Deep Learning Homework 6 Report

Cheng-Liang Chi

May 6, 2025

Contents

1 Introduction

In this lab we build a conditional Denoising Diffusion Probabilistic Model that turns a set of color-shape phrases (e.g., “red sphere”, “yellow cube”) into a 64×64 scene containing exactly the requested objects. We will use a pretrained ResNet-18 model to evaluate the accuracy of the generated images.

We will use the `CLEVR` dataset, which contains 3D shapes and their corresponding color-shape phrases. The dataset is available here.

2 Implementation Details

For this homework, we implemented a conditional Denoising Diffusion Probabilistic Model (DDPM) using PyTorch. The model is designed to generate images based on color-shape phrases, such as "red sphere" or "yellow cube". The implementation consists of several key components, including the model architecture, training process, and evaluation metrics.

I modulized the code into several files to improve readability and maintainability. The main components of the implementation are as follows:

- `model.py`: This file contains the implementation of the conditional DDPM model. Most of the model architecture is using Hugging Face Diffusers library [1], which provides a high-level interface for building and training diffusion models.
- `dataset.py`: This file defines the dataset class for loading and preprocessing the training data, including color-shape phrases and corresponding images.
- `train.py`: This file contains the training loop, including the optimization process and logging of training metrics.
- `test.py`: This file implements the evaluation process, including generating images from the trained model and evaluating the generated image using the pretrained ResNet-18 classifier provided by the TA.

I will provide a brief overview of each component in the following sections.

2.1 Model Architecture

The model architecture is based on a U-Net structure, which is commonly used in image generation tasks. The U-Net consists of an encoder-decoder structure with skip connections, allowing the model to capture both local and global features in the image.

Since the color-shape phrase (label) is a limited set of words, I used one-hot encoding to represent the phrases. The one-hot encoded vector is then using a linear layer to project it into a higher-dimensional space, which is then provided as an additional input to the U-Net model.

When dealing with higher-resolution inputs it is reasonable to use more down and up-blocks, and keep the attention layers only at the lowest resolution (bottom) layers to reduce memory usage. However, in this homework, I used a smaller model with only 6 down and up-blocks to keep the training time reasonable and also get good results.

The implementation of the U-Net model is based on the Hugging Face Diffusers library, which provides a high-level interface for building and training diffusion models. The implementation of the U-Net model is as follows:

```python
class ConditionalDDPM(nn.Module):
    def __init__(self, num_classes=24, dim=512):
        super().__init__()
        channel = dim // 4
        self.UNet = UNet2DModel(
            sample_size=64,
            in_channels=3,
            out_channels=3,
            layers_per_block=2,
            block_out_channels=[
                channel,
                channel,
                channel * 2,
                channel * 2,
            ],  # type: ignore
            down_block_types=[
                "DownBlock2D",
                "DownBlock2D",
                "DownBlock2D",
                "AttnDownBlock2D",
            ],  # type: ignore
            up_block_types=[
                "AttnUpBlock2D",
                "UpBlock2D",
                "UpBlock2D",
                "UpBlock2D",
            ],  # type: ignore
            class_embed_type="identity",
        )
        self.label_embedding = nn.Linear(num_classes, dim)

    def forward(self, x, t, label):
        embedding_label = self.label_embedding(label)
        return self.UNet(x, t, embedding_label).sample
```

2.2 Training Process

The U-Net model is trained using a denoising diffusion process, where the model learns to predict the noise added to the image at each time step. The training process involves optimizing the model parameters using a combination of mean squared error (MSE) loss.

The training loop consists of the following steps:

- Load a batch of images and corresponding color-shape phrases from the dataset.
- For each image, add Gaussian noise to the image at a random time step.
- Pass the noisy image and the one-hot encoded phrase through the U-Net model to predict the noise.
- Compute the loss between the predicted noise and the actual noise added to the image.

- Backpropagate the loss and update the model parameters using an optimizer (AdamW).

