, affine=True)
          (1): SiLU()
          (2): Dropout(p=0.0, inplace=False)
          (3): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
        )
        (skip_connection): Conv2d(1024, 512, kernel_size=(1, 1), stride=(1,
1))
      )
    )
    (1): TimestepEmbedSequential(
      (0): ResBlock(
        (in_layers): Sequential(
          (0): GroupNorm32(32, 1024, eps=1e-05, affine=True)
          (1): SiLU()
          (2): Conv2d(1024, 512, kernel_size=(3, 3), stride=(1, 1), padding
=(1, 1))
        )
        (h_upd): Identity()
        (x_upd): Identity()
```

```
        (emb_layers): Sequential(
          (0): SiLU()
          (1): Linear(in_features=512, out_features=1024, bias=True)
        )
        (out_layers): Sequential(
          (0): GroupNorm32(32, 512, eps=1e-05, affine=True)
          (1): SiLU()
          (2): Dropout(p=0.0, inplace=False)
          (3): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
        )
        (skip_connection): Conv2d(1024, 512, kernel_size=(1, 1), stride=(1,
1))
      )
      (1): ResBlock(
        (in_layers): Sequential(
          (0): GroupNorm32(32, 512, eps=1e-05, affine=True)
          (1): SiLU()
          (2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
        )
        (h_upd): Upsample()
        (x_upd): Upsample()
        (emb_layers): Sequential(
          (0): SiLU()
          (1): Linear(in_features=512, out_features=1024, bias=True)
        )
        (out_layers): Sequential(
          (0): GroupNorm32(32, 512, eps=1e-05, affine=True)
          (1): SiLU()
          (2): Dropout(p=0.0, inplace=False)
          (3): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
        )
        (skip_connection): Identity()
      )
    )
    (2): TimestepEmbedSequential(
      (0): ResBlock(
        (in_layers): Sequential(
          (0): GroupNorm32(32, 1024, eps=1e-05, affine=True)
          (1): SiLU()
          (2): Conv2d(1024, 512, kernel_size=(3, 3), stride=(1, 1), padding
=(1, 1))
        )
        (h_upd): Identity()
        (x_upd): Identity()
        (emb_layers): Sequential(
          (0): SiLU()
          (1): Linear(in_features=512, out_features=1024, bias=True)
        )
        (out_layers): Sequential(
          (0): GroupNorm32(32, 512, eps=1e-05, affine=True)
          (1): SiLU()
          (2): Dropout(p=0.0, inplace=False)
          (3): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=
```

```
(1, 1))
        )
        (skip_connection): Conv2d(1024, 512, kernel_size=(1, 1), stride=(1,
1))
      )
      (1): AttentionBlock(
        (norm): GroupNorm32(32, 512, eps=1e-05, affine=True)
        (qkv): Conv1d(512, 1536, kernel_size=(1,), stride=(1,))
        (attention): QKVAttentionLegacy()
        (proj_out): Conv1d(512, 512, kernel_size=(1,), stride=(1,))
      )
    )
    (3): TimestepEmbedSequential(
      (0): ResBlock(
        (in_layers): Sequential(
          (0): GroupNorm32(32, 768, eps=1e-05, affine=True)
          (1): SiLU()
          (2): Conv2d(768, 512, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
        )
        (h_upd): Identity()
        (x_upd): Identity()
        (emb_layers): Sequential(
          (0): SiLU()
          (1): Linear(in_features=512, out_features=1024, bias=True)
        )
        (out_layers): Sequential(
          (0): GroupNorm32(32, 512, eps=1e-05, affine=True)
          (1): SiLU()
          (2): Dropout(p=0.0, inplace=False)
          (3): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
        )
        (skip_connection): Conv2d(768, 512, kernel_size=(1, 1), stride=(1,
1))
      )
      (1): AttentionBlock(
        (norm): GroupNorm32(32, 512, eps=1e-05, affine=True)
        (qkv): Conv1d(512, 1536, kernel_size=(1,), stride=(1,))
        (attention): QKVAttentionLegacy()
        (proj_out): Conv1d(512, 512, kernel_size=(1,), stride=(1,))
      )
      (2): ResBlock(
        (in_layers): Sequential(
          (0): GroupNorm32(32, 512, eps=1e-05, affine=True)
          (1): SiLU()
          (2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
        )
        (h_upd): Upsample()
        (x_upd): Upsample()
        (emb_layers): Sequential(
          (0): SiLU()
          (1): Linear(in_features=512, out_features=1024, bias=True)
        )
        (out_layers): Sequential(
```

