SGM_GTAP

July 8, 2025

1 Introduction to score-based generative models

Hugo Gangloff 12/06/25

Modified version of the SGM tutorial by Jakiw Pidstrigach: https://jakiw.com/sgm_intro.

Summary:

1.0.1 Preamble: negative impacts

Mainstream use of diffusion models is that of image generation which is by far the most carbon hungry type of inference (see graph below) and requires some of the biggest GPU architectures to run. Then we should also talk about water and raw material consumption, rebound effect and societal impacts, composition and curation of datasets, etc.

Power Hungry Processing: Watts Driving the Cost of AI Deployment?, Luccioni, Jernite, Strubell, https://arxiv.org/pdf/2311.16863

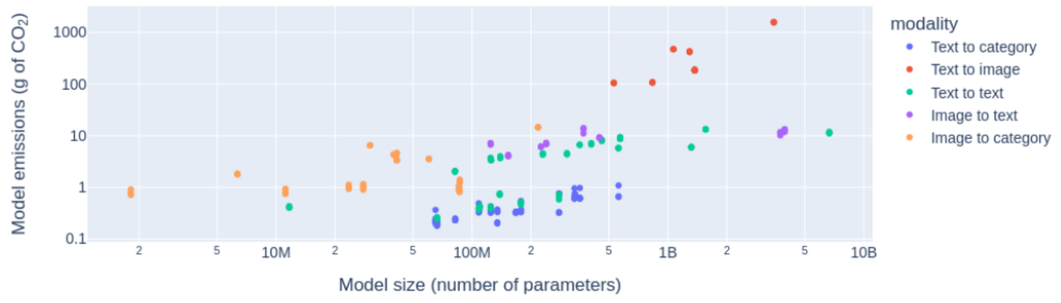


Fig. 2. The 5 modalities examined in our study, with the number of parameters of each model on the x axis and the average amount of carbon emitted for 1000 inferences on the y axis. NB: Both axes are in logarithmic scale.

Now we have text to video https://stability.ai/stable-video

But today, to explore key concepts, we will work in a relatively sober $\mathbb{R}^2$ space and we will not even need a GPU

---

2 Score-based generative models: key equations

The equations are written for unidimensional X_t

Given access to samples $\{x_i\}$ from a target density μ_{data}, generative models have the task to generate more samples from μ_{data}.

- The **forward SDE** adds noise to the data. The forward SDE is run until some terminal time T. Often, it is an Ornstein Ulhenbeck process.

$$\mathrm{d}X_t = -\frac{1}{2}\beta_t X_t + \sqrt{\beta_t}\mathrm{d}B_t,$$
$$X_0 \sim \mu_{\text{data}}.$$

We define $\alpha_t = \int_0^t \beta_s \mathrm{d}s$. Then the **transition kernel** of X_t is given as

$$p_{t|0}(x_t|x_0) = \mathcal{N}(m_t x_0, v_t),$$

where

m_t	$\exp(-\frac{1}{2}\alpha_t)$
v_t	$1 - \exp(-\alpha_t)$

- The **reverse SDE** enables to sample from μ_{data} if started in the distribution $Y_0 \sim p_T$.

$$dY_t = \frac{1}{2}\beta_{T-t}Y_t + \beta_{T-t}\nabla \log p_{T-t}(Y_t) + \sqrt{\beta_{T-t}}\mathrm{d}B_t.$$

- **Noise schedule**: in what's above, we will employ the function

$$\beta_t = \beta_{\min} + t(\beta_{\max} - \beta_{\min}).$$

  In this case we get

$$\alpha_t = t\beta_{\min} + \frac{1}{2}t^2(\beta_{\max} - \beta_{\min}).$$
$$\beta_{\min} = 0.001, \beta_{\max} = 3.$$

  (See Jakiw original tutorial for some references and more discussion)

- **Time interval**: we will always run the forward SDE until time $T = 1$. Therefore, the time interval for the backward SDE is also $[0, 1]$. We discretize this time interval into discrete times $(t_i)_{i=1}^R$, $t_0 = 0, t_R = 1$ and run the above scheme.

