

The Annotated Diffusion Model

In this blog post, we'll take a deeper look into **Denoising Diffusion Probabilistic Models** (also known as DDPMs, diffusion models, score-based generative models or simply [autoencoders](#)) as researchers have been able to achieve remarkable results with them for (un)conditional image/audio/video generation. Popular examples (at the time of writing) include [GLIDE](#) and [DALLE 2](#) by OpenAI, [Latent Diffusion](#) by the University of Heidelberg and [ImageGen](#) by Google Brain.

We'll go over the original DDPM paper by ([Ho et al., 2020](#)), implementing it step-by-step in PyTorch, based on Phil Wang's [implementation](#) - which itself is based on the [original TensorFlow implementation](#). Note that the idea of diffusion for generative modeling was actually already introduced in ([Sohl-Dickstein et al., 2015](#)). However, it took until ([Song et al., 2019](#)) (at Stanford University), and then ([Ho et al., 2020](#)) (at Google Brain) who independently improved the approach.

Note that there are [several perspectives](#) on diffusion models. Here, we employ the discrete-time (latent variable model) perspective, but be sure to check out the other perspectives as well.

Alright, let's dive in!

What is a diffusion model?

A (denoising) diffusion model isn't that complex if you compare it to other generative models such as Normalizing Flows, GANs or VAEs: they all convert noise from some simple distribution to a data sample. This is also the case here where **a neural network learns to gradually denoise data** starting from pure noise.

In a bit more detail for images, the set-up consists of 2 processes:

- a fixed (or predefined) forward diffusion process q of our choosing, that gradually adds Gaussian noise to an image, until you end up with pure noise
- a learned reverse denoising diffusion process p_θ , where a neural network is trained to gradually denoise an image starting from pure noise, until you end up with an actual image.

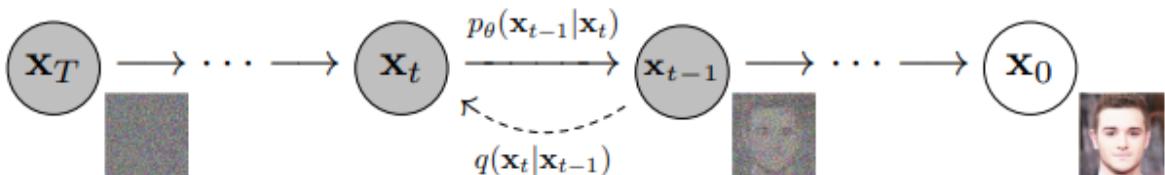


Figure 2: The directed graphical model considered in this work.

Both the forward and reverse process indexed by t happen for some number of finite time steps T (the DDPM authors use $T = 1000$). You start with $t = 0$ where you sample a real image \mathbf{x}_0 from your data distribution (let's say an image of a cat from ImageNet), and the forward process samples some noise from a Gaussian distribution at each time step t , which is added to the image of the previous time step. Given a sufficiently large T and a well behaved schedule for adding noise at each time step, you end up with what is called an [isotropic Gaussian distribution](#) at $t = T$ via a gradual process.

In more mathematical form

Let's write this down more formally, as ultimately we need a tractable loss function which our neural network needs to optimize.

Let $q(\mathbf{x}_0)$ be the real data distribution, say of "real images". We can sample from this distribution to get an image, $\mathbf{x}_0 \sim q(\mathbf{x}_0)$. We define the forward diffusion process $q(\mathbf{x}_t | \mathbf{x}_{t-1})$ which adds Gaussian noise at each time step t , according to a known variance schedule $0 < \beta_1 < \beta_2 < \dots < \beta_T < 1$ as

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

Recall that a normal distribution (also called Gaussian distribution) is defined by 2 parameters: a mean μ and a variance $\sigma^2 \geq 0$. Basically, each new (slightly noiser) image at time step t is drawn from a **conditional Gaussian distribution** with $\mu_t = \sqrt{1 - \beta_t} \mathbf{x}_{t-1}$ and $\sigma_t^2 = \beta_t$, which we can do by sampling $\epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ and then setting $\mathbf{x}_t = \sqrt{1 - \beta_t} \mathbf{x}_{t-1} + \sqrt{\beta_t} \epsilon$.

Note that the β_t aren't constant at each time step t (hence the subscript) --- in fact one defines a so-called "**variance schedule**", which can be linear, quadratic, cosine, etc. as we will see further (a bit like a learning rate schedule).

