DDPM Model Report

Model review

Diffusion models, such as DDPM, are a class of generative models that learn to produce new data samples by simulating a data-generating process. DDPM, which stands for "Denoising Diffusion Probabilistic Models", is a particular type of diffusion model that uses a Markov chain to gradually add noise to a data sample until it is completely obscured.

The model then learns to reverse this process, starting from a random noise vector and removing the noise in a series of steps to generate a new data sample. DDPM consists of a forward process and a reverse process. The forward process is a fixed procedure that adds noise to the data sample in a series of steps. The reverse process is a learned procedure that gradually removes the noise from the data sample. The reverse process is learned using a neural network, such as a U-Net, that is trained to predict the noise to be removed at each step based on the current state of the data sample.

Review of code files

unet_base.py

Components

The *unet_base.py* defines a set of PyTorch modules for a modified U-Net architecture with attention and time embedding. The key components are:

1. Time Embedding:

The get_time_embedding function generates a sinusoidal time embedding based on the input time steps.

2. DownBlock:

   This is a part of the U-Net architecture, which is responsible for down-sampling the feature map and increasing the number of feature maps. It contains four main components:

   - A list of residual convolutional layers (num_layers)
   - Time embedding layers (num_layers)
   - Residual input convolutional layers (num_layers)
   - A multi-head attention block (num_layers)
   - Down-sampling convolution (if down_sample is True)

3. MidBlock:

   This middle part of the U-Net applies the same attention mechanism as DownBlock.

4. UpBlock:

   This part of the U-Net up-samples the input feature tensor and decreases the number of feature maps. It consists of the following blocks:

   - Upsampling using '*ConvTranspose2d*'
   - A list of residual convolutional layers (num_layers) Time embedding layers (num_layers)
   - Residual input convolutional layers (num_layers)
   - Multi-head attention block (num_layers)

5. Unet:

   This is the primary class, which brings together the DownBlock, MidBlock, and UpBlock. The architecture is as follows:

   - Convolutional layer (im_channels -> down_channels[0])
   - DownBlock (num_down_layers)
   - MidBlock (num_mid_layers)
   - UpBlock (num_up_layers) (reversed and skip-connections)

Snippets of the code file are attached for better understanding:

```python
class Unet(nn.Module):
    r"""
    Unet model comprising
    Down blocks, Midblocks and Uplocks
    """
    def __init__(self, model_config):
        super().__init__()
        im_channels = model_config['im_channels']
        self.down_channels = model_config['down_channels']
        self.mid_channels = model_config['mid_channels']
        self.t_emb_dim = model_config['time_emb_dim']
        self.down_sample = model_config['down_sample']
        self.num_down_layers = model_config['num_down_layers']
        self.num_mid_layers = model_config['num_mid_layers']
        self.num_up_layers = model_config['num_up_layers']

        assert self.mid_channels[0] == self.down_channels[-1]
        assert self.mid_channels[-1] == self.down_channels[-2]
        assert len(self.down_sample) == len(self.down_channels) - 1

        # Initial projection from sinusoidal time embedding
        self.t_proj = nn.Sequential(
            nn.Linear(self.t_emb_dim, self.t_emb_dim),
            nn.SiLU(),
            nn.Linear(self.t_emb_dim, self.t_emb_dim)
        )

        self.up_sample = list(reversed(self.down_sample))
        self.conv_in = nn.Conv2d(im_channels, self.down_channels[0], kernel_size=3, padding=(1, 1))

        self.downs = nn.ModuleList([])
        for i in range(len(self.down_channels)-1):
            self.downs.append(DownBlock(self.down_channels[i], self.down_channels[i+1], self.t_emb_dim,
                                        down_sample=self.down_sample[i], num_layers=self.num_down_layers))

        self.mids = nn.ModuleList([])
        for i in range(len(self.mid_channels)-1):
            self.mids.append(MidBlock(self.mid_channels[i], self.mid_channels[i+1], self.t_emb_dim,
                                      num_layers=self.num_mid_layers))

        self.ups = nn.ModuleList([])
        for i in reversed(range(len(self.down_channels)-1)):
            self.ups.append(UpBlock(self.down_channels[i] * 2, self.down_channels[i-1] if i != 0 else 16,
                                    self.t_emb_dim, up_sample=self.down_sample[i], num_layers=self.num_up_layers))

        self.norm_out = nn.GroupNorm(8, 16)
        self.conv_out = nn.Conv2d(16, im_channels, kernel_size=3, padding=1)
```