```
          (0): GroupNorm32(32, 512, eps=1e-05, affine=True)
          (1): SiLU()
          (2): Dropout(p=0.0, inplace=False)
          (3): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
        )
        (skip_connection): Identity()
      )
    )
    (4): TimestepEmbedSequential(
      (0): ResBlock(
        (in_layers): Sequential(
          (0): GroupNorm32(32, 768, eps=1e-05, affine=True)
          (1): SiLU()
          (2): Conv2d(768, 256, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
        )
        (h_upd): Identity()
        (x_upd): Identity()
        (emb_layers): Sequential(
          (0): SiLU()
          (1): Linear(in_features=512, out_features=512, bias=True)
        )
        (out_layers): Sequential(
          (0): GroupNorm32(32, 256, eps=1e-05, affine=True)
          (1): SiLU()
          (2): Dropout(p=0.0, inplace=False)
          (3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
        )
        (skip_connection): Conv2d(768, 256, kernel_size=(1, 1), stride=(1,
1))
      )
    )
    (5): TimestepEmbedSequential(
      (0): ResBlock(
        (in_layers): Sequential(
          (0): GroupNorm32(32, 512, eps=1e-05, affine=True)
          (1): SiLU()
          (2): Conv2d(512, 256, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
        )
        (h_upd): Identity()
        (x_upd): Identity()
        (emb_layers): Sequential(
          (0): SiLU()
          (1): Linear(in_features=512, out_features=512, bias=True)
        )
        (out_layers): Sequential(
          (0): GroupNorm32(32, 256, eps=1e-05, affine=True)
          (1): SiLU()
          (2): Dropout(p=0.0, inplace=False)
          (3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
        )
        (skip_connection): Conv2d(512, 256, kernel_size=(1, 1), stride=(1,
```

```
1))
      )
      (1): ResBlock(
        (in_layers): Sequential(
          (0): GroupNorm32(32, 256, eps=1e-05, affine=True)
          (1): SiLU()
          (2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
        )
        (h_upd): Upsample()
        (x_upd): Upsample()
        (emb_layers): Sequential(
          (0): SiLU()
          (1): Linear(in_features=512, out_features=512, bias=True)
        )
        (out_layers): Sequential(
          (0): GroupNorm32(32, 256, eps=1e-05, affine=True)
          (1): SiLU()
          (2): Dropout(p=0.0, inplace=False)
          (3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
        )
        (skip_connection): Identity()
      )
    )
    (6): TimestepEmbedSequential(
      (0): ResBlock(
        (in_layers): Sequential(
          (0): GroupNorm32(32, 512, eps=1e-05, affine=True)
          (1): SiLU()
          (2): Conv2d(512, 256, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
        )
        (h_upd): Identity()
        (x_upd): Identity()
        (emb_layers): Sequential(
          (0): SiLU()
          (1): Linear(in_features=512, out_features=512, bias=True)
        )
        (out_layers): Sequential(
          (0): GroupNorm32(32, 256, eps=1e-05, affine=True)
          (1): SiLU()
          (2): Dropout(p=0.0, inplace=False)
          (3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
        )
        (skip_connection): Conv2d(512, 256, kernel_size=(1, 1), stride=(1,
1))
      )
    )
    (7): TimestepEmbedSequential(
      (0): ResBlock(
        (in_layers): Sequential(
          (0): GroupNorm32(32, 384, eps=1e-05, affine=True)
          (1): SiLU()
          (2): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=
```

