1 Solution

Summarizing work on the Pytorch friendly time-dependent ODE solver. Unfortunately, the solve_ivp function from Scipy does not work with Pytorch and it's recorded gradients. This is a real shame because it works with complex values and can take an arbitrary number of arguments for the differential function it's solving. The only Pytorch friendly ODE solver I could find is here, which has an admittedly annoying syntax, isn't complex compatible, and can only differentiate functions that take two inputs (dependent variable - t and initial conditions - y_0). Therefore I had to define a class called Applied_Hamiltonian which contains all of the parameters accompanying t and y_0, contains the method generating $\Omega_k(t) = \sum_{m=1,\dots M} A_{m,k} e^{-(t-t_m)^2/a^2}$ for $H(t) = \sum_k \Omega_k(t) * \sigma_k^{\gamma_k}$, and contains methods implementing the set of real differential equations - Eq. 2 - defining the Hamiltonian evolution according to Eq. 1.

$$dU(t) = -i(H_0 + H(t))U(t) \tag{1}$$

Where $d \equiv \frac{d}{dt}$ and H_0 is the static Hamiltonian assumed to be real. Defining $U(t) = U_r(t) + U_i(t)$ and $H(t) = H_r(t) + iH_i(t)$ and dropping the (t) dependence notation for convenience, the above can be written as:

$$d[U_r + iU_i] = -i(H_0 + H_r + iH_i)(U_r + iU_i) = -i(H_0U_r + iH_0U_i + H_rU_r + iH_rU_i + iH_iU_r - H_iU_i)$$

$$d[U_r + iU_i] = i(H_iU_i - H_0U_r - H_rU_r) + (H_0U_i + H_rU_i + H_iU_r)$$

Now, grouping the imaginary versus real terms, we have the set of real differential equations:

$$\begin{aligned} dU_r &= H_0U_i + H_rU_i + H_iU_r \\ dU_i &= H_iU_i - H_0U_r - H_rU_r \end{aligned} \tag{2}$$

Where $H = Hr + Hi$ is generated via $H(t) = \sum_k \Omega_k(t) * \sigma_k^{\gamma_k}$ where $\Omega_k(t) = \sum_{m=1,\dots M} A_{m,k} e^{-(t-t_m)^2/a^2}$.

2 Syntax

First of all, make sure to install torchdiffeq to your anaconda environment: with Conda env activated, do:
which pip

Should output .../anaconda3/envs/Pytorch/bin/pip (if not do conda install pip)
Use that pip to install the library as:
.../anaconda3/envs/Pytorch/bin/pip install torchdiffeq
Now the syntax is going to be very different due to needing to optimize the class Applied_Hamiltonian and it's parameter A. If initializing the class as:

Ht = Applied_Hamiltonian(A, T, gates, H0)

Here $H0$ is the static Hamiltonian (must be of type torch.double), the gates corresponds to the $\sigma_k^{\gamma_k}$ that the Gaussian Pulses are applied to as defined above (sorry if this isn't an accurate variable name), and T is the total/ending time. A is the [M, $N*$(# of 'gates')] tensor of amplitudes, is torch.float, and is set to be optimized (which means all optimizer and gradient calls must be changed from $[R]$ to $Ht.A$).
Once this is defined, we can go on solving the differential equation, this is done by calling the odeint on the Schrodinger_eq method in Applied_Hamiltonian given an evaluation time $t\_list = [0, t_{eval}]$ and an initial condition $U_0 = [U_{0r}, U_{0i}]$. The is done by the call UT = odeint(Ht.Schrodinger_eq, U0, t_list) which will return a tensor of size [len(t_list), 2, 2^L,2^L], the final unitary is then retrieved by reforming the complex unitary from the last time evaluation - i.e. $U_Exp = UT[-1, 0, ...] + 1j * UT[-1, 1, ...]$. This does track the gradient and does converge fairly well (see Fig.1) - though the algorithm is quite a bit slower.

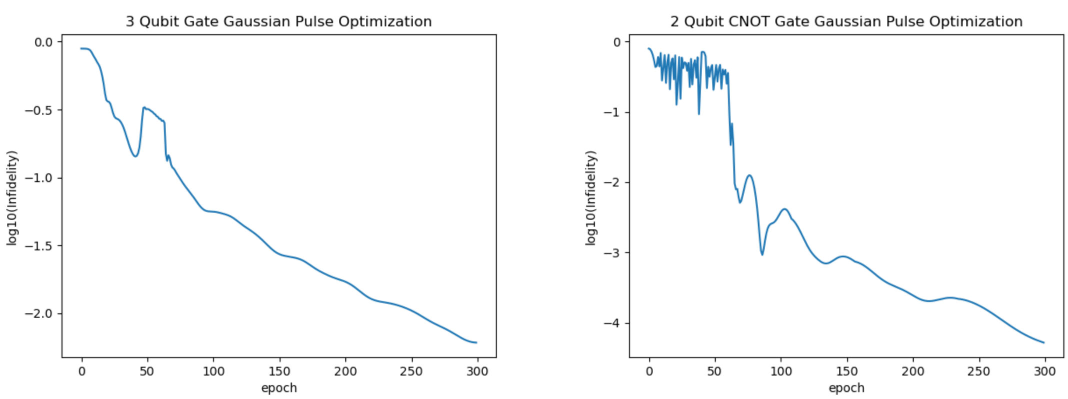

Figure 1: Average Infidility of the Gaussian pulse optimization routine using the Pytorch friendly ODE solver on Eq. 2. Left: Average infidelity of a $M = 12$ Gaussian pulse optimization with a 3 qubit Toffoli gate target, final evaluation time $= \pi$, $J = 1$, $B = 1$, and 300 iterations. Right: Average infidelity of a $M = 6$ Gaussian pulse optimization on a 2 qubit CNOT target gate, with all other parameters identical to the above simulation.