So starting from \mathbf{x}_0 , we end up with $\mathbf{x}_1, \dots, \mathbf{x}_t, \dots, \mathbf{x}_T$, where \mathbf{x}_T is pure Gaussian noise if we set the schedule appropriately.

Now, if we knew the conditional distribution $p(\mathbf{x}_{t-1} | \mathbf{x}_t)$, then we could run the process in reverse: by sampling some random Gaussian noise \mathbf{x}_T , and then gradually "denoise" it so that we end up with a sample from the real distribution \mathbf{x}_0 .

However, we don't know $p(\mathbf{x}_{t-1} | \mathbf{x}_t)$. It's intractable since it requires knowing the distribution of all possible images in order to calculate this conditional probability. Hence, we're going to leverage a neural network to **approximate (learn) this conditional probability distribution**, let's call it $p_\theta(\mathbf{x}_{t-1} | \mathbf{x}_t)$, with θ being the parameters of the neural network, updated by gradient descent.

Ok, so we need a neural network to represent a (conditional) probability distribution of the backward process. If we assume this reverse process is Gaussian as well, then recall that any Gaussian distribution is defined by 2 parameters:

- a mean parametrized by μ_θ ;
- a variance parametrized by Σ_θ ;

so we can parametrize the process as

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

where the mean and variance are also conditioned on the noise level t .

Hence, our neural network needs to learn/represent the mean and variance. However, the DDPM authors decided to **keep the variance fixed, and let the neural network only learn (represent) the mean μ_θ of this conditional probability distribution**. From the paper:

First, we set $\Sigma_\theta(\mathbf{x}_t, t) = \sigma_t^2 \mathbf{I}$ to untrained time dependent constants. Experimentally, both $\sigma_t^2 = \beta_t$ and $\sigma_t^2 = \tilde{\beta}_t$ (see paper) had similar results.

This was then later improved in the [Improved diffusion models](#) paper, where a neural network also learns the variance of this backwards process, besides the mean.

So we continue, assuming that our neural network only needs to learn/represent the mean of this conditional probability distribution.

Defining an objective function (by reparametrizing the mean)

To derive an objective function to learn the mean of the backward process, the authors observe that the combination of q and p_θ can be seen as a variational auto-encoder (VAE) ([Kingma et al., 2013](#)). Hence, the **variational lower bound** (also called ELBO) can be used to minimize the negative log-likelihood with respect to ground truth data sample \mathbf{x}_0 (we refer to the VAE paper for details regarding ELBO). It turns out that the ELBO for this process is a sum of losses at each time step t , $L = L_0 + L_1 + \dots + L_T$. By construction of the forward q process and backward process, each term (except for L_0) of the loss is actually the **KL divergence between 2 Gaussian distributions** which can be written explicitly as an L2-loss with respect to the means!

A direct consequence of the constructed forward process q , as shown by Sohl-Dickstein et al., is that we can sample \mathbf{x}_t at any arbitrary noise level conditioned on \mathbf{x}_0 (since sums of Gaussians is also Gaussian). This is very convenient: we don't need to apply q repeatedly in order to sample \mathbf{x}_t . We have that

$$q(\mathbf{x}_t | \mathbf{x}_0) = \mathcal{N}(\mathbf{x}_t; \sqrt{\bar{\alpha}_t} \mathbf{x}_0, (1 - \bar{\alpha}_t) \mathbf{I})$$

with $\alpha_t := 1 - \beta_t$ and $\bar{\alpha}t := \prod_{s=1}^t \alpha_s$. Let's refer to this equation as the "nice property". This means we can sample Gaussian noise and scale it appropriately and add it to \mathbf{x}_0 to get \mathbf{x}_t directly. Note that the $\bar{\alpha}_t$ are functions of the known β_t variance schedule and thus are also known and can be precomputed. This then allows us, during training, to **optimize random terms of the loss function L** (or in other words, to randomly sample t during training and optimize L_t).