The problems We cannot access the marginals p_t of the forward SDE, since we do not know $p_0 = \mu_{\text{data}}$. This leads to the following two problems.

1. **We do not have access to** p_T, the initial condition of the the reverse SDE,
2. For the same reason **we do not know** p_t **and therefore the drift** $\nabla \log p_{T-t}$ **of the reverse SDE**.

As you may have guessed, there are answers to the above:

1. The marginals of an Ornstein-Uhlenbeck converge to $\mathcal{N}(0, I)$ at an exponential speed $p_t \to \mathcal{N}(0, I)$, independently of p_0. **Therefore, we can start the reverse SDE in** $q_0 = \mathcal{N}(0, I) \approx p_T$**. Notably, we do not need the forward SDE in our process**.
2. **We train a neural network approximation** $s_\theta(x, t) \approx \nabla \log p_t(x)$ **using score matching,** to be able to run the reverse SDE.

The loss We train a Neural Network to approximate $\nabla \log p_t$. We do this by so-called Score-Matching Techniques. The optimal loss we would like to minimize is

$$L(\theta, t) = \mathbb{E}_{x \sim p_t(x)}[\|\nabla \log p_t(x) - s_\theta(x, t)\|^2] = \mathbb{E}_{x_0 \sim \mu_{\text{data}}} \mathbb{E}_{x \sim p_{t,0}(x|x_0)}[\|\nabla \log p_t(x) - s_\theta(x, t)\|^2]$$

But its usage is not possible. We can show that (see original Jakiw tutorial) we can use the **denoising score matching objective** objective

$$\bar{L}(\theta, t) = \mathbb{E}_{x_0 \sim \hat{\mu}_{\text{data}}} \mathbb{E}_{x \sim p_{t,0}(x|x_0)}[\|\nabla \log p_{t,0}(x_t|x_0) - s_\theta(x, t)\|^2],$$

Since $p_{t,0}$ is Gaussian we can fully evaluate the above gradient as

$$\nabla \log p_{t,0}(x|x_0) = \nabla \log \left((2\pi v_t)^{-d/2} \exp(-\frac{\|x - m_t x_0\|^2}{2v_t}) \right) = -\frac{(x - m_t x_0)}{v_t}.$$

Finally, we want to optimize the network for all t, not just one specific t, and therefore define

$$\bar{L}(\theta) = \mathbb{E}_{t \sim U[0,1]}[\bar{L}(\theta, t)].$$

This loss can now be approximated by randomly choosing datapoints from the training batch (as samples from $\hat{\mu}_{\text{data}}$ and also randomly generating times $t \sim U[0, 1]$).

Note that in the code below, the loss will be implemented with the simplification:

$$\mathbb{E}_{x_0 \sim \hat{\mu}_{\text{data}}} \mathbb{E}_{x \sim p_{t,0}(x|x_0)}[\|\nabla \log p_{t,0}(x_t|x_0) - s_\theta(x, t)\|^2] = \mathbb{E}_{x_0 \sim \hat{\mu}_{\text{data}}} \mathbb{E}_{z \sim N(0,1)}[\| - \frac{(\sqrt{v_t} z + m_t x_0 - m_t x_0)}{v_t} - s_\theta(x, t)\|^2]$$

Hence

$$\mathbb{E}_{x_0 \sim \hat{\mu}_{\text{data}}} \mathbb{E}_{x \sim p_{t,0}(x|x_0)}[\|\nabla \log p_{t,0}(x_t|x_0) - s_\theta(x, t)\|^2] \propto \mathbb{E}_{x_0 \sim \hat{\mu}_{\text{data}}} \mathbb{E}_{z \sim N(0,1)}[\|z + \sqrt{v_t} s_\theta(x, t)\|^2]$$

3 Score-based generative models: a simple JAX implementation

3.0.1 Setting up the environment

The most straightforward approach is to create a new Python environment

`{bash} conda create -n sgm python=3.13 conda activate sgm pip install jax[cuda12] jaxlib equinox optax matplotlib jupyter jaxtyping tqdm`

