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() #
    # ######################################################################
```

```
    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_losses'
    #  (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_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()
```

```
total_ptime = end_time - start_time

return hist_D_losses, hist_G_losses, hist_G_L1_losses
```

1.2.2 Report Results

Figure 1: Result Visualization after 20 training epoches

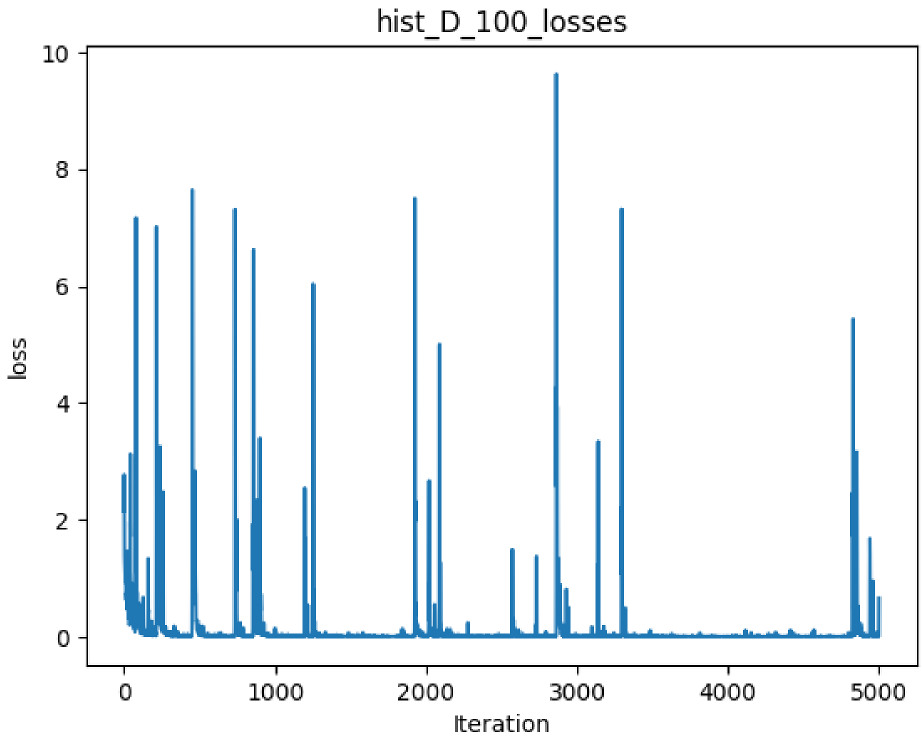

Figure 2: Discriminator BCE loss

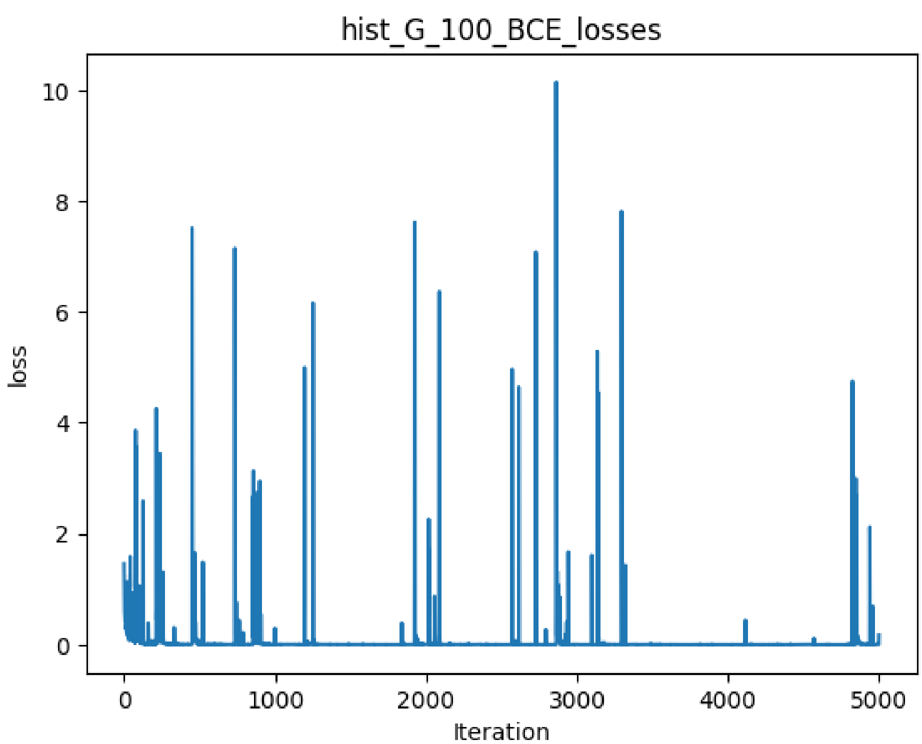

Figure 3: Generator BCE loss

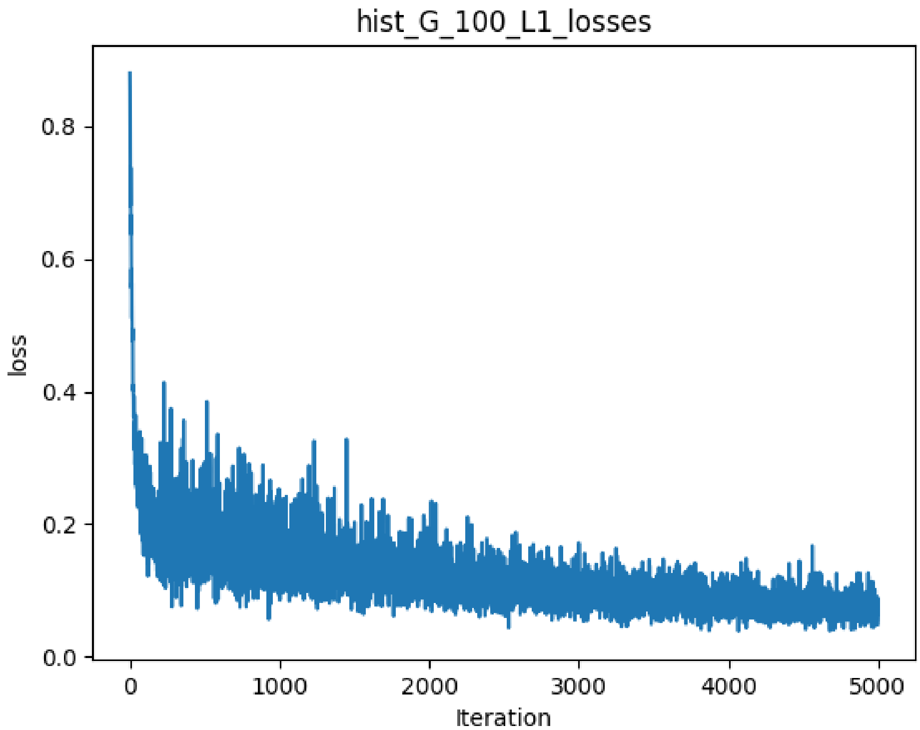

Figure 4: Generator L1 loss

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,
```

```python
                 **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
```

```
            x_start: torch.Tensor, random noise input
            measurement: torch.Tensor, our corrupted observation
            measurement_cond_fn: conditional function used to perform conditional sampling, is None for unconditional sampling
            record: Bool, save intermediate results if True
            save_root: str, root of the directoy to save the results
            uncond: Bool, perform unconditional sampling if True, else perform conditional sampling
        """
        if not uncond:
            assert measurement is not None and measurement_cond_fn is not None, \
                "measurement and measurement conditional function is required for conditional sampling"

        img = x_start   # start from random noise
        device = x_start.device

        ############ Start TODO #############
        # Implement the sample loop
        # Call p_sample for every iteration
        # It requires only one line of code implementation here

        pbar = tqdm(list(range(self.num_timesteps))[::-1])
        for idx in pbar:
            time = torch.tensor([idx] * img.shape[0], device=device)

            img = self.p_sample(model=model, x=img, t=time)['x_t_minus_1']

            img = img.detach_()

            if record:
                if idx % 10 == 0:
                    file_path = os.path.join(save_root, f"progress/x_{str(idx).zfill(4)}.png")
                    plt.imsave(file_path, clear_color(img))

        return img

    def p_sample(self, model, x, t):
        """
        Posterior sampling process, when given the model, x_t and timestep t, it returns predicted
        x_0 and x_t_minus_1

        We have already provide you with the function to get the log of the variance.
        Use self.var_processor.get_variance(var_values, t), where var_values is
        the 3:6 channels of the direct output of the model.
        example usage: log_variance = self.var_processor.get_variance(var_values, t)