I/O

The input to the Unet model is a tensor x of shape B x C x H x W, where B is the batch size, C is the number of channels, H is the height, and W is the width. The input is passed through an initial convolutional layer, and then through a series of downsampling, middle, and upsampling blocks.

The output of the Unet model is a tensor of shape B x C x H x W, where C is the number of output channels. The Unet model also takes a time tensor t as input, which is used to condition the model's output. The time tensor t is first converted into an embedding using the get_time_embedding function, and then passed through a projection layer to obtain the time embedding t_emb. The time embedding t_emb is then used in the downsampling, middle, and upsampling blocks to condition the model's output.

mnist_dataset.py

Components

The *mnist_dataset.py* defines a custom dataset class for your images. The class, named MnistDataset, is designed to allow for easy replacement of the source dataset. This class inherits the PyTorch Dataset class and includes functionalities to load, preprocess, and transform the images.

The primary parts of the code are

- Initialization:

  In the __init__ method, the dataset loads the image files and their respective labels. The user can specify the dataset split (train/test), image path, and the expected image file extension.

- Number of images:

  The __len__ method returns the number of images present in the dataset. This works in tandem with the DataLoader to determine the number of iterations (batches) during the training process.

- Accessing images:

  The __getitem__ method retrieves the desired image from the dataset. It performs the following steps:

  1. Loads the image file from the specified path.
  2. Performs a center square crop of size 2448x2448.
  3. Resizes the image to a size of 64x64.
  4. Converts the image to a PyTorch tensor.
  5. Applies normalization using the ImageNet (RGB) statistics (mean: [0.5, 0.5, 0.5], std: [0.5, 0.5, 0.5]).

Snippets of the code file are attached for better understanding:

```python
class MnistDataset(Dataset):
    r"""
    Nothing special here. Just a simple dataset class for mnist images.
    Created a dataset class rather using torchvision to allow
    replacement with any other image dataset
    """
    def __init__(self, split, im_path, im_ext='png'):
        r"""
        Init method for initializing the dataset properties
        :param split: train/test to locate the image files
        :param im_path: root folder of images
        :param im_ext: image extension. assumes all
        images would be this type.
        """
        self.split = split
        self.im_ext = im_ext
        self.images, self.labels = self.load_images(im_path)
```

```python
def load_images(self, im_path):
    r"""
    Gets all JPG images from the path specified
    and stacks them all up
    :param im_path: Path to the directory containing images
    :return: List of image file paths and corresponding labels
    """
    assert os.path.exists(im_path), "Images path {} does not exist".format(im_path)
    ims = []
    labels = []
    for d_name in tqdm(os.listdir(im_path)):
      for fname in glob.glob(os.path.join(im_path, d_name, '*.jpg')):
        ims.append(fname)
        #labels.append(int(d_name))
    print('Found {} images for split {}'.format(len(ims), self.split))
    return ims, labels
```

```python
def __len__(self):
    return len(self.images)

def __getitem__(self, index):
    im = Image.open(self.images[index])

    # Perform center square crop of size 2448x2448
    width, height = im.size
    left = (width - 2448) // 2
    top = (height - 2448) // 2
    right = (width + 2448) // 2
    bottom = (height + 2448) // 2
    im = im.crop((left, top, right, bottom))

    # Resize to 64x64
    im = im.resize((64, 64), Image.ANTIALIAS)

    # Convert to tensor and normalize
    transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5])  # For RGB images
    ])
    im_tensor = transform(im)

    return im_tensor
```