```
(1, 1))
        )
        (h_upd): Identity()
        (x_upd): Identity()
        (emb_layers): Sequential(
          (0): SiLU()
          (1): Linear(in_features=512, out_features=512, bias=True)
        )
        (out_layers): Sequential(
          (0): GroupNorm32(32, 256, eps=1e-05, affine=True)
          (1): SiLU()
          (2): Dropout(p=0.0, inplace=False)
          (3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
        )
        (skip_connection): Conv2d(384, 256, kernel_size=(1, 1), stride=(1,
1))
      )
      (1): ResBlock(
        (in_layers): Sequential(
          (0): GroupNorm32(32, 256, eps=1e-05, affine=True)
          (1): SiLU()
          (2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
        )
        (h_upd): Upsample()
        (x_upd): Upsample()
        (emb_layers): Sequential(
          (0): SiLU()
          (1): Linear(in_features=512, out_features=512, bias=True)
        )
        (out_layers): Sequential(
          (0): GroupNorm32(32, 256, eps=1e-05, affine=True)
          (1): SiLU()
          (2): Dropout(p=0.0, inplace=False)
          (3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
        )
        (skip_connection): Identity()
      )
    )
    (8): TimestepEmbedSequential(
      (0): ResBlock(
        (in_layers): Sequential(
          (0): GroupNorm32(32, 384, eps=1e-05, affine=True)
          (1): SiLU()
          (2): Conv2d(384, 128, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
        )
        (h_upd): Identity()
        (x_upd): Identity()
        (emb_layers): Sequential(
          (0): SiLU()
          (1): Linear(in_features=512, out_features=256, bias=True)
        )
        (out_layers): Sequential(
```

```
          (0): GroupNorm32(32, 128, eps=1e-05, affine=True)
          (1): SiLU()
          (2): Dropout(p=0.0, inplace=False)
          (3): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
        )
        (skip_connection): Conv2d(384, 128, kernel_size=(1, 1), stride=(1,
1))
      )
    )
    (9): TimestepEmbedSequential(
      (0): ResBlock(
        (in_layers): Sequential(
          (0): GroupNorm32(32, 256, eps=1e-05, affine=True)
          (1): SiLU()
          (2): Conv2d(256, 128, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
        )
        (h_upd): Identity()
        (x_upd): Identity()
        (emb_layers): Sequential(
          (0): SiLU()
          (1): Linear(in_features=512, out_features=256, bias=True)
        )
        (out_layers): Sequential(
          (0): GroupNorm32(32, 128, eps=1e-05, affine=True)
          (1): SiLU()
          (2): Dropout(p=0.0, inplace=False)
          (3): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
        )
        (skip_connection): Conv2d(256, 128, kernel_size=(1, 1), stride=(1,
1))
      )
      (1): ResBlock(
        (in_layers): Sequential(
          (0): GroupNorm32(32, 128, eps=1e-05, affine=True)
          (1): SiLU()
          (2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
        )
        (h_upd): Upsample()
        (x_upd): Upsample()
        (emb_layers): Sequential(
          (0): SiLU()
          (1): Linear(in_features=512, out_features=256, bias=True)
        )
        (out_layers): Sequential(
          (0): GroupNorm32(32, 128, eps=1e-05, affine=True)
          (1): SiLU()
          (2): Dropout(p=0.0, inplace=False)
          (3): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
        )
        (skip_connection): Identity()
      )
```