```

```python
    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)


```

```python
    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)
```

```

        #############################################
        #####   TODO    #####
        ##### Here updated sample x_t_minus_one refers to the noisy image, where the known region is    #####
        ##### obtained by adding noise to GT, and the unknown region is obtained from                   #####
        ##### x_t_minus_one_unknown (which is the predicted sample from previous timestep) and the      #####
        #####  known and unknown region are combined using the ground truth mask (gt_keep_mask).        #####
        ##### Compelete the implementation to compute the updated sample (x_t_minus_one) for the        #####
        ##### timestep t. Make use of the variables gt_keep_mask and gt, to access the                  #####
        ##### ground-truth image. and the ground-truth mask.                                            #####
        #############################################

        if conf["inpa_inj_sched_prev"]:

            if pred_xstart is not None:

                gt_keep_mask = model_kwargs['gt_keep_mask']
                if gt_keep_mask is None:
                    gt_keep_mask = conf.get_inpa_mask(x_t_minus_one_unknown)

                gt = model_kwargs['gt']

                # Get x_t_minus_one_known
                if t > 1:
                    epsilon = torch.randn(x_t_minus_one_unknown.shape, device=x_t_minus_one_unknown.device)
                else:
                    epsilon = torch.zeros_like(x_t_minus_one_unknown, device=x_t_minus_one_unknown.device)
                alpha_bar = extract_and_expand(self.alphas_cumprod, t, x_t_minus_one_unknown)
                x_t_minus_one_known = torch.sqrt(alpha_bar) * gt + torch.sqrt(1 - alpha_bar) * epsilon

                # Get x_t_minus_one
                x_t_minus_one = gt_keep_mask * x_t_minus_one_known + (1 - gt_keep_mask) * x_t_minus_one_unknown


            else:
                x_t_minus_one = x_t_minus_one_unknown

        # TODO #####
        # One-step denoising using the model: Perform a forward pass on the model.
```

```
        # Remember to scale the timestep 't' using '_scale_timesteps' method.
        # 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.

        model_output = model.forward(x_t_minus_one, self._scale_timesteps(t))
        pred_score, var_values = torch.split(model_output, 3, dim=1)

        model_mean, pred_xstart = self.mean_processor.get_mean_and_xstart(x_t_minus_one, t, pred_score)
        log_variance = self.var_processor.get_variance(var_values, t)   # get the log  of variance

        ##############################################
        ##### TODO                               #####
        ##### Compute the noisy sample for timestep 't'     #####
        ##### You should use the 'log_variance' to calculate the variance of noise  #####
        ##### to be added.                       #####
        ##### Assign the sample to the variable 'sample'    #####
        ##############################################

        if t > 1:
            z = torch.randn(x_t_minus_one_unknown.shape, device=x_t_minus_one_unknown.device)
        else:
            z = torch.zeros_like(x_t_minus_one_unknown, device=x_t_minus_one_unknown.device)

        sample = model_mean + torch.sqrt(torch.exp(log_variance)) * z



        result = {"sample": sample,
                  "pred_xstart": pred_xstart, 'gt': model_kwargs.get('gt')}

        return result


    def p_sample_loop(
        self,
        model,
        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

Submitted by Wensong Hu on April 4th, 2024.