Another beauty of this property, as shown in Ho et al. is that one can (after some math, for which we refer the reader to [this excellent blog post](#)) instead **reparametrize the mean to make the neural network learn (predict) the added noise (via a network $\epsilon_\theta(\mathbf{x}_t, t)$ for noise level t** in the KL terms which constitute the losses. This means that our neural network becomes a noise predictor, rather than a (direct) mean predictor. The mean can be computed as follows:

$$\mu_\theta(\mathbf{x}_t, t) = \frac{1}{\sqrt{\alpha_t}} \left(\mathbf{x}_t - \frac{\beta_t}{\sqrt{1 - \bar{\alpha}_t}} \epsilon_\theta(\mathbf{x}_t, t) \right)$$

The final objective function L_t then looks as follows (for a random time step t given $\epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$):

```
$$$ \mathbf{\epsilon} - \mathbf{\epsilon}_\theta(\mathbf{x}_t) |^2 = \| \mathbf{\epsilon} - \mathbf{\epsilon}_\theta(\sqrt{\bar{\alpha}} \mathbf{x}_0 + \sqrt{1-\bar{\alpha}} \mathbf{x}_t) \| \mathbf{\epsilon}, t) $$$
```

Here, \mathbf{x}_0 is the initial (real, uncorrupted) image, and we see the direct noise level t sample given by the fixed forward process. ϵ is the pure noise sampled at time step t , and $\epsilon_\theta(\mathbf{x}_t, t)$ is our neural network. The neural network is optimized using a simple mean squared error (MSE) between the true and the predicted Gaussian noise.

The training algorithm now looks as follows:

Algorithm 1 Training

- 1: **repeat**
 - 2: $\mathbf{x}_0 \sim q(\mathbf{x}_0)$
 - 3: $t \sim \text{Uniform}(\{1, \dots, T\})$
 - 4: $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
 - 5: Take gradient descent step on

$$\nabla_{\theta} \left\| \boldsymbol{\epsilon} - \boldsymbol{\epsilon}_{\theta} (\sqrt{\bar{\alpha}_t} \mathbf{x}_0 + \sqrt{1 - \bar{\alpha}_t} \boldsymbol{\epsilon}, t) \right\|^2$$
 - 6: **until** converged

In other words:

- we take a random sample \mathbf{x}_0 from the real unknown and possibly complex data distribution $q(\mathbf{x}_0)$
 - we sample a noise level t uniformly between 1 and T (i.e., a random time step)
 - we sample some noise from a Gaussian distribution and corrupt the input by this noise at level t using the nice property defined above
 - the neural network is trained to predict this noise based on the corrupted image \mathbf{x}_t , i.e. noise applied on \mathbf{x}_0 based on known schedule β_t

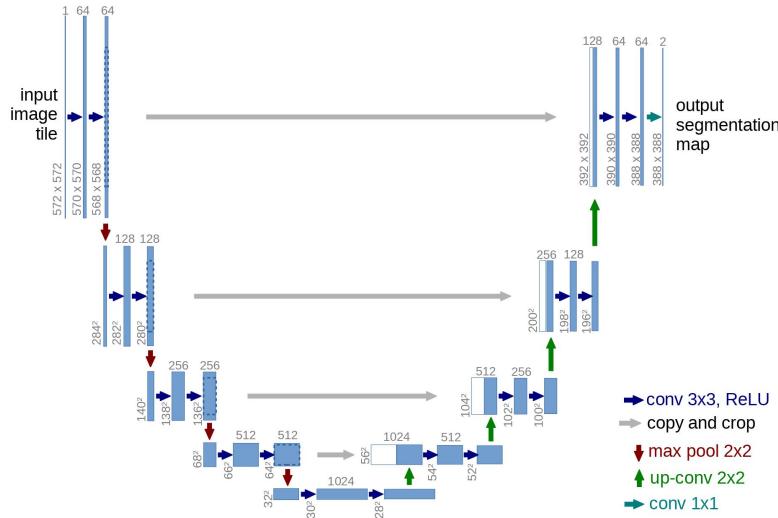
In reality, all of this is done on batches of data as one uses stochastic gradient descent to optimize neural networks.

The neural network

The neural network needs to take in a noised image at a particular time step and return the predicted noise. Note that the predicted noise is a tensor that has the same size/resolution as the input image. So technically, the network takes in and outputs tensors of the same shape. What type of neural network can we use for this?

What is typically used here is very similar to that of an [Autoencoder](#), which you may remember from typical "intro to deep learning" tutorials. Autoencoders have a so-called "bottleneck" layer in between the encoder and decoder. The encoder first encodes an image into a smaller hidden representation called the "bottleneck", and the decoder then decodes that hidden representation back into an actual image. This forces the network to only keep the most important information in the bottleneck layer.

