Neural network programming: diffusion models

Erik Spence

SciNet HPC Consortium

13 May 2025

Today's code and slides

You can get the slides and code for today's class at the SciNet Education web page.

`https://scinet.courses/1375`

Click on the link for the class, and look under "Lectures", click on "Diffusion models".

Today's class

This class will cover the following topics:

- Diffusion model approach.
- Develop our cost function.
- Example.

Please ask questions if something isn't clear.

Problems with GANs

GANs were state-of-the-art for many years. They are powerful, and have impressive results. But GANs have serious problems associated with them.

- Because there are two networks that are trained independently, there is no single cost-function global minimum that we can search for.
- This results in an unstable situation, as we are searching for a Nash equilibrium.
- Consequently, these models can be very hard to train. Optimization of hyperparameters, and some luck, is necessary for success.

We'd like an approach that is more robust, training-wise.

Diffusion models (2015/2020)

Diffusion models are a type of generative model.

- Originally introduced in 2015, but forgotten for a few years during the height of the GANs excitement.
- These were originally used to generate images, though in principle you could use it to generate any type of continuous data.
- They sample from $\boldsymbol{P}(\mathbf{x})$ by gradually reversing a 'noising' process.
- Suppose we start with an image, $\mathbf{x_0}$, and add noise to it through a series of $\boldsymbol{T}$ steps. Eventually all you have left is pure noise. This is called the 'forward diffusion'.
- The goal of the model is to reverse this process.
- Thus, we start with just pure noise, and the model eventually converts it into an image drawn from $\boldsymbol{P}(\mathbf{x})$.

The starting point is the same as that of GANs: just noise.

SciNet

Forward diffusion

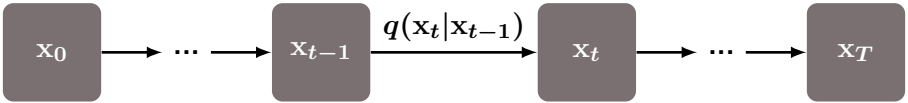
To train these models we need a specially-modified version of whatever data set we are using.

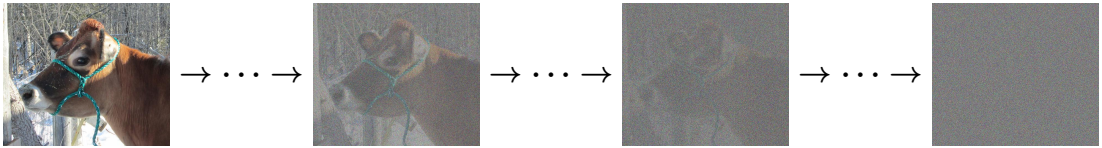
- We start with a data point sampled from our data set, $\mathbf{x_0}$.
- We need to define the process by which we create $\mathbf{x_t}$ from $\mathbf{x_{t-1}}$. This is a Markov process.
- Specifically, at each step we add Gaussian noise with variance $\boldsymbol{\beta_t}$ to $\mathbf{x_{t-1}}$. The variance can be constant, but usually varies with $\boldsymbol{t}$.
- This produces our new data, $\mathbf{x_t}$, with a probability distribution of $\boldsymbol{q}(\mathbf{x_t}|\mathbf{x_{t-1}})$.

$$\boldsymbol{q}(\mathbf{x_t}|\mathbf{x_{t-1}}) = \boldsymbol{N}(\sqrt{\mathbf{1}-\boldsymbol{\beta_t}}\mathbf{x_{t-1}}, \boldsymbol{\beta_t})$$

Where $\boldsymbol{N}$ is our normal distribution, and we're assuming a diagonal variance matrix with value $\boldsymbol{\beta_t}$.

Forward diffusion, illustrated





Forward diffusion, continued

This is ok, except for the fact that we need to generate noise and add it to $\mathbf{x_0}$ many times. Can't we do better?