```
    )
    (10-11): 2 x TimestepEmbedSequential(
      (0): ResBlock(
        (in_layers): Sequential(
          (0): GroupNorm32(32, 256, eps=1e-05, affine=True)
          (1): SiLU()
          (2): Conv2d(256, 128, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
        )
        (h_upd): Identity()
        (x_upd): Identity()
        (emb_layers): Sequential(
          (0): SiLU()
          (1): Linear(in_features=512, out_features=256, bias=True)
        )
        (out_layers): Sequential(
          (0): GroupNorm32(32, 128, eps=1e-05, affine=True)
          (1): SiLU()
          (2): Dropout(p=0.0, inplace=False)
          (3): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
        )
        (skip_connection): Conv2d(256, 128, kernel_size=(1, 1), stride=(1,
1))
      )
    )
  )
  (out): Sequential(
    (0): GroupNorm32(32, 128, eps=1e-05, affine=True)
    (1): SiLU()
    (2): Conv2d(128, 6, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  )
)
```

(3) Now we need to implement the posterior sampling of DDPM as shown in line 4 in Algorithm 2. As you can notice that in the original sampling algorithm of DDPM, the model is trained to predict the noise ϵ.

Our pretrained model takes x_t and t as input and predict the noise ϵ.

What's more, our model is also trained to predict the variance σ_t. The model output is a torch tensor of shape (B, C, H, W), where B is the batch size, C is the number of channels and H, W are the height and width respectively. C here for our model is 6, with the first 3 channels for the noise prediction ϵ and the last 3 chennels for σ_t.

Implement the `p_sample` function of `DDPMDiffusion` in `guided_diffusion/simple_diffusion.py` for unconditional posterior sampling. Follow the sampling algorithm. Attach your code to the report.

(4) Now we have everything we need to perform unconditional sampling!

Implement the `p_sample_loop` of `DDPMDiffusion` in `simple_diffusion.py` for unconditional sampling, using the DDPM sampling algorithm.

(6) Run the code below to see what we can get from unconditional distillation. Include your results in your report.

In [5]:
```
from guided_diffusion.simple_diffusion import *
import torchvision
from util.img_utils import clear_color, mask_generator
from torchvision.transforms.functional import to_pil_image, pil_to_tensor
from util.img_utils import clear_color, mask_generator
from PIL import Image
```

In [6]:
```
diffusion_config = {
    'sampler': 'ddpm',
    'steps': 1000,
    'noise_schedule': 'linear',
    'model_mean_type': 'epsilon',
    'model_var_type': 'learned_range',
    'dynamic_threshold': False,
    'clip_denoised': True,
    'rescale_timesteps': False,
    'timestep_respacing': 1000}

sampler = create_sampler(**diffusion_config) # Instantiate DDPMDiffusion
sample_fn = partial(sampler.p_sample_loop, model=model, measurement_cond_fn=

x_start = torch.randn((1, 3, 256, 256), device=device)
out_path = os.path.join(save_dir, 'uncond_ddpm')
```

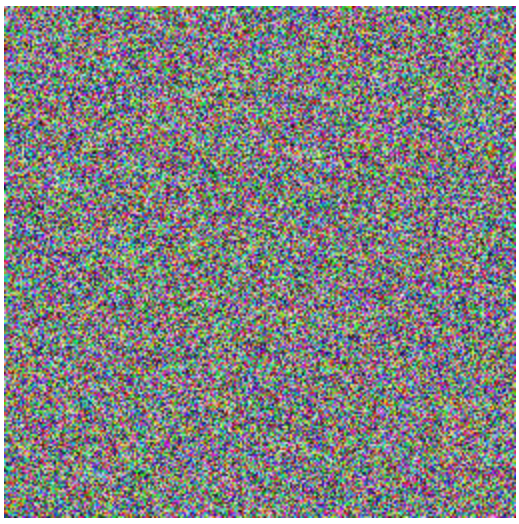
In [13]:
```
sample = sample_fn(x_start=x_start, measurement=None, record=True, save_root
```

```
  0%|          | 0/1000 [00:00<?, ?it/s]
100%|██████████| 1000/1000 [00:53<00:00, 18.86it/s]
```

We start from random noise of the same size as our output image:

In [15]:
```
torchvision.transforms.functional.to_pil_image((x_start[0] + 1)/2)
```

Out[15]:



We can then generate human faces with our pretrained model when given the noise input:

In [16]:
```
plt.imsave(os.path.join(out_path, 'output', '0.png'), clear_color(sample))
torchvision.transforms.functional.to_pil_image((sample[0] + 1)/2)
```

Out[16]:

Task 2: Unconditional Samping with DDIM

In this task, you will implement an improved sampling algorithm from Denoising Diffusion Implicit Models(DDIM) paper. DDIM sampling applies an improved update rule to sample from $p(x_{t-1}|x_t, x_0)$. The update rule is given by

Leveraging the above improved update rule, DDIM can be used to accelerate the sampling algorithm by only using a subset of the timesteps as before.

In `simple_diffusion.py` update the method `p_sample` under the class `DDIMDiffusion`, to implement the above update rule for DDIM sampling.

In [16]:
```
timestep_spacing = 100

diffusion_config = {
    'sampler': 'ddim',
    'steps': 1000,
    'noise_schedule': 'linear',
    'model_mean_type': 'epsilon',
    'model_var_type': 'learned_range',
    'dynamic_threshold': False,
    'clip_denoised': True,
    'rescale_timesteps': True,
    'timestep_respacing': f'ddim{timestep_spacing}'}