In terms of architecture, the DDPM authors went for a [U-Net](#), introduced by ([Ronneberger et al., 2015](#)) (which, at the time, achieved state-of-the-art results for medical image segmentation). This network, like any autoencoder, consists of a bottleneck in the middle that makes sure the network learns only the most important information. Importantly, it introduced residual connections between the encoder and decoder, greatly improving gradient flow (inspired by ResNet in [He et al., 2015](#)).



As can be seen, a U-Net model first downsamples the input (i.e. makes the input smaller in terms of spatial resolution), after which upsampling is performed.

Below, we implement this network, step-by-step.

Network helpers

First, we define some helper functions and classes which will be used when implementing the neural network. Importantly, we define a `Residual` module, which simply adds the input to the output of a particular function (in other words, adds a residual connection to a particular function).

We also define aliases for the up- and downsampling operations.

Position embeddings

As the parameters of the neural network are shared across time (noise level), the authors employ sinusoidal position embeddings to encode t , inspired by the Transformer ([Vaswani et al., 2017](#)). This makes the neural network "know" at which particular time step (noise level) it is operating, for every image in a batch.

The `SinusoidalPositionEmbeddings` module takes a tensor of shape `(batch_size, 1)` as input (i.e. the noise levels of several noisy images in a batch), and turns this into a tensor of shape `(batch_size, dim)`, with `dim` being the dimensionality of the position embeddings. This is then added to each residual block, as we will see further.

ResNet/ConvNeXT block

Next, we define the core building block of the U-Net model. The DDPM authors employed a Wide ResNet block ([Zagoruyko et al., 2016](#)), but Phil Wang decided to also add support for a ConvNeXT block ([Liu et al., 2022](#)), as the latter has achieved great success in the image domain. One can choose one or another in the final U-Net architecture.

Attention module

Next, we define the attention module, which the DDPM authors added in between the convolutional blocks. Attention is the building block of the famous Transformer architecture ([Vaswani et al., 2017](#)), which has shown great success in various domains of AI, from NLP and vision to [protein folding](#). Phil Wang employs 2 variants of attention: one is regular multi-head self-attention (as used in the Transformer), the other one is a [linear attention variant](#) ([Shen et al., 2018](#)), whose time- and memory requirements scale linear in the sequence length, as opposed to quadratic for regular attention.

For an extensive explanation of the attention mechanism, we refer the reader to Jay Allamar's [wonderful blog post](#).

Group normalization

The DDPM authors interleave the convolutional/attention layers of the U-Net with group normalization ([Wu et al., 2018](#)). Below, we define a `PreNorm` class, which will be used to apply groupnorm before the attention layer, as we'll see further. Note that there's been a [debate](#) about whether to apply normalization before or after attention in Transformers.

Conditional U-Net

Now that we've defined all building blocks (position embeddings, ResNet/ConvNeXT blocks, attention and group normalization), it's time to define the entire neural network. Recall that the job of the network $\epsilon_\theta(\mathbf{x}_t, t)$ is to take in a batch of noisy images + noise levels, and output the noise added to the input. More formally:

- the network takes a batch of noisy images of shape `(batch_size, num_channels, height, width)` and a batch of noise levels of shape `(batch_size, 1)` as input, and returns a tensor of shape `(batch_size, num_channels, height, width)`

The network is built up as follows:

- first, a convolutional layer is applied on the batch of noisy images, and position embeddings are computed for the noise levels
- next, a sequence of downsampling stages are applied. Each downsampling stage consists of 2 ResNet/ConvNeXT blocks + groupnorm + attention + residual connection + a downsample operation
- at the middle of the network, again ResNet or ConvNeXT blocks are applied, interleaved with attention
- next, a sequence of upsampling stages are applied. Each upsampling stage consists of 2 ResNet/ConvNeXT blocks + groupnorm + attention + residual connection + an upsample operation
- finally, a ResNet/ConvNeXT block followed by a convolutional layer is applied.

Ultimately, neural networks stack up layers as if they were lego blocks (but it's important to [understand how they work](#)).