Note: get rid of `[cuda12]` for a CPU installation

3.0.2 Python imports

[1]:
```python
import os
os.environ["JAX_PLATFORMS"]="cpu"
```

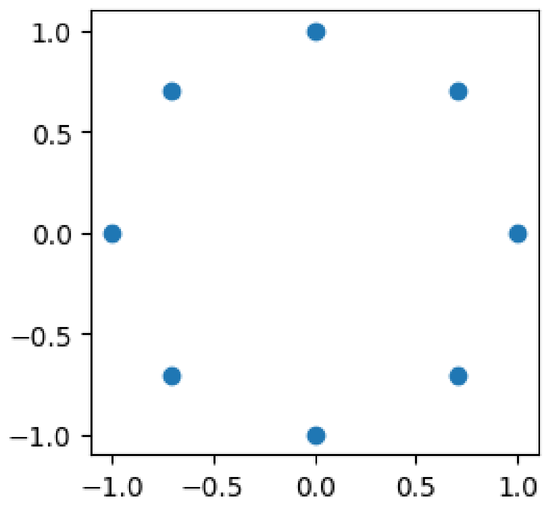
[2]:
```python
from functools import partial

import jax
import jax.numpy as jnp
from jax import jit, grad, vmap
import equinox as eqx
import matplotlib.pyplot as plt
import optax
from jaxtyping import PyTree, Array, Key, Float
from tqdm import tqdm
plt.rcParams["figure.figsize"] = (3, 3)
```

[3]:
```python
key = jax.random.key(0) # create the original key from seed=0
```

3.0.3 Generate samples from μ_{data}

[4]:
```python
def sample_sphere(J: int):
    """
    2 dimensional sample

    N_samples: Number of samples
    Returns a (N_samples, 2) array of samples
    """
    alphas = jnp.linspace(0, 2*jnp.pi * (1 - 1/J), J)
    xs = jnp.cos(alphas)
    ys = jnp.sin(alphas)
    mf = jnp.stack([xs, ys], axis=1)
    return mf
```

[5]:
```python
J = 8
mf = sample_sphere(J)
plt.scatter(mf[:, 0], mf[:, 1])
plt.show()
```



3.0.4 Time discretization

[6]:
```
R = 1000
train_times = jnp.arange(1, R) / (R-1)
```

3.0.5 Some helper functions

[7]:
```
beta_min = 0.001
beta_max = 3

def beta_t(t: float):
    """
    t: time
    returns beta_t as explained above
    """
    return beta_min + t*(beta_max - beta_min)

def alpha_t(t: float):
    """
    t: time
    returns alpha_t as explained above
    """
    return t*beta_min + 0.5 * t**2 * (beta_max - beta_min)

def drift(x: Float[Array, "n_samples N"], t:float):
    """
    x: location of J particles in N dimensions, shape (J, N)
    t: time
    returns the drift of a time-changed OU-process for each batch member, shape␣
  ↪(J, N)
    """
    return -0.5*beta_t(t)*x

def dispersion(t:float):
    """
    t: time
    returns the dispersion
    """
    return jnp.sqrt(beta_t(t))
```

3.0.6 Initialize the neural network

equinox: One of the three big JAX neural network libraries (with Haiku and Flax). We like it because the modules are Python immutable dataclasses: strict rules that play nicely with JAX for code clarity and efficiency. [link to the documentation]

[8]:
```
# We need a random key for the neural network module
# initialization: explicit random key handling
key, subkey = jax.random.split(key)

#initialize the model weights
score_model = eqx.nn.MLP(
    in_size=6, # x (2D) + t (4D) concatenated in entry. Will have some time␣
  ↪embedding additional input
    out_size=2, # \nabla log p_{T-t}(x_{T-t}) is 2D if x is 2D
    width_size=256,
    depth=4,
    activation=jax.nn.relu,
    key=subkey
)
```

Now, a point that is specific to equinox:

[9]:
```
# score_model includes several things that aren't parameters!
# We need to split our model into the bit we want to differentiate (its␣
  ↪parameters),
# and the bit we don't (everything else), this is done with the filter spec␣
  ↪argument
score_params_init, score_static = eqx.partition(score_model, filter_spec=eqx.
  ↪is_array)
# Later on, we reconstruct score_model with eqx.partition to apply it
```

3.0.7 Initialize the optimizer

optax: Most popular optimization library in the JAX ecosystem. Quite similar to Pytorch optimizers but again, we need to explicitly take care of the optimizer state that is updated along the epoch (same philosophy as JAX random keys). We like it because optax is built as functions that transform optimizer states (Python NamedTuples), and nothing more! [link to the documentation]

[10]:
```
#Initialize the optimizer
batch_size = 16
optimizer = optax.adam(1e-3)
opt_state = optimizer.init(score_params_init)
```

3.0.8 Loss function and Update step

Define a loss function and an update step. The update step basically takes the gradient of the loss function and then applies a gradient descent (or rather ADAM) step, to update the current weights with the attained gradient. The full update_step function can be jitted (compiled), making the whole process very fast.

Vectorized computations with JAX: As you may have noticed, we did not mention the batch dimension when instanciating the neural network. This is the classical way to work in JAX: functions are defined (mathematically), for one sample, and we extend them to work on batchs afterwards with JAX vmap function: [link to the doc].

[11]:
```
def loss_fn(
    score_params: PyTree,
    score_static: PyTree,
    key: Key,
    batch_x0: Float[Array, "batch_size 2"]
) -> Float[Array, " "]:
    r"""
    returns an random (MC) approximation to the loss \bar{L} explained above
    """
    score_model = eqx.combine(score_params, score_static) # reconstruct the NN␣
 ↪to be able to apply it
    # vectorization of the score_model.__call__() function with vmap
    # score_model signature: R^6 -> R^2
    vectorized_score_model = jax.vmap(score_model)
    # vectorized_score_model signature: R^bx6 -> R^bx2 where b is the batch␣
 ↪dimension

    key, subkey = jax.random.split(key)

    t = jax.random.choice(subkey, train_times, (batch_size, 1)) # t \sim U[0, 1]

    mean_coeff = jnp.exp(-0.5 * alpha_t(t)) # mean of p_{t|0}(x_t | x_0)
    stds = jnp.sqrt(1 - jnp.exp(-alpha_t(t))) # std of p_{t|0}(x_t | x_0)

    key, subkey = jax.random.split(key)
    noise = jax.random.normal(subkey, batch_x0.shape)

    # 2D sampling of Xt:
```

```
    # mean_coeff and stds are (batch_size, 1) thus broadcastable with
    # batch_x0 and noise which are (batch_size, 2)
    Xt = batch_x0 * mean_coeff + noise * stds # Xt \sim p_{t,0}(x | x_0)

    # encode t with some Fourier features before feeding it to the score NN
    t_enc = jnp.concatenate([t - 0.5, jnp.cos(2*jnp.pi*t), jnp.sin(2*jnp.pi*t),␣
↪-jnp.cos(4*jnp.pi*t)],axis=1)

    s_x_t = vectorized_score_model(jnp.concatenate([Xt, t_enc], axis=1)) #␣
↪s_theta(x,t)

    loss = jnp.mean(jnp.sum((stds * s_x_t + noise)**2, axis=-1))

    return loss
```

Computing derivatives with JAX: the functional approach is very mathematically pleasing when it comes to differentiation. Just call df=jax.grad(f) and you get back a function df that is simply the derivative of f. There are functions to compute derivatives in JAX with many different options: [link to the documentation].