It turns out that the answer is yes, using our old friend the reparameterization trick, but doing so recursively. This allows us to jump from $\mathbf{x_0}$ straight to $\mathbf{x_t}$.

$$\begin{aligned}
\mathbf{x_t} &= \sqrt{\mathbf{1-\beta_t}}\mathbf{x_{t-1}} + \sqrt{\boldsymbol{\beta_t}}\boldsymbol{\epsilon_t} \\
&= \sqrt{\mathbf{1-\beta_t}}\left[\sqrt{\mathbf{1-\beta_{t-1}}}\mathbf{x_{t-2}} + \sqrt{\boldsymbol{\beta_{t-1}}}\boldsymbol{\epsilon_{t-1}}\right] + \sqrt{\boldsymbol{\beta_t}}\boldsymbol{\epsilon_t} \\
&\vdots \\
&= \sqrt{\bar{\boldsymbol{\alpha}}_t}\mathbf{x_0} + \sqrt{\mathbf{1-\bar{\alpha}_t}}\boldsymbol{\epsilon_t}
\end{aligned}$$

where we assume that $\boldsymbol{\epsilon_t} = \boldsymbol{\epsilon_{t-1}} = \cdots = \boldsymbol{\epsilon} \sim \boldsymbol{N(0,1)}$, and $\boldsymbol{\alpha_t} = \mathbf{1} - \boldsymbol{\beta_t}$, $\bar{\boldsymbol{\alpha}}_t = \prod_1^t \boldsymbol{\alpha_t}$.

Forward diffusion, continued more

So rather than use $q(\mathbf{x}_t|\mathbf{x}_{t-1}) = N(\sqrt{1-\beta_t}\mathbf{x}_{t-1}, \beta_t)$, we instead use

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

We thus don't need to compute every value, $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_{t-1}$, to calculate $\mathbf{x}_t$.

- Since β_t is a hyperparameter we can precompute $\bar{\alpha}_t$ for all t.
- The variance parameter β_t can be a constant, or it can change on a schedule.
- The original DDPM authors used a linear schedule, with $\beta_1 = 10^{-4}$ and $\beta_T = 0.02$.
- More recently it's been demonstrated that a cosine schedule works better.

This class of diffusion models is known as DDPM (Denoising Diffusion Probabilistic Models).

Reverse diffusion

Now we want to reverse the process: start with purge Gaussian noise, and end up with an image.

- To reverse the process we will need to learn $q(\mathbf{x}_{t-1}|\mathbf{x}_t)$.
- If T is large then $\mathbf{x}_T$ is essentially just noise.
- Thus, if we have $q(\mathbf{x}_{t-1}|\mathbf{x}_t)$ we can start with just noise and reverse the process until we end up with $\mathbf{x}_0$.
- This is equivalent to sampling from $P(\mathbf{x})$, the probability distribution of the original data set.

But how do we do that? How do we get $q(\mathbf{x}_{t-1}|\mathbf{x}_t)$?

Reverse diffusion, continued

Using an approach similar to the one we used for Variational Autoencoders, we can model $q(\mathbf{x}_{t-1}|\mathbf{x}_t)$ with a different function, $p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)$.

- If β_t is small enough, we can take $q(\mathbf{x}_{t-1}|\mathbf{x}_t)$ to be Gaussian, with $q(\mathbf{x}_{t-1}|\mathbf{x}_t) = N(\mu_q(\mathbf{x}_t, t), \Sigma_q(\mathbf{x}_t, t))$
- Since $q(\mathbf{x}_{t-1}|\mathbf{x}_t)$ is Gaussian, we choose $p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)$ to be Gaussian as well.
- $p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t) = N(\mu_\theta(\mathbf{x}_t, t), \Sigma_\theta(\mathbf{x}_t, t))$

Finally, applying the reverse process to $\mathbf{x}_T$ to arrive at $\mathbf{x}_0$ we have

$$p_\theta(\mathbf{x}_{0:T}) = p(\mathbf{x}_T) \prod_{t=1}^{T} p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)$$

where $p_\theta(\mathbf{x}_{0:T})$ is known as a trajectory, and $p(\mathbf{x}_T) = N(\mathbf{0}, \mathbf{1})$.