sampler = create_sampler(**diffusion_config)
sample_fn = partial(sampler.p_sample_loop, model=model, measurement_cond_fn=
```

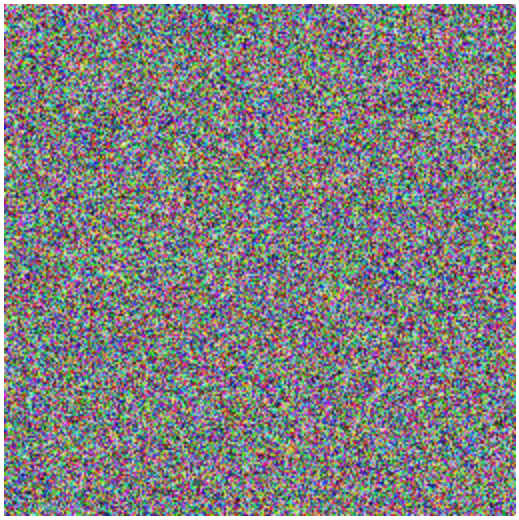
```
x_start = torch.randn((1, 3, 256, 256), device=device)
out_path = os.path.join(save_dir, 'uncond_ddim')
save_path = os.path.join(out_path, "progress")
os.makedirs(save_path, exist_ok=True)
```

In [12]:
```
sample = sample_fn(x_start=x_start, measurement=None, record=True, save_root
```

```
100%|██████████| 100/100 [00:05<00:00, 17.73it/s]
```

In [13]:
```
x_start_plot = to_pil_image((x_start[0] + 1)/2)
x_start_plot
```

Out[13]:



In [14]:
```
sample_plot = to_pil_image((sample[0] + 1)/2)
sample_plot
```

Out[14]:

Task 3: Inverse problem with RePaint

In this task, you will be applying the generative DDPM to solve an interesting problem of Image Inpainting. Image inpainting refers to filling out regions of the image that are unknown apriori. Here, we assume that a mask m indicating the known region is given to us.

Repaint Diffusion applies an update rule to the input image as shown below,

where the known region is sampled using

$$x_{t-1}^{known} \sim N\left(\bar{\alpha}_t x_0,\ (1-\bar{\alpha}_t)\, I\right)$$

and the unknown region is sampled from the diffusion model as

$$x_{t-1}^{unknown} \sim N\left(\mu_\theta(x_t,\ t),\ \Sigma_\theta(x_t,\ t)\right)$$

and the new sample x_{t-1} is obtained using

$$x_{t-1} = m \odot x_{t-1}^{known} + (1-m) \odot x_{t-1}^{unknown}$$

In `simple_diffusion.py` update the method `p_sample` under the class `Repaint`, to implement the above update rule for inpainting.

The summarized algorithm is given by

In [23]:
```
repaint_beta = get_named_beta_schedule('linear', 1000)

model_config = {
    'image_size': 256,
    'num_channels': 128,
    'num_res_blocks': 1,
    'channel_mult': "",
    'learn_sigma': True,
    'class_cond': False,
    'use_checkpoint': False,
    'attention_resolutions': 16,
    'num_heads': 4,
    'num_head_channels': 64,
    'num_heads_upsample': -1,
    'use_scale_shift_norm': True,
    'dropout': 0.0,
    'resblock_updown': True,
    'use_fp16': False,
    'use_new_attention_order': False,
    'model_path': 'models/ffhq_10m.pt'
}


diffusion_config = {
    'sampler': 'repaint',
    'steps': 1000,
    'noise_schedule': 'linear',
    'model_mean_type': 'epsilon',
    'model_var_type': 'learned_range',
    'dynamic_threshold': False,
```

```
    'clip_denoised': True,
    'rescale_timesteps': True,
    'timestep_respacing': 250}

repaint_conf = {
    "name": "face_example",
    "inpa_inj_sched_prev": True,
    "n_jobs": 1,
    "print_estimated_vars": True,
    "inpa_inj_sched_prev_cumnoise": False,
    "class_cond": False,
    "schedule_jump_params":{
        "t_T": 250,
        "n_sample": 1,
        "jump_length": 10,
        "jump_n_sample": 10,
    }
}

repaint_model = create_model(betas=repaint_beta, **model_config)
repaint_model = repaint_model.to(device)
repaint_model.eval()
print("Model Loaded")

gt_path = "data/datasets/gts/face/000000.png"
gt_mask_path = "data/datasets/gt_keep_masks/face/000000.png"
```