Just in Time compilation with JAX: One of JAX key concepts is Just-In-Time (JIT) compilation which greatly improves performances by pre computing some operations. Below the update step is decorated in order to be jitted. Some constrains apply: not all functions can be jitted (they need to be pure) and arguments that cannot be traced (~ do not mix well with the precomputations, i.e., all non-valid JAX types, i.e., all that is not jnp.arrays, PyTree of jnp.arrays or custom class with explicit flatten/unflatten) should be marked as static: [link to the documentation]

[12]:
```
@partial(jit, static_argnums=[1])
def update_step(
    score_params: PyTree,
    score_static: PyTree,
    key: Key,
    batch_x0: Float[Array, "batch_size 2"],
    opt_state: optax.OptState
) -> tuple[Float[Array, " "], PyTree, optax.OptState]:
    """

    takes the gradient of the loss function and updates the model weights␣
↪(params) using it. Returns
    the value of the loss function (for metrics), the new params and the new␣
↪optimizer state
    """
    # here we use the jax.value_and_grad function. jax.value_and_grad(loss_fn)␣
↪is itself a function with
    # the same arguments as loss_fn which returns two values the value of␣
↪loss_fn and the derivative of loss_fn
    # when evaluated at (score_params, score_static, key, batch_x0)
    val, grads = jax.value_and_grad(loss_fn)(score_params, score_static, key,␣
 ↪batch_x0)
    updates, opt_state = optimizer.update(grads, opt_state)
    score_params = optax.apply_updates(score_params, updates)
    return val, score_params, opt_state
```

3.1 Neural network training

[13]:
```
score_params = score_params_init # rerun this cell to reset parameters
```

[14]:
```
N_epochs = 60000 # 60000 we start to see overfitting to the eight-sample␣
 ↪dataset -> Early stopping seems a critical aspect here
train_size = mf.shape[0]
batch_size = train_size
# batch_size = min(train_size, batch_size)
steps_per_epoch = train_size // batch_size
losses = []
for k in tqdm(range(N_epochs)):
    key, subkey = jax.random.split(key)

    # Note that the dataset is ridiculously small and all the manipulations␣
 ↪below are not really needed
    perms = jax.random.permutation(subkey, train_size)
    perms = perms[:steps_per_epoch * batch_size]  # skip incomplete batch
    perms = perms.reshape((steps_per_epoch, batch_size))
    for perm in perms:
        batch_x0 = mf[perm, :]
        key, subkey = jax.random.split(key)
        loss, score_params, opt_state = update_step(score_params, score_static,␣
 ↪subkey, batch_x0, opt_state)
        losses.append(loss)
    if (k+1) % 5000 == 0:
        mean_loss = jnp.mean(jnp.array(losses))
        print("Epoch %d \t, Loss %f " % (k+1, mean_loss))
        losses = []
```

```
  9%|                                  | 5112/60000 [00:05<01:39, 549.88it/s]

Epoch 5000      , Loss 1.013629

 17%|                                 | 10111/60000 [00:11<00:59, 841.77it/s]

Epoch 10000     , Loss 0.869420

 25%|                                | 15143/60000 [00:16<01:10, 638.78it/s]

Epoch 15000     , Loss 0.815488

 34%|                               | 20165/60000 [00:21<01:03, 629.51it/s]

Epoch 20000     , Loss 0.787861
```

```
 42%|                              | 25159/60000 [00:26<01:04, 541.63it/s]
Epoch 25000     , Loss 0.761295
 50%|                             | 30140/60000 [00:31<00:31, 942.66it/s]
Epoch 30000     , Loss 0.751818
 59%|                            | 35191/60000 [00:36<00:38, 640.45it/s]
Epoch 35000     , Loss 0.750592
 67%|                           | 40105/60000 [00:41<00:33, 587.99it/s]
Epoch 40000     , Loss 0.747150
 75%|                          | 45091/60000 [00:45<00:21, 693.23it/s]
Epoch 45000     , Loss 0.739544
 84%|                         | 50108/60000 [00:50<00:13, 721.10it/s]
Epoch 50000     , Loss 0.741000
 92%|                        | 55169/60000 [00:55<00:07, 639.68it/s]
Epoch 55000     , Loss 0.730837
100%|                       | 60000/60000 [01:00<00:00, 989.14it/s]
Epoch 60000     , Loss 0.731699
```

- Running time on a Nvidia T600 Laptop GPU with Cuda 12.8: [01:41<00:00, 592.12it/s]
- Running time on Intel i7 CPU: [01:00<00:00, 989.14it/s]