I/O

The class takes three parameters during initialization:

1. split (a string to specify whether to load the training or testing set)
2. im_path (a string specifying the root folder of images)
3. im_ext (a string specifying the image file extension)

The methods of class have the following I/O:

1. __init__: Initializes the dataset by loading the images from the specified path.
2. load_images: A helper method to read all the image files in the given directory and return them as a list of file paths and a list of corresponding labels.
3. __len__: Returns the total number of images in the dataset.
4. __getitem__: Returns the image and its corresponding label from the dataset given an index.

LinearNoiseScheduler

Components

The *linear_noise_scheduler.py* file defines a PyTorch-based LinearNoiseScheduler class, which is used for noise scheduling in the Denoising Diffusion Probabilistic Models (DDPM). This architecture introduces a markov chain to convert the data to noise gradually. Then, it trains a neural network to predict the noise and utilizes that network to recover the original data.

The primary parts of the 'LinearNoiseScheduler' class are:

- Initialization (__init__ method):

  In this method, the linear noise schedule is created by calculating the values of betas, alphas, alpha_cum_prod, sqrt_alpha_cum_prod, and sqrt_one_minus_alpha_cum_prod.

- add_noise method:

  This method is responsible for applying noise to the original input. With the given t index, the corresponding sqrt_alpha_cum_prod and sqrt_one_minus_alpha_cum_prod is fetched to apply diffusion and produce a noisy version of the original input.

- sample_prev_timestep method:

  This method samples the value of the previous timestep using the noise prediction from the model, xt, and the current time step (t). The input is the original data corrupted with Gaussian noise, and the output is the denoised output.

- The Linear Noise Scheduler is used in DDPM for training and sampling, as explained below:

  In training, the objective is to minimize the mean squared error between the network's predicted noise and the actual noise added to the original data.

  During sampling, the goal is to reverse the Markov chain and convert the noise back to the original data. This starts from a random noise and recursively applies the sampled_prev_timestep until the original_data is recovered.

Snippets of the code file are attached for better understanding:

```python
def add_noise(self, original, noise, t):
    r"""
    Forward method for diffusion
    :param original: Image on which noise is to be applied
    :param noise: Random Noise Tensor (from normal dist)
    :param t: timestep of the forward process of shape -> (B,)
    :return:
    """
    original_shape = original.shape
    batch_size = original_shape[0]

    sqrt_alpha_cum_prod = self.sqrt_alpha_cum_prod.to(original.device)[t].reshape(batch_size)
    sqrt_one_minus_alpha_cum_prod = self.sqrt_one_minus_alpha_cum_prod.to(original.device)[t].reshape(batch_size)

    # Reshape till (B,) becomes (B,1,1,1) if image is (B,C,H,W)
    for _ in range(len(original_shape) - 1):
        sqrt_alpha_cum_prod = sqrt_alpha_cum_prod.unsqueeze(-1)
    for _ in range(len(original_shape) - 1):
        sqrt_one_minus_alpha_cum_prod = sqrt_one_minus_alpha_cum_prod.unsqueeze(-1)

    # Apply and Return Forward process equation
    return (sqrt_alpha_cum_prod.to(original.device) * original
            + sqrt_one_minus_alpha_cum_prod.to(original.device) * noise)
```

```python
def sample_prev_timestep(self, xt, noise_pred, t):
    r"""
        Use the noise prediction by model to get
        xt-1 using xt and the noise predicted
    :param xt: current timestep sample
    :param noise_pred: model noise prediction
    :param t: current timestep we are at
    :return:
    """
    x0 = ((xt - (self.sqrt_one_minus_alpha_cum_prod.to(xt.device)[t] * noise_pred)) /
          torch.sqrt(self.alpha_cum_prod.to(xt.device)[t]))
    x0 = torch.clamp(x0, -1., 1.)

    mean = xt - ((self.betas.to(xt.device)[t]) * noise_pred) / (self.sqrt_one_minus_alpha_cum_prod.to(xt.device)[t])
    mean = mean / torch.sqrt(self.alphas.to(xt.device)[t])

    if t == 0:
        return mean, x0
    else:
        variance = (1 - self.alpha_cum_prod.to(xt.device)[t - 1]) / (1.0 - self.alpha_cum_prod.to(xt.device)[t])
        variance = variance * self.betas.to(xt.device)[t]
        sigma = variance ** 0.5
        z = torch.randn(xt.shape).to(xt.device)

        # OR
        # variance = self.betas[t]
        # sigma = variance ** 0.5
        # z = torch.randn(xt.shape).to(xt.device)
        return mean + sigma * z, x0
```