Getting the reverse probability

We already determined, some slides ago, that

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

Using this, and Bayes' theorem, we can condition $q(\mathbf{x}_{t-1}|\mathbf{x}_t)$ on $\mathbf{x}_0$:

$$\begin{aligned} q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0) &= \frac{q(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{x}_0) q(\mathbf{x}_{t-1}|\mathbf{x}_0)}{q(\mathbf{x}_t|\mathbf{x}_0)} \\ &= \frac{N(\sqrt{\alpha_t}\mathbf{x}_{t-1}, (1-\alpha_t)) N(\sqrt{\bar{\alpha}_{t-1}}\mathbf{x}_0, (1-\bar{\alpha}_{t-1}))}{N(\sqrt{\bar{\alpha}_t}\mathbf{x}_0, (1-\bar{\alpha}_t))} \\ &\vdots \\ &\propto N\left(\frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})\mathbf{x}_t + \sqrt{\bar{\alpha}_{t-1}}(1-\alpha_t)\mathbf{x}_0}{1-\bar{\alpha}_t}, \frac{(1-\alpha_t)(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}\right) \end{aligned}$$

SciNet

Getting the reverse probability, continued

Ok, so we have that

$$\mu_q(\mathbf{x}_t, \mathbf{x}_0, t) = \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})\mathbf{x}_t + \sqrt{\bar{\alpha}_{t-1}}(1-\alpha_t)\mathbf{x}_0}{1-\bar{\alpha}_t}$$

But recall that

$$\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\epsilon_t$$

and thus, after some algebra, we have

$$\mu_q(\mathbf{x}_t, t) = \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon_t\right)$$

Getting the reverse probability, continued more

Ok, we've got a simplified version of $q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0)$:

$$q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0) = N\left(\mu_q(\mathbf{x}_t, t), \frac{(1-\alpha_t)(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}\right)$$

with

$$\mu_q(\mathbf{x}_t, t) = \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon_t\right)$$

How does that help us? But we still need $p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)$. What's the plan there?

We can follow an approach similar to the one we used for Variational Autoencoders. We can try to maximize the log of $p_\theta(\mathbf{x})$.

Bayesian inference, again

We now follow a similar calculation which we used for Variational Autoencoders.

$$
\begin{aligned}
\log\left(p_{\boldsymbol{\theta}}(\mathbf{x})\right) &= \log \int p_{\boldsymbol{\theta}}(\mathbf{x}_{0:T}) d\mathbf{x}_{1:T} \\
&= \log \int p_{\boldsymbol{\theta}}(\mathbf{x}_{0:T}) \frac{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}{q(\mathbf{x}_{1:T}|\mathbf{x}_0)} d\mathbf{x}_{1:T} \\
&= \log \mathbb{E}_{q(\mathbf{x}_{1:T}|\mathbf{x}_0)} \left[\frac{p_{\boldsymbol{\theta}}(\mathbf{x}_{0:T})}{q(\mathbf{x}_{1:T}|\mathbf{x}_0)} \right] \\
&\geq \mathbb{E}_{q(\mathbf{x}_{1:T}|\mathbf{x}_0)} \left[\log \left(\frac{p_{\boldsymbol{\theta}}(\mathbf{x}_{0:T})}{q(\mathbf{x}_{1:T}|\mathbf{x}_0)} \right) \right]
\end{aligned}
$$

Where we have used Jensen's inequality in the last step.

Bayesian inference, again, continued

$$
\begin{aligned}
\log\left(p(\mathbf{x})\right) &\geq \mathbb{E}_{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}\left[\log\left(\frac{p_\theta(\mathbf{x}_{0:T})}{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}\right)\right] \\
&\geq \mathbb{E}_{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}\left[\log\left(\frac{p_\theta(\mathbf{x}_T)\prod_{t=1}^{T} p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)}{\prod_{t=1}^{T} q(\mathbf{x}_t|\mathbf{x}_{t-1})}\right)\right] \\
&\vdots \\
&\geq \mathbb{E}_{q(\mathbf{x}_1|\mathbf{x}_0)}\left[\log\left(p_\theta(\mathbf{x}_0|\mathbf{x}_1)\right)\right] - D_{KL}(q(\mathbf{x}_T|\mathbf{x}_0)||p_\theta(\mathbf{x}_T)) - \\
&\quad \sum_{t=2}^{T} \mathbb{E}_{q(\mathbf{x}_t|\mathbf{x}_0)}\left[D_{KL}\left(q(\mathbf{x}_{t-1}|\mathbf{x}_t,\mathbf{x}_0)||p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)\right)\right]
\end{aligned}
$$

where we have skipped many many steps in the derivation. See the references on the last slide for details.

Understanding the terms

It may not look like it, but this represents progress. Each term can now be calculated from at most one random variable at a time. How do we interpret these terms?