3.2 Sampling from the reverse SDE with the score $s_\theta(x, t)$

The reverse SDE is implemented using the Euler-Maryuama scheme. To advance the SDE by δt, we implement the following iteration,

$$Y_{t_{i+1}} = Y_{t_i} + (t_{i+1} - t_i)\left(\frac{1}{2}\beta_{T-t_i} Y_{t_i} + \beta_{T-t_i} s_\theta(Y_{t_i}, t_i)\right) + Z_{t_{i+1}-t_i},$$

where $Z_{t_{i+1}-t_i}$ is a random variable with distribution

$$Z_{t_{i+1}-t_i} \sim \mathcal{N}(0, (t_{i+1} - t_i)\sqrt{\beta_{T-t}}),$$

and where we hope that

$$s_\theta(Y_{t_i}, t_i) \approx \nabla \log p_{T-t_i}(Y_{t_i}).$$

The time interval for the backward SDE is also $[0, T]$, we set $T = 1$. We keep the discretization in R time steps defined above.

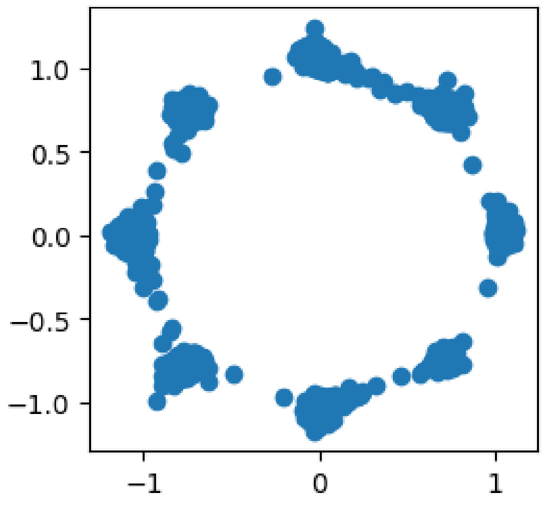
Recall that we set $T = 1$.

Optimized for loops with JAX: when for loops are unavoidable, for example in purely iterative algorithms such as Euler Maruyama, JAX offers the jax.lax.scan function whose content is automatically jitted. [link to the documentation]

[15]:
```
T = 1

def reverse_sde(
    key: Key,
    N: int,
    n_samples: int,
    score_model: eqx.Module
) -> Float[Array, "n_samples N"]:
    """
    Eular Maruyama in N dimensions

    Run a scan loop for all the n_samples samples in parallel
    """
    vectorized_score_model = jax.vmap(score_model)

    def f(carry, times):
        t, dt = times
        x, key = carry
        disp = dispersion(T - t)
        t = T - jnp.ones((x.shape[0], 1)) * t
        t_enc = jnp.concatenate([t - 0.5, jnp.cos(2*jnp.pi*t), jnp.sin(2*jnp.
 ↪pi*t), -jnp.cos(4*jnp.pi*t)],axis=1)

        reverse_drift = -drift(x, T - t) + disp ** 2 *␣
 ↪vectorized_score_model(jnp.concatenate([x, t_enc], axis=1))

        key, subkey = jax.random.split(key)
        noise = jax.random.normal(subkey, x.shape)

        x = x + dt * reverse_drift + jnp.sqrt(dt) * disp * noise
        return (x, key), x

    key, subkey = jax.random.split(key)
    initial = jax.random.normal(subkey, (n_samples, N))
    dts = train_times[1:] - train_times[:-1]
    times = jnp.stack([train_times[:-1], dts], axis=1)
    key, subkey = jax.random.split(key)
    (x, _), samples = jax.lax.scan(f, (initial, subkey), times)
    return samples
```

[16]:
```
key, subkey = jax.random.split(key)
score_model = eqx.combine(score_params, score_static)
samples = reverse_sde(subkey, 2, 5000, score_model)
```

```
[17]: plt.scatter(samples[-1, :, 0], samples[-1, :, 1])
      plt.show()
```


1.0
0.5
0.0
−0.5
−1.0
−1
0
1