O'REILLY®

Hands-On Generative AI with Transformers and Diffusion Models

Early Release

RAW & UNEDITED

Pedro Cuenca,
Apolinário Passos,
Omar Sanseviero &
Jonathan Whitaker

O'REILLY®

Hands-On Generative AI with Transformers and Diffusion Models

Early Release

RAW & UNEDITED

Pedro Cuenca,
Apolinário Passos,
Omar Sanseviero &
Jonathan Whitaker

Hands-On Generative AI with Transformers and Diffusion Models

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Pedro Cuenca, Apolinário Passos, Omar Sanseviero, and Jonathan Whitaker

Beijing • Boston • Farnham • Sebastopol • Tokyo

Hands-On Generative AI with Transformers and Diffusion Models

by Pedro Cuenca, Apolinário Passos, Omar Sanseviero, and Jonathan Whitaker

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

- Acquisitions Editor: Nicole Butterfield
- Development Editor: Jill Leonard
- Production Editor: Gregory Hyman
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- Illustrator: Kate Dullea
- September 2024: First Edition

Revision History for the Early Release

- 2023-03-16: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781098149246* for release details.

978-1-098-14924-6

Chapter 1. Diffusion Models

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the third chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at jleonard@oreilly.com.

In late 2020 a little-known class of models called diffusion models began causing a stir in the machine-learning world. Researchers figured out how to use these models to generate synthetic images at higher quality than any produced by previous techniques. A flurry of papers followed, proposing improvements and modifications that pushed the quality up even further. By late 2021 there were models like GLIDE that showcased incredible results on text-to-image tasks, and a few months later, these models had entered the mainstream with tools like DALL-E 2 and Stable Diffusion. These models made it easy for anyone to generate images just by typing in a text description of what they wanted to see.

In this chapter, we're going to dig into the details of how these models work. We'll outline the key insights that make them so powerful, generate images with existing models to get a feel for how they work, and then train our own models to deepen this understanding further. The field is still rapidly evolving, but the topics covered here should give you a solid foundation to build on. Chapter 5 will explore more advanced techniques

through the lens of a model called Stable Diffusion, and chapter 6 will explore applications of these techniques beyond simple image generation.

The Key Insight: Iterative Refinement

So what is it that makes diffusion models so powerful? Previous techniques, such as VAEs or GANs, generate their final output via a single forward pass of the model. This means the model must get everything right on the first try. If it makes a mistake, it can't go back and fix it. Diffusion models, on the other hand, generate their output by iterating over many steps. This 'iterative refinement' allows the model to correct mistakes made in previous steps and gradually improve the output. To illustrate this, let's look at an example of a diffusion model in action.

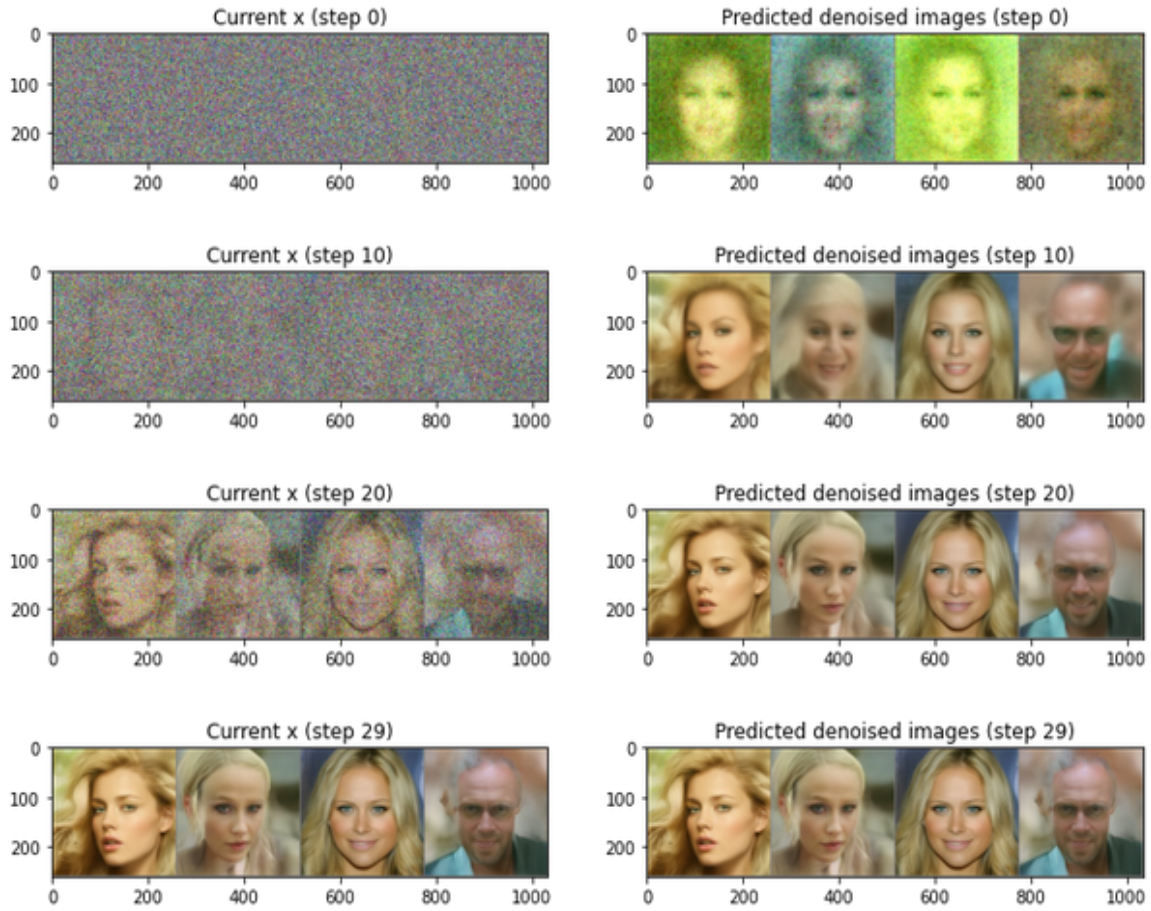
We can load a pre-trained model using the Hugging Face diffusers library. The pipeline can be used to create images directly, but this doesn't show us what is going on under the hood:

```
# Load the pipeline
image_pipe = DDPMPipeline.from_pretrained("google/ddpm-celebahq-256")
image_pipe.to(device);

# Sample an image
image_pipe().images[0]
```

We can re-create the sampling process step by step to get a better look at what is happening as the model generates images. We initialize our sample x with random noise and then run it through the model for 30 steps. On the right, you can see the model's prediction for what the final image will look like at specific steps - note that the initial predictions are not particularly good! Instead of jumping right to that final predicted image, we only modify x by a small amount in the direction of the prediction (shown on the left). We then feed this new, slightly better x through the model again for the next step, hopefully resulting in a slightly improved prediction, which can be used to update x a little more, and so on. With enough steps, the model can produce some impressively realistic images.

```
# The random starting point for a batch of 4 images
x = torch.randn(4, 3, 256, 256).to(device)

# Set the number of timesteps lower
image_pipe.scheduler.set_timesteps(num_inference_steps=30)

# Loop through the sampling timesteps
for i, t in enumerate(image_pipe.scheduler.timesteps):

    # Get the prediction given the current sample x and the timestep t
    with torch.no_grad():
        noise_pred = image_pipe.unet(x, t)["sample"]

    # Calculate what the updated sample should look like with the scheduler
    scheduler_output = image_pipe.scheduler.step(noise_pred, t, x)

    # Update x
    x = scheduler_output.prev_sample

    # Occasionally display both x and the predicted denoised images
    if i % 10 == 0 or i == len(image_pipe.scheduler.timesteps) - 1:
        fig, axs = plt.subplots(1, 2, figsize=(12, 5))

        grid = torchvision.utils.make_grid(x, nrow=4).permute(1, 2, 0)
        axs[0].imshow(grid.cpu().clip(-1, 1) * 0.5 + 0.5)
        axs[0].set_title(f"Current x (step {i})")

        pred_x0 = scheduler_output.pred_original_sample
        grid = torchvision.utils.make_grid(pred_x0, nrow=4).permute(1, 2, 0)
        axs[1].imshow(grid.cpu().clip(-1, 1) * 0.5 + 0.5)
```

```
    axs[1].set_title(f"Predicted denoised images (step {i})")
    plt.show()
```



> **NOTE**
>
> Don't worry if that code looks a bit intimidating - we'll explain how this all works over the course of this chapter. For now, just focus on the results.

This core idea of learning how to refine a 'corrupted' input gradually can be applied to a wide range of tasks. In this chapter, we'll focus on unconditional image generation - that is, generating images that resemble the training data, with no additional controls over what these generated samples look like. Diffusion models have also been applied to audio, video, text and more. And while most implementations use some variant of the

‘denoising’ approach that we’ll cover here, new approaches utilizing different types of ‘corruption’ together with iterative refinement are emerging that may move the field beyond the current focus on denoising diffusion specifically. Exciting times!

Training a Diffusion Model

In this section, we’re going to train a diffusion model from scratch to gain a better understanding of how they work. We’ll start by using components from the Hugging Face diffusers library. As the chapter progresses, we’ll gradually demystify how each component works. Training a diffusion model is relatively straightforward compared to other types of generative models. We repeatedly:

- Load in some images from the training data.
- Add noise in different amounts. Remember, we want the model to do a good job estimating how to ‘fix’ (denoise) both extremely noisy images and images that are close to perfect.
- Feed the noisy versions of the inputs into the model.
- Evaluate how well the model does at denoising these inputs.
- Use this information to update the model weights.

To generate new images with a trained model, we begin with a completely random input and repeatedly feed it through the model, updating the input on each iteration by a small amount based on the model prediction. As we’ll see, there are a number of sampling methods that try to streamline this process so that we can generate good images with as few steps as possible.

The Data

For this example, we’ll use a dataset of images from the Hugging Face Hub- specifically, this collection of 1000 butterfly pictures. Later on, in the projects section, you will see how to use your own data.

```
dataset = load_dataset("huggan/smithsonian_butterflies_subset", split="train")
```

We need to do some preparation before this data can be used to train a model. Images are typically represented as a grid of ‘pixels’, with color values between 0 and 255 for each of the three color channels (Red, Green and Blue). To process these and make them ready for training, we: - Resize them to a fixed size - (Optional) Add some augmentation by randomly flipping them horizontally, effectively doubling the size of our dataset - Convert them to a PyTorch tensor (which represents the color values as floats between 0 and 1) - Normalize them to have a mean of 0, with values between -1 and 1

We can do all of this with `torchvision.transforms`:

```
image_size = 64

# Define data augmentations
preprocess = transforms.Compose(
    [
        transforms.Resize((image_size, image_size)),  # Resize
        transforms.RandomHorizontalFlip(),  # Randomly flip (data
augmentation)
        transforms.ToTensor(),  # Convert to tensor (0, 1)
        transforms.Normalize([0.5], [0.5]),  # Map to (-1, 1)
    ]
)
```

Next, we need to create a dataloader to load the data in batches with these transforms applied:

```
batch_size = 32

def transform(examples):
    images = [preprocess(image.convert("RGB")) for image in examples["image"]]
    return {"images": images}


dataset.set_transform(transform)

train_dataloader = torch.utils.data.DataLoader(
    dataset, batch_size=batch_size, shuffle=True
)
```

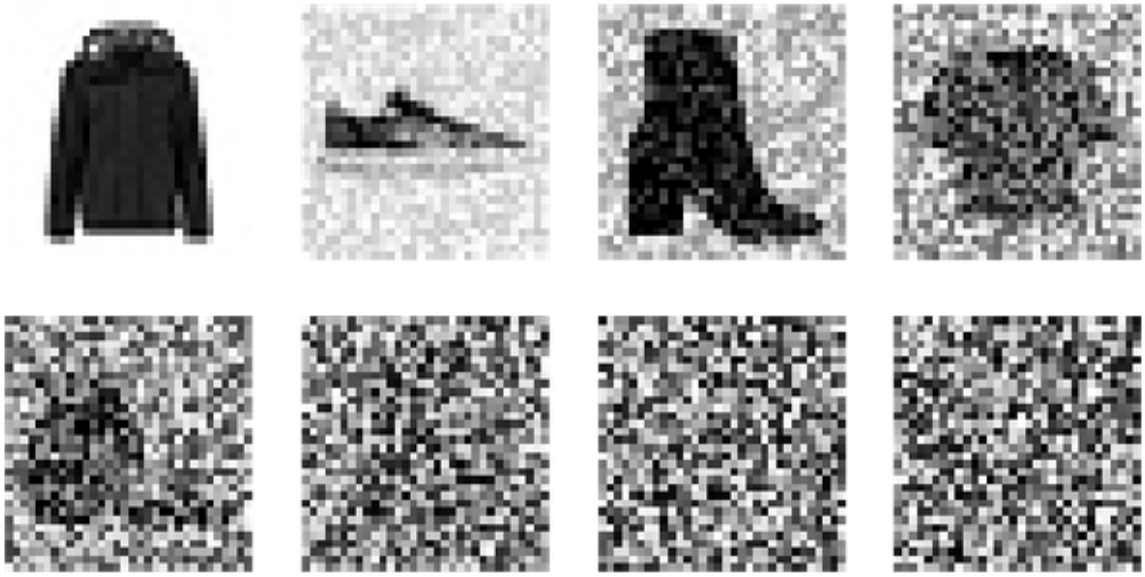
We can check that this worked by loading a single batch and inspecting the images.

```
batch = next(iter(train_dataloader))
print('Shape:', batch['images'].shape,
      '\nBounds:', batch['images'].min().item(), 'to',
batch['images'].max().item())
show_images(batch['images'][:8]*0.5 + 0.5) # NB: we map back to (0, 1) for
display
Shape: torch.Size([32, 3, 64, 64])
Bounds: -0.9921568632125854 to 1.0
```

Adding Noise

How do we gradually corrupt our data? The most common approach is to add noise to the images. The amount of noise we add is controlled by a noise schedule. Different papers and approaches tackle this in different ways, which we'll explore later in the chapter. For now, let's see one common approach in action based on the paper "Denoising diffusion probabilistic models" by Ho et al. In the diffusers library, adding noise is handled by something called a scheduler, which takes in a batch of images and a list of 'timesteps' and determines how to create the noisy versions of those images:

```
scheduler = DDPMScheduler(num_train_timesteps=1000, beta_start=0.001,
beta_end=0.02)
timesteps = torch.linspace(0, 999, 8).long()

x = batch['images'][:8]
noise = torch.rand_like(x)
noised_x = scheduler.add_noise(x, noise, timesteps)
show_images((noised_x*0.5 + 0.5).clip(0, 1))
```



During training, we'll pick the timesteps at random. The scheduler takes some parameters (beta_start and beta_end) which it uses to determine how much noise should be present for a given timestep. We will cover schedulers in more detail in section X.

The UNet

UNet is a convolutional neural network invented for tasks such as image segmentation, where the desired output has the same spatial extent as the input. It consists of a series of 'downsampling' layers that reduce the spatial size of the input, followed by a series of 'upsampling' layers that increase the spatial extent of the input again. The downsampling layers are also typically followed by a 'skip connection' that connects the downsampling layer's output to the upsampling layer's input. This allows the upsampling layers to 'see' the higher-resolution representations from earlier in the

network, which is useful for tasks with image-like outputs where this high-resolution information is especially useful.

The UNet architecture used in the diffusers library is more advanced than the original UNet proposed in 2015 by Ronneberger et al, with additions like attention and residual blocks. We'll take a closer look later, but the key feature here is that it can take in an input (the noisy image) and produce a prediction that is the same shape (the predicted noise). For diffusion models, the UNet typically also takes in the timestep as additional conditioning, which again we will explore in the UNet deep dive section.

Here's how we might create a UNet and feed our batch of noisy images through it:

```
# Create a UNet2DModel
model = UNet2DModel(
    in_channels=3,  # 3 channels for RGB images
    sample_size=64,  # Specify our input size
    block_out_channels=(64, 128, 256, 512), # N channels per layer
    down_block_types=("DownBlock2D", "DownBlock2D",
                      "AttnDownBlock2D", "AttnDownBlock2D"),
    up_block_types=("AttnUpBlock2D", "AttnUpBlock2D",
                    "UpBlock2D", "UpBlock2D"),
)

# Pass a batch of data through
with torch.no_grad():
    out = model(noised_x, timestep=timesteps).sample
out.shape
torch.Size([8, 3, 64, 64])
```

Note that the output is the same shape as the input, which is exactly what we want.

Training

Now that we have our model and our data ready, we can train it. We'll use the AdamW optimizer with a learning rate of 3e-4. For each training step, we:

- Load a batch of images.
- Add noise to the images, choosing random timesteps to determine how much noise is added.
- Feed the noisy images into the model.
- Calculate the loss, which is the mean squared error between the model's predictions and the target - which in this case is the *noise* that we added to the images. This is called the noise or 'epsilon' objective. You can find more information on the different training objectives in section X.
- Backpropagate the loss and update the model weights with the optimizer.

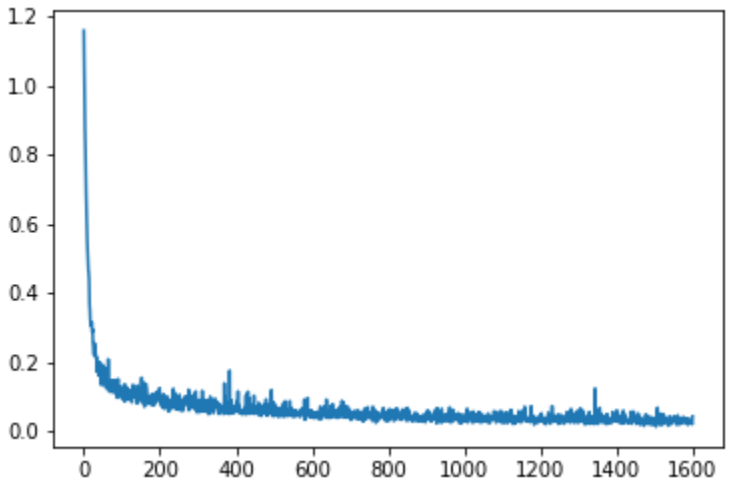
Here's what all of that looks like in code:

```
num_epochs = 50 # How many runs through the data should we do?
lr = 1e-4 # What learning rate should we use
model = model.to(device) # The model we're training (defined in the previous
section)
optimizer = torch.optim.AdamW(model.parameters(), lr=lr) # The optimizer
losses = [] # somewhere to store the loss values for later plotting

# Train the model (this takes a while!)
for epoch in range(num_epochs):
    for step, batch in enumerate(train_dataloader):

        # Load the input images
        clean_images = batch["images"].to(device)

        # Sample noise to add to the images
        noise = torch.randn(clean_images.shape).to(clean_images.device)

        # Sample a random timestep for each image
        timesteps = torch.randint(
            0,
            scheduler.num_train_timesteps,
            (clean_images.shape[0],),
            device=clean_images.device,
        ).long()

        # Add noise to the clean images according timestep
```

```
        noisy_images = scheduler.add_noise(clean_images, noise, timesteps)

        # Get the model prediction for the noise
        noise_pred = model(noisy_images, timesteps, return_dict=False)[0]

        # Compare the prediction with the actual noise:
        loss = F.mse_loss(noise_pred, noise)

        # Store the loss for later plotting
        losses.append(loss.item())

        # Update the model parameters with the optimizer based on this loss
        loss.backward(loss)
        optimizer.step()
        optimizer.zero_grad()
```

It takes an hour or so to run the above code on a GPU, so get some tea while you wait or lower the number of epochs. Here's what the loss curve looks like after training:

```
# Plot the loss curve:
plt.plot(losses);
```



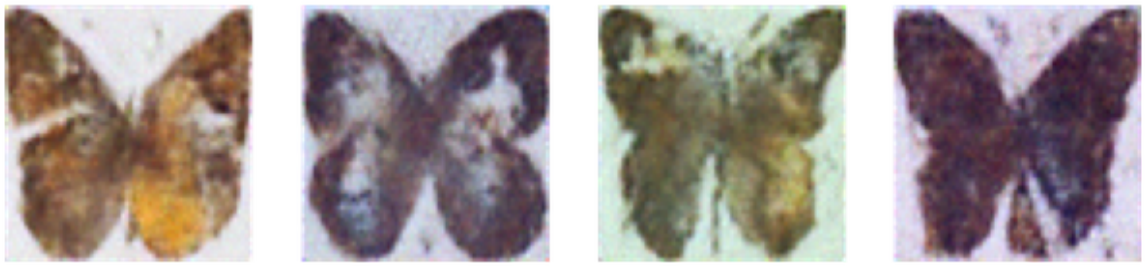
The loss curve trends downwards as the model learns to denoise the images. The curve is fairly noisy, thanks to different amounts of noise being added to the images based on the random sampling of timesteps for each iteration. It is hard to tell just by looking at the mean squared error of the noise

predictions whether this model will be any good at generating samples, so let's move on to the next section and see how well it does.

Sampling

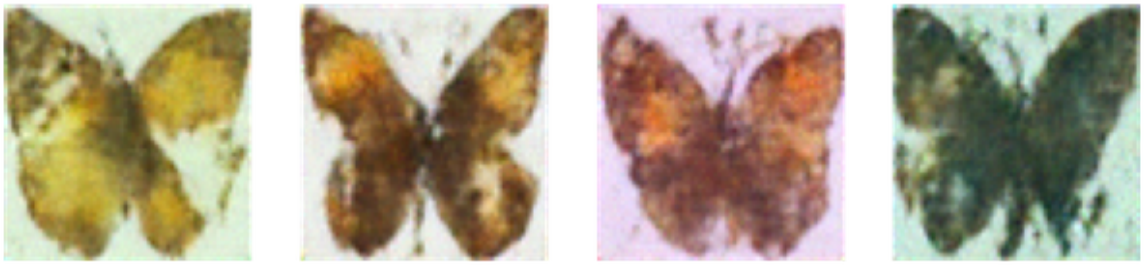
The diffusers library uses the idea of 'pipelines' which bundle together all of the components needed to generate samples with a diffusion model:

```
pipeline = DDPMPipeline(unet=model, scheduler=scheduler)
ims = pipeline(batch_size=4).images
show_images(ims, nrows=1)
```



Of course, offloading the job of creating samples to the pipeline doesn't really show us what is going on. So, here is a simple sampling loop that shows how the model is gradually refining the input image:

```
# Random starting point (4 random images):
sample = torch.randn(4, 3, 64, 64).to(device)

for i, t in enumerate(scheduler.timesteps):

    # Get model pred
    with torch.no_grad():
        noise_pred = model(sample, t).sample

    # Update sample with step
    sample = scheduler.step(noise_pred, t, sample).prev_sample

show_images(sample.clip(-1, 1)*0.5 + 0.5, nrows=1)
```



This is the same code we used at the beginning of the chapter to illustrate the idea of iterative refinement, but hopefully, now you have a better understanding of what is going on here. We start with a completely random input, which is then refined by the model in a series of steps. Each step is a small update to the input, based on the model's prediction for the noise at that timestep. We're still abstracting away some complexity behind the call to `pipeline.scheduler.step()` - in a later chapter we will dive deeper into different sampling methods and how they work.

Evaluation

Generative model performance can be evaluated using FID scores (Fréchet Inception Distance). FID scores measure how closely generated samples match real-world samples by comparing statistics between feature maps extracted from both sets of data using a pre-trained neural network. The lower the score, the better the quality and realism of generated images produced by a given model. FID scores are popular due to their ability to provide an 'objective' comparison metric for different types of generative networks without relying on human judgment.

As convenient as FID scores are, there are some important caveats to be aware of:

- The FID score for a given model depends on the number of samples used to calculate it, so when comparing between model,s we need to make sure both reported scores are calculated using the same number of samples. Common practice is to use 50,000 samples for this purpose, although to save time, you may evaluate on a smaller number

of samples during development and only do the full evaluation once you're ready to publish the results.

- When calculating FID, images are resized to 299px square images. This makes it less useful as a metric for extremely low-res or high-res images. There are also minor differences between how resizing is handled by different deep learning frameworks, which can result in small differences in the FID score! We recommend using a library such as `clean-fid` to standardize the FID calculation.
- The network used as a feature extractor for FID is typically a model trained on the Imagenet classification task. When generating images in a different domain, the features learned by this model may be less useful. A more accurate approach would be to somehow train a classification network on domain-specific data first, but this would make it harder to compare scores between different papers and approaches, so for now the imagenet model is the standard choice.
- If you save generated samples for later evaluation, the format and compression can again affect the FID score. Avoid low-quality JPEG images where possible.

Even if you account for all these caveats, FID scores are just a rough measure of quality and do not perfectly capture the nuances of what makes images look more 'real'. So, use them to get an idea of how one model performs relative to another but also look at the actual images generated by each model to get a better sense of how they compare. Human preference is still the gold standard for quality in what is ultimately a fairly subjective field!

In Depth: Noise Schedules

In the training example above, one of the steps was 'add noise, in different amounts'. We achieved this by picking a random timestep between 0 and 1000 and then relying on the scheduler to add the appropriate amount of noise. Likewise, during sampling, we again relied on the scheduler to tell us

which timesteps to use and how to move from one to the next given the model predictions. It turns out that choosing how much noise to add is an important design decision that can drastically affect the performance of a given model. In this section, we'll see why this is the case and explore some of the different approaches that are used in practice.

Why Add Noise?

At the start of this chapter, we said that the key idea behind diffusion models is that of iterative refinement. During training, we 'corrupt' an input by different amounts. During inference, we begin with a 'maximally corrupted' input and iteratively 'de-corrupt' it, in the hopes that we will eventually end up with a nice final result.

So far, we've focused on one specific kind of 'corruption': adding Gaussian noise. One reason for this is the theoretical underpinnings of diffusion models - if we use a different corruption method we are no longer technically doing 'diffusion'! However, a paper titled *Cold Diffusion* by Bansal et al dramatically demonstrated that we do not necessarily need to constrain ourselves to this method just for theoretical convenience. They showed that a diffusion-model-like approach works for many different 'corruption' methods (see Figure 1-1). More recently, models like MUSE, MaskGIT and PAELLA have used random token masking or replacement as an equivalent 'corruption' method for quantized data - that is, data that is represented by discrete tokens rather than continuous values.

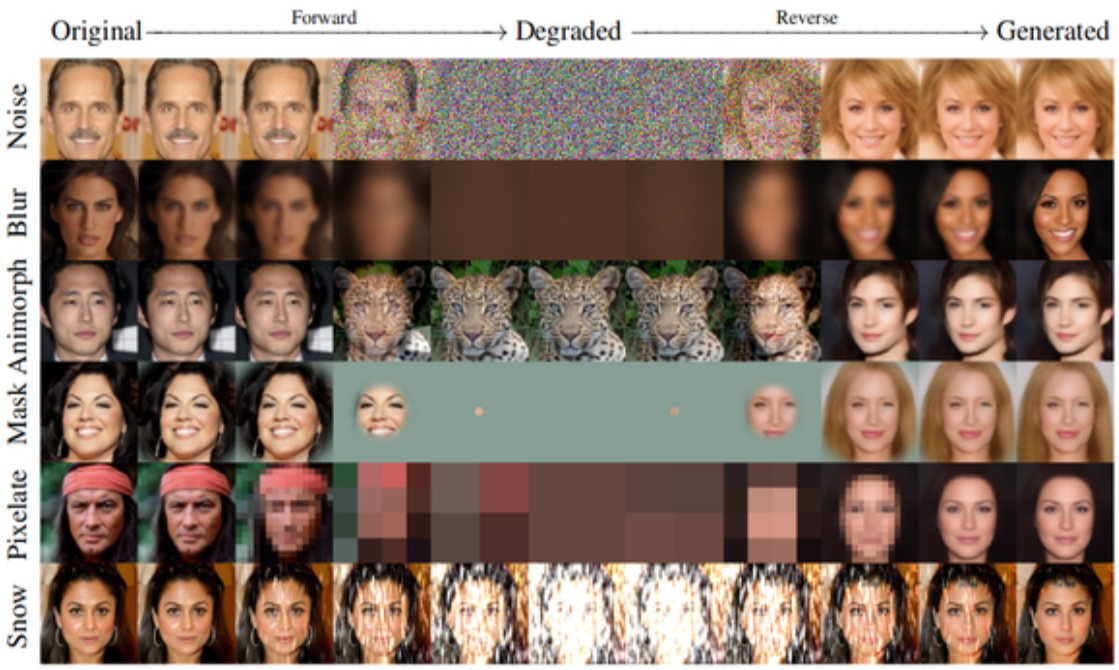


Figure 1: Demonstration of the forward and backward processes for both hot and cold diffusions. While standard diffusions are built on Gaussian noise (top row), we show that generative models can be built on arbitrary and even noiseless/cold image transforms, including the ImageNet-C *snowification* operator, and an *animorphosis* operator that adds a random animal image from AFHQ.

Figure 1-1. Illustration of the different degradations used in the Cold Diffusion Paper

Nonetheless, adding noise remains the most popular approach for several reasons:

- We can easily control the amount of noise added, giving a smooth transition from ‘perfect’ to ‘completely corrupted’. This is not the case for something like reducing the resolution of an image, which may result in ‘discrete’ transitions.
- We can have many valid random starting points for inference, unlike some methods which may only have a limited number of possible initial (fully corrupted) states, such as a completely black image or a single-pixel image.

So, for the moment at least, we’ll stick with adding noise as our corruption method. Next, let’s take a closer look at how we add noise to our images.

Starting Simple

We have some images (x) and we'd like to combine them somehow with some random noise.

```
x = next(iter(train_dataloader))['images'][:8]
noise = torch.rand_like(x)
```

One way we could do this is to linearly interpolate (lerp) between them by some amount. This gives us a function that smoothly transitions from the original image x to pure noise as the 'amount' varies from 0 to 1:

```
def corrupt(x, noise, amount):
  amount = amount.view(-1, 1, 1, 1) # make sure it's broadcastable
  return x*(1-amount) + noise*amount
```

Let's see this in action on a batch of data, with the amount of noise varying from 0 to 1:

```
amount = torch.linspace(0, 1, 8)
noised_x = corrupt(x, noise, amount)
show_images(noised_x*0.5 + 0.5)
```

This seems to be doing exactly what we want, smoothly transitioning from the original image to pure noise. Now, we've created a noise schedule here that takes in a value for 'amount' from 0 to 1. This is called the 'continuous time' approach, where we represent the full path on a time scale from 0 to 1. Other approaches use a discrete time approach, with some large integer number of 'timesteps' used to define the noise scheduler. We can wrap our function into a class that converts from continuous time to discrete timesteps and adds noise appropriately:

```
class SimpleScheduler():
  def __init__(self):
    self.num_train_timesteps = 1000
  def add_noise(self, x, noise, timesteps):
    amount = timesteps / self.num_train_timesteps
    return corrupt(x, noise, amount)

scheduler = SimpleScheduler()
timesteps = torch.linspace(0, 999, 8).long()
noised_x = scheduler.add_noise(x, noise, timesteps)
show_images(noised_x*0.5 + 0.5)
```

Now we have something that we can directly compare to the schedulers used in the diffusers library, such as the DDPMScheduler we used during training. Let's see how it compares:

```
scheduler = DDPMScheduler(beta_end=0.01)
timesteps = torch.linspace(0, 999, 8).long()
noised_x = scheduler.add_noise(x, noise, timesteps)
show_images((noised_x*0.5 + 0.5).clip(0, 1))
```

The Maths

There are many competing notations and approaches in the literature. For example, some papers parametrize the noise schedule in *continuous-time* where t runs from 0 (no noise) to 1 (fully corrupted) - just like our `corrupt` function in the previous section. Others use a *discrete-time* approach with integer timesteps running from 0 to some large number T, typically 1000. It is possible to convert between these two approaches the way we did with our `SimpleScheduler` class - just make sure you're consistent when comparing different models. We'll stick with the discrete-time approach here.

A good place to start for pushing deeper into the maths is the DDPM paper mentioned earlier. You can find an annotated implementation here which is a great additional resource for understanding this approach.

The paper begins by specifying a single noise step to go from timestep t-1 to timestep t. Here's how they write it:

$$q\left(\mathbf{x}_t|\mathbf{x}_{t-1}\right) = \mathcal{N}\left(\mathbf{x}_t; \sqrt{1-\beta_t}\mathbf{x}_{t-1}, \beta_t\mathbf{I}\right).$$

Here β_t is defined for all timesteps t and is used to specify how much noise is added at each step. This notation can be a little intimidating, but what this equation tells us is that the noisier $\mathbf{x}_t$ is a *distribution* with a mean of $\sqrt{1-\beta_t}\mathbf{x}_{t-1}$ and a variance of β_t. In other words, $\mathbf{x}_t$ is a mix of $\mathbf{x}_{t-1}$ (scaled by $\sqrt{1-\beta_t}$) and some random noise, which we can think of as unit-variance noise scaled by $\sqrt{\beta_t}$. Given x_{t-1} and some noise ϵ, we can sample from this distribution to get x_t with:

$\mathbf{x}_t = \sqrt{1-\beta_t}\mathbf{x}_{t-1} + \sqrt{\beta_t}\epsilon$

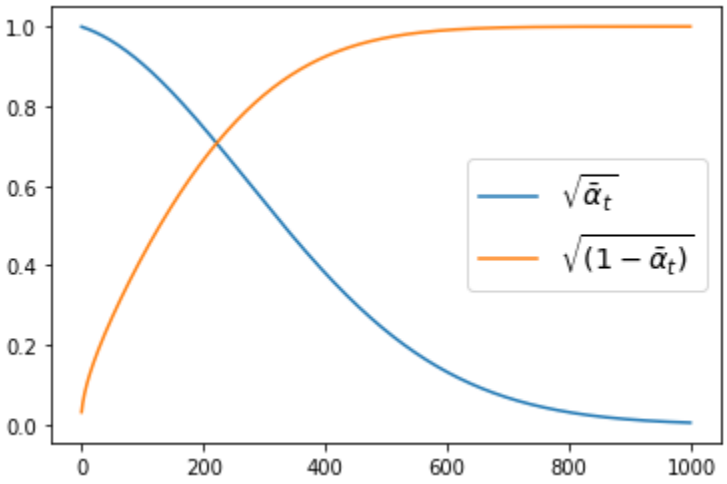
To get the noisy input at timestep t, we could begin at t=0 and repeatedly apply this single step, but this would be very inefficient. Instead, we can find a formula to move to any timestep t in one go. We define $\alpha_t = 1 - \beta_t$ and then use the following formula:

$x_t = \sqrt{\alpha_t}x_0 + \sqrt{1-\alpha_t}\epsilon$

where - ϵ is some gaussian noise with unit variance - α ('alpha_bar') is the cumulative product of all the α values up to the time t.

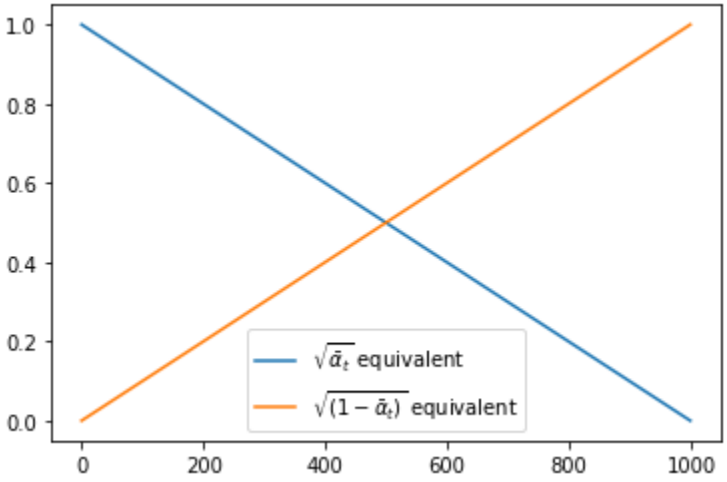
So x_t is a mixture of x_0 (scaled by $\sqrt{\alpha_t}$) and ϵ (scaled by $\sqrt{1-\alpha_t}$). In the diffusers library the α values are stored in `scheduler.alphas_cumprod`. Knowing this, we can plot the scaling factors for the original image x_0 and the noise ϵ across the different timesteps for a given scheduler:

```
plot_scheduler(DDPMScheduler()) # The default scheduler
```



Our SimpleScheduler above just linearly mixes between the original image and noise, as we can see if we plot the scaling factors (equivalent to $\sqrt{\alpha_t}$ and $\sqrt{(1-\alpha_t)}$ in the DDPM case):

```
plot_scheduler(SimpleScheduler())
```



A good noise schedule will ensure that the model sees a mix of images at different noise levels. The best choice will differ based on the training data. Visualizing a few more options, note that:

- Setting beta_end too low means we never completely erase the image, so the model will never see anything like the random noise used as a starting point for inference.

- Setting beta_end extremely high means that most of the timesteps are spent on almost complete noise, which will result in poor training performance.
- Different beta schedules give different curves.

The ‘cosine’ schedule is a popular choice, as it gives a smooth transition from the original image to the noise.

```
fig, (ax) = plt.subplots(1, 1, figsize=(8, 5))
plot_scheduler(DDPMScheduler(beta_schedule="linear"),
               label = 'default schedule', ax=ax, plot_both=False)
plot_scheduler(DDPMScheduler(beta_schedule="squaredcos_cap_v2"),
               label = 'cosine schedule', ax=ax, plot_both=False)
plot_scheduler(DDPMScheduler(beta_end=0.003, beta_schedule="linear"),
               label = 'Low beta_end', ax=ax, plot_both=False)
plot_scheduler(DDPMScheduler(beta_end=0.1, beta_schedule="linear"),
               label = 'High beta_end', ax=ax, plot_both=False)
```

NOTE

All of the schedules shown here are called ‘Variance Preserving’ (VP), meaning that the variance of the model input is kept close to 1 across the entire schedule. You may also encounter ‘Variance Exploding’ (VE) formulations where noise is simply added to the original image in different amounts (resulting in high-variance inputs). We’ll go into this more in the chapter on sampling. Our SimpleScheduler is almost a VP schedule, but the variance is not quite preserved due to the linear interpolation.

As with many diffusion-related topics, there is a constant stream of new papers exploring the topic of noise schedules, so by the time you read this there will likely be a large collection of options to try out!

Effect of Input Resolution and Scaling

One aspect of noise schedules that was mostly overlooked until recently is the effect of input size and scaling. Many papers test potential schedulers on small-scale datasets and at low resolution, and then use the best-performing scheduler to train their final models on larger images. The problem with this is can be seen if we add the same amount of noise to two images of different sizes.

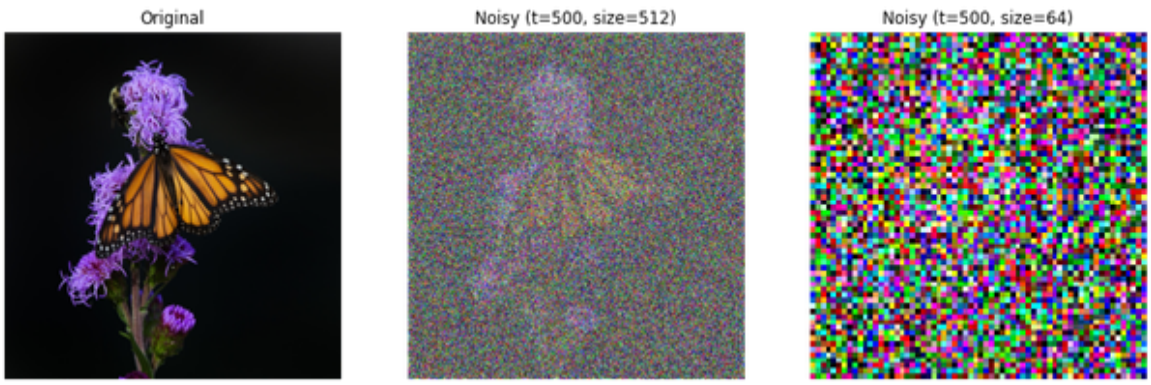


Figure 1-2. Comparing the effect of adding noise to images of different sizes

Images at high resolution tend to contain a lot of redundant information. This means that even if a single pixel is obscured by noise, the surrounding pixels contain enough information to reconstruct the original image. This is not the case for low-resolution images, where a single pixel can contain a lot of information. This means that adding the same amount of noise to a low-resolution image will result in a much more corrupted image than adding the equivalent amount of noise to a high-resolution image.

This effect was thoroughly investigated in two independent papers, both of which came out in January 2023. Each used the new insights to train models capable of generating high-resolution outputs without requiring any of the tricks that have previously been necessary. *Simple diffusion* by Hoogeboom et al introduced a method for adjusting the noise schedule

based on the input size, allowing a schedule optimized on low-resolution images to be appropriately modified for a new target resolution. A paper called "On the Importance of Noise Scheduling for Diffusion Models" by Ting Chen performed similar experiments, and noted another key variable: input scaling. That is, how do we represent our images? If the images are represented as floats between 0 and 1 then they will have a lower variance than the noise (which is typically unit variance) and thus the signal-to-noise ratio will be lower for a given noise level than if the images were represented as floats between -1 and 1 (which we used in the training example above) or something else. Scaling the input images shifts the signal-to-noise ratio, and so modifying this scaling is another way we can adjust when training on larger images.

In Depth: UNets and Alternatives

Now let's address the actual model that makes the all-important predictions! To recap, this model must be capable of taking in a noisy image and estimating how to denoise it. This requires a model that can take in an image of arbitrary size and output an image of the same size. Furthermore, the model should be able to make precise predictions at the pixel level, while also capturing higher-level information about the image as a whole. A popular approach is to use an architecture called a UNet. UNets were invented in 2015 for medical image segmentation, and have since become a popular choice for various image-related tasks. Like the AutoEncoders and VAEs we looked at in the previous chapter, UNets are made up of a series of 'downsampling' and 'upsampling' blocks. The downsampling blocks are responsible for reducing the size of the image, while the upsampling blocks are responsible for increasing the size of the image. The downsampling blocks are typically made up of a series of convolutional layers, followed by a pooling or downsampling layer. The upsampling blocks are typically made up of a series of convolutional layers, followed by an upsampling or 'transposed convolution' layer. The transposed convolution layer is a special type of convolutional layer that increases the size of the image, rather than reducing it.

The reason a regular AutoEncoder or VAE is not a good choice for this task is that they are less capable of making precise predictions at the pixel level since the output must be entirely re-constructed from the low-dimensional latent space. In a UNet, the downsampling and upsampling blocks are connected by ‘skip connections’, which allow information to flow directly from the downsampling blocks to the upsampling blocks. This allows the model to make precise predictions at the pixel level, while also capturing higher-level information about the image as a whole.

A Simple UNet

To better understand the structure of a UNet, let’s build a simple UNet from scratch.

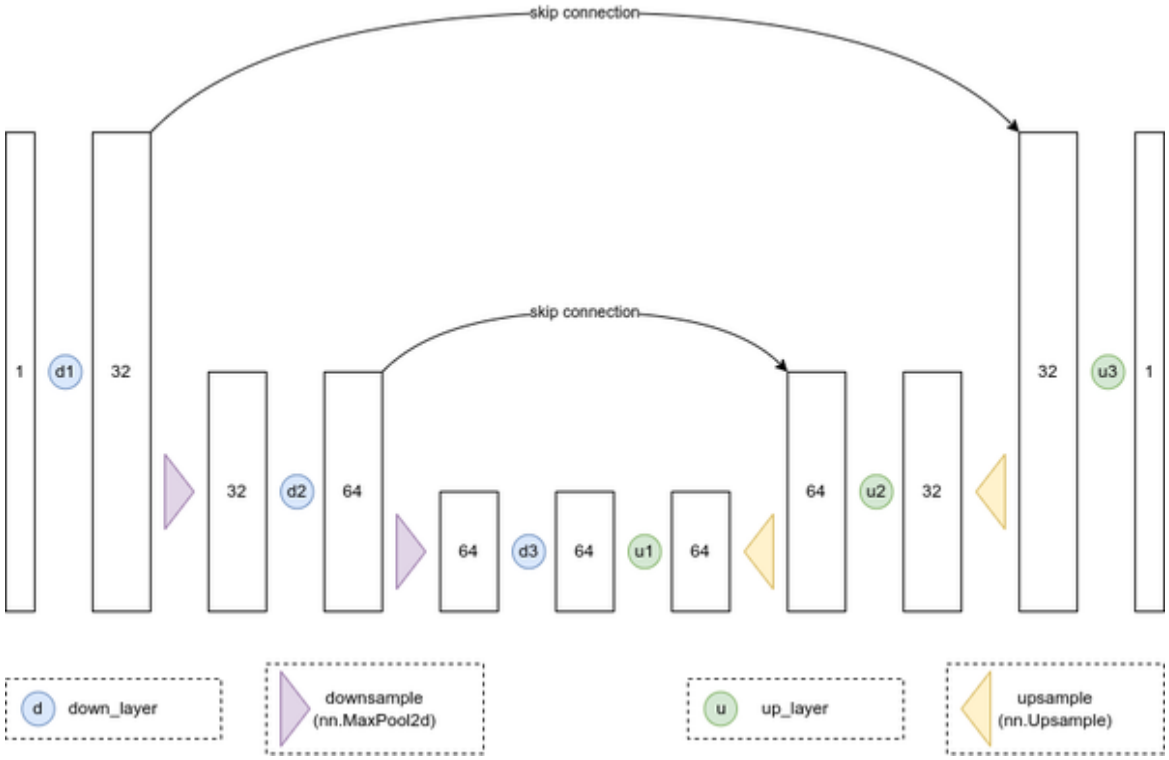


Figure 1-3. Our simple UNet architecture

This UNet takes single-channel inputs at 32px resolution and outputs single-channel outputs at 32px resolution, which we could use to build a diffusion model for the MNIST dataset. There are three layers in the encoding path, and three layers in the decoding path. Each layer consists of

a convolution followed by an activation function and an upsampling or downsampling step (depending on whether we are in the encoding or decoding path). The skip connections allow information to flow directly from the downsampling blocks to the upsampling blocks, and are implemented by adding the output of the downsampling block to the input of the corresponding upsampling block. Some UNets instead concatenate the output of the downsampling block to the input of the corresponding upsampling block, and may also include additional layers in the skip connections. Here's what this network looks like in code:

```
from torch import nn

class BasicUNet(nn.Module):
    """A minimal UNet implementation."""
    def __init__(self, in_channels=1, out_channels=1):
        super().__init__()
        self.down_layers = torch.nn.ModuleList([
            nn.Conv2d(in_channels, 32, kernel_size=5, padding=2),
            nn.Conv2d(32, 64, kernel_size=5, padding=2),
            nn.Conv2d(64, 64, kernel_size=5, padding=2),
        ])
        self.up_layers = torch.nn.ModuleList([
            nn.Conv2d(64, 64, kernel_size=5, padding=2),
            nn.Conv2d(64, 32, kernel_size=5, padding=2),
            nn.Conv2d(32, out_channels, kernel_size=5, padding=2),
        ])
        self.act = nn.SiLU() # The activation function
        self.downscale = nn.MaxPool2d(2)
        self.upscale = nn.Upsample(scale_factor=2)

    def forward(self, x):
        h = []
        for i, l in enumerate(self.down_layers):
            x = self.act(l(x)) # Through the layer and the activation function
            if i < 2: # For all but the third (final) down layer:
              h.append(x) # Storing output for skip connection
              x = self.downscale(x) # Downscale ready for the next layer

        for i, l in enumerate(self.up_layers):
            if i > 0: # For all except the first up layer
              x = self.upscale(x) # Upscale
              x += h.pop() # Fetching stored output (skip connection)
            x = self.act(l(x)) # Through the layer and the activation function
```

```
        return x
```

A diffusion model trained with this architecture on MNIST produces the following samples (code included in the supplementary material but omitted here for brevity):

Improving the UNet

This simple UNet works for this relatively easy task, but it is far from ideal. So, what can we do to improve it?

- Add more parameters. This can be accomplished by using multiple convolutional layers in each block, by using a larger number of filters in each convolutional layer, or by making the network deeper.
- Add residual connections. Using ResBlocks instead of regular convolutional layers can help the model learn more complex functions while keeping training stable.
- Add normalization, such as batch normalization. Batch normalization can help the model learn more quickly and reliably, by ensuring that the outputs of each layer are centered around 0 and have a standard deviation of 1.
- Add regularization, such as dropout. Dropout helps by preventing the model from overfitting to the training data, which is important when working with smaller datasets.
- Add attention. By introducing self-attention layers we allow the model to focus on different parts of the image at different times, which can help it learn more complex functions. The addition of transformer-like attention layers also lets us increase the number of learnable parameters, which can help the model learn more complex functions. The downside is that attention layers are much more expensive to compute than regular convolutional layers at higher resolutions, so we

typically only use them at lower resolutions (i.e. the lower resolution blocks in the UNet).

- Add an additional input for the timestep, so that the model can tailor its predicitons according to the noise level. This is called timestep conditioning, and is used in almost all recent diffusion models. We'll see more on conditional models in the next chapter.

For comparison, here are the results on MNIST when using the UNet implementation in the diffusers library, which features all of the above improvements:

> **WARNING**
>
> This section will likely be expanded with results and more details in the future. We just haven't gotten around to training variants with the different improvements yet!

Alternative Architectures

More recently, a number of alternative architectures have been proposed for diffusion models. These include:

- Transformers. The DiT paper ("Scalable Diffusion Models with Transformers") by Peebles and Xie showed that a transformer-based architecture can be used to train a diffusion model, with great results. However, the compute and memory requirements of the transformer architecture remain a challenge for very high resolutions.
- The *UViT* architecture from the Simple Diffusion paper link aims to get the best of both worlds by replacing the middle layers of the UNet with a large stack of transformer blocks. A key insight of this paper was that focusing the majority of the compute at the lower resolution blocks of the UNet allows for more efficient training of high-resolution diffusion models. For very high resolutions, they do some additional pre-processing using something called a wavelet transform to reduce the spatial resolution of the input image while keeping as much

information as possible through the use of additional channels, again reducing the amount of compute spent on the higher spatial resolutions.

- Recurrent Interface Networks. The RIN paper (Jabri et al) takes a similar approach, first mapping the high-resolution inputs to a more manageable and lower-dimensional 'latent' representation which is then processed by a stack of transformer blocks before being decoded back out to an image. Additionally, the RIN paper introduces an idea of 'recurrence' where information is passed to the model from the previous processing step, which can be beneficial for the kind of iterative improvement that diffusion models are designed to perform.

It remains to be seen whether transformer-based approaches completely supplant UNets as the go-to architecture for diffusion models, or whether hybrid approaches like the UViT and RIN architectures will prove to be the most effective.

In Depth: Objectives and Pre-Conditioning

We've spoken about diffusion models taking a noisy input and "learning to denoise" it. At first glance, you might assume that the natural prediction target for the network is the denoised version of the image, which we'll call `x0`. However, in the code, we compared the model prediction with the unit-variance noise that was used to create the noisy version (often called the epsilon objective, `eps`). The two appear mathematically identical since if we know the noise and the timestep we can derive `x0` and vice versa. While this is true, the choice of objective has some subtle effects on how large the loss is at different timesteps, and thus which noise levels the model learns to denoise best. To gain some intuition, let's visualize some different objectives across different timesteps:

At extremely low noise levels, the `x0` objective is trivially easy while predicting the noise accurately is almost impossible. Likewise, at extremely high noise levels, the `eps` objective is easy while predicting the denoised image accurately is almost impossible. Neither case is ideal, and so additional objectives have been introduced that have the model predict a mix of `x0` and `eps` at different timesteps. The `v` objective (introduced in "Progressive distillation for fast sampling of diffusion models." by Salimans and Ho) is one such objective, which is defined as $v = \sqrt{\alpha} \cdot \epsilon + \sqrt{1 - \alpha} \cdot x_0$. The EDM paper by Karras et al introduce a similar idea via a parameter called `c_skip`, and unify the different diffusion model formulations into a consistent framework. If you're interested in learning more about the different objectives, scalings and other nuances of the different diffusion model formulations, we recommend reading their paper for a more in-depth discussion.

Project Time: Train Your Own Diffusion Model

Now that you have an understanding of the basics of diffusion models, it's time to train some for yourself! The supplementary material for this chapter includes a notebook that walks you through the process of training a diffusion model on your own dataset. As you work through it, check back with this chapter and see how the different pieces fit together. The notebook

also includes lots of suggested changes you can make to better explore how different model architectures and training strategies affect the results.

Summary

In this chapter, we've seen how the idea of iterative refinement can be applied to train a diffusion model capable of turning noise into beautiful images. You've seen some of the design choices that go into creating a successful diffusion model, and hopefully put them into practice by training your own model. In the next chapter, we'll take a look at some of the more advanced techniques that have been developed to improve the performance of diffusion models and to give them extraordinary new capabilities!

References

Ho, Jonathan, Ajay Jain, and Pieter Abbeel. "Denoising diffusion probabilistic models." Advances in Neural Information Processing Systems 33 (2020): 6840-6851.

Ronneberger, O., Fischer, P. and Brox, T., 2015. U-net: Convolutional networks for biomedical image segmentation. In Medical Image Computing and Computer-Assisted Intervention–MICCAI 2015: 18th International Conference, Munich, Germany, October 5-9, 2015, Proceedings, Part III 18 (pp. 234-241). Springer International Publishing.

Bansal, Arpit, Eitan Borgnia, Hong-Min Chu, Jie S. Li, Hamid Kazemi, Furong Huang, Micah Goldblum, Jonas Geiping, and Tom Goldstein. "Cold diffusion: Inverting arbitrary image transforms without noise." arXiv preprint arXiv:2208.09392 (2022).

Hoogeboom, Emiel, Jonathan Heek, and Tim Salimans. "simple diffusion: End-to-end diffusion for high resolution images." arXiv preprint arXiv:2301.11093 (2023).

Chang, Huiwen, Han Zhang, Jarred Barber, A. J. Maschinot, Jose Lezama, Lu Jiang, Ming-Hsuan Yang et al. "Muse: Text-To-Image Generation via

Masked Generative Transformers.” arXiv preprint arXiv:2301.00704 (2023).

Chang, Huiwen, Han Zhang, Lu Jiang, Ce Liu, and William T. Freeman. “Maskgit: Masked generative image transformer.” In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 11315-11325. 2022.

Rampas, Dominic, Pablo Pernias, Elea Zhong, and Marc Aubreville. “Fast Text-Conditional Discrete Denoising on Vector-Quantized Latent Spaces.” arXiv preprint arXiv:2211.07292 (2022).

Chen, Ting “On the Importance of Noise Scheduling for Diffusion Models.” arXiv preprint arXiv:2301.10972 (2023).

Peebles, William, and Saining Xie. “Scalable Diffusion Models with Transformers.” arXiv preprint arXiv:2212.09748 (2022).

Jabri, Allan, David Fleet, and Ting Chen. “Scalable Adaptive Computation for Iterative Generation.” arXiv preprint arXiv:2212.11972 (2022).

Salimans, Tim, and Jonathan Ho. “Progressive distillation for fast sampling of diffusion models.” arXiv preprint arXiv:2202.00512 (2022).)

Karras, Tero, Miika Aittala, Timo Aila, and Samuli Laine. “Elucidating the design space of diffusion-based generative models.” arXiv preprint arXiv:2206.00364 (2022).

Chapter 2. Building up to Stable Diffusion

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the fourth chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at jleonard@oreilly.com.

In the previous chapter, we introduced diffusion models and the underlying idea of iterative refinement. By the end of the chapter, we could generate images, but training the model was time-consuming and we had no control over the images that were generated. In this chapter, we'll see how to go from this to text-conditioned models that can efficiently generate images based on text descriptions, with a model called Stable Diffusion (SD) as a case study. Before we get to SD, though, we'll first look at how conditional models work and go over some of the innovations that lead up to the text-to-image models we have today.

Adding Control: Conditional Diffusion Models

Before we deal with the problem of generating images from text descriptions (a very challenging task!), let's focus on something slightly easier first. We'll see how we can steer our models outputs towards specific types or *classes* of images. We can use a method called *conditioning*, where the idea is to ask the model to generate not just any image, but an image belonging to a pre-defined class.

Model *conditioning* is a simple but effective idea. We'll start from the same diffusion model we used in Chapter 3, with just a couple of changes. First, we'll use a new dataset called Fashion MNIST instead of butterflies so that we can identify categories easily. Then, crucially, we'll run two inputs through the model. Instead of just showing it how *real* images look like, we'll also tell it the class every image belongs to. We hope the model will learn to associate images and labels, so it gets an idea about the distinctive features of sweaters, boots and the like.

Note that we are not interested in solving a classification problem – we don't want the model to tell us the class, given an input image –. We still want it to perform the same task as in Chapter 3, namely: *please, generate plausible images that look like they came from this dataset*. The only difference is that we are giving it additional information about those images. We'll use the same loss function and training strategy, as it's the same task as before.

Preparing the Data

We need a dataset with distinct groups of images. Datasets intended for computer vision classification tasks are ideal for this purpose. We could start with something like the ImageNet dataset, which contains millions of images across 1000 classes. However, training models on this dataset would take an extremely long time. When approaching a new problem, it's often a good idea to start with a smaller dataset first, to make sure everything

works as expected. This keeps the feedback loop short, so we can iterate quickly and make sure we're on the right track.

For this example, we could choose MNIST as we did in Chapter 3. To make things just a little bit different, we'll choose Fashion MNIST instead. Fashion MNIST, developed and open-sourced by Zalando, is a replacement for MNIST that shares some of the same characteristics: a compact size, black & white images, and 10 classes. The main difference is that instead of being digits, classes correspond to different types of clothing and the images contain more detail than simple handwritten digits.

Let's look at some examples.

```
from datasets import load_dataset

fashion_mnist = load_dataset("fashion_mnist")
clothes = fashion_mnist["train"]["image"][:8]
classes = fashion_mnist["train"]["label"][:8]
show_images(clothes, titles=classes, figsize=(4,2.5))
```

So class *0* means t-shirt, *2* is a sweater and *9* means boot. Here's a list of the 10 categories in Fashion MNIST: *https://huggingface.co/datasets/fashion_mnist#data-fields*. We prepare our dataset and dataloader similarly to how we did it in Chapter 3, with the main difference that we'll also include the class information as an input. Instead of resizing, in this case we'll pad our image inputs (which have a size of *28* × *28* pixels) to *32* × *32*, as we did in Chapter 3.

```
preprocess = transforms.Compose([
    transforms.RandomHorizontalFlip(),   # Randomly flip (data augmentation)
    transforms.ToTensor(),               # Convert to tensor (0, 1)
```

```
    transforms.Pad(2),                  # Add 2 pixels on all sides
    transforms.Normalize([0.5], [0.5]),  # Map to (-1, 1)
])
batch_size = 256

def transform(examples):
    images = [preprocess(image.convert("L")) for image in examples["image"]]
    return {"images": images, "labels": examples["label"]}

train_dataset = fashion_mnist["train"].with_transform(transform)

train_dataloader = torch.utils.data.DataLoader(
    train_dataset, batch_size=batch_size, shuffle=True
)
```

Creating a Class-Conditioned Model

If we use the *UNet* model from the diffusers library, we can provide our own custom conditioning information. Here we create a similar model to the one we used in Chapter 3, but we add a *num_class_embeds* argument to the *UNet* constructor. This argument tells the model that we'd like to use class labels as additional conditioning. We'll use 10, because we have 10 classes in Fashion MNIST.

```
model = UNet2DModel(
    in_channels=1,   # 1 channel for grayscale images
    out_channels=1,  # output channels must also be 1
    sample_size=32,
    block_out_channels=(32, 64, 128, 256),
    norm_num_groups=8,
    num_class_embeds=10, # Enable class conditioning
)
```

To make predictions with this model, we must pass in the class labels as additional inputs to the *forward* method:

```
x = torch.randn((1, 1, 32, 32))
with torch.no_grad():
    out = model(x, timestep=7, class_labels=torch.tensor([2])).sample
out.shape
torch.Size([1, 1, 32, 32])
```

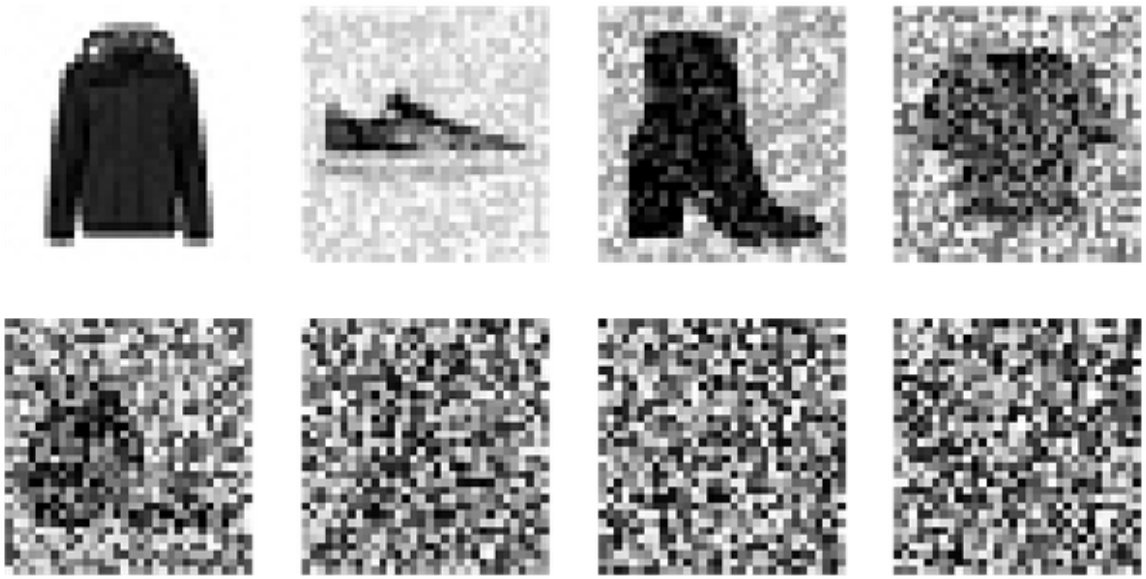
> **NOTE**
>
> You'll notice we also pass something else to the model as conditioning: the timestep! That's right, even the model from Chapter 3 can be considered a conditional diffusion model! We condition it on the timestep in the hopes that knowing how far we are in the diffusion process will help it generate more realistic images.

Internally, both the timestep and the class label are turned into **embeddings** that the model uses during its forward pass. At multiple stages throughout the UNet, these embeddings are projected onto a dimension that matches the number of channels in a given layer and are then added to the outputs of that layer. This means the conditioning information is fed to every block of the UNet, giving the model ample opportunity to learn how to use it effectively.

Training the Model

Adding noise works just as well on greyscale images as it did on the butterflies from Chapter 3.

```
scheduler = DDPMScheduler(num_train_timesteps=1000, beta_start=0.0001,
beta_end=0.02)
timesteps = torch.linspace(0, 999, 8).long()
batch = next(iter(train_dataloader))
x = batch['images'][:8]
noise = torch.rand_like(x)
noised_x = scheduler.add_noise(x, noise, timesteps)
show_images((noised_x*0.5 + 0.5).clip(0, 1))
```



Our training loop is also almost exactly the same as in Chapter 3 too, except that we now pass the class labels for conditioning. Note that this is just additional information for the model, but it doesn't affect our loss function in any way.

```
for step, batch in enumerate(train_dataloader):
        # Load the input images
        clean_images = batch["images"].to(device)
        class_labels = batch["labels"].to(device)

        # *Sample noise to add to the images*
        # *Sample a random timestep for each image*
        # *Add noise to the clean images according to the timestep*

        # Get the model prediction for the noise - note the use of
class_labels
        noise_pred = model(noisy_images, timesteps, class_labels=class_labels,
return_dict=False)[0]

        # *Calculate the loss and update the parameters as before*
        ...
```

In this case we train for 25 epochs - full code can be found in the supplementary material.

Sampling

Now we have a model that expects two inputs when making predictions: the image and the class label. We can create samples by beginning with random noise and then iteratively denoising, passing in whatever class label we'd like to generate:

```
def generate_from_class(class_to_generate, n_samples=8):
    sample = torch.randn(n_samples, 1, 32, 32).to(device)
    class_labels = [class_to_generate] * n_samples
    class_labels = torch.tensor(class_labels).to(device)

    for i, t in tqdm(enumerate(scheduler.timesteps)):
        # Get model pred
        with torch.no_grad():
            noise_pred = model(sample, t, class_labels=class_labels).sample

        # Update sample with step
        sample = scheduler.step(noise_pred, t, sample).prev_sample

    return sample.clip(-1, 1)*0.5 + 0.5
# Generate t-shirts (class 0)
images = generate_from_class(0)
show_images(images, nrows=2)
1000it [00:21, 47.25it/s]
```

```
# Now generate some sneakers (class 7)
images = generate_from_class(7)
show_images(images, nrows=2)
```
```
1000it [00:21, 47.20it/s]
```

```
# ...or boots (class 9)
images = generate_from_class(9)
show_images(images, nrows=2)
```
```
1000it [00:21, 47.26it/s]
```

As you can see, the generated images are far from perfect. They'd probably get much better if we explored the architecture and trained for longer. But it's amazing that the model not only learnt the shapes of different types of clothing, but also realized that shape *9* looks different than shape *0*, just by sending this information alongside the training data. To put it in a slightly different way: the model is used to seeing the number *9* accompanying boots. When we ask it to generate an image and provide the *9*, it responds with a boot. We have successfully built a class-conditioned model capable of generating images **conditioned** on class labels from fasionMNIST!

Improving Efficiency: Latent Diffusion

Now that we can train a conditional model, all we need to do is scale it up and condition it on text instead of class labels, right? Well, not quite. As image size grows, so does the computational power required to work with those images. This is especially pronounced in an operation called self-attention, where the amount of operations grows quadratically with the number of inputs. A 128px square image has 4x as many pixels as a 64px square image, and so requires 16x (i.e. 4^2) the memory and compute in a self-attention layer. This is a problem for anyone who'd like to generate high-resolution images!

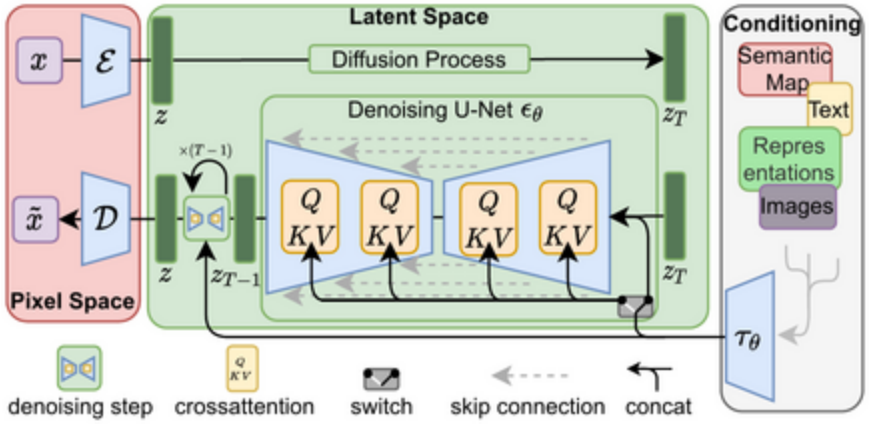


Figure 2-1. The architecture introduced in the Latent Diffusion Models paper. Note the VAE encoder and decoder on the left for translating between pixel space and latent space

Latent diffusion tries to mitigate this issue by using a separate model called a Variational Auto-Encoder (VAE). As we saw in Chapter 2, VAEs can compress images to a smaller spatial dimension. The rationale behind this is that images tend to contain a large amount of redundant information - given enough training data, a VAE can hopefully learn to produce a much smaller representation of an input image and then reconstruct the image based on this small latent representation with a high degree of fidelity. The VAE used in SD takes in 3-channel images and produces a 4-channel latent representation with a reduction factor of 8 for each spatial dimension. That is, a 512px square input image will be compressed down to a 4x64x64 latent.

By applying the diffusion process on **these smaller latent representations** rather than on full-resolution images, we can get many of the benefits that would come from using smaller images (lower memory usage, fewer layers needed in the UNet, faster generation times…) and still decode the result back to a high-resolution image once we're ready to view it. This innovation dramatically lowers the cost to train and run these models. The paper that introduced this idea (High-Resolution Image Synthesis with Latent Diffusion Models by Rombach et al) demonstrated the power of this technique by training models conditioned on segmentation maps, class labels and text. The impressive results led to further collaboration between the authors and partners such as RunwayML, LAION, and EleutherAI to train a more powerful version of the model, which later became Stable Diffusion.

Stable Diffusion: Components in Depth

Stable Diffusion is a text-conditioned latent diffusion model. Thanks to its popularity, there are hundreds of websites and apps that let you use it to create images with no technical knowledge required. It's also very well-supported by libraries like `diffusers`, which let us sample an image with SD using a user-friendly pipeline:

```
pipe("Watercolor illustration of a rose").images[0]
```

```
0%|          | 0/50 [00:00<?, ?it/s]
```

In this section we'll explore all of the components that make this possible.

The Text Encoder

So how does Stable Diffusion understand text? Earlier on we showed how feeding additional information to the UNet allows us to have some additional control over the types of images generated. Given a noisy version of an image, the model is tasked with predicting the denoised version based on additional clues such as a class label. In the case of SD, the additional *clue* is the text prompt. At inference time, we can feed in the description of an image we'd like to see and some pure noise as a starting point, and the

model does its best to *denoise* the random input into something that matches the caption.

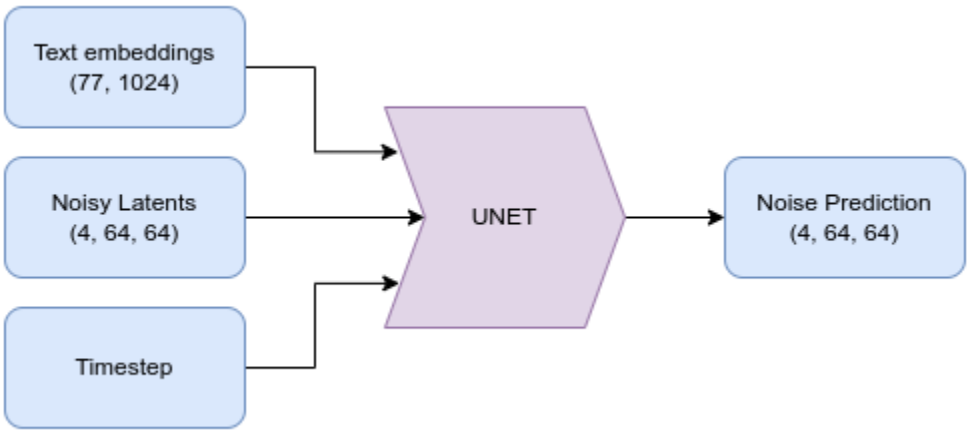


Figure 2-2. The text encoder turns an input string into text embeddings which are fed into the UNet along with the timestep and the noisy latents.

For this to work, we need to create a numeric representation of the text that captures relevant information about what it describes. To do this, SD leverages a pre-trained transformer model based on CLIP, which was also introduced in Chapter 2. The text encoder is a transformer model that takes in a sequence of tokens and produces a 1024-dimensional vector for each token (0r 768-dimensional in the case of SD version 1 which we're using for the demonstrations in this section). Instead of combining these vectors into a single representation, we keep them separate and use them as conditioning for the UNet. This allows the UNet to make use of the information in each token separately, rather than just the overall meaning of the entire prompt. Because we're extracting these text embeddings from the internal representation of the CLIP model, they are often called the "encoder hidden states". Figure 3 shows the text encoder architecture.

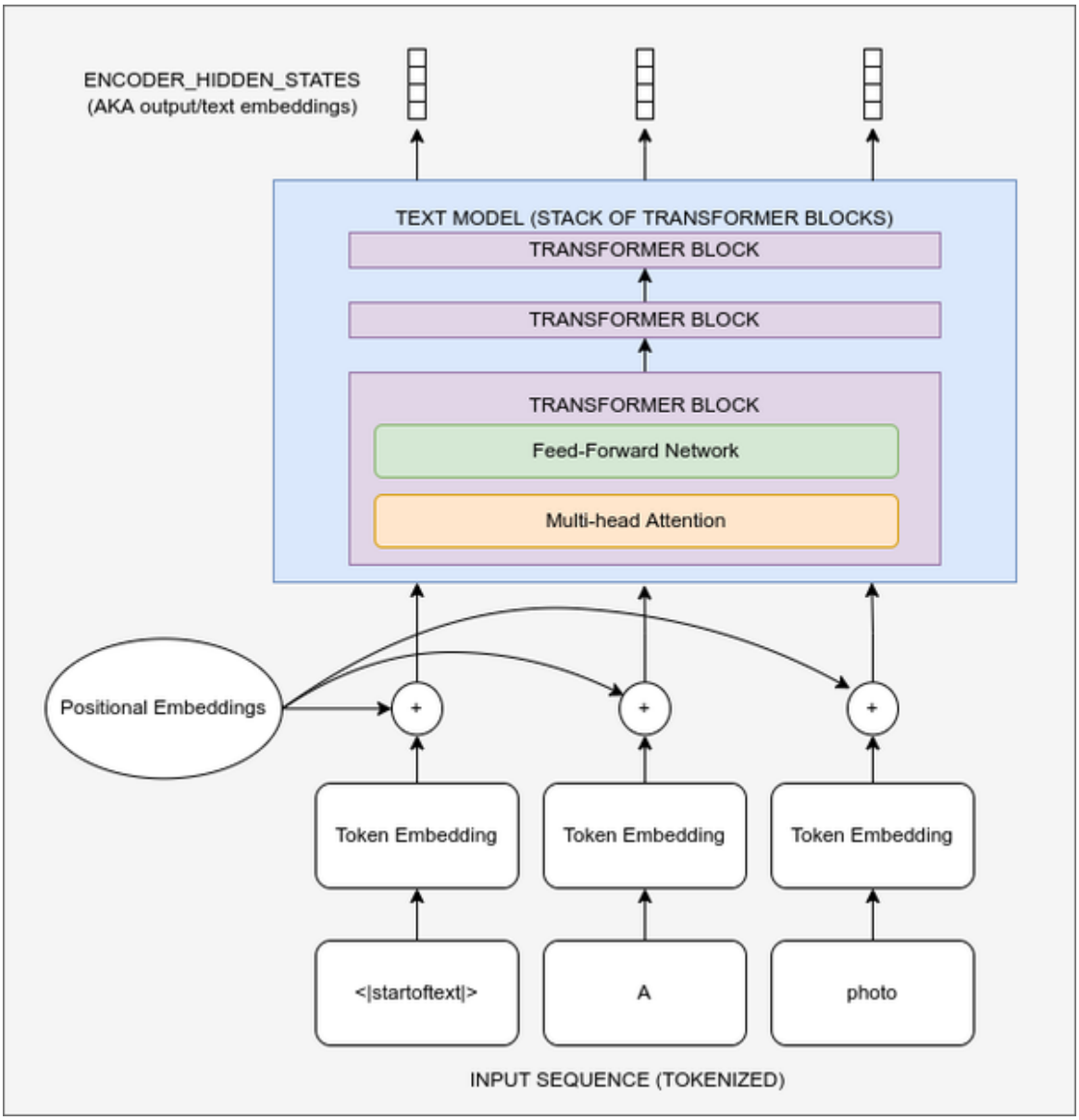


Figure 2-3. Diagram showing the text encoding process which transforms the input prompt into a set of text embeddings (the encoder_hidden_states) which can then be fed in as conditioning to the UNet.

The first step to encode text is to follow a process called **tokenization**. This converts a sequence of characters into a sequence of numbers, where each number represents a group of various characters. Characters that are usually found together (like most common words) can be assigned a single token that represents the whole word or group. Long or complicated words, or words with many inflections, may be translated to multiple tokens, where each one usually represents a meaningful section of the word.

There is no single “best” tokenizer; instead, each language model comes with its own one. Differences reside in the number of tokens supported, and on the tokenization strategy – do we use single characters, as we just described, or should we consider different primitive units. In the following example we see how the tokenization of a phrase works with Stable Diffusion’s tokenizer. Each word in our sentence is assigned a unique token number (for example, `photograph` happens to be `8853` in the tokenizer’s vocabulary). There are also additional tokens that are used to provide additional context, such as the point where the sentence ends.

```
prompt = 'A photograph of a puppy'
# Turn the text into a sequence of tokens:
text_input = pipe.tokenizer(prompt, padding="max_length",
                            max_length=pipe.tokenizer.model_max_length,
                            truncation=True, return_tensors="pt")

# See the individual tokens
for t in text_input['input_ids'][0][:8]: # We'll just look at the first 7
    print(t, pipe.tokenizer.decoder.get(int(t)))
tensor(49406) <|startoftext|>
tensor(320) a</w>
tensor(8853) photograph</w>
tensor(539) of</w>
tensor(320) a</w>
tensor(6829) puppy</w>
tensor(49407) <|endoftext|>
tensor(49407) <|endoftext|>
```

Once the text is tokenized, we can pass it through the text encoder to get the final text embeddings that will be fed into the UNet:

```
# Grab the output embeddings
text_embeddings = pipe.text_encoder(text_input.input_ids.to(device))[0]
print('Text embeddings shape:', text_embeddings.shape)
Text embeddings shape: torch.Size([1, 77, 768])
```

We’ll go into more detail about how a transformer model processes a string of tokens in the chapters focusing on transformer models.

Classifier-free guidance

It turns out that even with all of the effort put into making the text conditioning as useful as possible, the model still tends to default to relying mostly on the noisy input image rather than the prompt when making its predictions. In a way, this makes sense - many captions are only loosely related to their associated images and so the model learns not to rely too heavily on the descriptions! However, this is undesirable when it comes time to generate new images - if the model doesn't follow the prompt then we may get images out that don't relate to our description at all.

Figure 2-4. Images generated from the prompt "An oil painting of a collie in a top hat" with CFG scale 0, 1, 2 and 10 (left to right)

To fix this, we use a trick called Classifier-Free Guidance (CGF). During training, text conditioning is sometimes kept blank, forcing the model to learn to denoise images with no text information whatsoever (unconditional generation). Then at inference time, we make two separate predictions: one with the text prompt as conditioning and one without. We can then use the difference between these two predictions to create a final combined prediction that pushes even further in the direction indicated by the text-conditioned prediction according to some scaling factor (the guidance scale), hopefully resulting in an image that better matches the prompt. Figure 4 shows the outputs for a prompt at different guidance scales - as you can see, higher values result in images that better match the description.

The VAE

The VAE is tasked with compressing images into a smaller latent representation and back again. The VAE used with Stable Diffusion is a

truly impressive model. We won't go into the training details here, but in addition to the usual reconstruction loss and KL divergence described in Chapter 2 they use an additional patch-based discriminator loss to help the model learn to output plausible details and textures. This adds a GAN-like component to training and helps to avoid the slightly blurry outputs that were typical in previous VAEs. Like the text encoder, the VAE is usually trained separately and used as a frozen component during the diffusion model training and sampling process.

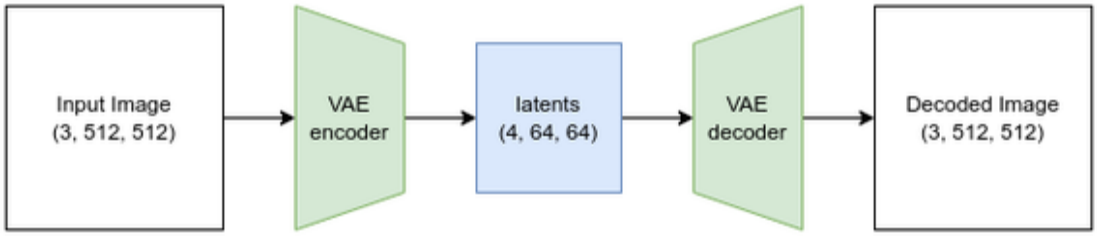


Figure 2-5. Encoding and decoding and image with the VAE

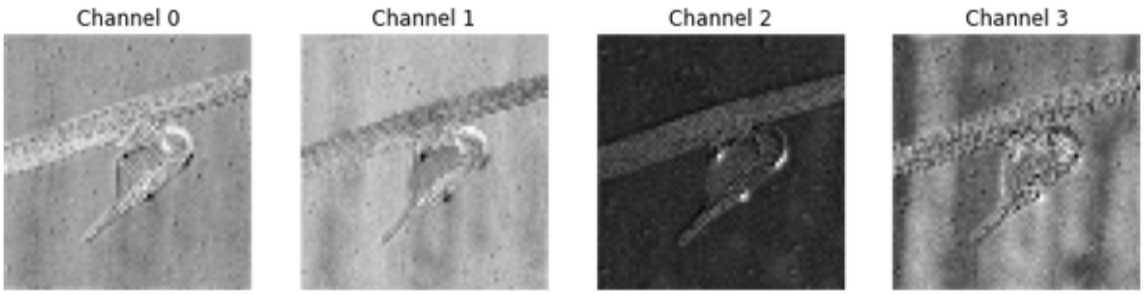
Let's load an image and see what it looks like after being compressed and decompressed by the VAE:

```
# NB, this will be our own image as part of the supplementary material to
avoid external URLs
im = load_image('https://images.pexels.com/photos/14588602/pexels-photo-
14588602.jpeg', size=(512, 512))
show_image(im);
```

```
# Encode the image
with torch.no_grad():
    tensor_im = transforms.ToTensor()(im).unsqueeze(0).to(device)*2-1
    latent = vae.encode(tensor_im.half()) # Encode the image to a distribution
    latents = latent.latent_dist.sample() # Sampling from the distribution
    latents = latents * 0.18215 # This scaling factor was introduced by the SD
authors to reduce the variance of the latents

latents.shape
torch.Size([1, 4, 64, 64])
```

Visualizing the low-resolution latent representation, we can that some of the rough structure of the input image is still visible in the different channels:

```
# Plot the individual channels of the latent representation
show_images([l for l in latents[0]], titles=[f'Channel {i}' for i in
range(latents.shape[1])], ncols=4)
```



And decoding back to image space, we get an output image that is almost identical to the original. Can you spot the difference?

```
# Decode the image
with torch.no_grad():
    image = vae.decode(latents / 0.18215).sample
image = (image / 2 + 0.5).clamp(0, 1)
show_image(image[0].float());
```

When generating images from scratch, we create a random set of latents as the starting point. We iteratively refine these noisy latents to generate a sample, and then the VAE decoder is used to decode these final latents into an image we can view. The encoder is only used if we'd like to start the process from an existing image, something we'll explore in chapter 5.

The UNet

The UNet used in stable diffusion is somewhat similar to the one we used in chapter 3 for generating images. Instead of taking in a 3-channel image as the input we take in a 4-channel latent. The timestep embedding is fed in in the same way as the class conditioning was in the example at the start of this chapter. But this UNet also needs to accept the text embeddings as additional conditioning. Scattered throughout the UNet are cross-attention layers. Each spatial location in the UNet can *attend* to different tokens in the text conditioning, bringing in relevant information from the prompt. The diagram in Figure 7 shows how this text conditioning (as well as timestep-based conditioning) is fed in at different points.

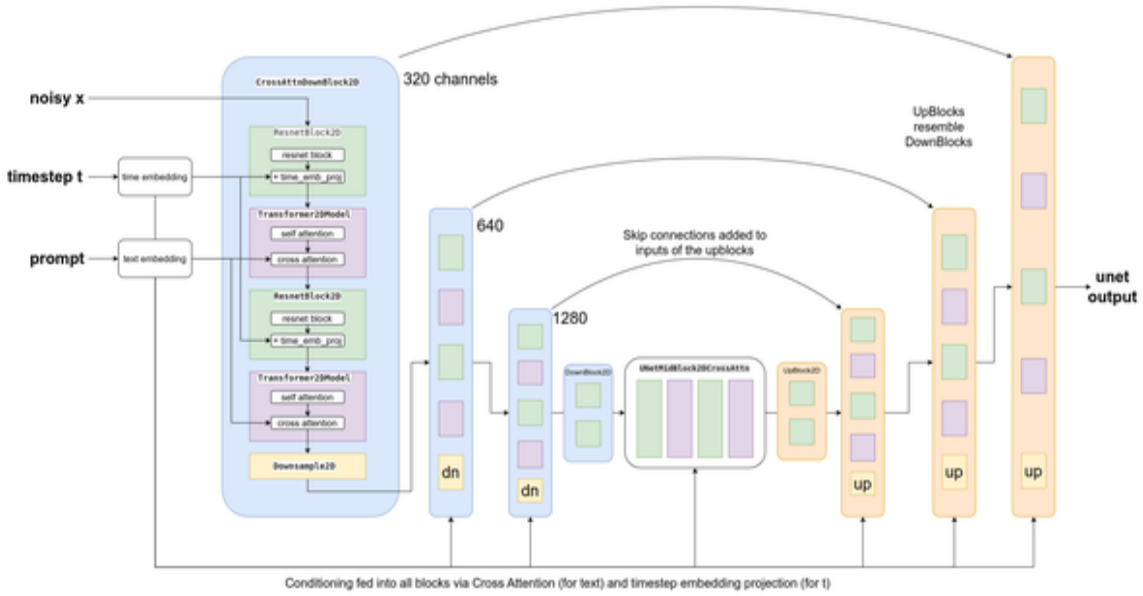


Figure 2-6. The Stable Diffusion UNet

The UNet for Stable Diffusion version 1 and 2 has around 860 million parameters. The more recent SD XL has even more, at around (details TBC), with most of the additional parameters being added at the lower-resolution stages via additional channels in the residual blocks (N vs 1280 in the original) and additional transformer blocks.

NB: Stable Diffusion XL has not yet been publically released, so this section will be updated when more information is public.

Putting it All Together: Annotated Sampling Loop

Now that we know what each of the components does, let's put them together to generate an image without relying on the pipeline. Here are the settings we'll use:

```
# Some settings
prompt = ["Acrylic palette knife painting of a flower"] # What we want to
generate
height = 512                              # default height of Stable Diffusion
width = 512                               # default width of Stable Diffusion
num_inference_steps = 30                  # Number of denoising steps
```

```
guidance_scale = 7.5                # Scale for classifier-free guidance
seed = 42                           # Seed for random number generator
```

The first step is to encode the text prompt. Because we plan to do classifier-free guidance, we'll actually create two sets of text embeddings: one with the prompt, and one representing an empty string. You can also encode a *negative prompt* in place of the empty string, or combine multiple prompts with different weightings, but this is the most common usage:

```
# Tokenize the input
text_input = pipe.tokenizer(prompt, padding="max_length",
max_length=pipe.tokenizer.model_max_length, truncation=True,
return_tensors="pt")

# Feed through the text encoder
with torch.no_grad():
    text_embeddings = pipe.text_encoder(text_input.input_ids.to(device))[0]

# Do the same for the unconditional input (a blank string)
uncond_input = pipe.tokenizer("", padding="max_length",
max_length=pipe.tokenizer.model_max_length, return_tensors="pt")
with torch.no_grad():
    uncond_embeddings = pipe.text_encoder(uncond_input.input_ids.to(device))
[0]

# Concatenate the two sets of text embeddings embeddings
text_embeddings = torch.cat([uncond_embeddings, text_embeddings])
```

Next we create our random initial latents and set up the scheduler to use the desired number of inference steps:

```
# Prepare the Scheduler
pipe.scheduler.set_timesteps(num_inference_steps)

# Prepare the random starting latents
latents = torch.randn(
    (1, pipe.unet.in_channels, height // 8, width // 8), # Shape of the latent
representation
    generator=torch.manual_seed(32),  # Seed the random number generator
).to(device).half()
latents = latents * pipe.scheduler.init_noise_sigma
```

Now we loop through the sampling steps, getting the model prediction at each stage and using this to update the latents:

```
# Sampling loop
for i, t in enumerate(pipe.scheduler.timesteps):

    # Create two copies of the latents to match the two text embeddings
(unconditional and conditional)
    latent_model_input = torch.cat([latents] * 2)
    latent_model_input = pipe.scheduler.scale_model_input(latent_model_input,
t)

    # predict the noise residual for both sets of inputs
    with torch.no_grad():
        noise_pred = pipe.unet(latent_model_input, t,
encoder_hidden_states=text_embeddings).sample

    # Split the prediction into unconditional and conditional versions:
    noise_pred_uncond, noise_pred_text = noise_pred.chunk(2)

    # perform classifier-free guidance
    noise_pred = noise_pred_uncond + guidance_scale * (noise_pred_text -
noise_pred_uncond)

    # compute the previous noisy sample x_t -> x_t-1
    latents = pipe.scheduler.step(noise_pred, t, latents).prev_sample
```

Notice the classifier-free guidance step. Our final noise prediction is *noise_pred_uncond + guidance_scale * (noise_pred_text - noise_pred_uncond)*, pushing the prediction *away* from the unconditional prediction towards the prediction made based on the prompt. Try changing the guidance scale to see how this affects the output.

By the end of the loop the latents should hopefully now represent a plausible image that matches the prompt. The final step is to decode the latents into an image using the VAE so that we can see the result:

```
# scale and decode the image latents with vae
latents = 1 / 0.18215 * latents
with torch.no_grad():
    image = vae.decode(latents).sample
image = (image / 2 + 0.5).clamp(0, 1)
```

```
# Display
show_image(image[0].float());
```

If you explore the source code for the *StableDiffusionPipeline* you'll see that the code above closely matches the *call* method used by the pipeline. Hopefully this annotated version shows that there is nothing too magical going on behind the scenes! Use this as a reference for when we encounter additional pipelines that add additional tricks to this foundation.

Training Data for Text-To-Image Models (TBD)

NB: We may add a more in-depth section here with the history and technical details of how LAION came together, and some of the nuances and debate around training on public data scraped from the internet.

Open Data, Open Models

The LAION-5B dataset includes over 5 billion image-caption pairs scraped from the internet. This dataset was created by and for the open-source community, which saw the need for a publically-accessible dataset of this kind. Before the LAION initiative, only a handful of research labs at large companies had access to such data. These organizations kept the details of

their private datasets to themselves, which made their results impossible to validate or replicate. By creating a publically available source of training data, LAION enabled a wave of smaller communities and organizations to train models and perform research that would otherwise have been impossible.

Figure 2-7. "An explosion of artistic creativity" - Image generated by the authors using Stable Diffusion

Stable Diffusion was one such model, trained on a subset of LAION as part of a collaboration between the researchers who had invented latent diffusion models and an organization called Stability AI. Training a model like SD requires a significant amount of GPU time. Even with the freely-available LAION dataset, there aren't many who could afford the investment. This is why the public release of the model weights and code was such a big deal - it marked the first time a powerful text-to-image model with similar capabilities to the best closed-source alternatives was available to all. Stable Diffusion's public availability has made it the go-to choice for researchers and developers looking to explore this technology over the past year. Hundreds of papers build upon the base model, adding new capabilities or finding innovative ways to improve its speed and quality. And innumerable startups have found ways to integrate these rapidly-improving tools into their products, spawning an entire ecosystem of new applications.

The months after the introduction of Stable Diffusion demonstrated the impact of sharing these technologies in the open. SD is not the best text-to-

image model, but it IS the best model most of us had access to, so thousands of people have spent their time making it better and building upon that open foundation. We hope this example encourages others to follow suit and share their work with the open-source community in the future!

Summary

In this chapter we've seen how *conditioning* gives us new ways to control the images generated by diffusion models. We've seen how latent diffusion lets us train diffusion models more efficiently. We've seen how a text encoder can be used to condition a diffusion model on a text prompt, enabling powerful text-to-image capabilities. And we've explored how all of this comes together in the Stable Diffusion model by digging into the sampling loop and seeing how the different components work together. In the next chapter, we'll show some of the many additional capabilities that can be added to diffusion models such as SD to take them beyond simple image generation. And later, in part 2 of the book, you'll learn how to fine-tune SD to add new knowledge or capabilities to the model.

About the Authors

Pedro Cuenca is a machine learning engineer who works on diffusion software, models, and applications at Hugging Face.

Apolinário Passos is a machine learning art engineer at Hugging Face, working across teams on multiple machine learning for art and creativity use cases.

Omar Sanseviero is a lead machine learning engineer at Hugging Face, where he works at the intersection of open source, community, and product. Previously, Omar worked at Google on Google Assistant and TensorFlow.

Jonathan Whitaker is a data scientist and deep learning researcher focused on generative modeling. Besides his research and consulting work, his main focus is on sharing knowledge, which he does via the DataScienceCastnet YouTube channel and various free online resources he has created.