I/O

The LinearNoiseScheduler class in this code takes in three parameters:

1. num_timesteps: An integer that represents the number of time steps in the diffusion process.
2. beta_start: A scalar value that represents the starting value for the linear noise schedule.

3. beta_end: A scalar value that represents the ending value for the linear noise schedule.

The methods of LinearNoiseScheduler class have the following I/O:

__init__:

The constructor method takes in the three parameters and initializes the class variables. It calculates the betas, alphas, cumulative product of alphas, and square roots of these cumulative products.

add_noise:

This method takes in three parameters:

1. original: A tensor that represents the image to which noise is to be added.
2. noise: A tensor that represents the random noise drawn from the normal distribution.
3. t: An integer that represents the time step in the forward process.

The method returns the noisy image obtained by applying the forward process equation to the original image.

sample_prev_timestep:

This method takes in three parameters:

1. xt: A tensor that represents the current time step sample.
2. noise_pred: A tensor that represents the noise prediction made by the model.
3. t: An integer that represents the current time step.

The method returns the previous time step's sample and x0 calculated using the noise prediction, the current time step, and the previous time step's alpha. If t is 0, the method returns the mean and x0 only. If t is greater than 0, the method returns the mean and x0 after calculating the variance and the standard deviation.

train_ddpm.py

Components

The *train_ddpm.py* file is a PyTorch implementation of a training script for Denoising Diffusion Probabilistic Models (DDPM) using a U-Net architecture with a time scheduler and noise addition mechanism. DDPM is a class of generative models that applies a

progressive denoising procedure to gradually convert random noise into concrete data by a learned process.

The script includes several components necessary for DDPM training:

1. Data Loading:

   It leverages the MNIST dataset to train the model, with configurations such as the image path specified in the config file.

2. Neural Network:

   The Unet class implements the U-Net-based architecture for DDPM, with specific configurations and hyperparameters defined in the model_params section of the config file.

3. Noise Scheduler:

   The script uses a linear noise scheduler (LinearNoiseScheduler class) to progressively convert data to noise and recover original data. This is done by calculating the values of betas, alphas, alpha_cum_prod, sqrt_alpha_cum_prod, and sqrt_one_minus_alpha_cum_prod in the __init__ method.

4. Training Procedure:

   The main train function reads config parameters, instantiates the dataset, model, noise scheduler, and handles model optimization and checkpoints. The loop continues for the specified number of epochs, with each iteration involving the following steps:

   - Sample random noise
   - Sample a random time step
   - Add noise to images using the selected time step
   - Calculate the loss between the predicted and the actual noise
   - Perform backpropagation

Snippets of the code file are attached for better understanding:

```python
def train(args):
    # Read the config file #
    with open(args.config_path, 'r') as file:
        try:
            config = yaml.safe_load(file)
        except yaml.YAMLError as exc:
            print(exc)
    print(config)
    ########################

    diffusion_config = config['diffusion_params']
    dataset_config = config['dataset_params']
    model_config = config['model_params']
    train_config = config['train_params']

    # Create the noise scheduler
    scheduler = LinearNoiseScheduler(num_timesteps=diffusion_config['num_timesteps'],
                                     beta_start=diffusion_config['beta_start'],
                                     beta_end=diffusion_config['beta_end'])

    # Create the dataset
    mnist = MnistDataset('train', im_path=dataset_config['im_path'])
    mnist_loader = DataLoader(mnist, batch_size=train_config['batch_size'], shuffle=True, num_workers=4)

    # Instantiate the model
    model = Unet(model_config).to(device)
    model.train()

    # Create output directories
    if not os.path.exists(train_config['task_name']):
        os.mkdir(train_config['task_name'])
```

```python
    # Load checkpoint if found
    if os.path.exists(os.path.join(train_config['task_name'],train_config['ckpt_name'])):
        print('Loading checkpoint as found one')
        model.load_state_dict(torch.load(os.path.join(train_config['task_name'],
                                                      train_config['ckpt_name']), map_location=device))
    # Specify training parameters
    num_epochs = train_config['num_epochs']
    optimizer = Adam(model.parameters(), lr=train_config['lr'])
    criterion = torch.nn.MSELoss()

    # Run training
    for epoch_idx in range(num_epochs):
        losses = []
        for im in tqdm(mnist_loader):
            optimizer.zero_grad()
            im = im.float().to(device)

            # Sample random noise
            noise = torch.randn_like(im).to(device)

            # Sample timestep
            t = torch.randint(0, diffusion_config['num_timesteps'], (im.shape[0],)).to(device)

            # Add noise to images according to timestep
            noisy_im = scheduler.add_noise(im, noise, t)
            noise_pred = model(noisy_im, t)

            loss = criterion(noise_pred, noise)
            losses.append(loss.item())
            loss.backward()
            optimizer.step()
        print('Finished epoch:{} | Loss : {:.4f}'.format(
            epoch_idx + 1,
            np.mean(losses),
        ))
        torch.save(model.state_dict(), os.path.join(train_config['task_name'],
                                                    train_config['ckpt_name']))

    print('Done Training ...')
```

I/O

Inputs:

1. The primary input is a configuration YAML file specified by --config (default: config/default.yaml). The config file contains various parameters needed for the dataset, model, and training.
2. The training process uses your custom dataset (as implemented in the dataset.mnist_dataset package) and a custom denoising diffusion model (implemented in the models.unet_base package).

Outputs:

1. The trained model is saved in the specified task directory, specified in the train_config['task_name'] parameter. The script uses the initial checkpoint (if available) and saves the model's state dictionary after completing the training.
2. The script prints training progress to the console, including the current epoch number, loss for the epoch, and other metrics.

In essence, the aim of this script is to train a denoising diffusion model using your custom dataset for a specified number of epochs. It stores the trained model and its corresponding metrics for future use.

sample_ddpm.py

Components

This Python code is the implementation of the Denoising Diffusion Probabilistic Models (DDPM) inference procedure on the dataset using a U-Net model and a linear noise scheduler.

The script consists of the following main steps:

1. Import required modules and tools.

2. Define functions to train and infer the DDPM model:

    - sample():

      This function takes the trained DDPM model, the linear noise scheduler, the training configurations, the U-Net model configurations, and the

DDPM configurations as inputs. It loops through the noise schedule, predicts the noise at each time step, and recovers the initial image from the previous time step using the scheduler. The sampled images are saved on disk as you go backward in time.

- infer():

  This function reads the configuration file, loads the trained DDPM model, initializes the noise scheduler, and processes the inference using the sample function.

3. Parse command line arguments to input the path of the configuration file.

4. Run the inference by processing the content of the configuration file, which contains the U-Net architecture, diffusion parameters, and training parameters.