- $\mathbb{E}_{q(\mathbf{x}_1|\mathbf{x}_0)}\left[\log\left(p_\theta(\mathbf{x}_0|\mathbf{x}_1)\right)\right]$: this is analogous to the first term in our VAE derivation. This is the term to be maximized.
- $D_{KL}(q(\mathbf{x}_T|\mathbf{x}_0)||p_\theta(\mathbf{x}_T))$: this estimates how close the noisified input, $\mathbf{x}_T$, is to the standard Gaussian prior. This can be taken to be zero.
- $\mathbb{E}_{q(\mathbf{x}_t|\mathbf{x}_0)}\left[D_{KL}\left(q(\mathbf{x}_{t-1}|\mathbf{x}_t,\mathbf{x}_0)||p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)\right)\right]$: this represents how well we're denoising. We are trying to determine $p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)$ as approximated by $q(\mathbf{x}_{t-1}|\mathbf{x}_t,\mathbf{x}_0)$. This term is minimized when the denoising steps are close to each other, as measured by the KL divergence.

The training work lies in approximating $q(\mathbf{x}_{t-1}|\mathbf{x}_t,\mathbf{x}_0)$ using $p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)$.

Understanding the terms, continued

Recall that both $q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0)$ and $p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)$ can be represented by Gaussians. As such, the KL divergence can be calculated exactly.

$$\begin{aligned} D_{KL}\left(q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0)||p(\mathbf{x}_{t-1}|\mathbf{x}_t)\right) &= D_{KL}\left(N(\mu_q, \Sigma_q(t))||N(\mu_\theta, \Sigma_\theta(t))\right) \\ &\vdots \\ &= \frac{1-\bar{\alpha}_t}{2(1-\alpha_t)(1-\bar{\alpha}_{t-1})}\,||\mu_\theta - \mu_q||_2^2 \end{aligned}$$

where we have assumed the form

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

which should be fine, since $\mathbf{x}_t$ is available at training time.

Understanding the terms, continued more

We will model $\epsilon_\theta(\mathbf{x}_t, t)$ using our neural network. We already know that

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

$$\begin{aligned} D_{KL}\left(q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0)||p(\mathbf{x}_{t-1}|\mathbf{x}_t)\right) &= \frac{1-\bar{\alpha}_t}{2(1-\alpha_t)(1-\bar{\alpha}_{t-1})}\,||\mu_q - \mu_\theta||_2^2 \\ &\vdots \\ &= \frac{(1-\alpha_t)}{2(1-\bar{\alpha}_{t-1})\alpha_t}\,||\epsilon_t - \epsilon_\theta(\mathbf{x}_t, t)||_2^2 \end{aligned}$$

Understanding the terms, continued even more

We already know that

$$\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\epsilon_t$$

$$\begin{aligned} L &= \mathbb{E}_{q(\mathbf{x}_t|\mathbf{x}_0)}\left[\frac{(1-\alpha_t)}{2(1-\bar{\alpha}_{t-1})\alpha_t}\left|\left|\epsilon_t - \epsilon_\theta(\mathbf{x}_t, t)\right|\right|_2^2\right] \\ &= \mathbb{E}_{q(\mathbf{x}_t|\mathbf{x}_0)}\left[\frac{(1-\alpha_t)}{2(1-\bar{\alpha}_{t-1})\alpha_t}\left|\left|\epsilon_t - \epsilon_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\epsilon_t, t)\right|\right|_2^2\right] \end{aligned}$$

The original authors found that the prefactor in front is unnecessary. So we instead use

$$L = \mathbb{E}_{q(\mathbf{x}_t|\mathbf{x}_0)}\left[\left|\left|\epsilon_t - \epsilon_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\epsilon_t, t)\right|\right|_2^2\right]$$

as our loss function.

Diffusion models

How are such models trained?

- We first need to calculate our noise variance schedule, $\{\beta_1, \ldots, \beta_T\}$. From this we get $\alpha_t = 1 - \beta_t$ and $\bar{\alpha}_t = \prod_{i=1}^{t} \alpha_i$.
- We then need a data set on which to train our model. Each data sample in our training batch is created by
  - randomly sampling an $\mathbf{x_0}$ from our original data set,
  - randomly sampling a noise step, t,
  - randomly sampling some noise, to get ϵ_t,
  - combining $\mathbf{x_0}$ and ϵ to create $\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\mathbf{x_0} + \sqrt{1 - \bar{\alpha}_t}\epsilon_t$.
- The model is then trained to predict the noise of the image, ϵ_t, given $\mathbf{x}_t$ and t.
- A mean-squared (quadratic) loss function is used.