Defining the forward diffusion process

The forward diffusion process gradually adds noise to an image from the real distribution, in a number of time steps T . This happens according to a **variance schedule**. The original DDPM authors employed a linear schedule:

We set the forward process variances to constants increasing linearly from $\beta_1 = 10^{-4}$ to $\beta_T = 0.02$.

However, it was shown in ([Nichol et al., 2021](#)) that better results can be achieved when employing a cosine schedule.

Below, we define various schedules for the T timesteps, as well as corresponding variables which we'll need, such as cumulative variances.

To start with, let's use the linear schedule for $T = 200$ time steps and define the various variables from the β_t which we will need, such as the cumulative product of the variances $\bar{\alpha}_t$. Each of the variables below are just 1-dimensional tensors, storing values from t to T . Importantly, we also define an `extract` function, which will allow us to extract the appropriate t index for a batch of indices.

We'll illustrate with a cats image how noise is added at each time step of the diffusion process.

Noise is added to PyTorch tensors, rather than Pillow Images. We'll first define image transformations that allow us to go from a PIL image to a PyTorch tensor (on which we can add the noise), and vice versa.

These transformations are fairly simple: we first normalize images by dividing by 255 (such that they are in the $[0, 1]$ range), and then make sure they are in the $[-1, 1]$ range. From the DPPM paper:

We assume that image data consists of integers in $\{0, 1, \dots, 255\}$ scaled linearly to $[-1, 1]$. This ensures that the neural network reverse process operates on consistently scaled inputs starting from the standard normal prior $p(\mathbf{x}_T)$.

Output:

```
torch.Size([1, 3, 128, 128])
```

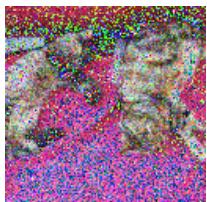
We also define the reverse transform, which takes in a PyTorch tensor containing values in $[-1, 1]$ and turn them back into a PIL image:

Let's verify this:



We can now define the forward diffusion process as in the paper:

Let's test it on a particular time step:



Let's visualize this for various time steps:



This means that we can now define the loss function given the model as follows:

The `denoise_model` will be our U-Net defined above. We'll employ the Huber loss between the true and the predicted noise.

Define a PyTorch Dataset + DataLoader

Here we define a regular [PyTorch Dataset](#). The dataset simply consists of images from a real dataset, like Fashion-MNIST, CIFAR-10 or ImageNet, scaled linearly to $[-1, 1]$.

Each image is resized to the same size. Interesting to note is that images are also randomly horizontally flipped. From the paper:

We used random horizontal flips during training for CIFAR10; we tried training both with and without flips, and found flips to improve sample quality slightly.

Here we use the 😊 [Datasets library](#) to easily load the Fashion MNIST dataset from the [hub](#). This dataset consists of images which already have the same resolution, namely 28x28.

Next, we define a function which we'll apply on-the-fly on the entire dataset. We use the `with_transform` functionality for that. The function just applies some basic image preprocessing: random horizontal flips, rescaling and finally make them have values in the $[-1, 1]$ range.

Sampling

As we'll sample from the model during training (in order to track progress), we define the code for that below. Sampling is summarized in the paper as Algorithm 2:

Algorithm 2 Sampling

```
1:  $\mathbf{x}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ 
2: for  $t = T, \dots, 1$  do
3:    $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$  if  $t > 1$ , else  $\mathbf{z} = \mathbf{0}$ 
4:    $\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}} \left( \mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}} \mathbf{z}_\theta(\mathbf{x}_t, t) \right) + \sigma_t \mathbf{z}$ 
5: end for
6: return  $\mathbf{x}_0$ 
```

Generating new images from a diffusion model happens by reversing the diffusion process: we start from T , where we sample pure noise from a Gaussian distribution, and then use our neural network to gradually denoise it (using the conditional probability it has learned), until we end up at time step $t = 0$. As shown above, we can derive a slightly less denoised image \mathbf{x}_{t-1} by plugging in the reparametrization of the mean, using our noise predictor. Remember that the variance is known ahead of time.

Ideally, we end up with an image that looks like it came from the real data distribution.

The code below implements this.

Note that the code above is a simplified version of the original implementation. We found our simplification (which is in line with Algorithm 2 in the paper) to work just as well as the [original, more complex implementation](#).