```
pretrained model loaded!
Model Loaded
```

In [24]:
```
model_kwargs_keys = ['gt', 'gt_keep_mask']

pil_gt_image = Image.open(gt_path)
gt_tensor = (pil_to_tensor(pil_gt_image) / 127.5 - 1.0).to(device = 'cuda')

pil_gt_mask = Image.open(gt_mask_path)
gt_mask_tensor = (pil_to_tensor(pil_gt_mask) / 255.0).to(device = 'cuda')

model_kwargs = {
    'gt': gt_tensor,
    'gt_keep_mask': gt_mask_tensor
}
```

In [25]:
```
sampler = create_sampler(**diffusion_config)
sample_fn = partial(sampler.p_sample_loop, model=repaint_model, shape=(1, 3,

x_start = torch.randn((1, 3, 256, 256), device=device)
out_path = os.path.join(save_dir, 'repaint')
save_path = os.path.join(out_path, "progress")
os.makedirs(save_path, exist_ok=True)
```

In [26]:
```
pil_gt_image
```

Out[26]:

In [27]:
```
pil_gt_mask
```

Out[27]:

In [28]:
```
sample = sample_fn(noise=x_start, progress = True)
to_pil_image((sample[0] + 1)/2)
```

```
100%|██████████| 4570/4570 [02:29<00:00, 30.51it/s]
```

Out[28]:

Task 4: Inverse problem with DPS

In this task, you will implement the sampling algorithm with posterior sampling using a pre-trained diffusion model. This is prettey much similar to the unconditional sampling algorithm except that we now are given the corrupted input. Let's implement the algorithm below.

The algorithm follows the same process as unconditional sampling. The only difference here is that we need to use the prior provided by the diffusion model and optimize the **image** directly by take taking the derivative in line 7.

(1) We have already implemented the conditional sampling part of `p_sample_loop` and `p_sample` of `DPSDiffusion` in `guided_diffusion/simple_diffusion.py` for conditional posterior sampling. But we do encourage you to take a look that into that.

(2)You will need to implment the `PosteriorSampling` in `guided_diffusion/condition_methods.py`.

(3) Run the following code. You should report one sample(including the raw image, corrupted image input and the algorithm output) of the inpainting task.

(4) [Optional] Play around with other task configurations and operate the algorithm to see how the results look like. Report one sample(including the raw image, corrupted image input and the algorithm output) of the following task: motion deblur, gaussain deblur and super resolution. Compare the results and discuss how the algorithm perform in each task. [Hint: Change `task_config` to paly with different tasks]

In [29]:
```
from guided_diffusion.measurements import get_noise, get_operator
from guided_diffusion.condition_methods import get_conditioning_method
from util.img_utils import clear_color, mask_generator
from PIL import Image
```

In [30]:
```
# Prepare dataloader
# data_config = task_config['data']
data_config = {
    'name': 'ffhq',
    'root': './data/samples/'}
transform = transforms.Compose([transforms.ToTensor(),
                                transforms.Normalize((0.5, 0.5, 0.5), (0.5,
dataset = get_dataset(**data_config, transforms=transform)
loader = get_dataloader(dataset, batch_size=1, num_workers=0, train=False)

# configuration of inpainting task
task_config_inpainting = {'conditioning':
            {'method': 'ps',
             'params': {'scale': 0.5}},
        'measurement':
            {'operator': {'name': 'inpainting'},
             'mask_opt':
             {'mask_type': 'random',
              'mask_prob_range': (0.3, 0.7),
```

```
              'image_size': 256},
              'noise': {'name': 'gaussian', 'sigma': 0.05}}
             }

# configuration of motion-deblur task
task_config_motion_deblur = {'conditioning':
            {'method': 'ps',
             'params': {'scale': 0.3}},
        'measurement':
            {'operator': {
                'name': 'motion_blur',
                'kernel_size': 61,
                'intensity': 0.5},
             'noise': {'name': 'gaussian', 'sigma': 0.05}}
             }