This class of diffusion models is known as DDPM (Denoising Diffusion Probabilistic Models).

Example

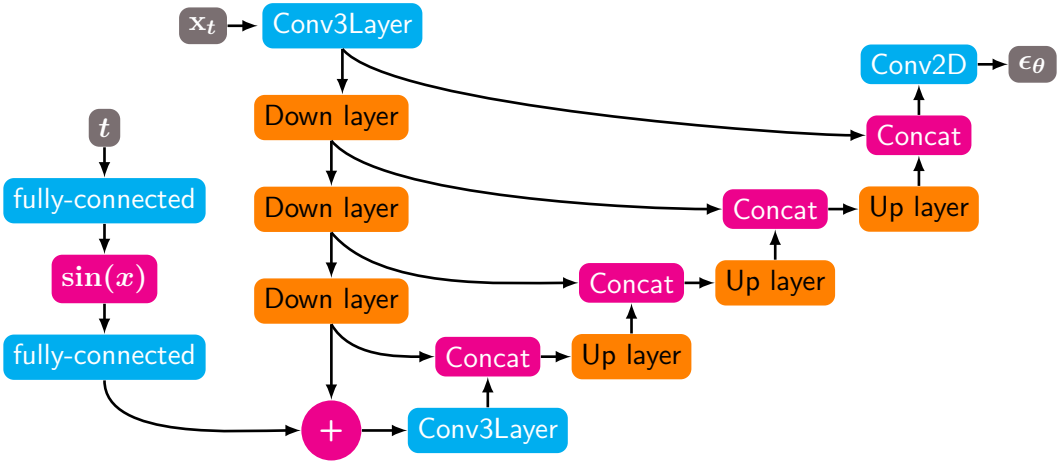
Let's do something more ambitious than MNIST. We will use the cats from the cats-and-dogs data set.

- Colour images of cats.
- The images have been resized to 64 x 64 pixels only (very grainy!).
- We will set $\beta_1 = 10^{-4}$, $\beta_T = 0.02$, with $T = 1000$.

We will use a modified U-Net architecture for our model.

SciNet

U-Net


$\mathbf{x}_t$
Conv3Layer
Conv2D
ϵ_θ
t
Down layer
Concat
fully-connected
Down layer
Concat
Up layer
$\sin(x)$
Down layer
Concat
Up layer
fully-connected
Concat
Up layer
+
Conv3Layer

Diffusion network example, code

```python
# unet.py
import tensorflow as tf
import tensorflow.keras.models as km
import tensorflow.keras.layers as kl
import tensorflow.keras.saving as ks

@ks.register_keras_serializable(package = "MyLayers")
class Conv3Layer(kl.Layer):
  def __init__(self, num_chan, is_res = False, **kwargs):
    super().__init__(**kwargs)
    self.num_chan = num_chan
    self.Conv2D_1 = kl.Conv2D(num_chan, kernel_size = (3, 3),
      padding = 'same', activation = 'linear')
    self.Norm_1 = kl.GroupNormalization(groups = 8)
    self.Conv2D_2 = kl.Conv2D(num_chan, kernel_size = (3, 3),
      padding = 'same', activation = 'linear')
    self.Norm_2 = kl.GroupNormalization(groups = 8)
    self.is_res = is_res
```

```python
# unet.py, continued

  def call(self, inputs):
    x = self.Conv2D_1(inputs)
    x = self.Norm_1(x)
    x1 = kl.ReLU()(x)

    x = self.Conv2D_2(x1)
    x = self.Norm_2(x)
    x = kl.ReLU()(x)

    if self.is_res: x = (x1 + x)

    return x
```