The training process is implemented in the `train.py` file, which contains the training loop and logging of training metrics. The implementation of the training loop is as follows:

```
class Trainer:
    def __init__(self, args):
        self.train_loader = DataLoader(
            IclevrDataset(args.dataset, "train"),
            batch_size=args.batch_size,
            shuffle=True,
            num_workers=args.num_workers,
        )
        self.val_loader = DataLoader(
            IclevrDataset(args.dataset, "test"),
            batch_size=1,
            shuffle=False,
            num_workers=args.num_workers,
        )

        self.criteria = nn.MSELoss()
        self.model = ConditionalDDPM().to(args.device)
        self.noise_scheduler =
        ↪  DDPMScheduler(num_train_timesteps=args.time_steps)
        self.time_steps = args.time_steps

        self.optimizer = torch.optim.Adam(self.model.parameters(),
        ↪  lr=args.lr)
        self.epochs = args.epochs
        self.save_dir = args.save_dir
        self.save_ckpt_period = args.save_ckpt_period
        self.save_img_period = args.save_img_period
        self.device = args.device
        self.writer = SummaryWriter()

        self.best_loss: float = float("inf")
        self.epoch = 0

        os.makedirs(self.save_dir, exist_ok=True)
        os.makedirs(os.path.join(self.save_dir, "imgs"), exist_ok=True)

    def save_checkpoint(self, path):
        torch.save(
            {
                "epoch": self.epoch,
                "best_loss": self.best_loss,
                "model": self.model.state_dict(),
                "optimizer": self.optimizer.state_dict(),
                "time_steps": self.time_steps,
            },
            path,
```

```
        )
        tqdm.write(f"Model saved at {path}")

    def train(self):
        for epoch in trange(self.epochs, desc="Training",
        ↪  dynamic_ncols=True):
            self.epoch = epoch
            train_loss = self.train_one_epoch()
            self.writer.add_scalar("Loss/train", train_loss, epoch)
            tqdm.write(f"Epoch {epoch}, Train Loss: {train_loss}")

            if (epoch + 1) % self.save_ckpt_period == 0:
                self.save_checkpoint(
                    os.path.join(self.save_dir, f"model_epoch_{epoch}.pth")
                )

            if (epoch + 1) % self.save_img_period == 0:
                self.save_images()

    def train_one_epoch(self):
        self.model.train()
        train_loss = []
        for i, (img, label) in enumerate(
            tqdm(
                self.train_loader,
                desc=f"Epoch: {self.epoch}",
                dynamic_ncols=True,
            )
        ):
            batch_size = img.shape[0]
            img, label = img.to(self.device), label.to(self.device)
            noise = torch.randn_like(img)

            timesteps = (
                torch.randint(0, self.time_steps,
                ↪  (batch_size,)).long().to(self.device)
            )
            noisy_x = self.noise_scheduler.add_noise(img, noise, timesteps)
            ↪  # type: ignore
            output = self.model(noisy_x, timesteps, label)

            loss = self.criteria(output, noise)

            self.optimizer.zero_grad()
            loss.backward()
            self.optimizer.step()

            train_loss.append(loss.item())

            self.writer.add_scalar(
```

```
                "Loss/train-step", loss.item(), self.epoch *
                ↪ len(self.train_loader) + i
            )

        os.makedirs(
            os.path.join(self.save_dir, "imgs", f"ep{self.epoch}"),
            ↪ exist_ok=True
        )
        self.writer.add_scalar("Loss/epoch", np.mean(train_loss),
        ↪ self.epoch)
        return np.mean(train_loss)

    def save_images(self):
        self.model.eval()

        for idx, (y, label) in enumerate(
            tqdm(self.val_loader, desc=f"Epoch: {self.epoch}",
            ↪ dynamic_ncols=True)
        ):
            y = y.to(self.device)
            x = torch.randn(1, 3, 64, 64).to(self.device)
            denoising_result = []
            for i, t in enumerate(self.noise_scheduler.timesteps):
                with torch.no_grad():
                    residual = self.model(x, t, y)

                x = self.noise_scheduler.step(residual, t, x).prev_sample
                ↪ # type: ignore
                if i % (len(self.noise_scheduler.timesteps) // 10) == 0:
                    denoising_result.append(x.squeeze(0))

            denoising_result.append(x.squeeze(0))
            denoising_result = torch.stack(denoising_result)
            row_image = make_grid(
                (denoising_result + 1) / 2, nrow=denoising_result.shape[0],
                ↪ pad_value=0
            )
            save_image(
```

2.3 Evaluation Process

The evaluation process involves generating images from the trained model and evaluating the generated images using a pretrained ResNet-18 classifier. The evaluation process consists of the following steps:

- Load the trained model and the pretrained ResNet-18 classifier.
- Generate images from the trained model using a set of color-shape phrases.

- Evaluate the generated images using the pretrained ResNet-18 classifier to compute the accuracy of the generated images.
- Save the generated images and their corresponding accuracy scores.

The evaluation process is implemented in the `test.py` file, which contains the evaluation loop and logging of evaluation metrics. The implementation of the evaluation loop is as follows:

```python
class Tester:
    def __init__(self, args):
        self.test_loader = DataLoader(
            IclevrDataset(args.dataset, "test"),
            batch_size=1,
            shuffle=False,
            num_workers=args.num_workers,
        )
        self.new_test_loader = DataLoader(
            IclevrDataset(args.dataset, "new_test"),
            batch_size=1,
            shuffle=False,
            num_workers=args.num_workers,
        )
        self.manual_test_loader = DataLoader(
            IclevrDataset(args.dataset, "manual_test"),
            batch_size=1,
            shuffle=False,
            num_workers=args.num_workers,
        )

        self.device = args.device
        self.model = ConditionalDDPM().to(self.device)
        self.model.load_state_dict(torch.load(args.ckpt)["model"])
        self.model.eval()

        self.eval_model = evaluation_model()

        self.time_steps = torch.load(args.ckpt)["time_steps"]
        self.noise_scheduler =
        ↪  DDPMScheduler(num_train_timesteps=self.time_steps)

        self.save_dir = args.save_dir
        os.makedirs(self.save_dir, exist_ok=True)
        os.makedirs(os.path.join(self.save_dir, "test"), exist_ok=True)
        os.makedirs(os.path.join(self.save_dir, "new_test"), exist_ok=True)
        os.makedirs(os.path.join(self.save_dir, "manual_test"),
        ↪  exist_ok=True)

    def test(self):
        test_acc = self.inference(
            self.test_loader,
```

```
            os.path.join(self.save_dir, "test"),
        )
        new_test_acc = self.inference(
            self.new_test_loader,
            os.path.join(self.save_dir, "new_test"),
        )
        manual_test_acc = self.inference(
            self.manual_test_loader,
            os.path.join(self.save_dir, "manual_test"),
        )
        print(f"Test accuracy: {test_acc:.4f}")
        print(f"New test accuracy: {new_test_acc:.4f}")
        print(f"Manual test accuracy: {manual_test_acc:.4f}")

    def inference(self, loader, save_dir):
        all_results = []
        accs = []
        for idx, (y, label) in enumerate(pbar := tqdm(loader)):
            y = y.to(self.device)
            x = torch.randn(1, 3, 64, 64).to(self.device)
            denoising_results = []
            for i, t in enumerate(self.noise_scheduler.timesteps):
                with torch.no_grad():
                    residual = self.model(x, t, y)

                x = self.noise_scheduler.step(residual, t, x).prev_sample
                ↪  # type: ignore
                if i % (self.time_steps // 10) == 0:
                    denoising_results.append(x.squeeze(0))
            acc = self.eval_model.eval(x, y)
            tqdm.write(f"image: {idx}, label: {label}, accuracy:
            ↪  {acc:.4f}")
            accs.append(acc)
            pbar.set_postfix_str(f"image: {idx}, accuracy:
            ↪  {np.mean(accs):.4f}")
            pbar.refresh()

            denoising_results.append(x.squeeze(0))
            denoising_results = torch.stack(denoising_results)
            row_image = make_grid(
                (denoising_results + 1) / 2,
                nrow=denoising_results.shape[0],
                pad_value=0,
            )
            save_image(
                row_image,
                os.path.join(save_dir, f"{idx}.png"),
            )
            all_results.append(denoising_results[-1])
        all_results = torch.stack(all_results)
```

```
        save_image(
            make_grid(
                (all_results + 1) / 2,
                nrow=8,
                pad_value=0,
            ),
            os.path.join(save_dir, "all_results.png"),
        )
        return np.mean(accs)
```

2.4 Hyperparameters

The hyperparameters used in the training process are as follows:

- `batch_size`: 32
- `learning_rate`: 1e-4
- `epochs`: 10
- `num_steps`: 1000

2.5 Dataset

The dataset used for training the model is the CLEVR dataset, which contains 3D shapes and their corresponding color-shape phrases. I used PyTorch's `DataLoader` class to load the dataset in batches and apply necessary transformations, such as resizing and normalization. The dataset class is implemented in the `dataset.py` file, which defines the dataset class for loading and preprocessing the training data. The implementation of the dataset class is as follows:

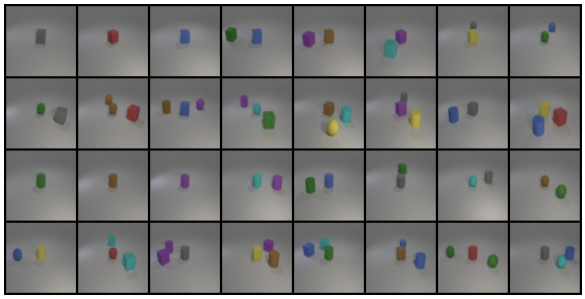
```
def transform_img(img):
    transform = transforms.Compose(
        [
            transforms.Resize((64, 64)),
            transforms.ToTensor(),
            transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
        ]
    )
    return transform(img)


class IclevrDataset(Dataset):
    def __init__(self, root: str, mode="train"):
        super().__init__()
        assert mode in [
            "train",
            "test",
            "new_test",
            "manual_test",
```

```
        ], f"mode {mode} not in [train, test, new_test, manual_test]"
        assert os.path.exists(root), f"{root} does not exist"

        self.root = root
        self.mode = mode

        json_path = os.path.join(root, f"{mode}.json")
        with open(json_path, "r") as json_file:
            self.json_data = json.load(json_file)
            match mode:
                case "train":
                    self.img_paths = list(self.json_data.keys())
                    self.labels = list(self.json_data.values())
                case _:
                    self.labels = list(self.json_data)

        with open(os.path.join(root, "objects.json"), "r") as json_file:
            self.objects_dict = json.load(json_file)
        self.labels_one_hot = torch.zeros(len(self.labels),
        ↪  len(self.objects_dict))

        for i, label in enumerate(self.labels):
            self.labels_one_hot[i][[self.objects_dict[j] for j in label]] =
            ↪  1

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

    def __getitem__(self, index) -> tuple[torch.Tensor, torch.Tensor |
    ↪  list]:
        match self.mode:
            case "train":
                img_path = os.path.join(self.root, "iclevr",
                ↪  self.img_paths[index])
                img = Image.open(img_path).convert("RGB")
                img = transform_img(img)
                label_one_hot = self.labels_one_hot[index]
                return img, label_one_hot
            case _:
                label_one_hot = self.labels_one_hot[index]
                semantic_label = self.labels[index]
                return label_one_hot, semantic_label
```

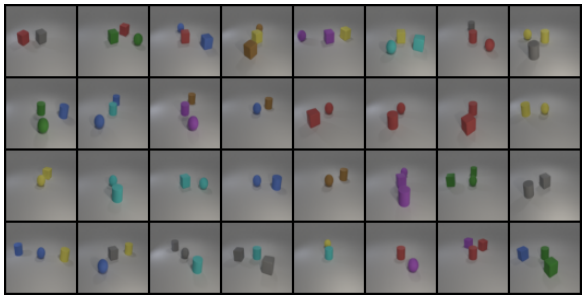
3 Discussion

3.1 Default Configuration

The results of the default configuration are shows in Figure 1. The default configuration is using four down and up-blocks, and keep the attention layers only at the lowest resolution (bottom) layers to reduce memory usage. And the model is trained for 300 epochs with a batch size of 32 and time steps of 1000.



(a) Default Configuration Results on Test Set with accuracy of 0.9010



(b) Default Configuration Results on New Test Set with accuracy of 0.9271

Figure 1: Default Configuration Results and Accuracy

And the results of the manual test with “red sphere”, “cyan cylinder”, and “cyan cube” are shown in Figure 2.

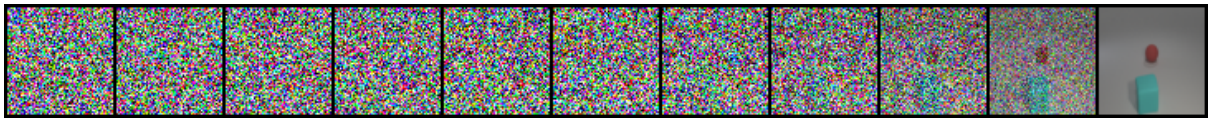


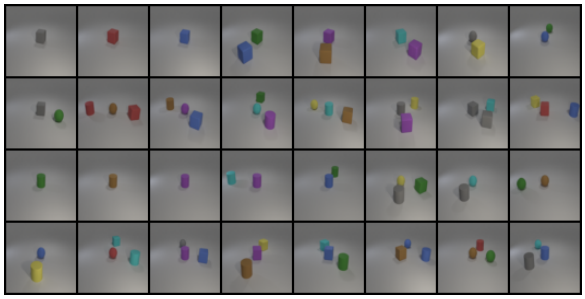
Figure 2: Default Configuration Results on Manual Test Set with accuracy of 1.0000

And the loss curve of the default configuration is shown in Figure 7.

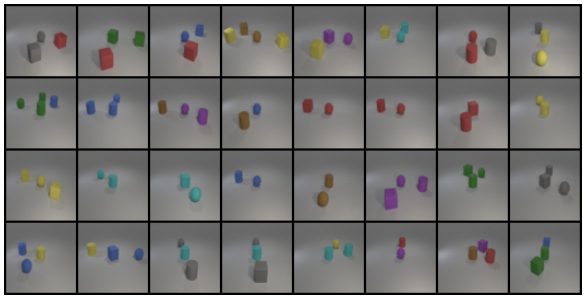
3.2 Same model with different time steps

3.2.1 500 Time Steps

The results of the same model with different time steps are shown in Figure 3. The model is trained for 300 epochs with a batch size of 32 and time steps of 500.



(a) 500 Time Steps Results on Test Set with accuracy of 0.9740



(b) 500 Time Steps Results on New Test Set with accuracy of 0.8906

Figure 3: 500 Time Steps Results and Accuracy

And the results of the manual test with "red sphere", "cyan cylinder", and "cyan cube" are shown in Figure 4.

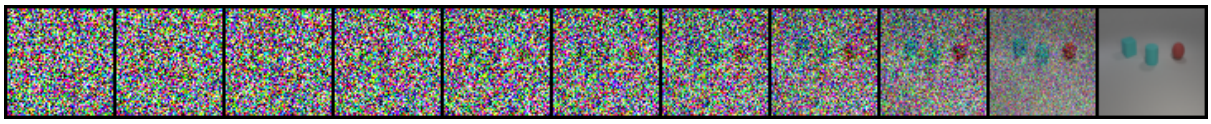
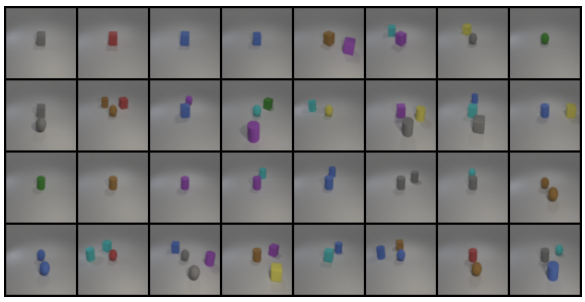


Figure 4: 500 Time Steps Results on Manual Test Set with accuracy of 1.0000

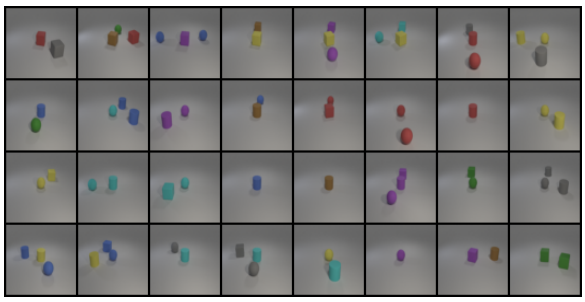
And the loss curve of the 500 time steps is shown in Figure 7.

3.2.2 100 Time Steps

The results of the same model with different time steps are shown in Figure 5. The model is trained for 300 epochs with a batch size of 32 and time steps of 100.



(a) 100 Time Steps Results on Test Set with accuracy of 0.8021



(b) 100 Time Steps Results on New Test Set with accuracy of 0.7865

Figure 5: 100 Time Steps Results and Accuracy

And the results of the manual test with "red sphere", "cyan cylinder", and "cyan cube" are shown in Figure 6.

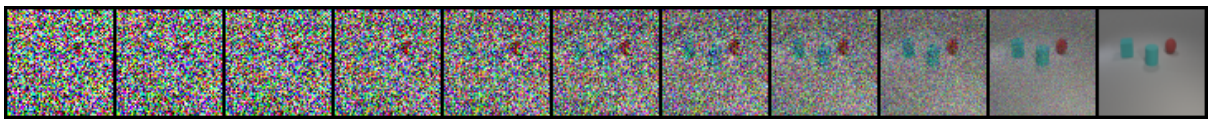


Figure 6: 100 Time Steps Results on Manual Test Set with accuracy of 1.0000

And the loss curve of the 100 time steps is shown in Figure 7.

3.3 Discussion of Results

The results of the default configuration and the same model with different time steps are shown in Figure 7. The model with 1000 time steps has the lowest loss, and the model with 500 or 100 time steps has a higher loss. We can see that the more time steps we have, the lower the loss is.

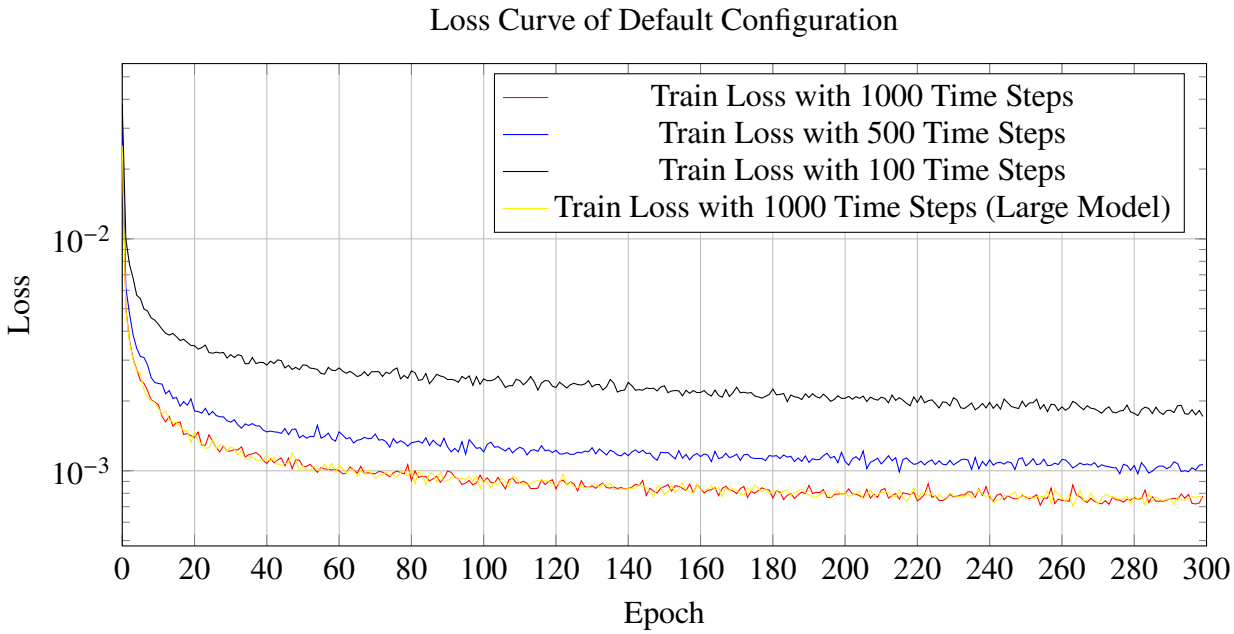


Figure 7: Loss Curve of Different Time Steps

3.4 Larger Model

I also tried to use 6 down and up-blocks, and keep the attention layers only at the lowest two resolution (bottom) layers to reduce memory usage. The model is trained for 300 epochs with a batch size of 32 and time steps of 1000.

Since the results shows similar with the previous model, I didn't include the results here. But I will include the loss curve of the larger model. The loss curve is shown in Figure **??**.

References

[1] P. Pernias, D. Rampas, M. L. Richter, C. J. Pal, and M. Aubreville, *Wuerstchen: An efficient architecture for large-scale text-to-image diffusion models*, 2023. arXiv: 2306.00637 [cs.CV].

[2] J. Ho, A. Jain, and P. Abbeel, "Denoising diffusion probabilistic models," *CoRR*, vol. abs/2006.11239, 2020. arXiv: 2006.11239. [Online]. Available: https://arxiv.org/abs/2006.11239.

[3] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," *CoRR*, vol. abs/1312.6114, 2013. [Online]. Available: https://api.semanticscholar.org/CorpusID:216078090.

[4] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, *et al.*, "Generative adversarial nets," in *Neural Information Processing Systems*, 2014. [Online]. Available: https://api.semanticscholar.org/CorpusID:261560300.