Train the model

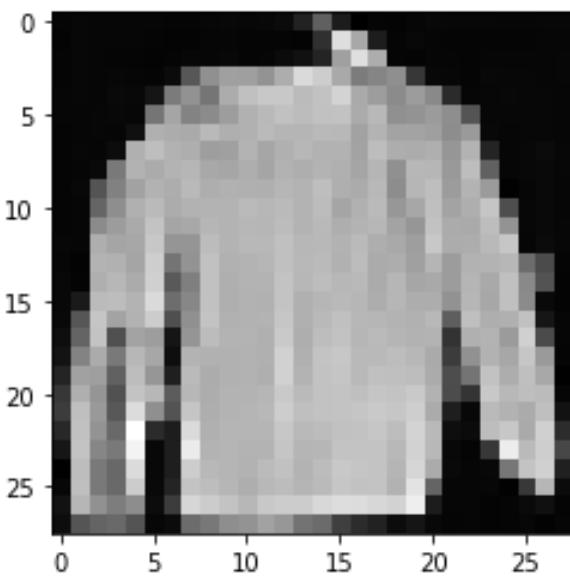
Next, we train the model in regular PyTorch fashion. We also define some logic to periodically save generated images, using the `sample` method defined above.

Below, we define the model, and move it to the GPU. We also define a standard optimizer (Adam).

Let's start training!

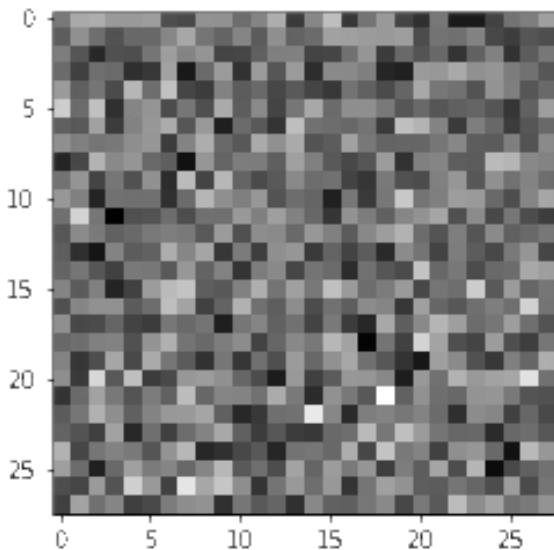
Sampling (inference)

To sample from the model, we can just use our `sample` function defined above:



Seems like the model is capable of generating a nice T-shirt! Keep in mind that the dataset we trained on is pretty low-resolution (28x28).

We can also create a gif of the denoising process:



Follow-up reads

Note that the DDPM paper showed that diffusion models are a promising direction for (un)conditional image generation. This has since then (immensely) been improved, most notably for text-conditional image generation. Below, we list some important (but far from exhaustive) follow-up works:

- Improved Denoising Diffusion Probabilistic Models ([Nichol et al., 2021](#)): finds that learning the variance of the conditional distribution (besides the mean) helps in improving performance
- Cascaded Diffusion Models for High Fidelity Image Generation ([Ho et al., 2021](#)): introduce cascaded diffusion, which comprises a pipeline of multiple diffusion models that generate images of increasing resolution for high-fidelity image synthesis
- Diffusion Models Beat GANs on Image Synthesis ([Dhariwal et al., 2021](#)): show that diffusion models can achieve image sample quality superior to the current state-of-the-art generative models by improving the U-Net architecture, as well as introducing classifier guidance
- Classifier-Free Diffusion Guidance ([Ho et al., 2021](#)): shows that you don't need a classifier for guiding a diffusion model by jointly training a conditional and an unconditional diffusion model with a single neural network
- Hierarchical Text-Conditional Image Generation with CLIP Latents (DALL-E 2) ([Ramesh et al., 2022](#)): use a prior to turn a text caption into a CLIP image embedding, after which a diffusion model decodes it into an image
- Photorealistic Text-to-Image Diffusion Models with Deep Language Understanding (ImageGen) ([Saharia et al., 2022](#)): shows that combining a large pre-trained language model (e.g. T5) with cascaded diffusion works well for text-to-image synthesis

Note that this list only includes important works until the time of writing, which is June 7th, 2022.

For now, it seems that the main (perhaps only) disadvantage of diffusion models is that they require multiple forward passes to generate an image (which is not the case for generative models like GANs). However, there's [research going on](#) that enables high-fidelity generation in as few as 10 denoising steps.