SciNet

Diffusion network example, code, continued

```python
# unet.py, continued

@ks.register_keras_serializable(
  package = "MyLayers")
class UnetDownLayer(kl.Layer):
  def __init__(self, num_chan, **kwargs):
    super().__init__(**kwargs)
    self.num_chan = num_chan
    self.Conv3 = Conv3Layer(num_chan)
    self.MaxPool2D = kl.MaxPool2D(
      pool_size = (2, 2), strides = 2)

  def call(self, inputs):
    x = self.Conv3(inputs)
    x = self.MaxPool2D(x)
    return x
```

```python
# unet.py, continued
@ks.register_keras_serializable(package = "MyLayers")
class UnetUpLayer(kl.Layer):
  def __init__(self, num_chan, **kwargs):
    super().__init__(**kwargs)
    self.num_chan = num_chan
    self.Conv2DTranspose = kl.Conv2DTranspose(num_chan,
      kernel_size = (3, 3), padding = 'same',
      activation = 'linear')
    self.Conv3_1 = Conv3Layer(num_chan)
    self.Conv3_2 = Conv3Layer(num_chan)

  def call(self, inputs):
    img_input, skip = inputs
    x = tf.concat([img_input, skip], -1)
    x = self.Conv2DTranspose(x)
    x = self.Conv3_1(x);     x = self.Conv3_2(x)
    return x
```

SciNet

Diffusion network example, code, continued more

```
# unet.py, continued
def NaiveUnet(input_shape, n_feat):
  img_in = kl.Input(shape = input_shape)
  t_in = kl.Input(shape = (1,))

  x = Conv3Layer(n_feat, is_res = True)(img_in)

  d1 = UnetDownLayer(n_feat)(x)
  d2 = UnetDownLayer(2 * n_feat)(d1)
  d3 = UnetDownLayer(2 * n_feat)(d2)

  thro = kl.AveragePooling2D(
    pool_size = (4, 4))(d3)

  thro = kl.ReLU()(thro)

  t_emb = TimeLayer(2 * n_feat)(t_in)
  t_emb = kl.Reshape((1, 1, 2 * n_feat))(t_emb)
```

```
# unet.py, continued
  thro = kl.Conv2DTranspose(2 * n_feat,
    kernel_size = (3, 3), padding = 'same',
    activation = 'relu')(thro + t_emb)
  thro = kl.GroupNormalization(groups = 8)(thro)
  thro = kl.ReLU()(thro)

  up1 = UnetUpLayer(2 * n_feat)([thro, d3])
  up2 = UnetUpLayer(2 * n_feat)([up1, d2])
  up3 = UnetUpLayer(2 * n_feat)([up2, d1])

  out = kl.Concatenate(axis = -1)([up3, x])

  out = kl.Conv2D(3, kernel_size = (3, 3),
    padding = 'same', activation = 'linear')(out)

  return km.Model(outputs = out,
    inputs = [img_in, t_in])
```

SciNet

Diffusion network example, code, continued even more

```python
# diffusion_model.py
from unet import NaiveUnet
from ddpm import ddpm_schedule
import tensorflow.keras.models as km

n_epochs, batch_size = 300, 64
n_feats = 512
image_shape = (64, 64, 3)

beta1, beta2, T = 1e-4, 0.02, 1000

sched = ddpm_schedule(beta1, beta2, T)

# Go get the data.  Normalize.

model = NaiveUnet(image_shape, n_feat)
model.compile(loss = 'mse',
  metrics = ['mse'], optimizer = 'adam')
```

```python
# diffusion_model.py, continued
for i in range(n_epochs):
  loss = 0
  for image_batch in train_data:
    this_batch = image_batch.shape[0]
    this_size = list(image_shape).insert(0, this_batch)
    eps = tf.random.normal(this_size)

    t_ind = tf.random.uniform([this_batch],
      minval = 1, maxval = T + 1, dtype = tf.int32)
    sqrtab = tf.gather(sched['sqrtab'], t_ind)
    sqrtmab = tf.gather(sched['sqrtmab'], t_ind)
    sqrtab = tf.reshape(sqrtab, (this_batch, 1, 1, 1))
    sqrtmab = tf.reshape(sqrtmab, (this_batch, 1, 1, 1))

    x_t = sqrtab * image_batch + sqrtmab * eps
    t_input = t_ind / T
    loss += model.train_on_batch([x_t, t_input], eps)[0]
```

Training our Diffusion model

This takes many many days of training on a GPU (it has 129 million free parameters).

```
ejspence@mycomp ~>
ejspence@mycomp ~> python diffusion_model.py
Epoch :0 loss :387.72831055894494
Epoch :1 loss :14.70780010893941
Epoch :2 loss :11.63943213969469
⋮
Epoch :1797 loss :3.040174555964768
Epoch :1798 loss :2.9104982246644795
Epoch :1799 loss :2.8991684457287192
ejspence@mycomp ~>
```

Diffusion models, continued more

Ok, so we've got a network that can predict noise. How does that help?