# configuration of gaussian-deblur task
task_config_gaussian_deblur = {'conditioning':
            {'method': 'ps',
             'params': {'scale': 0.3}},
        'measurement':
            {'operator': {
                'name': 'gaussian_blur',
                'kernel_size': 61,
                'intensity': 3.0},
             'noise': {'name': 'gaussian', 'sigma': 0.05}}
             }

# configuration of super resolution task
task_config_super_resolution = {'conditioning':
            {'method': 'ps',
             'params': {'scale': 0.3}},
        'measurement':
            {'operator': {
                'name': 'super_resolution',
                'in_shape': (1, 3, 256, 256),
                'scale_factor': 4},
             'noise': {'name': 'gaussian', 'sigma': 0.05}}
             }

task_config = task_config_inpainting

measure_config = task_config['measurement']
operator = get_operator(device=device, **measure_config['operator'])
noiser = get_noise(**measure_config['noise'])
logger.info(f"Operation: {measure_config['operator']['name']} / Noise: {meas

# Prepare conditioning method
cond_config = task_config['conditioning']
cond_method = get_conditioning_method(cond_config['method'], operator, noise
measurement_cond_fn = cond_method.conditioning
logger.info(f"Conditioning method : {task_config['conditioning']['method']}"

diffusion_config = {
    'sampler': 'dps',
    'steps': 1000,
```

```
    'noise_schedule': 'linear',
    'model_mean_type': 'epsilon',
    'model_var_type': 'learned_range',
    'dynamic_threshold': False,
    'clip_denoised': True,
    'rescale_timesteps': False,
    'timestep_respacing': 1000}

sampler = create_sampler(**diffusion_config)
sample_fn = partial(sampler.p_sample_loop, model=model, measurement_cond_fn=

out_path = os.path.join(save_dir, measure_config['operator']['name'])
os.makedirs(out_path, exist_ok=True)
for img_dir in ['input', 'recon', 'progress', 'label']:
    os.makedirs(os.path.join(out_path, img_dir), exist_ok=True)
```

```
2024-04-09 01:15:50,275 [DPS] >> Operation: inpainting / Noise: gaussian
2024-04-09 01:15:50,277 [DPS] >> Conditioning method : ps
DPS Initialized!
```

In [31]:

```
if measure_config['operator']['name'] == 'inpainting':
        mask_gen = mask_generator(
           **measure_config['mask_opt']
        )

for i, ref_img in enumerate(loader):
        logger.info(f"Inference for image {i}")
        fname = str(i).zfill(5) + '.png'
        ref_img = ref_img.to(device)

        # Exception) In case of inpainging,
        if measure_config['operator'] ['name'] == 'inpainting':
            mask = mask_gen(ref_img)
            mask = mask[:, 0, :, :].unsqueeze(dim=0)
            measurement_cond_fn = partial(cond_method.conditioning, mask=mas
            sample_fn = partial(sample_fn, measurement_cond_fn=measurement_c

            # Forward measurement model (Ax + n)
            y = operator.forward(ref_img, mask=mask)
            y_n = noiser(y)

        else:
            # Forward measurement model (Ax + n)
            y = operator.forward(ref_img)
            y_n = noiser(y)

        # Sampling
        x_start = torch.randn(ref_img.shape, device=device).requires_grad_()
        sample = sample_fn(x_start=x_start, measurement=y_n, record=True, sa

        plt.imsave(os.path.join(out_path, 'input', fname), clear_color(y_n))
        plt.imsave(os.path.join(out_path, 'label', fname), clear_color(ref_i
        plt.imsave(os.path.join(out_path, 'recon', fname), clear_color(sampl

        break
```

```
2024-04-09 01:15:54,288 [DPS] >> Inference for image 0
100%|██████████| 1000/1000 [01:50<00:00,  9.07it/s]
```

Now let's visualize some results.

Runnig the code below to visualize the raw image:

In [32]:
```
Image.open(os.path.join(out_path, 'label', '00000.png'))
```

Out[32]:

And here is the corrupted image by random masks:

In [33]:
```
Image.open(os.path.join(out_path, 'input', '00000.png'))
```

Out[33]:

Now let's see how our algorithm works:

In [34]:
```
Image.open(os.path.join(out_path, 'recon', '00000.png'))
```

Out[34]: