Neural Networks

GLIDE: Towards Photorealistic Image Generation and Editing with Text-Guided Diffusion Models

Domenico Muller 1708331

16 September 2022

Contents

Introduction

Text-to-image generation has been one of the most active and exciting AI fields in recent years. Therefore, a tool capable of generating realistic images from natural language can empower humans to create rich and diverse visual content with unprecedented ease. The ability to edit images using natural language further allows for iterative refinement and fine-grained control. Recent text-conditional image models (like **GAN**) are capable of synthesizing images from free-form text prompts, and can compose unrelated objects in semantically plausible ways. However, they are not yet able to generate photorealistic images that capture all aspects of their corresponding text prompts.

In this paper the **OpenAI** team showed how they built a **Diffusion Model** capable of creating an image starting from a simple word or from a more complex text, **GLIDE**, which stands for **G**uided **L**anguage to **I**mage **D**iffusion for **G**eneration and **E**diting.

Before GLIDE, OpenAI's work on diffusion models generated images from class labels, but now, this next iteration has successfully **integrated textual information into the generation process**.

Diffusion models have emerged as a promising family of generative models, achieving state-of-the-art sample quality on a number of image generation benchmarks, especially when paired with a guidance technique to trade off diversity for fidelity.

We will see how they implemented two different guidance strategies: **CLIP guidance** and **classifier-free guidance**, and, additionally, how they fine-tuned their model to perform **image in-painting**, enabling powerful text-driven image editing.

Models, Techniques and Training

In this chapter we will see the theory behind the **used models and techniques**, starting from understanding what a Diffusion Model is, in order to understand how GLIDE works.

2.1 Diffusion Models

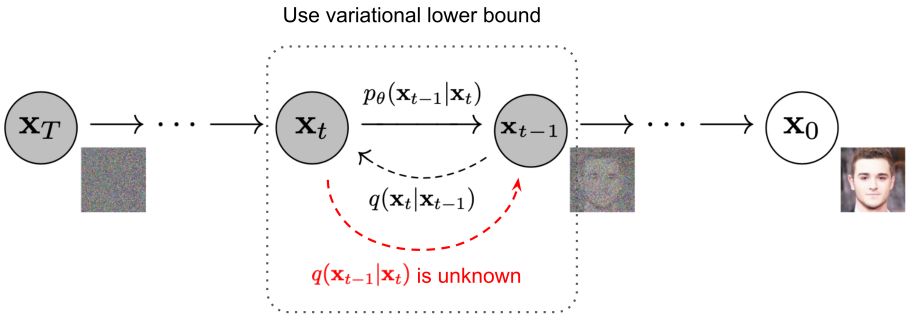
The key concept in **Diffusion Modelling** is that if we could build a learning model which can learn the systematic decay of information due to noise, then it should be possible to reverse the process and therefore, recover the information back from the noise. Diffusion models assume that the data is generated through a **reverse process** which is a Markov chain with transitions that successively denoises a latent $\mathbf{x}_T$ until you get a sample from the data distribution $\mathbf{x}_0$

$$p(\mathbf{x}_T) = \mathcal{N}(\mathbf{0},\mathbf{I})$$
$$p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t) = \mathcal{N}(\mu_\theta(\mathbf{x}_t, t), \Sigma_\theta(\mathbf{x}_t, t))$$

It is associated with the **forward process**, or diffusion process, fixed to a Markov chain that successively adds more Gaussian noise to an initial data point $\mathbf{x}_0$ over a sequence of timesteps

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

The data sample $\mathbf{x}_0$ gradually loses its distinguishable features as the $\mathbf{t}$ step becomes larger.



The model learns to minimise

$$E_q\left[-\log\frac{p_\theta(\mathbf{X}_{0:T})}{q(\mathbf{X}_{1:T}|\mathbf{X}_0)}\right]$$

which is the variational bound on the negative log likelihood

$$E\left[-\log p_\theta(\mathbf{x}_0)\right]$$

In practice you can train a model that predicts $(\mu_\theta(\mathbf{x}_t, t), \Sigma_\theta(\mathbf{x}_t, t))$, and sample for $\mathbf{T}$ successive steps from p_θ until you get to $\mathbf{x}_0$.

2.2 CLIP model

OpenAI developed a large scale model jointly trained on language and image pairs using a contrastive loss, the CLIP model, that consists of two separate pieces: an image encoder ***f(x)*** and a caption encoder ***g(c)***. You sample pairs ***(x,c)*** from a large dataset and train the model with contrastive cross-entropy loss that maximises the dot product ***f(x)·g(c)*** if ***c*** is paired with ***x*** and minimises it otherwise.

The model can then generate embeddings for an image or a text input in a shared embedding space. That means you can **find the cosine similarity of the embeddings for an image and some text**, such as a caption, so that you can determine the relevance of the text to the image.

This is quite powerful because you can train zero shot classifiers by just obtaining embeddings for an image and a bunch of labels, represented as text, and predicting the one with the highest cosine similarity. In GLIDE as well as DALL-E 2, CLIP is used to **supervise** the generation of images.

2.3 Guided diffusion

The vanilla diffusion model generates inputs from **randomly sampled latents**. Later papers introduce class conditional diffusion models where a class label is also input to the model such it has **mean** and the **variance**

$$(\mu_\theta(\mathbf{x}_t|y), \Sigma_\theta(\mathbf{x}_t|y))$$

In addition, during sampling you can perturb prediction of the model so as to guide the **sample towards the label**. You can also omit the label conditioning and use the label only to guide sampling. There are several approaches to guided sampling.

2.4 Classifier-based guidance

The mean is perturbed by the gradient of the log probability of the target class $\boldsymbol{y}$ predicted by a trained classifier $p_\phi(y|\mathbf{x}_t)$. The resulting new perturbed mean is

$$\hat{\mu}_\theta(\mathbf{x}_t|y) = \mu_\theta(\mathbf{x}_t|y) + s \cdot \Sigma_\theta(\mathbf{x}_t|y)\nabla_{x_t}\log\, p_\phi(y|\mathbf{x}_t)$$

where the coefficient $\boldsymbol{s}$ is called the **guidance scale** and increasing it improves sample quality at the cost of diversity.

2.5 CLIP guidance

As mentioned earlier, CLIP comprises two parts, an image encoder $\boldsymbol{f(x)}$ and a caption encoder $\boldsymbol{g(c)}$. The classifier is first trained on noised images to obtain the correct gradient in the reverse process, and during the diffusion sampling process, gradients from the classifier are used to guide the sample towards the label. This approach is similar to guided diffusion, here we perturb the reverse-process mean with the gradient of the dot product of the image and caption encodings with respect to the image:

$$\hat{\mu}_\theta(\mathbf{x}_t|y) = \mu_\theta(\mathbf{x}_t|y) + s \cdot \Sigma_\theta(\mathbf{x}_t|y)\nabla_{x_t}\log\, (f(\mathbf{x}_t) \cdot g(c))$$

To better match the classifier guidance defined as in the "Diffusion Models Beat GANs on Image Synthesis" paper from Dhariwal & Nichol (2021), they trained noised CLIP models with an image encoder $f(x_t, t)$ that receives noised images x_T and is otherwise trained with the same objective as the original CLIP model. They trained these models at 64×64 resolution with the same noise schedule as their base model.

2.6 Classifier-free guidance

To avoid having to train an extra model, a classifier-free guidance method is proposed. Here it is assumed that the prediction of the model is $\epsilon_\theta(\mathbf{x}_t|y)$. You **jointly train conditional and unconditional models** $\epsilon_\theta(\mathbf{x}_t|y)$ and $\epsilon_\theta(\mathbf{x}_t)$. You can do this by randomly letting the $\boldsymbol{y}$ be all zeros, in this case in the training procedure 20% of text token sequences are replaced with the empty sequence. Then the perturbation is then proportional to the difference between the predictions of the conditional and unconditional models.

$$\hat{\epsilon}_\theta(\mathbf{x}_t|y) = \epsilon_\theta(\mathbf{x}_t|\emptyset) + s \cdot (\epsilon_\theta(\mathbf{x}_t|y) - \epsilon_\theta(\mathbf{x}_t|\emptyset))$$

Classifier-free guidance has **two appealing properties**. First, it allows a single model to leverage its own knowledge during guidance, rather than relying on the knowledge of a separate (and sometimes smaller) classification model. Second, it simplifies guidance when conditioning on information that is difficult to predict with a classifier (such as text).

2.7 GLIDE

Now we can understand what GLIDE is and how it works. The GLIDE architecture can be boiled down to **three parts**: an Ablated Diffusion Model (ADM) trained to generate a 64 x 64 image, a text model (Transformer) that influences that image generation through a text prompt, and an upsampling model that takes our small 64 x 64 images to a 256 x 256 pixels format. The first two components interact

with one another to guide the image generation process to accurately reflect the text prompt, and the latter is necessary to make the images we produce more easy to interpret.

The roots of the GLIDE project, as already said, lie in "Diffusion Models Beat GANs on Image Synthesis", which showed that ADM approaches could outperform contemporarily popular, state-of-the-art generative models in terms of image sample quality. The GLIDE authors used the same ImageNet 64×64 model as Dhariwal & Nichol for the ADM, but unlike them, however, the GLIDE team wanted to be able to influence the image generation process more directly, so they augmented it with **text conditioning information**. For each noised image x_t and corresponding text caption c, their model predicts $p(\mathbf{x}_{t-1}|\mathbf{x}_t, c)$. To condition on the text, they first encode it into a sequence of K tokens, and feed these tokens into a Transformer model. The output of this transformer is used in **two ways**: first, the final token embedding is used in place of a class embedding in the ADM model; second, the last layer of token embeddings (a sequence of K feature vectors) is separately projected to the dimensionality of each attention layer throughout the ADM model, and then concatenated to the attention context at each layer. So additionally, each attention layer in the model is also attending to all the text tokens that the transformer produces when encoding the text.

They trained their base model on the same dataset as DALL-E and, as said, they used the same model architecture as the ImageNet 64×64 model from Dhariwal & Nichol, but scale the model width to 512 channels, resulting in roughly 2.3 billion parameters for the visual part of the model. For the text encoding Transformer, they used 24 residual blocks of width 2048, resulting in roughly 1.2 billion parameters, and finally they trained the 1.5 billion parameter upsampling diffusion model to go from 64×64 to 256×256 resolution. This model is conditioned on text in the same way as the base model, but uses a smaller text encoder with width 1024 instead of 2048. Otherwise, the architecture matches the ImageNet upsampler from Dhariwal & Nichol, except that they increased the number of base channels to 384. They trained the base model for 2.5M iterations at batch size 2048 and the upsam-

pling model for 1.6M iterations at batch size 512. The total training compute is roughly equal to that used to train DALL-E.

2.8 Inpainting

They fine-tune the model to perform image inpainting, finding that it is capable of making **realistic edits to existing images** using natural language prompts. It can be performed by sampling from the diffusion model as usual, but replacing the known region of the image with a sample from $q(x_t|x_0)$ after each sampling step. This has the disadvantage that the model cannot see the entire context during the sampling process (only a noised version of it), occasionally resulting in undesired edge artifacts in our early experiments.

To achieve better results, during fine-tuning, random regions of training examples are erased, and the remaining portions are fed into the model along with a mask channel as additional conditioning information. They modified the model architecture to have four additional input channels: a second set of RGB channels, and a mask channel. They initialized the corresponding input weights for these new channels to zero before fine-tuning. For the up-sampling model, however, they always provided the full low-resolution image, but only provided the unmasked region of the high-resolution image.

2.9 GLIDE(filtered)

OPENAI observed that their resulting model can significantly reduce the effort required to produce convincing disinformation or Deepfakes. To safeguard against these use cases while aiding future research, they released a smaller diffusion model and a noised CLIP model trained on filtered datasets. They gathered a dataset of several hundred million images from the internet, which is largely disjoint from the datasets used to train CLIP and DALL-E, and then applied several filters to this data. Training images containing people were filtered, to reduce the capabili-

ties of the model in many people-centric problematic use cases. They also filtered out several violent images and hate symbols as well. So they trained a small 300 million parameter model, which they refer as "GLIDE(filtered)". As just said, they also released a noised ViT-B CLIP model trained on a filtered dataset, that is a combination of the GLIDE(filtered) dataset with a filtered version of the original CLIP dataset. Being the only version released, all the study and all the experiments carried out by me are based on GLIDE(filtered).

Code analysis - Notebooks

In this chapter we will take a look at the implementation through the three notebooks, one for each used technique. We will start from CLIP guidance to see the general functioning of text-conditioning, and also its peculiarities, then going to Classifier-free and In-paint to notice the differences between them.

3.1 CLIP guidance

```
# Create base model.
options = model_and_diffusion_defaults()
options['use_fp16'] = has_cuda
options['timestep_respacing'] = '150' # Use 150 diffusion steps for
    fast sampling

model, diffusion = create_model_and_diffusion(**options)
model.eval()

if has_cuda:
    model.convert_to_fp16()

model.to(device)
model.load_state_dict(load_checkpoint('base', device))

print('total base parameters', sum(x.numel() for x in model.
   parameters()))
```

Listing 3.1: Python example

Here we are starting by loading in the base model, which will perform the CLIP guidance sampling. First, we use the provided function `model_and_diffusion_defaults()` to create our configuration settings for the base model. Next, `create_model_and_diffusion()` uses those params to create the Text2ImUnet and diffusion models to be used. In the constructor of the Text2ImUnet class we can see the difference of this model compared to the previous of OPENAI, the `Transformer`. `model.eval()` then puts the model in evaluation mode so we can use it for inference, while `model.to(device)` ensures that operations by the model will be performed with the GPU. Finally, `model.load_state_dict()` loads the checkpoint provided by OpenAI for the base model. Note the use of the `timestep_respacing` parameter to override the number

of diffusion steps for sampling, by default they're 1000.

```
# Create upsampler model.
options_up = model_and_diffusion_defaults_upsampler()
options_up['use_fp16'] = has_cuda
options_up['timestep_respacing'] = 'fast27' # use 27 diffusion
    steps for very fast sampling

model_up, diffusion_up = create_model_and_diffusion(**options_up)
model_up.eval()

if has_cuda:
     model_up.convert_to_fp16()

model_up.to(device)
model_up.load_state_dict(load_checkpoint('upsample', device))

print('total upsampler parameters', sum(x.numel() for x in model_up
    .parameters()))
```

Similar to above, we now need to load in the upsample model. This model will take the sample generated by the base model, and upsample it to 256 x 256. We use 'fast27' for the `timestep_respacing` here to quickly generate our enlarged photo from the sample. Just like with the baseline, we use the parameters, this time taken from `model_and_diffusion_defaults_upsampler()`, to create our upsample SuperResText2ImUNet model and diffusion, set the upsample model to evaluation mode, and then load in the checkpoint for the upsample model to our `model_up` variable.

```
# Create CLIP model.
clip_model = create_clip_model(device=device)
clip_model.image_encoder.load_state_dict(load_checkpoint('clip/
    image-enc', device))
clip_model.text_encoder.load_state_dict(load_checkpoint('clip/text-
    enc', device))
```

Now we create the CLIP model using the `create_clip_model()` function that load the model parameter from the `config.yaml` file, loading then both its `image_encoder` and its `text_encoder`. In this case for the CLIP model we have a different tokenizer, a SimpleTokenizer().

```
# Sampling parameters
prompt = "an oil painting of a corgi"
batch_size = 1
guidance_scale = 3.0
# Tune this parameter to control the sharpness of 256x256 images.
# A value of 1.0 is sharper, but sometimes results in grainy
   artifacts.
upsample_temp = 0.997
```

Here we can set the sampling parameters, the input prompt, the batch size to control how much samples to generate, the guidance scale coefficient, that as said affects how strongly the classifier guidance will try to act on the image, and the upsample temperature.

```
##############################
# Sample from the base model #
##############################

# Create the text tokens to feed to the model.
tokens = model.tokenizer.encode(prompt)
tokens, mask = model.tokenizer.padded_tokens_and_mask(
    tokens, options['text_ctx']
)

# Pack the tokens together into model kwargs.
model_kwargs = dict(
    tokens=th.tensor([tokens] * batch_size, device=device),
    mask=th.tensor([mask] * batch_size, dtype=th.bool, device=
   device),
)

# Setup guidance function for CLIP model.
cond_fn = clip_model.cond_fn([prompt] * batch_size, guidance_scale)

# Sample from the base model.
model.del_cache()
samples = diffusion.p_sample_loop(
    model,
    (batch_size, 3, options["image_size"], options["image_size"]),
    device=device,
    clip_denoised=True,
    progress=True,
    model_kwargs=model_kwargs,
    cond_fn=cond_fn,
)
model.del_cache()

# Show the output
show_images(samples)
```

Now that set up is complete, this cell contains everything needed to generate the 64 x 64 sample. First, we create the text tokens to feed into the model from the prompt using the model tokenizer, a pre-trained bpe encorder, and use the `text_ctx` to generate the conditional text tokens and mask for our model. There will be 128 tokens for the Transformer context but in the mask only the first 7 will be flagged as True. We then concatenate the set of token and mask in the `model_kwargs` dic-

tionary. Then we assign to `cond_fn` the conditioning function of the CLIP model. This will create the embedding for the prompt, a single token that is a rappresentation of our input, and will find the gradient, multiplied for the gradient scale, that is gonna move our image towards the part of the space where that image resembles the prompt much more closely compared to the unconditional case. Next we call the true reverse process of the diffusion model with the `p_sample_loop`. This function loop progressive and starts generating multiple samples. It starts from a normal distribution (completely noised image) and denoises until we obtain the final image. The internal `p_sample()` will sample x_{t-1} from the model at the given timestep by using the `p_mean_variance()` function and, in the CLIP guidance case, also the `condition_mean` results given by the previous `cond_fn` function. The `p_mean_variance()` function applies the model to get $p(x_{t-1}|x_t)$, as well as a prediction of the initial x, x_0. Again, the difference between the previous models of OPENAI is how text information is used to condition the model, and we can observe it in the `forward()` pass of the Text2ImUNet.

```
def get_text_emb(self, tokens, mask):
    assert tokens is not None

    if self.cache_text_emb and self.cache is not None:
        assert (
            tokens == self.cache["tokens"]
        ).all(), f"Tokens {tokens.cpu().numpy().tolist()} do not
   match cache {self.cache['tokens'].cpu().numpy().tolist()}"
        return self.cache

    xf_in = self.token_embedding(tokens.long())
    xf_in = xf_in + self.positional_embedding[None]
    if self.xf_padding:
        assert mask is not None
        xf_in = th.where(mask[..., None], xf_in, self.
   padding_embedding[None])
    xf_out = self.transformer(xf_in.to(self.dtype))
    if self.final_ln is not None:
        xf_out = self.final_ln(xf_out)
    xf_proj = self.transformer_proj(xf_out[:, -1])
    xf_out = xf_out.permute(0, 2, 1)  # NLC -> NCL

    outputs = dict(xf_proj=xf_proj, xf_out=xf_out)

    if self.cache_text_emb:
        self.cache = dict(
            tokens=tokens,
            xf_proj=xf_proj.detach(),
            xf_out=xf_out.detach() if xf_out is not None else None,
        )

    return outputs

def forward(self, x, timesteps, tokens=None, mask=None):
    hs = []
```

```
    # embed the time_steps
    emb = self.time_embed(timestep_embedding(timesteps, self.
   model_channels))
    if self.xf_width:
        text_outputs = self.get_text_emb(tokens, mask)
        xf_proj, xf_out = text_outputs["xf_proj"], text_outputs["
   xf_out"]
        emb = emb + xf_proj.to(emb)
    else:
        xf_out = None
    h = x.type(self.dtype)
    for module in self.input_blocks:
        h = module(h, emb, xf_out)
        hs.append(h)
    h = self.middle_block(h, emb, xf_out)
    for module in self.output_blocks:
        h = th.cat([h, hs.pop()], dim=1)
        h = module(h, emb, xf_out)
    h = h.type(x.dtype)
    h = self.out(h)
    return h
```

The final token embedding `x_proj` is used in place of a class embedding and added to the timestep embedding.

```
def forward(self, qkv, encoder_kv=None):
    """
    Apply QKV attention.

    :param qkv: an [N x (H * 3 * C) x T] tensor of Qs, Ks, and Vs.
    :return: an [N x (H * C) x T] tensor after attention.
    """
    bs, width, length = qkv.shape
    assert width % (3 * self.n_heads) == 0
    ch = width // (3 * self.n_heads)

    q, k, v = qkv.reshape(bs * self.n_heads, ch * 3, length).split(
   ch, dim=1)
    if encoder_kv is not None:
        assert encoder_kv.shape[1] == self.n_heads * ch * 2
        ek, ev = encoder_kv.reshape(bs * self.n_heads, ch * 2, -1).
   split(ch, dim=1)
        # we concatenate textual keys to image keys
        k = th.cat([ek, k], dim=-1)
        # we concatenate textual values to image values
        v = th.cat([ev, v], dim=-1)
    scale = 1 / math.sqrt(math.sqrt(ch))
    weight = th.einsum(
        "bct,bcs->bts", q * scale, k * scale
    )  # More stable with f16 than dividing afterwards
    weight = th.softmax(weight.float(), dim=-1).type(weight.dtype)
    a = th.einsum("bts,bcs->bct", weight, v)
    return a.reshape(bs, -1, length)
```

In addition, we can see it in the forward() of QKVAttention.unet.py, the last layer of K token embeddings, `x_out` are input to each attention layer of the UNetModel by projecting (`encoder_kv`), then concatenating to the key and value attention inputs.

Once this cell completes running, it will output a 64 x 64 generated image specified in your prompt. This image is then used by the upsample model to create our final 256 x 256 image.

```
##############################
# Upsample the 64x64 samples #
##############################

tokens = model_up.tokenizer.encode(prompt)
tokens, mask = model_up.tokenizer.padded_tokens_and_mask(
    tokens, options_up['text_ctx']
)

# Create the model conditioning dict.
model_kwargs = dict(
    # Low-res image to upsample.
    low_res=((samples+1)*127.5).round()/127.5 - 1,

    # Text tokens
    tokens=th.tensor(
        [tokens] * batch_size, device=device
    ),
    mask=th.tensor(
        [mask] * batch_size,
        dtype=th.bool,
        device=device,
    ),
)

# Sample from the base model.
model_up.del_cache()
up_shape = (batch_size, 3, options_up["image_size"], options_up["
   image_size"])
up_samples = diffusion_up.ddim_sample_loop(
    model_up,
    up_shape,
    noise=th.randn(up_shape, device=device) * upsample_temp,
    device=device,
    clip_denoised=True,
    progress=True,
    model_kwargs=model_kwargs,
    cond_fn=None,
)[:batch_size]
model_up.del_cache()

# Show the output
show_images(up_samples)
```

Finally, now that we have generated our initial image, it's time to use our diffusion upsampler to get the image in 256 x 256. First, we reassign our tokens and mask using `model_up`. We then pack the `samples` variable representing our generated image, the tokens and mask together as our `model_kwargs`. We then use our `diffusion_up` model to upsample the image (now stored as `low_res` in the `kwargs`) for the "fast" 27 steps.

3.2 Classifier-free guidance

Here we will analyze the notebook dedicated to Classifier-free guidance tecnique, going directly to show the differences in the cell dedicated to sampling.

```
##############################
# Sample from the base model #
##############################

# Create the text tokens to feed to the model.
tokens = model.tokenizer.encode(prompt)
tokens, mask = model.tokenizer.padded_tokens_and_mask(
    tokens, options['text_ctx']
)

# Create the classifier-free guidance tokens (empty)
full_batch_size = batch_size * 2
uncond_tokens, uncond_mask = model.tokenizer.padded_tokens_and_mask
    (
    [], options['text_ctx']
)

# Pack the tokens together into model kwargs.
model_kwargs = dict(
    tokens=th.tensor(
        [tokens] * batch_size + [uncond_tokens] * batch_size,
    device=device
    ),
    mask=th.tensor(
        [mask] * batch_size + [uncond_mask] * batch_size,
        dtype=th.bool,
        device=device,
    ),
)

# Create a classifier-free guidance sampling function
def model_fn(x_t, ts, **kwargs):
    half = x_t[: len(x_t) // 2]
    combined = th.cat([half, half], dim=0)
    model_out = model(combined, ts, **kwargs)
    eps, rest = model_out[:, :3], model_out[:, 3:]
    cond_eps, uncond_eps = th.split(eps, len(eps) // 2, dim=0)
    half_eps = uncond_eps + guidance_scale * (cond_eps - uncond_eps
    )
    eps = th.cat([half_eps, half_eps], dim=0)
    return th.cat([eps, rest], dim=1)

# Sample from the base model.
model.del_cache()
samples = diffusion.p_sample_loop(
    model_fn,
    (full_batch_size, 3, options["image_size"], options["image_size
    "]),
    device=device,
    clip_denoised=True,
    progress=True,
    model_kwargs=model_kwargs,
    cond_fn=None,
)[:batch_size]
model.del_cache()

# Show the output
show_images(samples)
```

As always, we start with the encoding of the input prompt, but this time we also

add the unconditional tokens and the relative mask to `model_kwargs` by encoding an empty prompt `"[]"`. It can also be seen how `batch_size` is doubled in order to process the output of the latter as well. The reverse process follows the same steps as the clip guidance but this time we define `model_fn`, the function that will be passed to the `p_sample_loop`. It will be called during the iterations of the reverse process by `p_mean_variance()` to apply the "trick" that force the diffusion model to use its own learned understanding of the inputs to influence the image generation procedure. This function will take half sample for the classical calculation, and then calculate the perturbation as the difference between the conditional and the unconditional epsilon. After that it will follow the upsample process identical to the one shown previously, which will generate the image in 256x256 format.

3.3 Inpaint

```
# Create base model.
options = model_and_diffusion_defaults()
options['inpaint'] = True
options['use_fp16'] = has_cuda
options['timestep_respacing'] = '100' # use 100 diffusion steps for
    fast sampling

model, diffusion = create_model_and_diffusion(**options)
model.eval()
if has_cuda:
    model.convert_to_fp16()
model.to(device)

model.load_state_dict(load_checkpoint('base-inpaint', device))
print('total base parameters', sum(x.numel() for x in model.
   parameters()))
```

Here we are loading in the inpaint base model, which will perform the inpainting. In order to work with this tecnique a different model is generated, which as mentioned before has been fine-tuned for this purpose. This is InpaintText2ImUNet, which is returned by setting the options['inpaint'] flag, inside which a different checkpoint than the base will be loaded. Same thing happens for the SuperResInpaintText2ImUnet which will take care of the upscaling.

```
# Sampling parameters
prompt = "a corgi in a field"
batch_size = 1
guidance_scale = 5.0

# Tune this parameter to control the sharpness of 256x256 images.
# A value of 1.0 is sharper, but sometimes results in grainy
    artifacts.
upsample_temp = 0.997

# Source image we are inpainting
source_image_256 = read_image('grass.png', size=256)
source_image_64 = read_image('grass.png', size=64)

# The mask should always be a boolean 64x64 mask, and then we
# can upsample it for the second stage.
source_mask_64 = th.ones_like(source_image_64)[:, :1]
source_mask_64[:, :, 20:] = 0
source_mask_256 = F.interpolate(source_mask_64, (256, 256), mode='
    nearest')

# Visualize the image we are inpainting
show_images(source_image_256 * source_mask_256)
```

We set the usual sampling parameters but this time the prompt determines what the inpainter will try to add or remove from the image. Then we load the source image that will be painted over and finally we generate our source masks. These are used to identify the regions in the photo to be inpainted.

```
##############################
# Sample from the base model #
##############################
# Create the text tokens to feed to the model.
tokens = model.tokenizer.encode(prompt)
tokens, mask = model.tokenizer.padded_tokens_and_mask(
    tokens, options['text_ctx']
)
# Create the classifier-free guidance tokens (empty)
full_batch_size = batch_size * 2
uncond_tokens, uncond_mask = model.tokenizer.padded_tokens_and_mask
    (
    [], options['text_ctx']
)
# Pack the tokens together into model kwargs.
model_kwargs = dict(
    tokens=th.tensor(
        [tokens] * batch_size + [uncond_tokens] * batch_size,
    device=device
    ),
    mask=th.tensor(
        [mask] * batch_size + [uncond_mask] * batch_size,
        dtype=th.bool,
        device=device,
    ),

    # Masked inpainting image
    inpaint_image=(source_image_64 * source_mask_64).repeat(
    full_batch_size, 1, 1, 1).to(device),
    inpaint_mask=source_mask_64.repeat(full_batch_size, 1, 1, 1).to
    (device),
)

# Create an classifier-free guidance sampling function
def model_fn(x_t, ts, **kwargs):
    half = x_t[: len(x_t) // 2]
    combined = th.cat([half, half], dim=0)
    model_out = model(combined, ts, **kwargs)
    eps, rest = model_out[:, :3], model_out[:, 3:]
    cond_eps, uncond_eps = th.split(eps, len(eps) // 2, dim=0)
    half_eps = uncond_eps + guidance_scale * (cond_eps - uncond_eps
    )
    eps = th.cat([half_eps, half_eps], dim=0)
    return th.cat([eps, rest], dim=1)
def denoised_fn(x_start):
    # Force the model to have the exact right x_start predictions
    # for the part of the image which is known.
    return (
        x_start * (1 - model_kwargs['inpaint_mask'])
        + model_kwargs['inpaint_image'] * model_kwargs['
    inpaint_mask']
    )

# Sample from the base model.
model.del_cache()
samples = diffusion.p_sample_loop(
    model_fn,
    (full_batch_size, 3, options["image_size"], options["image_size
    "]),
    device=device,
    clip_denoised=True,
    progress=True,
    model_kwargs=model_kwargs,
    cond_fn=None,
    denoised_fn=denoised_fn,
)[:batch_size]
model.del_cache()

# Show the output
show_images(samples)
```

This cell contains the sampling operations that we have seen in the section dedicated to the classifier-free guidance, but with some changes. We concatenate the two sets of tokens and masks in the `model_kwargs` dictionary, along with our inpaint image and its corresponding image mask. `model_fn()` is used this time to perform inpainting by sampling from the diffusion model to generate the image, but replacing the known region of the image with a sample from a noised version of the context after each sampling step. In practice, the generator learns to add the information suggested by the text prompt tokens to the masked portion of image. `denoised_fn()` is used instead to force the model to have the exact prediction for the known part of the image. Once this cell completes running, it will output a 64 x 64 generated image that contains the inpainting changes specified in your prompt. This image is then used by the upsample model to create our final 256 x 256 image. It works as the one seen in the classifier-free guidance but we pack in the `model_kwargs` also the inpainting image, and inpainting mask.

Results

As already mentioned, the GLIDE version I worked on for this project is a version with far fewer parameters than the original one, and trained on a filtered dataset. The removal of images, for example of people, not only affects the generation of samples belonging to that category, but removes many contexts related to them. Then I'll report the results obtained from the original model, and then show you what differences it gets in its filtered version.

Using human and automated evaluations, they found that classifier-free guidance yields higherquality images and also that samples are both photorealistic and reflect a wide breadth of world knowledge than those produced using CLIP guidance.

In the image above you can see the generation of various samples with the same input prompt, generated by the two techniques we have seen, and by DALL-E.

Also in these examples you can see the better details obtained by the classifier-free, compared both to CLIP guidance and to a previous state-of-the-art text-conditional image generation model like DALL-E, even though it is a much smaller model.

	DALL-E Temp.	Photo-realism	Caption Similarity
No reranking	1.0	91%	83%
	0.85	84%	80%
DALL-E reranked	1.0	89%	71%
	0.85	87%	69%
DALL-E reranked + GLIDE blurred	1.0	72%	63%
	0.85	66%	61%

This table shows the win probabilities of GLIDE in various tests. Two temperatures were used for DALL-E and various reranked techniques that should have favored it, allowing it to use a much larger amount of test-time compute, but on average when evaluated by human judges, GLIDE's samples are preferred to those from DALL-E 87% of the time when evaluated for photorealism, and 69% of the time when evaluated for caption similarity.

"a corgi wearing a bow tie and a birthday hat"

"a fire in the background"

On the inpainting task, they found that GLIDE can realistically modify existing images using text prompts, inserting new objects, shadows and reflections when necessary, matching also styles when editing objects into paintings.

"a cozy living room" "a painting of a corgi on the wall above a couch" "a round coffee table in front of a couch" "a vase of flowers on a coffee table" "a couch in the corner of a room"

In this figure they showed how they used GLIDE iteratively to produce a complex scene using a zero-shot generation followed by a series of inpaintining edits.

GLIDE often generates realistic shadows and reflections, as well as high-quality textures. It is also capable of producing illustrations in various styles as we will see in the examples proposed below, such as the style of a particular artist or painting, or in general styles like pixel art. OpenAI have generated a third model, to underline the importance of training on sampling quality, having the same reduced hyperparameters of the filtered one but trained on the entire dataset as the complete one. Studying GLIDE(filtered) i realized that in general the filtered version was not so good at responding to prompts as complex as in the examples in the paper and did not always match perfectly with the caption.

All of the models can often produce realistic images, but the two models trained on our full dataset are much better at combining unusual concepts, as can be also seen from the human evaluation below which compares the small model with the larger one

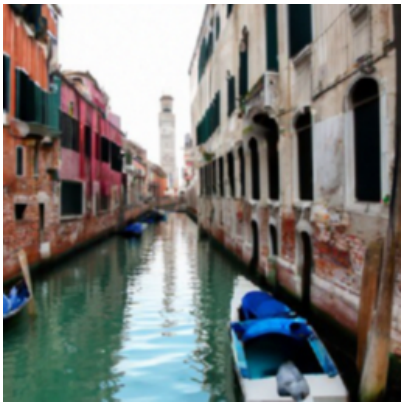
Size	Guide. Scale	Photorealism	Caption
300M	1.0	-131.8	-136.4
300M	4.0	28.2	70.9
3.5B	1.0	-23.9	-27.1
3.5B	3.0	**133.0**	**140.5**

For samples shown in this report, I followed the same paper parameters. So i set the base model using 150 diffusion steps, the upsampler to the special strided sampling schedule with only 27 diffusion steps, and the guidance scale to 3.


"a futuristic city in synthwave style" (GLIDE vs filtered + CLIP)


"a painting of a fox in the style of starry night" (GLIDE vs filtered + CLIP)


"a fall landscape with a small cottage next to a lake" (GLIDE vs filtered + CLIP)




"a boat in the canals of venice" (GLIDE vs filtered)

Conclusion

The results of GLIDE are too convincing, just look at this. The authors conducted human evaluation experiments where it is clear that the majority preferred GLIDE's generations over the blurrier and messier outputs of DALL-E but diffusion models are a much slower having to go through all the diffusion steps sequentially (150 in GLIDE's case), take much longer than GANs, for example, which produce images in a single forward pass and are thus more favorable for use in real-time applications. Finally, the cost of running the model is relatively high just for 256 x 256 images. It's reasonable to assume that this would only exponentially increase if we were to upsample further or use the expensive CLIP to guide the process. But OpenAI work to improve image-generation continued because now, at the time I am writing, a separate DALL-E 2 system is announced. It generates images with 4x greater resolution (compared to original DALL·E and GLIDE), now up to 1024×1024 pixels. The model behind the DALL·E 2 is a clever combination of CLIP and GLIDE, and the model itself (the full text-conditional image generation stack) is called unCLIP internally in the paper since it generates images by inverting the CLIP image encoder, and the diffusion decoder is a modified GLIDE with 3.5B parameters. There is still much room for improvement, but DALL-E 2 is a huge improvement over the first version of the system, DALL-E 1, and made in just one year thanks to GLIDE, so maybe we may call GLIDE as DALL·E 1.5.