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

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

We can use this to do the reverse diffusion:

- We start with straight-up noise, x_T. We set T to large.
- We then start using our model to predict ϵ_θ, given $\mathbf{x}_t$ and t. We then calculate μ_θ.
- We then generate $\mathbf{x}_{t-1} = \mu_\theta(\mathbf{x}_t, t) + \sqrt{\beta_t}\epsilon_t$. In practice the factor of $(1-\bar{\alpha}_{t-1})/(1-\bar{\alpha}_t)$ is about 1, and is dropped.
- We decrement: $t = t - 1$. We then repeat until we remove all of the noise ($t = 0$).

Typical values of T are about 1000.

SciNet

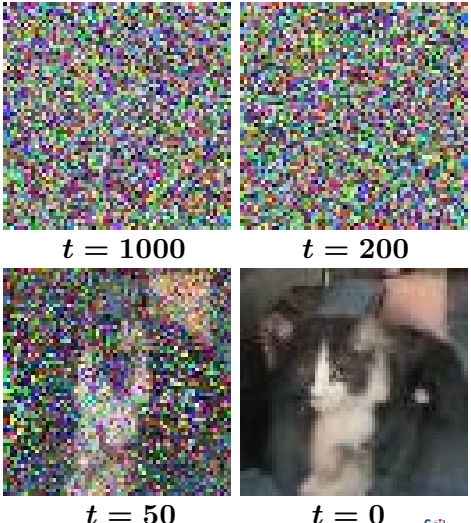
Reverse diffusion example, code

```
# reverse_diffusion.py
import tensorflow as tf, numpy as np
import tensorflow.keras.models as km
from ddpm import ddpm_schedule
from unet import *

def my_sample(image_shape = [64, 64, 3]
  n_sample = 1):

  # The noise schedule.
  beta1, beta2, T = 1e-4, 0.02, 1000
  sched = ddpm_schedule(beta1, beta2, T)

  this_size = list(image_shape)
  this_size.insert(0, n_sample)

  # Our pre-training model.
  model = km.load('model.keras')
```

```
# reverse_diffusion.py, continued
  # Starting image.
  x_t = tf.random.normal(this_size)

  for i in range(T, -1, -1):

    z = tf.random.normal(this_size) if i > 0 else: 0
    t = tf.repeat(tf.constant(i / T), n_sample)

    eps = model.predict([x_t, t], verbose = 0)

    x_t = sched['oneover_sqrta'][i] * \
      (x_t - sched['mab_over_sqrtmab'][i] * eps) +
      sched['sqrt_beta_t'][i] * z

  return np.array(x_t * 255).astype('int16')
```

Reverse diffusion example

```
In [1]:
In [1]: import matplotlib.pyplot as plt
In [2]: import reverse_diffusion as rd
In [3]:
In [3]: a = rd.my_sample()
1000
999
998
⋮
2
1
0
In [3]:
In [3]: plt.imshow(a[0])
In [4]:
```



Reverse diffusion example, continued

Some are good. Some are not

Improvements

Since re-discovery in 2020, a number of improvements to the basic diffusion network have been developed:

- Change the noise schedule, $\boldsymbol{\beta_t}$. We used linear, others have used
  - a cosine dependence for $\boldsymbol{\beta_t}$,
  - have the $\boldsymbol{\beta_t}$ dependence be learnable (quite complicated).
- Add category information to the image, and add that to the diffusion process. The reverse diffusion can then be guided to an image of a particular category. How?
  - Have a separate classification network that works with the diffusion network.
  - Have the classification be part of the diffusion process itself.
- Speed up the reverse diffusion process using 'strided sampling', by taking steps in $\boldsymbol{t}$ of size $\boldsymbol{S}$ instead of 1.

This an active area of research.

Linky goodness

Diffusion models:

- https://arxiv.org/abs/1503.03585
- https://arxiv.org/abs/2006.11239
- https://arxiv.org/abs/2006.09011
- https://arxiv.org/abs/2105.05233
- https://arxiv.org/abs/2208.11970
- https://theaisummer.com/diffusion-models
- https://arxiv.org/abs/2010.02502
- https://lilianweng.github.io/posts/2021-07-11-diffusion-models