5. The DDPM inference procedure in the script performs the following steps:
   1) Load the pre-trained U-Net model.
   2) Instantiate the linear noise scheduler according to the provided diffusion parameters.
   3) Reverse the time steps and, for each time step:

      a. Generate a noise tensor with the current time step.
      b. Predict the noise at this time step using the model.
      c. Recover the image from the previous time step using the scheduler.
      d. Save the predicted image.

Finally, it calls the inference function to run the DDPM inference procedure on the provided dataset.

Snippets of the code file are attached for better understanding:

```python
def sample(model, scheduler, train_config, model_config, diffusion_config):
    r"""
    Sample stepwise by going backward one timestep at a time.
    We save the x0 predictions
    """
    xt = torch.randn((train_config['num_samples'],
                      model_config['im_channels'],
                      model_config['im_size'],
                      model_config['im_size'])).to(device)
    for i in tqdm(reversed(range(diffusion_config['num_timesteps']))):
        # Get prediction of noise
        noise_pred = model(xt, torch.as_tensor(i).unsqueeze(0).to(device))

        # Use scheduler to get x0 and xt-1
        xt, x0_pred = scheduler.sample_prev_timestep(xt, noise_pred, torch.as_tensor(i).to(device))

        # Save x0
        ims = torch.clamp(xt, -1., 1.).detach().cpu()
        ims = (ims + 1) / 2
        grid = make_grid(ims, nrow=train_config['num_grid_rows'])
        img = torchvision.transforms.ToPILImage()(grid)
        if not os.path.exists(os.path.join(train_config['task_name'], 'samples')):
            os.mkdir(os.path.join(train_config['task_name'], 'samples'))
        img.save(os.path.join(train_config['task_name'], 'samples', 'x0_{}.png'.format(i)))
        img.close()


def infer(args):
    # Read the config file #
    with open(args.config_path, 'r') as file:
        try:
            config = yaml.safe_load(file)
        except yaml.YAMLError as exc:
            print(exc)
    print(config)
    ########################
```

I/O

Inputs:

1. The primary input is a configuration YAML file specified by -config (default: config/default.yaml). The config file contains various parameters needed for the dataset, model, and training.
2. The training process uses your custom dataset (as implemented in the dataset.mnist_dataset package) and a custom denoising diffusion model (implemented in the models.unet_base package).

Outputs:

1. The trained model is saved in the specified task directory, specified in the train_config['task_name'] parameter. The script uses the initial checkpoint (if available) and saves the model's state dictionary after completing the training.
2. The script prints training progress to the console, including the current epoch number, loss for the epoch, and other metrics.

In essence, the aim of this script is to train a denoising diffusion model using the MNIST dataset for a specified number of epochs. It stores the trained model and its corresponding metrics for future use.

Model Training

Dataset

We utilized the dataset provided to us.

Parameters

Diffusion

- Number of Timesteps: 1000
- Beta Start: 0.0001
- Beta End: 0.02

Model

- Image Channels: 3
- Image Size: 64
- Down Channels: [32, 64, 128, 256]
- Mid Channels: [256, 256, 128]
- Downsampling: [True, True, False]
- Time Embedding Dimension: 128
- Number of Down Layers: 2
- Number of Mid Layers: 2
- Number of Up Layers: 2
- Number of Heads: 4

Training

- Task Name: 'default'
- Batch Size: 6
- Number of Epochs: 400
- Number of Samples: 3
- Number of Grid Rows: 10
- Learning Rate: 0.0001
- Checkpoint Name: 'ddpm_ckpt.pth'

Results

After training the model with the provided dataset and parameters, we achieved the following results:

- Generated Images Resolution: 64x64 pixels
- Accuracy: 2 out of 3 images in a batch were accurate.

These results indicate the model generated images at the desired resolution, with a notable accuracy rate in the generated images.

Below attracted is a picture of the final out at timestep 0