

[핸드 배경]

Reviewer : 김정학

Time series 데이터가 모든 시간에

대해서 정보를 다 갖고 있지 않은

경우가 많다. 이를 해결하기 위해서

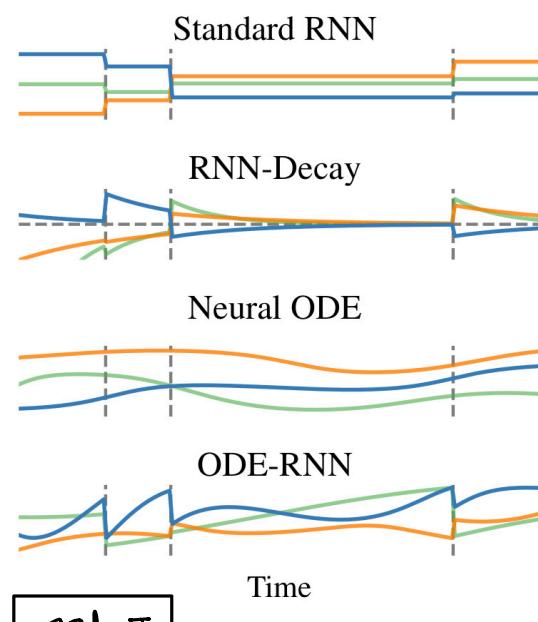
neuralODE를 사용해서 버어 있는

데이터도 알 수 있도록 한다.

(다른게 말하면 각 시간에 대한 정보를

유추할 수 있다)

[비교]



제 I

기준에 버어 있는 데이터가 있는 경우에 사용할 수 있는 방법 0)

'제 I'에 나와 있다. 기본 Standard RNN의 경우 버어 있는
데이터를 무시하는데 이를 통해 학습이 제대로 안될 수 있다.

RNN-Decay 방식은 정보가 있을 때 RNN을 사용하고
시간이 지나면 정보를 흐려지게 한다. (쉽게 얘기하면
data decay). 본문에서 소개하는 ODE-RNN은 정보가
있을 때 RNN으로 업데이트를 하고 없을 때는 neuralode
로 부드럽게 연결된다 (정보를 취약한 가리먼트)

[3단계 설명 (코드 포함)]

- Neural ODE에 대한 back ground가 있다고 가정.

① ODE-RNN

Algorithm 1 The ODE-RNN. The only difference, highlighted in blue, from standard RNNs is that the pre-activations h' evolve according to an ODE between observations, instead of being fixed.

```
Input: Data points and their timestamps  $\{(x_i, t_i)\}_{i=1..N}$ 
 $h_0 = \mathbf{0}$ 
for  $i$  in  $1, 2, \dots, N$  do
     $h'_i = \text{ODESolve}(f_\theta, h_{i-1}, (t_{i-1}, t_i))$                                  $\triangleright$  Solve ODE to get state at  $t_i$ 
     $h_i = \text{RNNCell}(h'_i, x_i)$                                           $\triangleright$  Update hidden state given current observation  $x_i$ 
end for
 $o_i = \text{OutputNN}(h_i)$  for all  $i = 1..N$ 
Return:  $\{o_i\}_{i=1..N}; h_N$ 
```

h_i 는 hidden vector.

(t_{i-1}, t_i) 에서는 데이터가 없어도 ODESolve로 hidden vector를 업데이트한다. t_i 에 데이터가 생기면 (존재하면) RNN 모델을 통해 h_i 를 업데이트한다. 여기서 시간에 대한 정보 t 를 업데이트하는데 사용하지 않는다. 이 방식을 사용하면 RNN-Decay처럼 h_i 가 계속 변하기만 parameter에 의해 시 흐름에 따라 유동적으로 (flexible) 바뀐다.

2 Latent ODEs

ODE-RNN은 실제 데이터의 불확실성을 고려하기
역보존하다. (내 생각에는 h_i 를 다시 복구할 때
제대로 복구가 안되는 것 같은데)

이를 해결하기 위해 Latent ODE를 소개한다.

작동방식은 같다. (의미로.. 코드를 보고야 이해)

① x_i, t_i 로 부터 z 의 μ, σ^2 를 학습. (인코더)

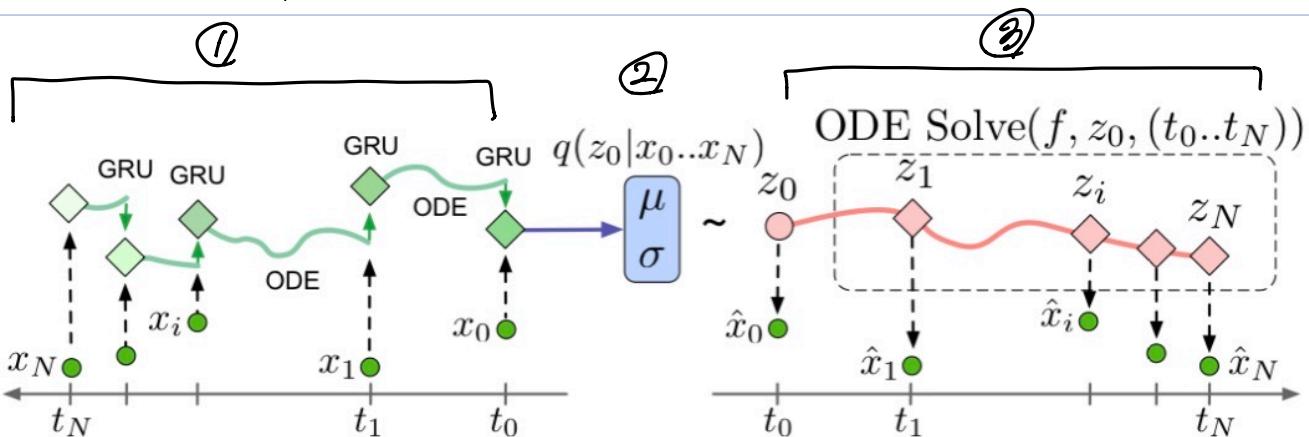
여기에서 μ, σ^2 를 뽑아내기 위해 사용하는 것이
ODE-RNN이다.

② z 로 부터 t_i 에의 x_i 를 학습 (디코더)

①과 ②을 섞으로 나누면 이해와 같다.

$$q(z_0 | \{x_i, t_i\}_{i=0}^N) = \mathcal{N}(\mu_{z_0}, \sigma_{z_0}) \quad \text{where} \quad \mu_{z_0}, \sigma_{z_0} = g(\text{ODE-RNN}_\phi(\{x_i, t_i\}_{i=0}^N))$$

오늘은 3시학입니다 아래와 같다.



① 이 부분에서 ODE-RNN을 사용한다. ($t_N \sim t_0$ 순으로 한다)
 첫번째 \diamond 는 0이다. 관측치가 있을 때는
 RNN으로 업데이트를 해주고 업데이트되는 neural-ode로
 적용한다. 이 때 계속 μ, σ^2 을 업데이트해라.
 마지막이 되었을 때 이 데이터의 μ, σ^2 에 대한
 정보를 가진다.

② ①에서 나온 결과값을 fc-layer를 통해
 차원 축소한다. 뽑아내야 하는 time-stamp
 차원 만큼 μ, σ^2 이 있다.

③ 구해낸 μ, σ^2 을 그대로 사용하는 게 아니라.
 trajectory-num 수 만큼. 복제하고 이를 이용해
 Decoder를 통과해서 t_i 에서 \hat{x}_i 를 얻어낸다.

$$\text{ELBO}(\theta, \phi) = \mathbb{E}_{z_0 \sim q_\phi(z_0 | \{x_i, t_i\}_{i=0}^N)} [\log p_\theta(x_0, \dots, x_N)] - \text{KL} [q_\phi(z_0 | \{x_i, t_i\}_{i=0}^N) || p(z_0)]$$

위 식이 Loss 같다. 앞 부분은 추정한 값이
지나와 가깝게 하기 위함이고 뒤는 추정한
 μ, σ^2 이 너무 벗어나지 않도록 위함으로 보인다(두 부분은
확실치 않다)

[2 외]

1 Poisson process likelihood

시간까지 학습을 해서 정해진 시간외에 알고 싶은
시간에서의 데이리를 구할 수 있음.

2 Batching and computational complexity

각 레이터마다 버어 있는 게 일정하지 않은 데 레이터
길이가 다른 수 있다. 모든 time point를 하나로
합쳐서 사용한다. runtime 크게 영향을 안 준다 함.

3 또 다른 특징

$t_0 \sim t_N$ 내에서의 값을 얻어내는 interpolation
하고 그 뒤를 예측하는 extrapolation에도 효과적이라고 함.